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

March 5, 1994

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

Andrew W. Appel	Department of Computer Science, Princeton University
Dave Berry	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
John H. Reppy	AT&T Bell Laboratories

Part I

Discussion

Chapter 1

Introduction

NOTE: THIS IS AN INCOMPLETE DRAFT

This is a proposal for a replacement of the Standard ML initial basis.

Following the hint of Berry, Milner, et al., we are assuming that the initial basis of the Definition [MTH90] can be entirely revamped. This is our own proposal.

Philosophy

[[Everything should belong to a particular structure (except for overloading and infix]]

Summary

Summary of our 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.
- Both constructive reals and floating-point; implementations must implement at least one of these. Floating-point semantics specified in more detail, and with more operators, than in the Definition [MTH90].
- More comprehensive operators on strings.
- Mutable arrays and immutable vectors, with constant-time random-access.
- Input/output, and other operating-system interface.

1.1 Conventions

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 data constructors should be capitalized to distinguish them from variables; there seems to be wide agreement on this point. Since we are revamping the initial basis, this is the logical time to alter the capitalization of nil, true, and false.

To write down a proposal, we had to choose a capitalization convention. We don't wish to debate capitalization; feel free to make an alternate proposal for capitalization, and let's try to keep that issue separate from the semantics.

The convention we use is:

- Alphanumeric value variables in lower-case; words separated by underscore. Examples: `map`, `open_in`.
- Alphanumeric constructors in all caps: `SOME`, `NONE`.
- Type identifiers following the same rules as value variables.
- Signature identifiers in all caps, words separated by underscore.
- Structure and functor identifiers with initial letter capitalized.

While capitalization is a touchy subject, we strongly believe that data 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.

The initial basis is contained in a set of structures. Some of these structures are initially opened.

1.2 Overview

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.

Table 1.1: List of required generic signatures

Signature	Description
CONVERT_REAL	Conversions between two real representations.
CONVERT_INT	Conversions between two integer representations.
CONVERT_REAL_INT	Conversions between integer and real 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.

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 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. The key to the three *status* columns is:

O is the structure partially open at top-level?

L is the structure pre-loaded in the interactive environment?

M does the structure require special compiler or run-time system support?

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

Table 1.2: List of required structures

Module	Signature	Status O	L	M	Description
`Array`	`ARRAY`		Y	Y	Mutable polymorphic arrays.
`BinIO`	`BIN_IO`			Y	Binary input/output streams and operations.
`Char`	`CHAR`		Y	Y	Characters
`CharArray`	`MONO_ARRAY`		Y	Y	Mutable arrays of characters
`CharVector`	`MONO_VECTOR`		Y	Y	Immutable vectors of characters
`CType`	`CTYPE`				Character classification operations.
`Date`	`DATE`			Y	Calander operations
`FmtDate`	`FMT_DATE`			Y	Formatting dates
`General`	`GENERAL`	Y	Y	Y	General-purpose types, exceptions and miscellaneous operations.
`Integer`	`INTEGER`	Y	Y	Y	Default interger structure.
`List`	`LIST`	Y			Useful utility functions on lists.
`Math`	`MATH`				Default math structure.
`OS`	`OS`		Y	Y	Basic operating system services.
`OS.FileSys`	`FILE_SYS`		Y		File status and directory operations
`OS.Path`	`PATH`				Pathname operations
`OS.Process`	`PROCESS`				Simple process manipulation operations
`PrimIO`	`PRIM_IO`		Y	Y	Primitive input/output operations.
`Real`	`REAL`	Y	Y	Y	Default real structure.
`String`	`STRING`	Y	Y	Y	Strings (cf., `CharVector`)
`StringUtil`	`STRING_UTIL`		?		String utility functions
`TextIO`	`TEXT_IO`	Y	Y	Y	Text input/output streams and operations.
`Time`	`TIME`			Y	Representation of time values
`Timer`	`TIMER`			Y	Timing operations
`Vector`	`VECTOR`		Y	Y	Immutable polymorphic vectors.

