A New Initial Basis for Standard ML
(DRAFT — DO NOT DISTRIBUTE)

June 26, 1995

Contents

Preface

The *Initial Basis* defined in the *Definition of Standard ML* [MTH90] is probably the weakest aspect of the definition. In addition to the expected operators on the standard types (e.g., **`int`**, **`real`**, etc.), it defines a small, and random, collection of utility functions. This basis is woefully inadequate for serious programming, and as a result, each implementation of Standard ML has developed its own extensions. This document is a proposal for a new, richer initial basis for SML, which we hope will be adopted as a replacement for Appendices C and D of the Definition.

This document is organized into two parts. The first discusses the various pieces of the proposed basis, and gives some rationale for the design. The second part is a complete set of manual pages for each proposed module.

Contributors

This document is the result of a collaboration of the Standard ML of New Jersey effort and Harlequin (what is the full name?):

Andrew W. Appel	Princeton University, USA
Matthew Arcus	Harlequin
Nick Barnes (ne Haines)	Harlequin
Dave Berry	Harlequin
Richard Brooksby	Harlequin
Emden R. Gansner	AT&T Bell Laboratories
Lal George	AT&T Bell Laboratories
Lorenz Huelsbergen	AT&T Bell Laboratories
Dave MacQueen	AT&T Bell Laboratories
Brian Monahan	Harlequin
John H. Reppy	AT&T Bell Laboratories
Jon Thackray	Harlequin
Peter Sestoft	Royal Veterinary and Agricultural University, Denmark.

In addition, Peter Lee and Mads Tofte provided helpful comments on drafts of this document.

Part I

Discussion

Chapter 1

Introduction

NOTE: THIS IS AN INCOMPLETE DRAFT

[[Need some words of introduction.]]

Summary

Summary of the proposal:

- Capitalization convention; rules for extensions of initial basis.
- Both arbitrary and fixed-precision integers; implementations are required to implement at least one of these.
- Unsigned integers (called *words*), with literals.
- Multiple precisions of IEEE floating-point allowed. Floating-point semantics specified in more detail, and with more operators, than in the Definition [MTH90].
- Mutable arrays and immutable vectors, with constant-time random-access.
- More comprehensive operators on lists, strings, arrays, vectors, etc.
- Industrial strength input/output; support for both text and binary I/O.
- A useful set of portable operating-system interfaces.
- Minor language changes: adding character literals, and adding overloading of integer, word and of real literals at multiple precisions.
- Admendments for operating specific APIs.

1.1 Conventions and design philosophy

As long as we are doing everything all over again, we can revise the capitalization conventions of the initial basis. We believe, for example, that value constructors should be capitalized to distinguish them from variables; there seems to be wide agreement on this point.

The capitalization convention we use is:

- Alphanumeric value variables in mixed-case, with a leading lower-case letter. Examples: `map`, `openIn`.
- Alphanumeric constructors with a leading upper-case letter. Examples: `SOME`, `NONE`, `Jan`, `Wed`. The only exceptions to this are the identifiers `nil`, `true`, and `false`, where we bow to tradition.
- Type identifiers are all lower case, with underscores.
- Signature identifiers in all caps, words separated by underscore.
- Structure and functor identifiers are mixed-case, with initial letter capitalized.

While capitalization is a touchy subject, we strongly believe that value constructors MUST have a different capitalization from variables. Otherwise, misspelling of a constructor in a pattern-match can result in an error not easily caught by the compiler.[1]

The initial basis is contained in a set of structures. Every type, exception constructor, and value belongs to some structure, although some are also bound in the initial top-level environment. Infix declarations and overloading are top-level definitions.

Functional arguments that are evaluated solely for their side-effects should be required to have a return type of unit. For example, the list application function should have the type:

```
val app : ('a -> unit) -> 'a list -> unit
```

We have tried to use consistent names and *type shapes* for similar operations. For example, the function `Array.app` has the type:

```
val app : ('a -> unit) -> 'a array -> unit
```

which has the same shape as `List.app`. Also, we rely heavily on the module system to structure the name space (e.g., `Array.app` and `List.app`). This means that programmers who use `open` liberally will have to change their ways.

[1]We believe that compilers should generate warning messages to enforce this convention.

Table 1.1: List of required generic signatures

Signature	Description
CONVERT_INT	Conversions between two integer representations.
CONVERT_REAL	Conversions between two real representations.
CONVERT_WORD	Conversions between two unsigned representations.
FLOAT	Generic IEEE floating-point module interface
INTEGER	Generic integer module interface.
MATH	Generic math library interface.
MONO_ARRAY	Mutable monomorphic arrays.
MONO_VECTOR	Immutable monomorphic vectors.
OS	Generic interface to basic operating system features
REAL	Generic real number interface.

1.2 Overview

[[This section is out of date]]

The proposal is organized in to chapters covering related collections of modules. These groupings are:

General General purpose definitions

Arithmetic Integer and real arithmetic and mathematical functions.

Text Strings and characters

Aggregates Arrays and vectors of various kinds.

System Generic operating system interfaces.

Input/Output This includes a low-level extensible I/O interface, and both text and binary I/O streams.

In addition, there is a chapter on the top-level environment and one on language issues, such as overloading and literal values.

We have divided the modules into *required* and *optional* modules. Any conforming implementation of SML will provided implementations of all of the required modules. In addition, if an implementation provides any of the services covered by the optional modules, then they shall conform to the given interfaces. Many of the optional structures are variations on some generic module (e.g., single and double-precision floating-point numbers); Table 1.1 gives a list of required generic signatures. The required structures (and their signatures) are listed in Table 1.2. In addition to the required structures, there are several required aliases:

```
structure LargestInt : INTEGER
structure LargestWord : WORD
structure LargestFloat : FLOAT
```

These are aliases for the largest representation of the given kind, and are used for converting between different sizes (the `LargestFloat` structure is only required if the implementation provides one or more `Float` structures).

[[Are the `SysInt` and `SysWord` structures aliases, or abstract?]]

Table 1.3, which follows the same format, gives the list of optional structures.

1.3 Things to discuss

In the discussion below, we use the term *base type* to refer to the scalar types provided by an implementation (e.g., `bool`, `int`, ...).

Packing/unpacking values

The `Pack`*N*`Big` and `Pack`*N*`Little` structures provide some support for marshalling/unmarshalling of data, but we may want to extend this to other types. The most important of these are the float types. We might add a pack and unpack operation to the `FLOAT` signature:

```
val pack   : Word8.word list -> real
val unpack : real -> Word8.word list
```

The byte order for these operations would be architecture independent (say most-significant byte first). The `pack` operation raises the exception `Pack` if the number of bytes is incorrect.

1.4 Known incompatiblities with the Definition

The revised basis is largely a conservative extension of the basis described in the *Definition*, but there are a few points of incompatiblity:

- The `Io` exception.
- The I/O interfaces. Operations are not at top-level, and some of the functions have changed.
- The semantics of overloading.
- The `implode` and `explode` functions.

- The types of `ord` and `chr`.
- The math functions (`sin`, etc.) are not bound at top-level.
- The addition of word and character literals.
- The overloading of literals and the addition of default overloadings.

Table 1.2: List of required structures

Module	**Signature**	**Description**
Array	ARRAY	Mutable polymorphic arrays.
BinIO	IMPERATIVE_IO	Binary input/output streams and operations.
BinIO.StreamIO	STREAM_IO	
BinIO.StreamIO.PrimIO	PRIM_IO	
Bool	BOOL	Operations on booleans.
Byte	BYTE	Conversions between Word8 and Char
Char	CHAR	Characters
CharArray	MONO_ARRAY	Mutable arrays of characters
CharVector	MONO_VECTOR	Immutable vectors of characters
Date	DATE	Calendar operations
General	GENERAL	General-purpose types, exceptions and miscellaneous operations.
Integer	INTEGER	Default interger structure.
List	LIST	Utility functions on lists.
ListPair	LIST_PAIR	Utility functions on pairs of lists.
Locale	LOCALE	Support for localization.
Math	MATH	Default math structure.
OS	OS	Basic operating system services.
OS.FileSys	OS_FILE_SYS	File status and directory operations
OS.Path	OS_PATH	Pathname operations
OS.Process	OS_PROCESS	Simple process manipulation operations
Real	REAL	Default real structure.
String	STRING	Utility functions on strings (cf., CharVector).
StringCvt	STRING_CVT	Basic string conversions.
Substring	SUB_STRING	Utility functions on pieces of strings.
TextIO	TEXT_IO	Text input/output streams and operations.
TextIO.StreamIO	STREAM_IO	
TextIO.StreamIO.PrimIO	PRIM_IO	
Time	TIME	Representation of time values
Timer	TIMER	Timing operations
Vector	VECTOR	Immutable polymorphic vectors.
Word8	WORD	8-bit unsigned integers
Word8Array	MONO_ARRAY	Arrays of 8-bit unsigned integers
Word8Vector	MONO_VECTOR	Vectors of 8-bit unsigned integers

Table 1.3: List of optional structures

Module	Signature	Description
BoolArray	MONO_ARRAY	Mutable arrays of booleans
BoolVector	MONO_VECTOR	Immutable vectors of booleans
Float	FLOAT	Default floating-point structure.
FloatArray	MONO_ARRAY	Mutable arrays of default floating-point numbers.
FloatMath	MATH	Default floating-point math library.
FloatVector	MONO_VECTOR	Immutable vectors of default floating-point numbers.
Float*n*	FLOAT	Floating-point numbers (*n*-bits, for $n \in \{32, 64, 96, 128\}$).
Float*n*Array	MONO_ARRAY	Mutable arrays of floating-point numbers (*n*-bit floats, $n \in \{32, 64, 96, 128\}$).
Float*n*Math	MATH	Floating-point math library (*n*-bit floats, $n \in \{32, 64, 96, 128\}$).
Float*n*Vector	MONO_VECTOR	Immutable vectors of floating-point numbers (*n*-bit floats, $n \in \{32, 64, 96, 128\}$).
Int*n*	INTEGER	*n*-bit, fixed precision integers
LargeInt	LARGE_INT	Arbitrary-precision integers.
POSIX	POSIX	POSIX 1003.1a binding
POSIX.FileSys	POSIX_FILE_SYS	File and directory operations
POSIX.IO	POSIX_IO	Input/output primitives.
POSIX.Process	POSIX_PROC_ENV	Process primitives
POSIX.ProcEnv	POSIX_PROCESS	Process environment primitives
POSIX.SysDB	POSIX_SYS_DB	System database primitives
POSIX.TTY	POSIX_TTY	Terminal device primitives
SmallInt	INTEGER	Fixed-precision integers.
Word	WORD	Unsigned machine integers
Word*n*	WORD	*n*-bit, unsigned machine integers
WordArray	MONO_ARRAY	Mutable arrays of unsigned machine integers
Word*n*Array	MONO_ARRAY	Mutable arrays of n-bit unsigned machine integers
WordVector	MONO_VECTOR	Immutable vectors of unsigned machine integers
Word*n*Vector	MONO_VECTOR	Immutable vectors of n-bit unsigned machine integers

Chapter 2

General

We include the definition of the ref type here, rather than in a separate signature. This is because the `Ref` structure would be trivial.

We do not include a specification of `type ref` because it has a "strange" equality property that can't be written down in a signature.

We include the datatype `option` because it is widely useful, and because we use it in some of the other structures in this proposal.

A number of common exceptions (`Subscript`, `Size`, `Overflow` and `Div`) are defined in `General`. These are the standard exceptions used by various modules to signal error conditions.

We include the exception `Interrupt`, but we believe it is a bad idea. Allowing an exception to be raised asynchronously, from a source other than the program itself, has a nasty semantics that defeats both compiler optimizations and human understanding of programs. In Standard ML of New Jersey we use a different mechanism (first-class continuations) to allow signals to be sent to programs; see [Rep90] for a more detailed discussion. In the absence of first-class continuations (which we are not proposing to be made Standard), implementations may (but are not required to) raise `Interrupt` upon an external interrupt signal.

Chapter 3

Arithmetic types

The Definition provides limited support for integer and real arithmetic, but does not address the important issue of supporting multiple representations. This chapter presents standard interfaces for integer and real types; the issue of literals is discussed in Section 10.2.

3.1 Integers

There are two possible implementations of integers:

- arbitrary precision ("bigints"),
- fixed precision ("smallints").

Either one is acceptable in a Standard ML compiler, but some implementations may provide both, and there should be a standard way to distinguish them.

We propose a signature `INTEGER` and two structures `LargeInt` and `SmallInt` matching the signature. Finally, a structure `Integer` will be bound to either `LargeInt` or `SmallInt` in any implementation. Implementations must provide at least one of the two integer structures.

[[Multiple fixed-precision integer representations may be provided. These will be named `Int`*n*, where *n* is the number of bits of precision (e.g., `Int32`).]]

3.2 Words

Words are an abstraction of the underlying hardware's machine word. They represent a sequence of `wordSize` bits; an unsigned integer; and a machine-dependent encoding of the `SmallInt.int` type.

The `Word` structure provides logical operations, both logical and arithmetic shifting, unsigned arithmetic, and conversions between the integer type.

[[Multiple word representations may be provided. These will be named `Word`*n*, where *n* is the number of bits of precision (e.g., `Word32`).]]

3.3 Real numbers

Real numbers provide a fairly challenging problem of interface design. There are several possible concrete implementations of "real" numbers:

- Constructive (infinite-precision) reals (e.g., [Vil88]);
- IEEE-754 floating point in several sizes, without infinities or NaN's;
- IEEE-754 floating point in several sizes, with infinities and NaN's;
- Vax, IBM 360, and other floating point representations.

Since the last of these seems to be going the way of the Dodo, we probably should concentrate on IEEE representations.

We require that an SML system provide an implementation of the `REAL` signature, which can use infinite-precision or floating-point representations.

The (optional) structure `ConReal:REAL` (possibly the same structure as `Real`) will be infinite-precision "Constructive Reals."

The implementation may, optionally, provide one or more implementations of the `FLOAT` signature providing various different precisions. These would be named:

`ShortFloat` Short precision (less than 32-bit) floating-point numbers represented as unboxed values to save time and space at the expense of accuracy.

`Float32` Single precision (32-bit) floating point.

`Float64` Double precision (64-bit) floating point.

`Float96, Float128` Higher precision (96 or 128-bit) floating point.

One of these (usually `Float64`) would also be bound to `Float`.

The standard mathematical functions (e.g., `sin`, `sqrt`, etc.) are found in the `Math` structure. For each different representation of reals (e.g., `ConReal`, `Float32`), there is an instance of the `Math` structure (e.g., `MathCon`, `Math32`). Thus, each representation of reals has its own mathematical functions.

3.4 Conversions

With various different representations available, there must be a way to convert between them. There are five different kinds of conversions that must be provided:

- conversions between different sizes of integers (`Cvt.Int`*N*`Int`*M*).
- conversions between different sizes of words (`Cvt.Word`*N*`Word`*M*).
- conversions between different sizes of floating-point numbers (`Cvt.Float`*N*`Float`*M*).
- conversions floating point numbers and integers (`Cvt.Float`*N*`Int`*M*).
- conversions between words and integers (`Cvt.Word`*N*`Int`*M*).

[[There will be a single structure `Cvt` that contains all of the conversion structures as sub-structures.]]

For each pair of float structures F, G (e.g., `Float32`, `Float64`, `Float96`), in the system, such that F`.precision` $<$ G`.precision`, there must also be a structure `Convert`FG matching the signature `CONVERTFLOAT`.

[[What is the behavior of the conversions between the real type of a structure and the default real type? Since the relative precision is not known, this would have to have some default behavior (e.g., `trunc`) when the default `real` type has more information than the target.]]

3.5 Floating-point arrays

For each floating-point structure `Float`*N*, there may be a monomorphic array struture called `Float`*N*`Array` that matches the `MONO_ARRAY` signature.

Chapter 4

Text

This chapter deals with characters and strings. The old basis uses the `int` type to represent single characters. This is unsatisfactory for several reasons:

- no symbolic names for pattern matching single characters
- character to string conversions require unecessary range checks

We propose that the single `string` type provided by the Definition be replaced with two types: `string` and `char`, where the `string` type is a *vector* of characters.

[[we need to think about Unicode]]

[[There should be a `CharVector` structure with `CharVector.vector` matching `String.string`. We may want to add `tabulate` to `String`]]

String conversions

There are conversions to and from strings for all of the base types. Each type has simple `toString` and `fromString` functions for default conversions, as well as more sophisticated `fmt` and `scan` functions. The `scan` functions are polymorphic over an abstract character stream; there general form is:

```
val scan : {getc : 'a -> (char * 'a) option} -> 'a -> (ty * 'a)
```

Chapter 5

Aggregates

This chapter describes various aggregate types that must be primitive in order to guarantee constant time updating and indexing. Implementations are required to provide polymorphic array and vector structures, and signatures for monomorphic arrays and vectors. The polymorphic and monomorphic versions of these types have the same basic operations.

Both vectors and arrays are indexed from `0`; each vector or array structures defines the integer variable `maxlen`, which defines the length of the longest allowed vector or array of that element type. We require that the default integer representation have sufficient precision to index every element of the largest possible array or vector.

5.1 Vectors

Vectors are immutable one-dimensional arrays of elements. Each vector structure provides two different ways to create a vector: `vector` takes a list of elements and makes a vector out of it, and `tabulate` takes a function from integers to vector elements, which it uses to initialize the vector elements. Given a vector, one can get its length (using `length`), get an element (using `sub`), or extract a sub-vector (using `extract`).

5.2 Arrays

Arrays are mutable one-dimensional arrays of elements. They have the same basic operations as vectors, with a couple of minor differences and extra operations. The `array` operation creates an array initialized to a given value, while the `arrayoflist` operation is used to make an array from a list. An array value can be modified using the `update` operation, which replaces a given element with another value. Lastly, the `extract` operation returns a vector of the corresponding vector type.

5.3 Monomorphic aggregates

An implementation may choose to provide various implementations of the `MONO_ARRAY` and `MONO_VECTOR` signatures. If an implementation provides either a monomorphic array or vector structure for a particular element type, then it should provide both structures.[1] The main reason for providing monomorphic vectors and arrays is that they allow more compact representations than the polymorphic versions (e.g., a `BoolVector` implementation might use one bit per element).

Character vectors

The `CharVector` structure defines a view of the `String` structure that matches to the `MONO_VECTOR` signature. The type `CharVector.vector` is the same as `String.string`.

Byte arrays and vectors

The `Byte` structure provides functions to extract strings from monomorphic arrays and vectors of `Word8.word`s. In addition, these types support additional operations for packing and unpacking larger sizes of words. These can be found in the `Pack`*N*`Big` and `Pack`*N*`Little` structures.

5.4 Lists

Polymorphic lists are traditionally an important class of aggregate in functional programming. As such, lists are often supported with a large collection of library functions. We have attempted to specify a somewhat smaller collection of operations that reflects common usage. The design philosophy behind the `List` module is:

- The `List` module should be "moderately" complete, meaning that most programs will not need to define any additional general list manipulation operations.
- A function should be included if both:
  - Proven useful
  - Complicated to implement, or significantly more concise or more efficient than an equivalent combination of the other list functions.
- No gratuitous name changes.

[1] Since the `MONO_ARRAY` structure refers to the corresponding vector type, one cannot have a monomorphic array structure without the vector structure.

- No equality types.
- Different SML implementations may still desire to provide list utility library modules, though if we have it right, they should be small.

Chapter 6

System interface

The system interface structures provide access to the underlying operating system features, and to other run-time facilities.

6.1 Operating system interface

We assume a structure `OS` that contains all of the operating system related interfaces. At a minimum, this structure must match the `OS` signature.

Input/Output

The I/O proposal is currently in a separate document.

6.2 Locale

Given that SML is an international language, we should support mechanisms for parameterizing the system by locale. For example, ANSI C allows string collating, formating of monetary and numeric values, and formatting of dates to be locale-specific.

At this time, we do not have a design proposal, but there seem to be two basic approaches: we can define an abstract `locale` type that is passed as an explicit argument to those functions that are locale-specific; or we can have a global notion of the current locale, with functions to get and change it. C does the latter, but the former is in keeping with the functional nature of SML.

6.3 Directories and paths

The `FileSys` structure provides operations for navigating the directory hierarchy, for listing the files in a directory, and some operations on files. The `Path` structure provides an abstract, system independent, view of pathnames.

6.4 Time

We propose three structures to support access to timing and dates: `Time`, `Date` and `Timer`.

The abstract type `Time.time` is used both to represent intervals of time, and to represent points in time, which are really just intervals starting at some common point (e.g., since 00:00, January 1, 1970 GMT). The `Time` structure provides mechanisms to convert between the `time` type and various concrete representations. The `Date` structure provides a mechanism for converting between time values (which are in *Universal Coordinated Time*) and the corresponding date in a particular time zone. The `Timer` structure provides timers for measuring both CPU and "wall-clock" times.

6.5 Misc. stuff

```
val implementation : string
val versionName : string
```

Chapter 7

UNIX interface

Since a large fraction of SML users work on UNIX systems, it is important to standardize access to UNIX system calls. This interface is based on the POSIX standard (IEEE standard 1003.1) [POS90], with some extensions from the 1003.1a version, which is currently being voted upon.

The interface consists of the `POSIX` structure, which is divided into six sub-structures, along the lines of the chapters of the POSIX standard. The sub-structures are:

Process operations for creating and managing processes.

ProcEnv operations on the process environment (e.g., process IDs, grocess groups).

FileSys operations on the file system.

PosixIO primitive I/O operations.

Device operations of terminal devices.

[[should this be called TermIO??]]

SysDB operations on the system data-base (e.g., passwords).

Chapter 8

The `Old` structure

To permit users to compile programs written under the old basis, we require that each implementation provide the structure `Old`. This structure contains the top-level bindings specified in the Definition, along with one or more substructures that define the top-level bindings of various implementations. For example, a user might write:

```
local
  open Old Old.NJ
in
  user's program
end
```

to compile a user's program under the old SML/NJ basis.

We expect that at some future point, the `Old` module will be deemed obsolete, and will be dropped from the standard basis.

Chapter 9

The top-level environment

This chapter describes the required top-level environment, which consists of: top-level identifiers, both the pre-loaded required modules and identifiers made available without qualification; infix identifiers; and overloading.

9.1 Pre-loaded modules

9.2 Top-level type, exception and value identifiers

[[add sharing constraints on types?]]

```
type unit
type int
type real
type char
type string
type substring
type exn
type 'a array
type 'a vector
type 'a ref
datatype bool = false | true
datatype 'a option = NONE | SOME of 'a
datatype ordering = LESS | EQUAL | GREATER
datatype 'a list = nil | :: of ('a * 'a list)
```

```
exception Bind = General.Bind
exception Match = General.Match
exception Subscript = General.Subscript
exception Size = General.Size
exception Overflow = General.Overflow
exception Div = General.Div
exception Sqrt = General.Sqrt
exception Ln = General.Ln
exception Fail = General.Fail
exception Io = ???

val ! = General.!
val (op =) = General.=
val (op <>) = General.<>
val (op :=) = General.:=
val (op o) = General.o
val (op before) = General.before
val ignore = General.ignore

val not = Bool.not

val chr = Char.chr
val ord = Char.ord

val size = String.size
val str = String.str
val concat = String.concat
val implode = String.implode
val explode = String.explode
val substring = String.substring
val ^ = String.^

val hd = List.hd
val tl = List.tl
val null = List.null
val length = List.length
val @ = List.@
val app = List.app
val map = List.map
val foldl = List.foldl
val foldr = List.foldr
val rev = List.rev
```

9.3 Infix identifiers

The top-level environment has the following infix identifiers:

```
infix  7  * / div mod quot rem
infix  6  + - ^
infixr 5  :: @
infix  4  = <> < <= > >=
infix  3  := o
infix  0  before
```

9.4 Overloaded identifiers

The following symbols are overloaded:

```
~
+
-
*
/
div
mod
quot
rem
<
<=
>
>=
```

Chapter 10

Language issues

While this proposal is not an attempt to define a new language, it does raise some issues that must be dealt with at the language definition level.

[[Imperative types?]]

10.1 Overloading

10.2 Literals

The new character type and the possibility of multiple implementations of the numeric types requires addressing the issue of literals.

10.3 Character literals

With the new character type, there should be a notation for character literals. We propose the notation

`#"`*c*`"`

where "*c*" is any legal single character string. This notation has the advantage that existing legal SML code will not be affected.

If Unicode characters are supported, then we will need additional syntax for them. We propose that the escape sequence "`\(`*n*)", where *n* is a non-negative integer literal, be recognized. Also, we will need syntax for Unicode strings.

10.4 Numeric literals

With the possibility of multiple representations of the numeric types in a given implementation (e.g., `SmallInt` and `LargeInt`), there needs to be a way to distinguish the different literals. There are a number of possible approaches to this problem:

- Many languages (e.g., C and Modula-3) use different notation for literals of different precision. For example, the `LargeInt` literal `0` might be written `0L`.
- We could make literals have the default type unless constrained to some other type. Thus, the top-level binding
  ```
  val x = 1
  ```
  would give `x` the type `Integer.int`, while
  ```
  val x = (1 : LargeInt.int)
  ```
  would give `x` the type `LargeInt.int`. If the default integer representation is `SmallInt.int`, then the following would result in a type error:
  ```
  val x = (1 : LargeInt.int)
  val y = x + 1
  ```
  since `x` has type `LargeInt.int` and `1` has type `SmallInt.int` (we are assuming that `+` is overloaded here).
- Literals might be viewed as overloaded symbols that default to the default representation. Thus, the top-level binding
  ```
  val x = 1
  ```
  would give `x` the type `Integer.int`, while
  ```
  val x = LargeInt.+(1, 0)
  ```
  would give `x` the type `LargeInt.int`. Unlike under the previous proposal, the following code would typecheck:
  ```
  val x = (1 : LargeInt.int)
  val y = x + 1
  ```
  assuming that `+` is overloaded.

We have decided on the last of these, because we think it is the least surprising to the user.

In addition, we propose adding notation for hexadecimal integer constants (as is already done in the SML/NJ compiler). Hexidecimal literals have the notation:

$$[\tilde{}]\mathtt{0x}[\mathtt{0123456789abcdefABCDEF}]^{+}$$

They are overloaded in the same way as ordinary integer literals.

Word literals will have a "`0w`" prefix; for example: `0w0`, `0w10`, or `0wxFF`. Word literals do not have a sign.

Real literals would be overloaded over the various R`.real` types (for structures R :`REAL`), defaulting to `Real.real`.

10.5 Vector literals

A related issue is the question of syntax for vectors in expressions and patterns. The SML/NJ compiler supports a modified version of the list notation for vector literals. The form is:

```
#[ ... ]
```

and can be used in both expressions and patterns.

Part II

Manual pages

NAME

Array — polymorphic mutable arrays

SYNOPSIS

signature ARRAY

structure Array : ARRAY

SIGNATURE

```
eqtype 'a array
eqtype 'a vector

val maxLen   : int

val array    : (int * '_a) -> '_a array
val tabulate : (int * (int -> '_a)) -> '_a array
val fromList : '_a list -> '_a array

val length   : 'a array -> int
val sub      : ('a array * int) -> 'a
val update   : ('a array * int * 'a) -> unit
val extract  : ('a array * int * int option) -> 'a vector

val copy     : {
                 src : 'a array, si : int, len : int option,
                 dst : 'a array, di : int
               } -> unit
val copyv    : {
                 src : 'a vector, si : int, len : int option,
                 dst : 'a array, di : int
               } -> unit

val app      : ('a -> unit) -> 'a array -> unit
val foldl    : (('a * 'b) -> 'b) -> 'b -> 'a array -> 'b
val foldr    : (('a * 'b) -> 'b) -> 'b -> 'a array -> 'b
val modify   : ('a -> 'a) -> 'a array -> unit

val appi     : ((int * 'a) -> unit) -> ('a array * int * int option) -> unit
val foldli   : ((int * 'a * 'b) -> 'b) -> 'b -> ('a array * int * int option) -> 'b
val foldri   : ((int * 'a * 'b) -> 'b) -> 'b -> ('a array * int * int option) -> 'b
val modifyi  : ((int * 'a) -> 'a) -> ('a array * int * int option) -> unit
```

DESCRIPTION

The `Array` structure provides polymorphic, one-dimensional, zero-based, updateable arrays.

`maxLen`

is the maximum length of arrays supported by the implementation.

`array (`*n*`, `*v*`)`

creates an n-element, zero-based array with each element initialized to v. Raises `Size` if $n < 0$ or if $n >$ `maxLen`.

`tabulate (`*n*`, `*f*`)`

create an n element array whose ith element is initialized to $f(i)$. The function *f* is called in increasing order of *i*. Raises `Size` if $n < 0$ or if $n >$ `maxLen`.

`arrayOfList` *l*

create an array whose elements are initialized to the elements of l. Raises `Size` if the list has more than `maxLen` elements.

`array0`

is the unique zero-length array.

`length` *arr*

the number of elements in the array *arr*.

`sub (`*arr*`, `*i*`)`

extracts (subscript) the ith element of array *arr*. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$.

`update (`*arr*`, `*i*`, `*v*`)`

replaces the ith element of *arr* by the value v. Raises `Subscript` if $i < 0$ or $i \geq \text{length}(a)$.

`extract (`*a*`, `*i*`, `*n*`)`

extracts the elements $a[i \ldots i + n - 1]$ as a vector of length n. The exception `Subscript` is raised if $i < 0 \vee n < 0 \vee |a| < i + n$.

`copy {`*src*`, `*si*`, `*len*`, `*dst*`, `*di*`}`

copies *len* elements from the source array *src* starting at index *si* into the destination array *dst* starting at index *di*. The exception `Subscript` is raised if $len < 0$, or if either $si < 0 \vee |src| < si + len$, or $di < 0 \vee |dst| < di + len$.

More precisely, let src' and dst' be the contents of *src* and *dst* immediately prior to the call to `copy`. Then upon successful completion of the call, for $0 \leq i < |dst|$:

$$dst_i = \begin{cases} src'_{si+(i-di)} & \text{if } di \leq i < di + len \\ dst'_i & \text{otherwise} \end{cases}$$

Moreover, if *src* and *dst* are different arrays, then for $0 \leq i < |src|$: $src_i = src'_i$.

`copyv` {*src*, *si*, *len*, *dst*, *di*}
is like `copy`, except that *src* is a vector.

Note that type α `array` is an equality type even if α is not. Thus, the `eqtype` specification in the signature `ARRAY` does not quite capture the equality semantics of arrays. All zero-length arrays are equal to each other. Nonzero-length arrays a and b, created by different calls to `array`, are always unequal, even if their elements are equal.

SEE ALSO

Vector(BASIS), MONO_ARRAY(BASIS)

NAME

Bool — Operations on booleans

SYNOPSIS

signature BOOL
structure Bool : BOOL

SIGNATURE

```
datatype bool = true | false

val not : bool -> bool

val fromString : string -> bool option
val toString   : bool -> string
val scan : {getc : 'a -> (char * 'a) option} -> 'a -> (bool * 'a) option
```

DESCRIPTION

NAME

Byte — unsigned 8-bit integers

SYNOPSIS

signature BYTE

structure Byte : BYTE

SIGNATURE

```
exception Ord

val byteToChar : Word8.word -> char
val charToByte : char -> Word8.word

val bytesToString : Word8Vector.vector -> string
val stringToBytes : string -> Word8Vector.vector

val unpackStringV : (Word8Vector.vector * int * int option) -> string
val unpackString  : (Word8Array.array * int * int option) -> string

val packStringV   : (substring * Word8Vector.vector * int) -> unit
val packString    : (substring * Word8Array.array * int) -> unit
```

DESCRIPTION

Bytes are unsigned 8-bit integers as provided by the `Word8` structure, but two additional operations are provided for conversion to and from ASCII characters.

The function `byteToChar` cannot fail: the range of character codes is guaranteed to be at least 0–255, but in SML implementations that use Unicode, some characters are not convertible to 8-bit integers; on these, `charToByte` will raise the `Ord` exception.

[[Under the wide character proposal, even this is not a problem]]

SEE ALSO

WORD(BASIS)

NAME

Char — character type and operations

SYNOPSIS

signature CHAR

structure Char : CHAR

open Char

SIGNATURE

```
eqtype char

exception Chr

val chr : int -> char
val ord : char -> int

val minChar : char
val maxChar : char
val maxOrd  : int

val succ : char -> char
val pred : char -> char

val <  : (char * char) -> bool
val <= : (char * char) -> bool
val >  : (char * char) -> bool
val >= : (char * char) -> bool

val compare : (char * char) -> ordering

val contains    : string -> char -> bool
val notContains : string -> char -> bool

val isLower    : char -> bool
val isUpper    : char -> bool
val isDigit    : char -> bool
val isAlpha    : char -> bool
val isHexDigit : char -> bool
val isAlphaNum : char -> bool
val isPrint    : char -> bool
val isSpace    : char -> bool
val isPunct    : char -> bool
val isGraph    : char -> bool
val isCntrl    : char -> bool
val isAscii    : char -> bool

val toUpper : char -> char
val toLower : char -> char
```

DESCRIPTION

The character type is a dense enumeration running from `minChar` to `maxChar`. We require that `ord(minChar)` be `0`, and that `ord(maxChar)` be `maxOrd`. The actual value of `maxOrd` is implementation dependent. For example, an ASCII-based implementation might use `255` for `maxOrd`. The mapping between characters and integers is provided by the following two operators:

`chr` *i*

returns the *i*th character. If $i < 0$ or `maxOrd` $< i$, then the exception `Chr` is raised.

`ord` *c*

returns the integer representation of the character. It should be the case that `chr(ord` *c*`)` $= c$, for all characters *c*.

The relational operators on characters are defined by:

```
fun (op f) (c1, c2) = (op f)(ord c1, ord c2)
```

where *f* is one of `<`, `<=`, `>` or `>=`.

SEE ALSO

String(BASIS)

NAME

CONVERT_INT — conversions between integer types

SYNOPSIS

signature CONVERT_INT

SIGNATURE

```
type to
type from

val to   : from -> to
val from : to -> from
```

DESCRIPTION

SEE ALSO

INTEGER (BASIS)

NAME

CONVERT_REAL — signature of floating-point conversions

SYNOPSIS

signature CONVERT_REAL

structure Cvt.Float*N*Float*M* : CONVERT_REAL

SIGNATURE

```
type small
  sharing type FloatN.real = small
type large
  sharing type FloatM.real = large

extend : small -> large
round  : large -> small
trunc  : large -> small
floor  : large -> small
ceil   : large -> small
```

DESCRIPTION

THis interface needs revision, but I’m not sure what the cuurrent proposal is.

SEE ALSO

FLOAT(BASIS)

NAME

CONVERT_WORD — signature of unsigned integer conversions

SYNOPSIS

signature CONVERT_WORD

SIGNATURE

```
type word
type to

val to     : word -> to
val extend : word -> to
val from   : to -> word
```

DESCRIPTION

This is the interface of conversions from some word type to a larger integer or word type (the type `to`).

`to` *w*

`extend` *w*

`from` *n*

SEE ALSO

WORD(BASIS)

NAME

Date — interface to local time and date information

SYNOPSIS

signature DATE

structure Date : DATE

SIGNATURE

```
datatype weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

datatype month
  = Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sep | Oct | Nov | Dec

datatype date = DATE of {
    year : int,                        (* e.g., 1995 *)
    month : month,
    day : int,                  (* 1-31 *)
    hour : int,                 (* 0-23 *)
    minute : int,               (* 0-59 *)
    second : int,               (* 0-61 (leap seconds) *)
    wday   : weekday option,
    yday   : int option,        (* 0-365 *)
    isDst  : bool option        (* daylight savings time in force *)
  }

exception Date

val fromTime : Time.time -> date
val fromUTC  : Time.time -> date
val toTime   : date -> Time.time

val toString   : date -> string
val fromString : string -> date option
val fmt        : string -> date -> string
val scan       : {getc : 'a -> (char * 'a) option} -> 'a -> (date * 'a) option

val compare : (date * date) -> ordering
```

DESCRIPTION

This interfaces follows the ANSI C semantics. The `compare` operation defines a lexical ordering using the `year`, `month`, `day`, `hour`, `minute`, and `second` fields. The other fields are ignored.

SEE ALSO

FmtDate(BASIS), Time(BASIS)

NAME

Float — floating-point arithmetic

SYNOPSIS

signature FLOAT

structure Float : FLOAT *(optional)*

structure Float64 : FLOAT *(optional)*

structure Float32 : FLOAT *(optional)*

structure Float*N* : FLOAT *etc.*

SIGNATURE

```
include REAL
val radix     : Integer.int          (* 2 for IEEE, Vax; 16 for IBM *)
val precision : Integer.int
    (* the number of digits (each 0..radix-1) in mantissa *)
val logb         : real -> Integer.int
    (* takes log to the base "radix", rounding towards negative infinity;
     * it is a fancy name for "extract exponent"
     *)
val scalb        : real * Integer.int -> real
    (* scalb(x,n) = x*radix^n *)
val nextAfter    : real * real -> real
    (* nextAfter(x, y) returns the next representable real after x in the
     * direction of y.  If x = y, then it returns x.
     *)
val maxFinite    : real   (* maximum finite number *)
val minPos       : real   (* minimum non-zero positive number *)
val minNormalPos : real   (* minimum non-zero normalized number *)
```

DESCRIPTION

[[If we assume IEEE representations, then do we need `radix`?]]

[[We should have operations to decompose float values]]

SEE ALSO

Real(BASIS), Math(BASIS)

NAME

General — basic definitions used in the pervasive environment

SYNOPSIS

signature GENERAL
structure General : GENERAL
open General

SIGNATURE

```
type exn
eqtype unit

exception Bind
exception Match
exception Interrupt  (* included for compatibility *)

exception Subscript
exception Size

exception Overflow
exception Div
exception Sqrt
exception Ln

exception Fail of string

val exnMessage : exn -> string
val exnName    : exn -> string

datatype 'a option = NONE | SOME of 'a

exception Option
val getOpt : ('a option * 'a) -> 'a
val isSome : 'a option -> bool
val valOf  : 'a option -> 'a

datatype ordering = LESS | EQUAL | GREATER

val =  : (''a * ''a) -> bool
val <> : (''a * ''a) -> bool

val !   : 'a ref -> 'a
val :=  : 'a ref * 'a -> unit

val o : (('b -> 'c) * ('a -> 'b)) -> ('a -> 'c)
val before : ('a * unit) -> 'a
```

```
val ignore : 'a -> unit
```

DESCRIPTION

NAME

INTEGER — Generic signature for integer arithmetic types and operations

SYNOPSIS

signature INTEGER
structure Integer : INTEGER
structure SmallInt : INTEGER *(optional)*
structure LargeInt : LARGE_INT *(optional)*
structure Int*N* : INTEGER *etc.*

SIGNATURE

```
eqtype int

(* infix 7 div mod *  *)
(* infix 6 + -        *)
(* infix 4 < > <= >=  *)

structure ToLarge : CONVERT_INT
  sharing type ToLarge.from = int
          type ToLarge.to = LargestInt.int
structure ToInt : CONVERT_INT
  sharing type ToInt.from = int
          type ToInt.to = Integer.int

val precision : int option
val minInt : int option
val maxInt : int option

val ~ : int -> int
val * : int * int -> int
val div : int * int -> int
val mod : int * int -> int
val quot : int * int -> int
val rem : int * int -> int
val + : int * int -> int
val - : int * int -> int
val abs : int -> int

val >  : int * int -> bool
val >= : int * int -> bool
val <  : int * int -> bool
val <= : int * int -> bool

val compare : (int * int) -> ordering
```

```
val min : (int * int) -> int
val max : (int * int) -> int

val sign     : int -> int
val sameSign : (int * int) -> bool

val toString   : int -> string
val fromString : string -> int option
val scan : StringCvt.radix -> {getc : 'a -> (char * 'a) option} -> 'a -> (int * 'a) option
val fmt  : StringCvt.radix -> int -> string
```

DESCRIPTION

The `Integer` structure is the same as either `LargeInt` (arbitrary precision integers) or `SmallInt` (standard size, fixed-precision integers). `SmallInt` is the same as `Int`N for some N.

The values `precision`, `minInt`, and `maxInt` are `NONE` in the `LargeInt` structure. In the `SmallInt` structure, `precision` is the number of bits used to represent an integer; `minInt` is the most negative integer, and `maxInt` is the most positive integer. In a two's complement implementation, it should be the case that:

$$\begin{aligned} 2^{precision-1} - 1 &= maxInt \\ -2^{precision-1} &= minInt. \end{aligned}$$

The operators `~`, `*`, `+`, `-`, and `abs` stand for integer negation, multiplication, addition, subtraction, and absolute value. The inequality comparison operators have the usual meaning. The equality operators are not listed explicitly in the signature, but note that `int` is an `eqtype`.

The operators `div` and `mod` are as in the Definition (i.e., `div` rounds toward negative infinity). But we also include operators `quot` and `rem`, which have the standard hardware semantics (i.e., round towards zero). More precisely, the following identities hold:

$$\begin{aligned} i \;\texttt{div}\; d &= q \\ i \;\texttt{mod}\; d &= r, \\ d \times q + r &= i \\ 0 \leq r < d \quad &\text{or} \quad d < r \leq 0 \end{aligned}$$

$$\begin{aligned} i \;\texttt{quot}\; d &= q' \\ i \;\texttt{rem}\; d &= r', \\ d \times q' + r' &= i \end{aligned}$$

$$0 \leq d \times q' \leq i \quad \text{or} \quad i \leq d \times q' \leq 0$$
$$0 \leq |r| < |d|$$

The operators `div`, `mod`, `quot`, and `rem` raise `Div` if their second argument is zero. If the second argument is nonzero but the result is too large to be representable, `Overflow` is raised.

`sign` *i*

returns -1, if $i < 0$; and 1, if $i \geq 0$.

`sameSign (`*i*`,` *j*`)`

returns true, if *i* and *j* have the same sign.

SEE ALSO

LargeInt(BASIS)

NAME

LargeInt — Arbitrary-precision integer structure

SYNOPSIS

signature LARGE_INT

structure LargeInt : LARGE_INT

SIGNATURE

```
include INTEGER

val divMod  : (int * int) -> (int * int)
val quotRem : (int * int) -> (int * int)
val exp : (int * Integer.int) -> int
val log2 : int -> Integer.int
```

DESCRIPTION

The `LargeInt` structure is one of the possible implementations of the `INTEGER` interface. In addition to the `INTEGER` operations, it provides some operations useful for programming with bignums.

The functions `divMod` and `quotRem` are defined by:

```
fun divMod (a, b) = (a div b, a mod b)
fun quotRem (a, b) = (a quot b, a rem b)
```

but are more efficient that doing both operations individually. These functions raise `Div`, if their second argument is zero. The function `exp` raises its first argument to the power of its second argument (which is a default integer). The function `log2` returns the log base-2 of its argument as a default integer.

SEE ALSO

INTEGER(BASIS)

NAME

List — List datatype and operations

SYNOPSIS

signature LIST

structure List : LIST

SIGNATURE

```
datatype 'a list = nil | :: of 'a * 'a list

exception Empty

val null : 'a list -> bool
val hd   : 'a list -> 'a
val tl   : 'a list -> 'a list
val last : 'a list -> 'a

val nth     : 'a list * int -> 'a

val take    : ('a list * int) -> 'a list
val drop    : ('a list * int) -> 'a list

val length : 'a list -> int

val rev : 'a list -> 'a list

val @         : 'a list * 'a list -> 'a list
val concat    : 'a list list -> 'a list
val revAppend : 'a list * 'a list -> 'a list

val app        : ('a -> unit) -> 'a list -> unit
val map        : ('a -> 'b) -> 'a list -> 'b list
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list

val find      : ('a -> bool) -> 'a list -> 'a option
val filter    : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> ('a list * 'a list)

val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

val exists : ('a -> bool) -> 'a list -> bool
val all    : ('a -> bool) -> 'a list -> bool

val tabulate : (int * (int -> 'a)) -> 'a list
```

DESCRIPTION

The list type is defined in both `General`, and in the `List` module. The list operations are described below; some of these may raise the `Empty` exception when applied to `nil`.

`null` *l*
: returns `true`, if the list *l* is `nil`.

`hd` *l*
: returns the first item of the list *l*; it raises `Empty` when applied to `nil`.

`tl` *l*
: returns the all but the first item of the list *l*; it raises `Empty` when applied to `nil`.

`last` *l*
: returns the last item of the list *l*; it raises `Empty` when applied to `nil`.

`nth (`*l*`,` *i*`)`
: returns the *i*th element of the list *l* counting from zero. If $i < 0 \vee |l| \leq i$, then the exception `Subscript` is raised.

`take (`*l*`,` *i*`)`
: Returns the first *i* elements of the list *l*. If $i < 0 \vee |l| < i$, then the exception `Subscript` is raised.

`drop (`*l*`,` *i*`)`
: Returns the tail of the list *l* starting at the *i*th element (i.e., it drops the first *i* elements). If $i < 0 \vee |l| < i$, then the exception `Subscript` is raised.

`length` *l*
: returns the number of elements in the list *l*.

`rev` *l*
: reverses the order of the elements of *l*.

l1 `@` *l2*
: appends the elements of list *l2* onto the end of *l1*.

`concat` *l*
: concatenates a list of lists.

`revAppend (`*l1*`,` *l2*`)`
: returns `(rev` *l1*`) @` *l2*.

`app` *f* *l*
: applies the function *f* to the elements of *l* in left-to-right order. Since *f* is being applied for its effect, it is constrained to return `unit`.

`map` *f l*

maps the function *f* over the elements of the list *l* in left-to-right order, returning the list of results.

`mapPartial` *f l*

maps the partial function *f* over the elements of the list *l* in left-to-right order, returning the list of results where *f* is defined. We say that *f* is partial in the sense that it returns `NONE` where it is not defined.

`find` *pred l*

returns the leftmost element of the list *l* that satisfies the predicate *pred*; it returns `NONE`, if there is no such element. The function *pred* is applied from left to right, and the search is terminated once an element has been found (i.e., *pred* is not applied to any elements to the right of the leftmost element satisfying *pred*).

`filter` *pred l*

returns a list of the elements that satisfy the predicate *pred*. The predicate is applied once to each element in left-to-right order, and the order of the result list respects the order of *l*.

`partition` *pred l*

partitions the list *l* into a list of elements that satisfy the predicate *pred*, and a list of elements that do not. The predicate is applied once to each element in left-to-right order, and the order of the result lists respects the order of *l*.

`foldl` *f init l*

computes $f(l_n, f(l_{n-1}, \ldots, f(l_1, init) \ldots))$, where the l_i are the elements of *l*. Note that *f* is applied to the elements in left-to-right order.

`foldr` *f init l*

computes $f(l_1, f(l_2, \ldots, f(l_n, init) \ldots))$, where the l_i are the elements of *l*. Note that *f* is applied to the elements in right-to-left order.

`exists` *pred l*

returns `true` if there is an element of *l* that satisfies the predicate *pred*. As with `find`, the predicate is tested from left-to-right, and the search is terminated once an element has been found.

`all` *pred l*

returns `true`, if all elements of the list *l* satisfy the predicate *pred*. It is equivalent to `not(exists (not o` *pred*`)` *l*`)`.

`tabulate (`*n*`,` *f*`)`

generates the list `[`*f* `0, ` *f* `1, ..., ` *f* `(`$n-1$`)]`. The function *f* is applied in left-to-right (increasing index) order. If $n < 0$, then the exception `Size` is raised.

SEE ALSO

General(Initial Basis), ListPair(Initial Basis)

NAME

ListPair — operations on pairs of lists and lists of pairs

SYNOPSIS

signature LIST_PAIR

structure ListPair : LIST_PAIR

SIGNATURE

```
val zip    : ('a list * 'b list) -> ('a * 'b) list
val unzip  : ('a * 'b) list -> ('a list * 'b list)
val map    : ('a * 'b -> 'c) -> ('a list * 'b list) -> 'c list
val app    : ('a * 'b -> unit) -> ('a list * 'b list) -> unit
val all    : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
val exists : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
```

DESCRIPTION

These are operations for computing with pairs of elements taken from a pair of lists.

`zip` (*l1*, *l2*)

combines the two lists *l1* and *l2* into a list of pairs, with the first element of each list comprising the first element of the result, the second elements comprising the second element of the result, and so on. If the lists are of unequal lengths, the excess elements from the tail of the longer one are ignored.

`unzip` *l*

returns a pair of lists formed by splitting the elements of *l*. This is the inverse of `zip`.

`map` *f* (*l1*, *l2*)

is equivalent to `List.map` *f* (`zip` (*l1*, *l2*)).

`app` *f* (*l1*, *l2*)

is equivalent to `List.app` *f* (`zip` (*l1*, *l2*)).

`all` *pred* (*l1*, *l2*)

is equivalent to `List.all` *pred* (`zip` (*l1*, *l2*)).

`exists` *pred* (*l1*, *l2*)

is equivalent to `List.exists` *pred* (`zip` (*l1*, *l2*)).

SEE ALSO

List(Initial Basis)

NAME

Locale — support for internationalization

SYNOPSIS

signature LOCALE

structure Locale : LOCALE

SIGNATURE

```
eqtype category
val collate     : category
val ctype       : category
val monetary    : category
val numeric     : category
val time        : category
val messages    : category

val all : category list

exception NoSuchLocale
val setLocale : (string * category list) -> unit
val getLocale : category -> string

datatype sign_posn
  = PAREN
  | PREC_ALL
  | PREC_CURR
  | FOLLOW_ALL
  | FOLLOW_CURR

type lconv
val conventions : unit -> lconv

val decimalPoint    : lconv -> char option          (* SOME(#".") *)
val thousandsSep    : lconv -> char option          (* NONE *)
val grouping        : lconv -> int list             (* [] *)
val currencySymbol  : lconv -> string               (* NONE *)
val intCurrSymbol   : lconv -> string               (* NONE *)
val monDecimalPoint : lconv -> char option          (* NONE *)
val monThousandsSep : lconv -> char option          (* NONE *)
val monGrouping     : lconv -> int list             (* [] *)
val positiveSign    : lconv -> string               (* NONE *)
val negativeSign    : lconv -> string               (* NONE *)
val intFracDigits   : lconv -> int option           (* NONE *)
val fracDigits      : lconv -> int option           (* NONE *)
val posCSPrecedes   : lconv -> bool option          (* NONE *)
val posSepBySpace   : lconv -> bool option          (* NONE *)
val negCSPrecedes   : lconv -> bool option          (* NONE *)
```

```
val negSepBySpace   : lconv -> bool option         (* NONE *)
val posSignPosn     : lconv -> sign_posn option    (* NONE *)
val negSignPosn     : lconv -> sign_posn option    (* NONE *)

val collateChr : (char * char) -> ordering
val collateStr : (substring * substring) -> ordering

exception NoSuchClass
val isClass : string -> char -> bool
```

DESCRIPTION

This is not the most recent version of this interface.

SEE ALSO

CAVEATS

NAME

MATH — signature of mathematical library functions

SYNOPSIS

signature MATH

SIGNATURE

```
type real

exception Sqrt
exception Trig
exception Ln

val pi : real
val e : real

val sqrt   : real -> real
val sin    : real -> real
val cos    : real -> real
val tan    : real -> real
val atan   : real -> real
val asin   : real -> real
val acos   : real -> real
val atan2  : (real * real) -> real
val exp    : real -> real
val pow    : (real * real) -> real
val ln     : real -> real
val log10  : real -> real
val sinh   : real -> real
val cosh   : real -> real
val tanh   : real -> real
```

DESCRIPTION

The `Math` structure is a substructure of the structures matching the `REAL` signature. The square root, exponential, and trigonometric functions are the same as those in the Definition, but we have added additional standard functions:

`pi`

The constant π in the full precision of the given real type.

`e`

The constant e in the full precision of the given real type.

`sqrt` *x*

returns $\sqrt{x}$, for $x \geq 0$. If $x < 0$, then the exception `Sqrt` is raised.

`sin` *x*

returns the sine of x, where x is in radians.

`cos` *x*

returns the cosine of x, where x is in radians.

`tan` *x*

returns the tangent of x, where x is in radians.

`acos` *x*

returns the arc cosine in the range 0 to π. If $|x| > 1$, then the exception `Trig` is raised.

`asin` *x*

returns the arc sine in the range $\frac{-\pi}{2}$ to $\frac{\pi}{2}$. If $|x| > 1$, then the exception `Trig` is raised.

`atan` *x*

returns the arc tangent in the range $\frac{-\pi}{2}$ to $\frac{\pi}{2}$.

`atan2 (`*y*`,` *x*`)`

returns the arc tangent of $\frac{y}{x}$ in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the result. This has the following properties:

$$\begin{aligned} \text{atan2}(0,0) &= 0 \\ \tan(\text{atan2}(y,x)) &= y/x, \text{ for } x \neq 0 \\ |\text{atan2}(y,0)| &= \pi/2, \text{ for } y \neq 0 \\ \text{sign}(\cos(\text{atan2}(y,x)) &= \text{sign}(x) \\ \text{sign}(\sin(\text{atan2}(y,x)) &= \text{sign}(y) \end{aligned}$$

`exp` *x*

returns e^x.

`pow (`*x*`,` *y*`)`

returns x^y.

`ln` *x*

returns the natural logarithm of x. If $x \leq 0$, then it raises the exception `Ln`.

`log10` *x*

returns the base-10 logarithm of x. If $x \leq 0$, then it raises the exception `Ln`.

SEE ALSO

Real(BASIS), Float(BASIS)

NAME

MONO_ARRAY — generic signature of monomorphic array structures

SYNOPSIS

signature MONO_ARRAY

SIGNATURE

```
eqtype array
eqtype elem
eqtype vector

val maxLen   : int

val array    : (int * elem) -> array
val tabulate : (int * (int -> elem)) -> array
val fromList : elem list -> array

val length   : array -> int
val sub      : (array * int) -> elem
val update   : (array * int * elem) -> unit
val extract  : (array * int * int option) -> vector

val copy     : {
                 src : array, si : int, len : int option,
                 dst : array, di : int
               } -> unit
val copyv    : {
                 src : vector, si : int, len : int option,
                 dst : array, di : int
               } -> unit

val app      : (elem -> unit) -> array -> unit
val foldl    : ((elem * 'a) -> 'a) -> 'a -> array -> 'a
val foldr    : ((elem * 'a) -> 'a) -> 'a -> array -> 'a
val modify   : (elem -> elem) -> array -> unit

val appi     : ((int * elem) -> unit) -> (array * int * int option) -> unit
val foldli   : ((int * elem * 'a) -> 'a) -> 'a -> (array * int * int option) -> 'a
val foldri   : ((int * elem * 'a) -> 'a) -> 'a -> (array * int * int option) -> 'a
val modifyi  : ((int * elem) -> elem) -> (array * int * int option) -> unit
```

DESCRIPTION

This is the generic signature of monomorphic arrays (e.g., `CharArray`). The equality type `array` is the monomorphic array type, which is indexed from `0`. The type `elem` is the element type, and the type `vector` is the type of the corresponding immutable vectors of the `elem` type. As in the case of polymorphic arrays, two arrays are equal if, and only if, they are the same array. For each monomorphic array type, there is a unique array of length zero. The other members of the structure are:

`maxLen`

is the maximum length supported for arrays of this type.

`array (`*n*`, `*v*`)`

creates an array of *n* elements intialized to *v*. This raises the `Size` exception, if *n* is either too large (> `maxLen`) or negative.

`tabulate (`*n*`, `*f*`)`

creates an array of *n* elements, where the *i*th element is initialized to `f(`*i*`)`. The function *f* is called in increasing order of *i*. This raises the `Size` exception, if *n* is either too large (> `maxLen`) or negative.

`arrayOfList` *l*

creates an array from the list of elements *l*. This raises the `Size` exception, if the *l* has more than `maxLen` elements. The zero-length array created by `arrayOfList []` is unique.

`length` *arr*

returns the length of the array *arr*.

`sub (`*arr*`, `*i*`)`

returns the *i*th element of *arr*. The exception `Subscript` is raised if *i* is out of bounds.

`update (`*arr*`, `*i*`, `*v*`)`

replaces the *i*th element of *arr* with *v*. The exception `Subscript` is raised if *i* is out of bounds.

`extract (`*arr*`, `*i*`, `*n*`)`

extracts a vector of length *n* from the array *arr*, starting with the *i*th element. The exception `Subscript` is raised if $i < 0 \vee n < 0 \vee |a| < i + n$.

`copy {`*src*`, `*si*`, `*len*`, `*dst*`, `*di*`}`

copies *len* elements from the source array *src* starting at index *si* into the destination array *dst* starting at index *di*. The exception `Subscript` is raised if $len < 0$, or if either $si < 0 \vee |src| < si + len$, or $di < 0 \vee |dst| < di + len$.

More precisely, let src' and dst' be the contents of *src* and *dst* immediately prior to the call to `copy`. Then upon successful completion of the call, for $0 \leq i < |dst|$:

$$dst_i = \begin{cases} src'_{si+(i-di)} & \text{if } di \leq i < di + len \\ dst'_i & \text{otherwise} \end{cases}$$

Moreover, if *src* and *dst* are different arrays, then for $0 \leq i < |src|$: $src_i = src'_i$.

`copyv` {*src*, *si*, *len*, *dst*, *di*}
: is like `copy`, except that *src* is a vector.

SEE ALSO

Array(BASIS), MONO_VECTOR(BASIS)

NAME

MONO_VECTOR — generic signature of monomorphic vector structures

SYNOPSIS

signature MONO_VECTOR

SIGNATURE

```
eqtype vector
eqtype elem

val maxLen   : int

val fromList : elem list -> vector
val tabulate : (int * (int -> elem)) -> vector

val length   : vector -> int
val sub      : (vector * int) -> elem
val extract  : (vector * int * int option) -> vector
val concat   : 'a vector list -> 'a vector

val app      : (elem -> unit) -> vector -> unit
val foldl    : ((elem * 'a) -> 'a) -> 'a -> vector -> 'a
val foldr    : ((elem * 'a) -> 'a) -> 'a -> vector -> 'a

val appi     : ((int * elem) -> unit) -> (vector * int * int option) -> unit
val foldli   : ((int * elem * 'a) -> 'a) -> 'a -> (vector * int * int option) -> 'a
val foldri   : ((int * elem * 'a) -> 'a) -> 'a -> (vector * int * int option) -> 'a
```

DESCRIPTION

This is the generic signature of monomorphic vectors (e.g., CharVector). The type `vector` is the monomorphic vector type, which is indexed from `0`. The type `elem` is the element type, and the type `vector` is the type of the corresponding immutable vectors of the `elem` type. The other members of the structure are:

`maxLen`

is the maximum length supported for vectors of this type.

`vector` *l*

creates an vector from the list of elements *l*. This raises the `Size` exception, if the *l* has more than `maxLen` elements.

`tabulate (`*n*`,` *f*`)`

creates an vector of *n* elements, where the *i*th element is initialized to `f(`*i*`)`. The function *f* is called in increasing order of *i*. This raises the `Size` exception, if *n* is either too large (> `maxLen`) or negative.

`length` *vec*

returns the length of the vector *vec*.

`sub (`*vec*`, `*i*`)`

returns the *i*th element of *vec*. The exception `Subscript` is raised if *i* is out of bounds.

`extract (`*vec*`, `*i*`, `*n*`)`

extracts a vector of length *n* from the vector *vec*, starting with the *i*th element. The exception `Subscript` is raised if $i < 0 \vee n < 0 \vee |vec| < i + n$.

`concat` *vl*

forms the concatenation of a list of vectors. If the sum of the lengths exceeds `maxLen`, then the `Size` exception is raised.

SEE ALSO

MONO_ARRAY(BASIS), Vector(BASIS)

NAME

OS — Generic interface to operating system

SYNOPSIS

signature OS

structure OS : OS

SIGNATURE

```
type syserror
val errorMessage : syserror -> string
val errorName    : syserror -> string

exception SysErr of (string * syserror option)

structure FileSys : OS_FILE_SYS
structure Path    : OS_PATH
structure Process : OS_PROCESS
```

DESCRIPTION

The type `syserror` represents a system dependent error code; the function `errorMsg` returns a useful error message from a `syserror`, while the function `errorName` returns the name used by the system for the error code. For example on UNIX systems, applying `errorMessage` to the `EACCES` error code might return `"Permission denied"`, while `errorName` would return `"EINTR"`. The exception `SysErr` is the general exception used by the system interfaces.

SEE ALSO

OS.FileSys(BASIS), OS.Path(BASIS), OS.Process(BASIS)

NAME

OS.FileSys — system independent file-system operations

SYNOPSIS

```
signature FILE_SYS

structure OS : OS =
  struct
    ...
    structure FileSys : OS_FILE_SYS
    ...
  end
```

SIGNATURE

```
type dirstream

val openDir   : string -> dirstream
val readDir   : dirstream -> string
val rewindDir : dirstream -> unit
val closeDir  : dirstream -> unit

val chDir  : string -> unit
val getDir : unit -> string
val mkDir  : string -> unit
val rmDir  : string -> unit
val isDir  : string -> bool

val isLink   : string -> bool
val readLink : string -> string

val realPath : string -> string
val fullPath : string -> string

val modTime : string -> Time.time
val setTime : (string * Time.time option) -> unit
val remove  : string -> unit
val rename  : {old : string, new : string} -> unit

datatype access = A_READ | A_WRITE | A_EXEC

val access : (string * access list) -> bool

val tmpName : {dir : string option, prefix : string option} -> string
```

DESCRIPTION

The `FileSys` structure provides a limited set of operations on directories and files, which are portable across operating systems.

Directories are viewed as a sequence of file names in some system dependent order. The `dirstream` type represents this abstraction; the operations are:

`openDir` *path*

  opens the specified directory stream.

`readDir` *ds*

  returns the next file name in the stream *ds*. If all of the file names in *ds* have been read, then the empty string is returned.

`rewindDir` *ds*

  rewinds the stream *ds* to the beginning.

`closeDir` *ds*

  closes the stream *ds*.

In addition to directory streams, the `Directory` structure provides operations for navigating the directory hierarchy:

`chDir` *path*

  changes the current working directory to the specified *path*.

`getDir` *path*

  returns the current working directory.

`mkDir` *path*

  creates the specified directory.

`rmDir` *path*

  removes the specified directory.

`isDir` *path*

  returns true if *path* names a directory. It raises the `SysErr` exception if *path* is invalid, does not exist, or there is a permission error.

The interface provides operations for canonicalizing pathnames:

`fullPath` *path*

  returns a canonical absolute physical path that names the object specified by *path*. This includes making relative paths absolute, expanding symbolic links, and removing empty, current and parent arcs. On file systems with case insensitive names, the arc names are case converted to the "reference" case. Note that this does *not* do tilde expansion on UNIX systems. If the path is ill-formed, the named object does not exist, or the user does not have access to some object on the path, then the `SysErr` exception is raised.

`realPath` *path*

returns a canonical physical path that names the object specified by *path*. If *path* is relative and names an object on the same volume as the current working directory, then a relative path is returned, otherwise this returns the same result as `fullPath`. If the path is ill-formed, the named object does not exist, or the user does not have access to some object on the path, then the `SysErr` exception is raised.

Several operations are provided on other files:

`modTime` *path*

`setTime (`*path*`,` *t*`)`

sets the file access and modification time (as returned by `modTime`) to *t* (if specified. If *t* is not specified (i.e., `NONE`), then it uses the current time. If the file does not exist, or is not readable, then the `SysErr` exception is raised with `ml_op` set to the string `"FileSys.setTime"`. On UNIX systems, this sets both the access and modification times.

`remove` *path*

Note that the effect of removing an open file is system dependent.

`rename` {*new*`,` *old*}

`access (`*path*`,` *acl*`)`

tests the access permissions associated with the named file. If *acl* is `nil`, then this tests for the existence of the named file.

`tmpName` {*dir*`,` *prefix*}

generates a pathname suitable for naming a temporary file. If *prefix* is specified, then the first few characters of *prefix* will be used as the beginning of the file name. The actual number of characters used from *prefix* depends on the underlying operating system. If *dir* is specified, and names a writable directory, then it is used as the location for the temporary file; otherwise a system dependent directory is used (e.g., `/usr/tmp` on UNIX systems).

SEE ALSO

OS(BASIS),Path(BASIS)

NAME

OS.Path — System independent interface to pathnames

SYNOPSIS

```
signature PATH

structure OS : OS =
  struct
    ...
    structure Path : OS_PATH
    ...
  end
```

SIGNATURE

```
exception Path

val parentArc  : string
val currentArc : string

val validVolume : {isAbs : bool, vol : string} -> bool

val fromString : string -> {isAbs : bool, vol : string, arcs : string list}
val toString   : {isAbs : bool, vol : string, arcs : string list} -> string

val getVolume   : string -> string
val getParent   : string -> string

val splitDirFile : string -> {dir : string, file : string}
val joinDirFile  : {dir : string, file : string} -> string
val dir          : string -> string
val file         : string -> string

val splitBaseExt : string -> {base : string, ext : string option}
val joinBaseExt  : {base : string, ext : string option} -> string
val base         : string -> string
val ext          : string -> string option

val mkCanonical : string -> string
val isCanonical : string -> bool

val mkAbsolute  : (string * string) -> string
val mkRelative  : (string * string) -> string
val isAbsolute  : string -> bool
val isRelative  : string -> bool

val isRoot      : string -> bool

val concat      : (string * string) -> string
```

DESCRIPTION

This is a system independent module for manipulating strings that represent paths in the directory structure. The description of these operations can be found elsewhere.

SEE ALSO

OS(BASIS)

NAME

OS.Process — System independent interface to process primitives

SYNOPSIS

```
signature PROCESS

structure OS : OS =
  struct
    ...
    structure Process : OS_PROCESS
    ...
  end
```

SIGNATURE

```
eqtype status

val success : status
val failure : status

val system  : string -> status

val atExit : (unit -> unit) -> unit

val exit      : status -> 'a
val terminate : status -> 'a

val getEnv : string -> string option
```

DESCRIPTION

`success`

the unique status value that signifies sucessful termination of a process.

`failure`

a status value that signifies an error during the execution of a process. Note that unlike **`sucess`**, the value **`failure`** is not necessarily the only error value for the type status. For example, on UNIX systems, any small non-zero integer signals failure.

`system` *cmd*

executes the command *cmd* as a sub-process of the calling SML program. The call to `system` returns when the sub-process has completed, and return status of the sub-process is returned as a result. The format of the string is system dependent.

`atExit` *act*

registers the action *act* to be executed when the SML program exits (e.g., calls `exit`). Exit actions are executed in the order that they were registered.

`exit` *sts*

Causes the SML program to terminate after first invoking the exit actions. The convention is that *sts* is `success` for successful termination, and is `failure` in the case of errors.

`terminate`

This causes the SML program to terminate *without* invoking the exit actions.

COMMENT: the exit actions could have type `status->unit` to allow them to test the return code.

SEE ALSO

OS(BASIS)

NAME

PACK_WORD — packing/unpacking of words in arrays of bytes

SYNOPSIS

signature PACK_WORD
structure Pack*n*Big : PACK_WORD
structure Pack*n*Little : PACK_WORD

SIGNATURE

```
val bytesPerElem : int
val isBigEndian  : bool

val subVec  : (Word8Vector.vector * int) -> LargestWord.word
val subVecX : (Word8Vector.vector * int) -> LargestWord.word

val subArr  : (Word8Array.array * int) -> LargestWord.word
val subArrX : (Word8Array.array * int) -> LargestWord.word

val update  : (Word8Array.array * int * LargestWord.word) -> unit
```

DESCRIPTION

The `Pack`*n*`Big` structure provides a big-endian view of a sequence of bytes as a sequence of *n*-bit word values, with extraction and update operations. Likewise, a `Pack`*n*`Little` structure provides little-endian view. Typically, implementations will provide these structures for sizes equal to a power of 2 number of bytes (e.g., 16, 32 and 64 bits).

`bytesPerElem`

The number of bytes per element. Most implementations will provide structures for powers of two numbers of bytes (e.g., 2, 4, and 8).

`isBig`

This is true, if this structure implements a big-endian view of the data.

`subVec (`*vec*`, `*i*`)`

this extracts the `bytesPerElem` bytes starting at index *i*`*bytesPerElem`.

`subVecX (`*vec*`, `*i*`)`

this extracts and sign extends the `bytesPerElem` bytes starting at index *i*`*bytesPerElem`.

`subArr (`*arr*`, `*i*`)`

this extracts the `bytesPerElem` bytes starting at index *i*`*bytesPerElem`.

`subArrX (`*arr*`, `*i*`)`

this extracts and sign extends the `bytesPerElem` bytes starting at index *i*`*bytesPerElem`.

```
update (arr, i, w)
```

SEE ALSO

Byte(BASIS), MONO_ARRAY(BASIS) MONO_VECTOR(BASIS), WORD(BASIS)

NAME

Real — generic interface to real arithmetic

SYNOPSIS

signature REAL

structure Real : REAL

SIGNATURE

```
type real

structure ToLarge : CVT_REAL_INT
  sharing type ToLarge.real = real
          type ToLarge.int = LargestInt.int

structure ToInt : CVT_REAL_INT
  sharing type ToInt.real = real
          type ToInt.int = Integer.int

val + : real * real -> real
val - : real * real -> real
val * : real * real -> real
val / : real * real -> real
val ~ : real -> real

val abs      : real -> real
val sign     : real -> int
val sameSign : (real * real) -> bool

val toDefault   : real -> Real.real
val fromDefault : Real.real -> real

val floor : real -> Integer.int  (* rounds toward negative infinity *)
val ceil  : real -> Integer.int  (* rounds toward positive infinity *)
val trunc : real -> Integer.int  (* rounds toward zero *)
val round : real -> Integer.int  (* rounds toward nearest, ties->nearest even *)
val real  : Integer.int -> real

val < : real * real -> bool
val <= : real * real -> bool
val > : real * real -> bool
val >= : real * real -> bool

val compare : (real * real) -> ordering

val toString   : real -> string
val fromString : string -> real option
val scan : {getc : 'a -> (char * 'a) option} -> 'a -> (real * 'a) option
val fmt  : StringCvt.realfmt -> real -> string
```

DESCRIPTION

[[Should `real` be an eqtype??]]

`sign` *r*
: returns -1, if $r < 0$; and 1, if $r \geq 0$.

`sameSign (`*x*`,` *y*`)`
: returns true, if *x* and *y* have the same sign.

SEE ALSO

Math(BASIS), CONVERT_REAL_INT(BASIS)

NAME

String — basic operations on strings

SYNOPSIS

signature STRING

structure String : STRING

SIGNATURE

```
eqtype string

val maxLen : int

val size        : string -> int
val sub         : (string * int) -> char
val substring   : (string * int * int) -> string
val extract     : (string * int * int option) -> string
val concat      : string list -> string
val ^           : (string * string) -> string
val str         : char -> string
val implode     : char list -> string
val explode     : string -> char list

val translate : (char -> string) -> string -> string
val tokens    : (char -> bool) -> string -> string list
val fields    : (char -> bool) -> string -> string list

val compare : (string * string) -> ordering
val collate : ((char * char) -> ordering) -> (string * string) -> ordering

val <   : (string * string) -> bool
val <=  : (string * string) -> bool
val >   : (string * string) -> bool
val >=  : (string * string) -> bool
```

DESCRIPTION

Strings are finite sequences of upto `maxLen` characters. A *substring* is a triple (*s*, *i*, *n*), where *s* is a string, *i* is the starting index of the substring in *s*, and *n* is the number of characters in the substring. We say that a substring (*s*, *i*, *n*) is *valid*, if $0 \leq i \leq i + n \leq |s|$.

size *s*

returns the number of characters in the string *s*.

sub (*s***,** *i***)**

returns the *i*th character in the string *s*. If *i* is out of range, then the exception `Subscript` is raised.

substring (*s*, *i*, *n*)

returns an *n* character substring starting at the *i*th character of *s*. If the substring (*s*, *i*, *n*) is not valid, then the exception `Subscript` is raised.

concat *sl*

returns the concatenation of the list of strings *sl*.

s1ˆs2

returns the concatenation of *s1* and *s2*. This is a left-associative infix operator with precedence level 6.

str *c*

returns the string consisting of the character *c*.

implode *cl*

returns a string consisting of the characters in the list *cl*. This is equivalent to the expression `concat o (map str)`.

explode *s*

explodes the string *s* into a list of its constituent characters.

`translate` *tr s*

`tokens` *p s*

`fields` *p s*

`cmp` (*s1*, *s2*)

SEE ALSO

Char(BASIS), MONO_VECTOR(BASIS), Substring(BASIS)

NAME

StringCvt — basic support for string conversions

SYNOPSIS

signature STRING_CVT

structure StringCvt : STRING_CVT

SIGNATURE

```
datatype radix = BIN | OCT | DEC | HEX

datatype realfmt
  = SCI of int option
  | FIX of int option
  | GEN of int option

val toBool     : string -> bool option
val toChar     : string -> char option
val toInt      : string -> int option
val toReal     : string -> real option
val toString   : string -> string option
val toWord     : string -> word option

val fromBool   : bool -> string
val fromChar   : char -> string
val fromInt    : int -> string
val fromReal   : real -> string
val fromString : string -> string
val fromWord   : word -> string

val padLeft  : char -> int -> string -> string
val padRight : char -> int -> string -> string

type cs
val scanString :
      ({getc : cs -> (char * cs) option} -> cs -> ('a * cs) option)
        -> string -> 'a option
```

DESCRIPTION

The type `cs` is an intermediate type for the stream of characters being supplied to the scanning operation. For example in the following implementation, `cs` is `int`:

```
fun scanString scanFn s = let
      val n = String.length s
      fun getc i = if (i < n) then SOME(String.sub(s, i), i+1) else NONE
      in
        case (scanFn getc = getc 0)
         of NONE => NONE
          | SOME(x, _) => SOME x
        (* end case *)
      end
```

`fromChar` *c*

this converts the character *c* to a printable string representation. If *c* is non-printable, or is the special character `#"\\"` or `#"\""`, then a standard ML escape sequence is returned.

`toChar` *s*

this scans and converts a character from the string *s*. The standard ML escape sequences are recognized. Note that unlike other scanning functions, this function does not skip leading white-space. If *s* starts with a non-printing character or a poorly formed escape character, then `NONE` is returned. If *s* starts with an escape character code that is out of range, the the `Chr` exception is raised.

SEE ALSO

String(BASIS)

NAME

Substring — substring manipulations

SYNOPSIS

signature SUBSTRING

structure Substring : STRING

SIGNATURE

```
type substring

val base : substring -> (string * int * int)

val string : substring -> string

val substring : (string * int * int) -> substring
val all : string -> substring

val isEmpty : substring -> bool

val getc  : substring -> (char * substring) option
val first : substring -> char option
val triml : int -> substring -> substring
val trimr : int -> substring -> substring

val sub     : (substring * int) -> char
val size    : substring -> int
val slice   : (substring * int * int option) -> substring
val concat  : substring list -> string
val explode : substring -> char list

val compare : (substring * substring) -> ordering
val collate : ((char * char) -> ordering) -> (substring * substring) -> ordering

val splitl  : (char -> bool) -> substring -> (substring * substring)
val splitr  : (char -> bool) -> substring -> (substring * substring)
val splitAt : (substring * int) -> (substring * substring)

val dropl : (char -> bool) -> substring -> substring
val dropr : (char -> bool) -> substring -> substring
val takel : (char -> bool) -> substring -> substring
val taker : (char -> bool) -> substring -> substring

val position : string -> substring -> substring

val translate : (char -> string) -> substring -> string

val tokens : (char -> bool) -> substring -> substring list
```

```
val fields : (char -> bool) -> substring -> substring list

val foldl : ((char * 'a) -> 'a) -> 'a -> substring -> 'a
val foldr : ((char * 'a) -> 'a) -> 'a -> substring -> 'a
val app : (char -> unit) -> substring -> unit
```

DESCRIPTION

A *substring* is an abstract representation of a contiguous subsequence of a string; we can think of a substring as a triple $\langle s, i, n \rangle$, where s is the underlying string, i is the starting index of the substring in s, and n is the number of characters in the substring. In the following discussion, we use the notation $\langle s, i, n \rangle$ to refer to an abstract substring. We say that a substring $\langle s, i, n \rangle$ is *valid*, if $0 \leq i \leq i + n \leq |s|$. The functions for creating substrings check validity, and the substring operators all preserve validity. This allows efficient implementations that can avoid bounds checking.

`base` $\langle s, i, n \rangle$

returns the concrete representation of the substring; i.e., the triple `(`s`,` i`,` n`)`.

`string` $\langle s, i, n \rangle$

extracts the substring out as a string. This is the same as `String.substring(`s`,` i`,` n`)`.

`substring (`s`,` i`,` n`)`

Returns the substring $\langle s, i, n \rangle$, if it is valid. Otherwise, it raises the `Subscript` exception. This function may also raise `Overflow`, if $i + n$ is not representable as an `Integer.int`.

`all` *s*

returns a substring covering the entire string *s*.

`isEmpty` *ss*

returns `true`, if the substring is empty (i.e., has zero length).

`getc` *ss*

returns `NONE`, if *ss* is empty, otherwise it returns the first character in the substring and the rest of the substring.

`first` *ss*

returns `NONE`, if *ss* is empty, otherwise it returns the first character in the substring.

`triml` *k ss*

trims *k* characters off the left of the substring *ss*. If *k* is greater than the length of *ss*, the rightmost empty substring of *ss* is returned; if $k < 0$, then the `Subscript` exception is raised.

`trimr` *k ss*

trims *k* characters off the right of the substring *ss*. If *k* is greater than the length of *ss*, the leftmost empty substring of *ss* is returned; if $k < 0$, then the `Subscript` exception is raised.

`sub` `(`$\langle s, i, n \rangle$`, ` *j*`)`

returns `String.sub(`*s*`, ` *i*`+`*j*`)`, if $0 \leq j < n$. Otherwise the `Subscript` exception is raised.

`size` $\langle s, i, n \rangle$

returns *n*.

SEE ALSO

Char(BASIS), String(BASIS)

NAME

Time — Representation of time values

SYNOPSIS

signature TIME

structure Time : TIME

SIGNATURE

```
eqtype time

exception Time

val zeroTime : time

val realToTime : real -> time
val timeToReal : time -> real

val toSeconds        : time -> int
val fromSeconds      : int -> time
val toMilliseconds   : time -> int
val fromMilliseconds : int -> time
val toMicroseconds   : time -> int
val fromMicroseconds : int -> time

val +  : (time * time) -> time
val -  : (time * time) -> time

val <  : (time * time) -> bool
val <= : (time * time) -> bool
val >  : (time * time) -> bool
val >= : (time * time) -> bool

val compare : (time * time) -> ordering

val now : unit -> time

val fmt  : int -> time -> string
val scan : {getc : 'a -> (char * 'a) option} -> 'a -> (time * 'a) option

val toString   : time -> string
val fromString : string -> time option
```

DESCRIPTION

The abstract type `time` is used to represent both intervals of time and absolute time values (which can be thought of as intervals since some time zero).

`zeroTime`

is the time representation of zero (e.g., `realToTime 0.0`).

`realToTime` *r*

converts a real number representing seconds to a time value. If $r < 0$, then the exception `Time` is raised.

`timeToReal` *t*

If the this is not representable as an `Real.real`, then the `Overflow` exception is raised.

`fromSeconds` *sec*

converts the integer number of seconds *sec* to a time value. If *sec* is negative, then the `Time` exception is raised.

`toSeconds` *t*

returns the integer number of seconds represented by the time value *t*. The conversion is done by truncation; fractional parts of a second are discarded. If the number of two seconds is two large to be represented as an `int`, then the `Overflow` exception is raised.

`toMilliseconds` *sec*

`fromMilliseconds` *t*

`toMicroseconds` *sec*

`fromMicroseconds` *t*

`(`*t1* `+` *t2*`)`

adds the time value *t2* to *t1*.

`(`*t1* `-` *t2*`)`

subtracts the time value *t2* from *t1*. If $t1 < t2$, then the `Time` exception is raised.

`(`*t1* `<` *t2*`)`

returns `true`, if $t1 < t2$.

`(`*t1* `<=` *t2*`)`

returns `true`, if $t1 <= t2$.

`(`*t1* `>` *t2*`)`

returns `true`, if $t1 > t2$.

`(`*t1* `>=` *t2*`)`

returns `true`, if $t1 >= t2$.

`now ()`

returns the current time of day. The interpretation of this value is system dependent, but the values returned by successive calls to `now` are monotonically increasing.

`fmt` *prec t*

converts the time value *t* to a string representation of the number of seconds. The integer *prec* specifies the number of decimal digits to report. If $prec \leq 0$, then no decimal digits are reported.

`scan {getc}` *charSrc*

`toString` *t*

Converts the time value *t* to a string with millisecond precision. It is equivalent to: `fmt 3`.

`fromStringing` *s*

This converts the string *s* to a time value; it returns `NONE`, if *s* is not valid, and raises `Overflow` if *s* is too large. It is equivalent to: `StringCvt.scanString scan`.

SEE ALSO

Date(BASIS), Timer(BASIS)

NAME

Timer — Interface to system timers

SYNOPSIS

signature TIMER

structure Timer : TIMER

SIGNATURE

```
type cpu_timer
type real_timer

val totalCPUTimer : unit -> cpu_timer
val startCPUTimer : unit -> cpu_timer
val checkCPUTimer : cpu_timer -> {usr : Time.time, sys : Time.time, gc : Time.time}

val totalRealTimer : unit -> real_timer
val startRealTimer : unit -> real_timer
val checkRealTimer : real_timer -> Time.time
```

DESCRIPTION

This module provides *timers* for measuring both CPU and real (wall-clock) time.

totalTimer ()

returns a timer that was started at system start-up.

startTimer ()

starts a new timer.

checkTimer *timer*

returns the current values of a timer. For CPU timing, this is broken out into user, system and garbage collector time.

SEE ALSO

Time(BASIS)

CAVEATS

Some systems may not provide a mechanism for measuring CPU time, in which case, real time should be substituted.

NAME

Vector — immutable polymorphic vectors

SYNOPSIS

signature VECTOR

structure Vector : VECTOR

SIGNATURE

```
eqtype 'a vector

val maxLen   : int

val fromList : 'a list -> 'a vector
val tabulate : (int * (int -> 'a)) -> 'a vector

val length   : 'a vector -> int
val sub      : ('a vector * int) -> 'a
val extract  : ('a vector * int * int option) -> 'a vector
val concat   : 'a vector list -> 'a vector

val app      : ('a -> unit) -> 'a vector -> unit
val foldl    : (('a * 'b) -> 'b) -> 'b -> 'a vector -> 'b
val foldr    : (('a * 'b) -> 'b) -> 'b -> 'a vector -> 'b

val appi     : ((int * 'a) -> unit) -> ('a vector * int * int option) -> unit
val foldli   : ((int * 'a * 'b) -> 'b) -> 'b -> ('a vector * int * int option) -> 'b
val foldri   : ((int * 'a * 'b) -> 'b) -> 'b -> ('a vector * int * int option) -> 'b
```

DESCRIPTION

The `Vector` structure provides one-dimensional, zero-based, immutable indexable arrays.

`maxLen`

is the maximum length supported for polymorphic vectors.

`vector` *l*

creates an vector from the list of elements *l*. This raises the `Size` exception, if the *l* has more than `maxLen` elements.

`tabulate (`*n*`,` *f*`)`

creates an vector of *n* elements, where the *i*th element is initialized to `f(`*i*`)`. The function *f* is called in increasing order of *i*. This raises the `Size` exception, if *n* is either too large (> `maxLen`) or negative.

`length` *vec*

returns the length of the vector *vec*.

`sub (`*vec*`, `*i*`)`

returns the *i*th element of *vec*. The exception `Subscript` is raised if *i* is out of bounds.

`extract (`*vec*`, `*i*`, `*n*`)`

extracts a vector of length *n* from the vector *vec*, starting with the *i*th element. The exception `Subscript` is raised if $i < 0 \vee n < 0 \vee |vec| < i + n$.

`concat` *vl*

forms the concatenation of a list of vectors. If the sum of the lengths exceeds `maxLen`, then the `Size` exception is raised.

SEE ALSO

Array(BASIS), MONO_VECTOR(BASIS)

NAME

Word — unsigned integers

SYNOPSIS

signature WORD

structure Word : WORD

structure Word*n* : WORD

SIGNATURE

```
eqtype word

val wordSize  : int

structure ToWord : CONVERT_WORD
  sharing type ToWord.word = word
          type ToWord.to = LargeWord.word

structure ToInt : CONVERT_WORD
  sharing type ToInt.word = word
          type ToInt.to = LargeInt.word

val orb  : word * word -> word
val xorb : word * word -> word
val andb : word * word -> word
val notb : word -> word

val shift : word * int -> word
val ashift : word * int -> word

val + : word * word -> word
val - : word * word -> word
val * : word * word -> word
val div : word * word -> word
val mod : word * word -> word

val >  : word * word -> bool
val <  : word * word -> bool
val >= : word * word -> bool
val <= : word * word -> bool

val compare : (word * word) -> ordering

val toString   : word -> string
val fromString : string -> word option
val scan : StringCvt.radix -> {getc : 'a -> (char * 'a) option} -> 'a -> (word * 'a) option
val fmt  : StringCvt.radix -> word -> string
```

DESCRIPTION

The `word` type represents integers modulo 2^n, where `wordSize` $= n$.

If the structure `SmallInt` is present, then

```
SmallInt.precision = SOME(Word.wordSize)
```

Also, if there are both `Int`*n* and `Word`*n* structures present, then

```
Intn.precision = SOME(Wordn.wordSize)
```

For the purposes of defining the semantics of the logical operations, the following definition is useful:

$$\text{bitwise}(\oplus) \;=\; (\sum_{i=0}^{n-1} 2^i(x_i \oplus y_i)) \text{ mod } 2^n,$$

where $x_i = \lfloor x/2^i \rfloor$ mod 2.

`intToWord` *i*
: yields a word w representing i mod 2^n. Cannot raise `Overflow`.

`wordToInt` *w*
: Returns a the smallest nonnegative integer i such that $\texttt{intToWord}(i) = w$, if i is representable as an `int`. Otherwise, returns the negative integer i of smallest absolute value such that $\texttt{intToWord}(i) = w$, if i is representable as an `int`. Otherwise, raises `Overflow`.

`signExtend` *w*
: If w mod $2^n = w$ mod 2^{n-1}, returns the smallest nonnegative integer i such that $\texttt{intToWord}(i) = w$.

  If w mod $2^n \neq w$ mod 2^{n-1}, returns the negative integer i of smallest absolute value such that $\texttt{intToWord}(i) = w$.

  If no such i is representable, raises `Overflow`.

`orb (`*x*`, `*y*`)`
: returns the bitwise or of *x* and *y*. That is, `orb` = bitwise$(\lambda(a, b).(1 - a)(1 - b))$.

`xorb (`*x*`, `*y*`)`
: bitwise exclusive-or, that is `xorb` = bitwise$(\lambda(a, b).(a + b)$ mod 2$)$.

`andb (`*x*`, `*y*`)`
: bitwise and, that is `andb` = bitwise$(\lambda(a, b).a \cdot b)$.

`notb` *w*
: returns the bitwise complement of *w*, that is `notb` $= \lambda w.$bitwise$(\lambda(a, b).1 - a)(w, w)$.

`shift(`*w*`,` *k*`)`

shifts w left k bits; or shifts right if k is negative. $\texttt{shift}(w, k) = \lfloor (w \bmod 2^n) \cdot 2^k \rfloor \bmod 2^n$.

`ashift(`*w*`,` *k*`)`

Arithemetic shift: shifts w left k bits; or shifts right if k is negative; copies the "sign bit" on right shifts.

$$\begin{aligned}\texttt{ashift}(w,k) &= \texttt{shift}(w,k) \text{ if } w \bmod 2^n = w \bmod 2^{(n-1)} \text{ or } k \geq 0 \\ \texttt{ashift}(w,k) &= -\texttt{shift}(-w,k) \quad \text{otherwise}\end{aligned}$$

`op + (`*w1*`,` *w2*`)`

returns $(w1 + w2) \bmod 2^n$.

`op - (`*w1*`,` *w2*`)`

returns $(w1 - w2) \bmod 2^n$.

`op * (`*w1*`,` *w2*`)`

returns $(w1 \times w2) \bmod 2^n$.

`op div (`*x*`,` *y*`)`

Unsigned division: returns $\left\lfloor \frac{x'}{y'} \right\rfloor$, where $x' = x \bmod 2^n \;\wedge\; 0 \leq x' < 2^n \;\wedge\; y' = y \bmod 2^n \;\wedge\; 0 \leq y' < 2^n$. Raises the `Div` exception if y' is `0`.

`op mod (`*x*`,` *y*`)`

returns $(x - y \cdot (x \;\texttt{div}\; y)) \bmod 2^n$. Raises the `Div` exception if y is `0`.

SEE ALSO

Byte(BASIS), Int(BASIS), SmallInt(BASIS), CONVERT_WORD(BASIS)

Part III

Amendment: POSIX 1003.1b-1993

NAME

POSIX — POSIX 1003.1 binding

SYNOPSIS

```
signature POSIX
structure Posix : POSIX
```

SIGNATURE

```
structure Error   : POSIX_ERROR
structure Signal  : POSIX_SIGNAL
structure Process : POSIX_PROCESS
structure ProcEnv : POSIX_PROC_ENV
structure FileSys : POSIX_FILE_SYS
structure IO      : POSIX_IO
structure SysDB   : POSIX_SYS_DB
structure TTY     : POSIX_TTY
  sharing type Process.pid = ProcEnv.pid = TTY.pid
      and type Process.signal = Signal.signal
      and type ProcEnv.file_desc = FileSys.file_desc
             = PrimIO.file_desc = TTY.file_desc
      and type FileSys.offset = IO.offset = PrimIO.offset
      and type FileSys.open_mode = IO.open_mode
      and type ProcEnv.uid = FileSys.uid = SysDB.uid
      and type ProcEnv.gid = FileSys.gid = SysDB.gid
```

DESCRIPTION

The `POSIX` structure defines an SML binding for the POSIX standard IEEE Std 1003.1b-1993 (with some 1003.1a extensions). The organization of the `POSIX` structure largely follows that of the standard; each substructure except for `Signal` and `Error` corresponds to a different section in the standard.

SEE ALSO

Posix.Error(BASIS), Posix.Signal(BASIS), Posix.Process(BASIS), Posix.ProcEnv(BASIS), Posix.FileSys(BASIS), Posix.IO(BASIS), Posix.SysDB(BASIS), Posix.TTY(BASIS), POSIX_FLAGS(BA

NAME

Posix.Error — system errors

SYNOPSIS

```
signature POSIX_ERROR

structure Posix : POSIX =
  struct
    ...
    structure Error : POSIX_ERROR
    ...
  end
```

SIGNATURE

```
eqtype syserror

val errorMsg   : syserror -> string
val wordOf     : syserror -> SystemWord.word
val syserror   : SystemWord.word -> syserror

val toobig      : syserror
val acces       : syserror
val again       : syserror
val badf        : syserror
val badmsg      : syserror
val busy        : syserror
val canceled    : syserror
val child       : syserror
val deadlk      : syserror
val dom         : syserror
val exist       : syserror
val fault       : syserror
val fbig        : syserror
val inprogress  : syserror
val intr        : syserror
val inval       : syserror
val io          : syserror
val isdir       : syserror
val loop        : syserror
val mfile       : syserror
val mlink       : syserror
val msgsize     : syserror
val nametoolong : syserror
val nfile       : syserror
val nodev       : syserror
val noent       : syserror
val noexec      : syserror
```

```
val nolck       : syserror
val nomem       : syserror
val nospc       : syserror
val nosys       : syserror
val notdir      : syserror
val notempty    : syserror
val notsup      : syserror
val notty       : syserror
val nxio        : syserror
val perm        : syserror
val pipe        : syserror
val range       : syserror
val rofs        : syserror
val spipe       : syserror
val srch        : syserror
val xdev        : syserror
```

DESCRIPTION

This structure encapsulates errors associated with POSIX system calls. In more typical implementations, these errors would be represented as values of the `errno` variable declared in `/usr/include/errno.h`. The declared `syserror` values correspond to the basic errors defined in the POSIX standard (cf. Section 2.4 of IEEE Std 1003.1b-1993). The function `errorMsg` maps an error code to an error message (e.g., `errorMsg enoent` might return the string `"No such file or directory"`). The `syserror` and `wordOf` functions provide access to the underlying representation of the error value. Values created by the former have the possibility of not being defined in all POSIX compliant systems.

SEE ALSO

Posix(BASIS)

NAME

POSIX_FLAGS — POSIX bit flags interface

SYNOPSIS

```
signature POSIX_FLAGS
```

SIGNATURE

```
eqtype flags

val toWord  : flags -> SystemWord.word
val wordTo : SystemWord.word -> flags

val flags  : flags list -> flags

val allSet : flags * flags -> bool
val anySet : flags * flags -> bool
```

DESCRIPTION

This signature specifies the common operations used for setting and testing flags used in POSIX functions. Typically, this signature is included in a substructure that also provides a collection of pre-defined flags (cf. Posix.IO.O). The function `flags` forms the union of all the flags set in its argument list. The call `allSet (f,f')` returns true if all the flags set in `f` are also set in `f'`, i.e., `f` is a subset of `f'`. The call `anySet (f,f')` returns true if any flag set in `f` is also set in `f'`, i.e., the intersection of `f` and `f'` is non-empty. The `wordTo` and `toWord` functions provide access to the underlying representation of the flags as bits set in a word. Values created by the former have the possibility of not being defined in all POSIX compliant systems.

SEE ALSO

Posix(BASIS), Posix.Process(BASIS), Posix.FileSys(BASIS), Posix.IO(BASIS)

NAME

Posix.FileSys — operations on the file system

SYNOPSIS

```
signature POSIX_FILE_SYS

structure Posix : POSIX =
  struct
    ...
    structure FileSys : POSIX_FILE_SYS
    ...
  end
```

SIGNATURE

```
eqtype uid
eqtype gid
eqtype file_desc

val fdToWord    : file_desc -> SystemWord.word
val wordToFD    : SystemWord.word -> file_desc

type nlink
type offset

type dirstream

val openDir   : string -> dirstream
val readDir   : dirstream -> string
val rewindDir : dirstream -> unit
val closeDir  : dirstream -> unit

val chdir  : string -> unit
val getcwd : unit -> string

val stdin  : file_desc
val stdout : file_desc
val stderr : file_desc

structure S :
  sig
    include POSIX_FLAGS

    type mode
      sharing type mode = flags

    val irwxu : mode
    val irusr : mode
```

```
    val iwusr : mode
    val ixusr : mode
    val irwxg : mode
    val irgrp : mode
    val iwgrp : mode
    val ixgrp : mode
    val irwxo : mode
    val iroth : mode
    val iwoth : mode
    val ixoth : mode
    val isuid : mode
    val isgid : mode
  end

datatype open_mode = O_RDONLY | O_WRONLY | O_RDWR

structure O :
  sig
    include POSIX_FLAGS

    val append   : flags
    val dsync    : flags
    val excl     : flags
    val noctty   : flags
    val nonblock : flags
    val rsync    : flags
    val sync     : flags
    val trunc    : flags
  end

val openf     : (string * open_mode * O.flags) -> file_desc
val createf   : (string * open_mode * O.flags * S.mode) -> file_desc
val creat     : (string * S.mode) -> file_desc
val umask     : S.mode -> S.mode
val link      : {old : string, new : string} -> unit
val mkdir     : string * S.mode -> unit
val mkfifo    : string * S.mode -> unit
val unlink    : string -> unit
val rmdir     : string -> unit
val rename    : {old : string, new : string} -> unit
val symlink   : {old : string, new : string} -> unit
val readlink  : string -> string

eqtype dev
val wordToDev : SystemWord.word -> dev
val devToWord : dev -> SystemWord.word

eqtype ino
val wordToIno : SystemWord.word -> ino
val inoToWord : ino -> SystemWord.word
```

```
eqtype file_type
val isDir  : file_type -> bool
val isChr  : file_type -> bool
val isBlk  : file_type -> bool
val isReg  : file_type -> bool
val isFIFO : file_type -> bool
val isLink : file_type -> bool
val isSock : file_type -> bool

structure ST :
  sig
    type stat

    val fileType   : stat -> file_type
    val mode       : stat -> S.mode
    val ino        : stat -> ino
    val dev        : stat -> dev
    val nlink      : stat -> nlink
    val uid        : stat -> uid
    val gid        : stat -> gid
    val size       : stat -> offset option
    val atime      : stat -> Time.time
    val mtime      : stat -> Time.time
    val ctime      : stat -> Time.time
  end

val stat : string -> ST.stat
val lstat : string -> ST.stat
val fstat : file_desc -> ST.stat

datatype access_mode = A_READ | A_WRITE | A_EXEC
val access : string * access_mode list -> bool

val chmod  : (string * S.mode) -> unit
val fchmod : (file_desc * S.mode) -> unit

val chown  : (string * uid * gid) -> unit
val fchown : (file_desc * uid * gid) -> unit

val utime : string * {actime : Time.time, modtime : Time.time} option -> unit

val ftruncate : file_desc * offset -> unit

val pathconf  : (string * string) -> SystemWord.word option
val fpathconf : (file_desc * string) -> SystemWord.word option
```

DESCRIPTION

This structure provides the basic POSIX operations on file systems, as described in Section 5 of IEEE Std 1003.1b-1993. The `wordToFD` and `fdToWord` functions provide access to the underlying arithmetic representation of a `file_desc` value. Similar statements hold for the functions `wordToDev`, `devToWord`, `wordToIno` and `inoToWord` and the types `dev` and `ino`. The substructure `S` implements the standard POSIX permission bits. Here also, the functions `S.wordTo` and `S.toWord` allow access to the underlying arithmetic representation. The functions `symlink`, `readlink`, `lstat` and `fchown` are provided as part of the POSIX standard 1003.1a, although this has not been officially accepted as yet. The functions `pathconf` and `fpathconf` return `NONE` if the corresponding value is unbounded.

SEE ALSO

Posix(BASIS), POSIX_FLAGS(BASIS)

NAME

Posix.IO — basic I/O operations

SYNOPSIS

```
signature POSIX_IO

structure Posix : POSIX =
  struct
    ...
    structure IO : POSIX_IO
    ...
  end
```

SIGNATURE

```
eqtype file_desc
eqtype offset
eqtype pid

val pipe : unit -> {infd : file_desc, outfd : file_desc}
val dup : file_desc -> file_desc
val dup2 : {old : file_desc, new : file_desc} -> unit
val close : file_desc -> unit

val readVec  : (file_desc * int) -> Word8Vector.vector
val readArr  : (file_desc * {buf : Word8Array.array, i : int, sz : int}) -> int
val writeVec : (file_desc * Word8Vector.vector * int) -> int
val writeArr : (file_desc * {buf : Word8Array.array, i : int, sz : int}) -> int

datatype whence = SEEK_SET | SEEK_CUR | SEEK_END

structure FD :
  sig
    include POSIX_FLAGS

    val cloexec : flags

  end

structure O :
  sig
    include POSIX_FLAGS

    val append   : flags
    val dsync    : flags
    val nonblock : flags
    val rsync    : flags
    val sync     : flags
```

```
  end

datatype open_mode = O_RDONLY | O_WRONLY | O_RDWR

val dupfd : {old : file_desc, new : file_desc} -> unit
val getfd : file_desc -> FD.flags
val setfd : (file_desc * FD.flags) -> unit
val getfl : file_desc -> (O.flags * open_mode)
val setfl : (file_desc * O.flags) -> unit

datatype lock_type = F_RDLCK | F_WRLCK | F_UNLCK

structure Flock :
  sig
    type flock

    val flock : { l_type : lock_type,
                  l_whence : whence,
                  l_start : offset,
                  l_len : offset,
                  l_pid : pid option} -> flock

    val ltype    : flock -> lock_type
    val whence   : flock -> whence
    val start    : flock -> offset
    val len      : flock -> offset
    val pid      : flock -> pid option
  end

val getlk  : (file_desc * Flock.flock) -> Flock.flock
val setlk  : (file_desc * Flock.flock) -> Flock.flock
val setlkw : (file_desc * Flock.flock) -> Flock.flock

val lseek : (file_desc * offset * whence) -> offset

val fsync : file_desc -> unit
```

DESCRIPTION

This structure provides the primitive POSIX I/O operations, as described in Section 6 of IEEE Std 1003.1b-1993. The functions **dupfd**, **getfd**, **setfd**, **getfl**, **setfl**, **getlk**, **setlk** and **setlkw** correspond to calls to the POSIX **fcntl** function with the commands F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK and F_SETLKW, respectively. The substructure FD implements sets of file descriptor flags, the only POSIX required value being **cloexec** corresponding to the C constant FD_CLOEXEC. Similarly, the substructure O implements sets of file status flags, with the supplied values **append**, **dsync**, **nonblock**, **rsync** and **sync** corresponding to the POSIX defined C constants O_APPEND, O_DSYNC, O_NONBLOCK, O_RSYNC and O_SYNC, respectively.

SEE ALSO

Posix(BASIS), POSIX_FLAGS(BASIS)

NAME

Posix.ProcEnv — operations on the process environment

SYNOPSIS

```
signature POSIX_PROC_ENV

structure Posix : POSIX =
  struct
    ...
    structure ProcEnv : POSIX_PROC_ENV
    ...
  end
```

SIGNATURE

```
eqtype pid
eqtype uid
eqtype gid
eqtype file_desc

val uidToWord : uid -> SystemWord.word
val wordToUid : SystemWord.word -> uid
val gidToWord : gid -> SystemWord.word
val wordToGid : SystemWord.word -> gid

val getpid  : unit -> pid
val getppid : unit -> pid

val getuid  : unit -> uid
val geteuid : unit -> uid
val getgid  : unit -> gid
val getegid : unit -> gid

val setuid : uid -> unit
val setgid : gid -> unit

val getgroups : unit -> gid list

val getlogin : unit -> string

val getpgrp : unit -> pid
val setsid  : unit -> pid
val setpgid : {pid : pid option, pgid : pid option} -> unit

val uname : unit -> (string * string) list

val time : unit -> Time.time
```

```
val times : unit -> {
        elapsed : Time.time,
        utime   : Time.time,
        stime   : Time.time,
        cutime  : Time.time,
        cstime  : Time.time
      }

val getenv  : string -> string option
val environ : unit -> string list

val ctermid : unit -> string
val ttyname : file_desc -> string
val isatty : file_desc -> bool

val sysconf : string -> SystemWord.word
```

DESCRIPTION

This structure encapsulates the POSIX operations on the process environment, as described in Section 4 of IEEE Std 1003.1b-1993. The `wordToUid`, `wordToGid`, `uidToWord` and `gidToWord` functions provide access to the underlying arithmetic representation of `uid` and `gid` values. The `sysconf` raises an exception if the corresponding feature is not supported by the underlying operating system.

SEE ALSO

Posix(BASIS)

NAME

Posix.Process — operations on processes

SYNOPSIS

```
signature POSIX_PROCESS

structure Posix : POSIX =
  struct
    ...
    structure Process : POSIX_PROCESS
    ...
  end
```

SIGNATURE

```
eqtype signal
eqtype pid

val wordToPid    : SystemWord.word -> pid
val pidToWord    : pid -> SystemWord.word

val fork : unit -> pid option

val exec  : string * string list -> 'a
val exece : string * string list * string list -> 'a
val execp : string * string list -> 'a

datatype waitpid_arg
= W_ANY_CHILD
| W_CHILD of pid
| W_SAME_GROUP
| W_GROUP of pid

datatype exit_status
= W_EXITED
| W_EXITSTATUS of Word8.word
| W_SIGNALED of signal
| W_STOPPED of signal

structure W :
  sig
    include POSIX_FLAGS

    val untraced   : flags

  end

val wait : unit -> pid * exit_status
```

```
val waitpid : waitpid_arg * W.flags list -> pid * exit_status
val waitpid_nh : waitpid_arg * W.flags list -> (pid * exit_status) option

val exit : Word8.word -> 'a

datatype killpid_arg
= K_PROC of pid
| K_SAME_GROUP
| K_GROUP of pid

val kill : killpid_arg * signal -> unit

val alarm : Time.time -> Time.time
val pause : unit -> unit
val sleep : Time.time -> Time.time
```

DESCRIPTION

This structure encapsulates the basic POSIX operations on processes, as described in Section 3 of IEEE Std 1003.1b-1993. The `wordToPid` and `pidToWord` functions provide access to the underlying representation of a `pid` value.

SEE ALSO

Posix(BASIS), POSIX_FLAGS(BASIS)

NAME

Posix.Signal — system signals

SYNOPSIS

```
signature POSIX_SIGNAL

structure Posix : POSIX =
  struct
    ...
    structure Signal : POSIX_SIGNAL
    ...
  end
```

SIGNATURE

```
eqtype signal

val toWord   : signal -> SystemWord.word
val fromWord : SystemWord.word -> signal

val abrt : signal
val alrm : signal
val bus  : signal
val fpe  : signal
val hup  : signal
val ill  : signal
val int  : signal
val kill : signal
val pipe : signal
val quit : signal
val segv : signal
val term : signal
val usr1 : signal
val usr2 : signal
val chld : signal
val cont : signal
val stop : signal
val tstp : signal
val ttin : signal
val ttou : signal
```

DESCRIPTION

This structure provides POSIX signals. The declared `signal` values correspond to the basic signals defined in Section 3.3 of the POSIX standard IEEE Std 1003.1b-1993. The `signal` and `wordOf` functions provide access to the underlying representation of the signal value. Values created by the former have the possibility of not being defined in all POSIX compliant systems.

SEE ALSO

Posix(BASIS)

NAME

Posix.SysDB — operations on the system data-base

SYNOPSIS

```
signature POSIX_SYS_DB

structure Posix : POSIX =
  struct
    ...
    structure SysDB : POSIX_SYS_DB
    ...
  end
```

SIGNATURE

```
eqtype uid
eqtype gid

structure Passwd :
  sig
    type passwd

    val name  : passwd -> string
    val uid   : passwd -> uid
    val gid   : passwd -> gid
    val home  : passwd -> string
    val shell : passwd -> string

  end

structure Group :
  sig
    type group

    val name    : group -> string
    val gid     : group -> gid
    val members : group -> string list

  end

val getgrgid : gid -> Group.group
val getgrnam : string -> Group.group
val getpwuid : uid -> Passwd.passwd
val getpwnam : string -> Passwd.passwd
```

DESCRIPTION

These are the operations described in Section 9 of the IEEE Std 1003.1b-1993.

SEE ALSO

Posix(BASIS)

NAME

Posix.Tty — operations on terminal devices

SYNOPSIS

```
signature POSIX_TTY

structure Posix : POSIX =
  struct
    ...
    structure TTY : POSIX_TTY
    ...
  end
```

SIGNATURE

```
eqtype pid       (* process ID *)
eqtype file_desc (* file descriptor *)

datatype c_iflag
  = BRIINT | ICRNL | IGNBRK | IGNCR | IGNPAR | INLCR
  | INPCK | ISTRIP | IXOFF | IXON | PARMRK
datatype c_oflag = OPOST
datatype cbits
  = CS5 | CS6 | CS7 | CS8
datatype c_cflag
  = CLOCAL | CREAD | CSIZE of cbits | CSTOPB | HUPCL
  | PARENB | PARODD
datatype c_lflag
  = ECHO | ECHOE | ECHOK | ECHONL | ICANON | IEXTEN
  | ISIG | NOFLSH | TOSTOP
datatype cc_item
  = VEOF | VEOL | VERASE | VINTR | VKILL | VMIN | VQUIT
  | VSUSP | VTIME | VSTART | VSTOP

type cc
val newcc    : (cc_item * string) list -> cc
val updatecc : (cc * (cc_item * string) list) -> cc
val subcc    : (cc * cc_item) -> string

type termios

datatype tcset_action = TCSANONE | TCSANOW | TCSADRAIN | TCSAFLUSH
datatype queue_sel = TCIFLUSH | TCOFLUSH | TCIOFLUSH
datatype flow_action = TCOOF | TCOON | TCIOFF | TCION
datatype speed
  = B0 | B50 | B75 | B110 | B134 | B150 | B200 | B300 | B600 | B1200
  | B1800 | B2400 | B4800 | B9600 | B19200 | B38400
```

```
val cfgetospeed : termios -> speed
val cfsetospeed : (termios * speed) -> unit
val cfgetispeed : termios -> speed
val cfsetispeed : (termios * speed) -> unit

val tcgetattr : file_desc -> termios
val tcsetattr : file_desc * tcset_action * termios -> unit
val tcsendbreak : file_desc * int -> unit
val tcdrain : file_desc -> unit
val tcflush : file_desc * queue_sel -> unit
val tcflow : file_desc * flow_action -> unit
val tcgetpgrp : file_desc -> pid
val tcsetpgrp : file_desc * pid -> unit
```

DESCRIPTION

These are the operations described in Section 7 of the IEEE Std 1003.1-1990.

SEE ALSO

Posix(BASIS)

Bibliography

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[POS90] IEEE. *POSIX – Part 1: System Application Program Interface*, 1990.

[Rep90] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Department of Computer Science, Cornell University, August 1990.

[Vil88] Villemin, J. Exact real computer arithmetic with continued fractions. In *Conference record of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, pp. 14–27.