Table 1.3: List of optional structures

Module	**Signature**	**Status** O	L	M	**Description**
BoolArray	MONO_ARRAY			Y	Mutable arrays of booleans
BoolVector	MONO_VECTOR			Y	Immutable vectors of booleans
ByteArray	BYTEARRAY			Y	Mutable arrays of bytes (8-bit integers).
Cvt	n.a.			Y	Contains various arithmetic conversion substructures.
DoubleFloat	FLOAT	Y		Y	Double-precision floating-point numbers.
DoubleFloatArray	MONO_ARRAY			Y	Mutable arrays of double-precision floating-point numbers.
DoubleFloatVector	MONO_VECTOR			Y	Immutable vectors of double-precision floating-point numbers.
DoubleMath	MATH				Double-precision floating-point math library.
ExtFloat	FLOAT	Y		Y	Extended-precision floating-point numbers.
ExtFloatArray	MONO_ARRAY			Y	Mutable arrays of extended-precision floating-point numbers.
ExtFloatVector	MONO_VECTOR			Y	Immutable vectors of extended-precision floating-point numbers.
ExtMath	MATH				Extended-precision floating-point math library.
Float	FLOAT	Y		Y	Default floating-point structure.
Int*n*	INTEGER	Y		Y	*n*-bit, fixed precision integers
LargeInt	LARGE_INT	Y		?	Arbitrary-precision integers.
POSIX	POSIX			Y	POSIX 1003.1a binding
POSIX.FileSys	POSIX_FILE_SYS			Y	File and directory operations
POSIX.IO	POSIX_IO			Y	Input/output primitives.
POSIX.Process	POSIX_PROC_ENV			Y	Process primitives
POSIX.ProcEnv	POSIX_PROCESS			Y	Process environment primitives
POSIX.SysDB	POSIX_SYS_DB			Y	System database primitives
POSIX.TTY	POSIX_TTY			Y	Terminal device primitives
RealArray	MONO_ARRAY			Y	Mutable arrays of the default real type
RealVector	MONO_VECTOR			Y	Immutable vectors of the default real type
SingleFloat	FLOAT	Y		Y	Single-precision floating-point numbers.
SingleFloatArray	MONO_ARRAY			Y	Mutable arrays of single-precision floating-point numbers.
SingleFloatVector	MONO_VECTOR			Y	Immutable vectors of single-precision floating-point numbers.
SingleMath	MATH				Single-precision floating-point math library.
SmallInt	INTEGER	Y		Y	Fixed-precision integers.
Word	WORD			Y	Unsigned machine integers
Word*n*	WORD			Y	*n*-bit, unsigned machine integers
WordArray	MONO_ARRAY			Y	Mutable arrays of unsigned machine integers
WordVector	MONO_VECTOR			Y	Immutable vectors of unsigned machine integers

Chapter 2

General

We include the definitions of the boolean, list, and ref types here, rather than in separate signatures. This is because we anticipate that libraries will have more complete `Bool`, `List`, and `Ref` structures.

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. The datatype `union` is a variant on the `result` type proposed by Harlequin, but with a more traditional naming scheme.

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 represenations. This chapter presents standard interfaces for integer and real types; the issue of literals is discussed in Chapter 10.

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. The represent a sequence of `wordSize` bits; an unsigned integer; and a machine-dependent encoding of the `SmallInt.int` type. This encoding is likely to be 2's complement, since essentially all current-day computers use this representation.

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 implmentation of the `REAL` signature, which can use infinite-precision or floating-point representations. 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.

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

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

`ExtendedFloat` Higher precision (96 or 128-bit) floating point.

One of these (usually `DoubleFloat`) 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., `SingleFloat`), there is an instance of the `Math` structure (e.g., `SingleMath`). 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 three different kinds of conversions that must be provided: conversions between integer and real representations, conversions between two different integer representations, and conversions between two different floating-point representations.

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

For each pair of float structures F`Float`, G`Float` (e.g., `SingleFloat`, `DoubleFloat`, `ExtendedFloat`), in the system, such that F`Float.precision` $<$ G`Float.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 default the `real` type has more information than the target.]]

3.5 Floating-point arrays

For each floating-point structure *S*`Float`, there may be a monomorphic array struture called *S*`FloatArray` 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 strings are immutable sequences 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` **]]**

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 was 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`.

Bytearrays

The `ByteArray` structure does not fit the general framework described above. It is included for reasons of both compatibility and usefulness.

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

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

Let's discuss the IO structure separately. However, we will propose that the `Io` exception be revised to take a more structured argument:

```
exception Io of {
    ml_op    : string,
    filename : string,
    os_op : string,
    reason   : SysError.syserror
  }
```

`ml_op` is the name of the Standard ML I/O function reporting the exception, e.g. `open_in`.

`filename` is the name of the stream in the file system. Thus, if `output(open_out "fn", s)` fails, the name `fn` will be reported even though it is not directly the argument of `output`.

`os_op` is the name of the operating system call that failed (e.g., `open`).

`reason` is the failure diagnostic reported by the operating system.

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`. Time values are represented by the following concrete datatype:

```
datatype time = TIME of {sec : int, usec : int}
```

We may want to consider going to nanoseconds for the second component, as the Draft POSIX Real-time standard does this.

Time values are 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).

6.5 Misc. stuff

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

Chapter 7

Input/Output

We propose that support for I/O be broken up into three levels: at the lowest level will be OS dependent operations on files and other I/O devices (e.g., sockets). Above this will be the `PrimIO` structure, which defines the SML I/O streams in terms of lower-level abstract *readers* and *writers*. Combining OS dependent implementations of readers and writers with the `PrimIO` structure gives the traditional `SML` I/O interface.

The reader and writer types are parameterized; the idea is that this parameter might specify system dependent information (e.g., the file descriptor for UNIX readers and writers).

[[We should make a distinction between text streams and binary streams.]]

[[Use a "offset" type for seek marks.]]

Chapter 8

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 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

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

Chapter 10

Literals

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

10.1 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.

10.2 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 hexidecimal integer constants (as is already done in the SML/NJ compiler).

10.3 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 arrayoflist : '_a list -> '_a array
val array0      : '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) -> 'a vector
```

DESCRIPTION

The `Array` structure provides 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 $f(i)$.

`arrayoflist` *l*

create an array whose elements are initialized to the elements of l.

`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 $i \geq$ length(a).

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

`extract` (*a*, *i*, *n*)
: extracts the elements $a[i \ldots i+n-1]$ as a vector of length n. This raises `Subscript` if either i, or $i+n-1$ is out of range.

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)

NAME

ByteArray — mutable arrays of 8-bit unsigned integers

SYNOPSIS

signature BYTE_ARRAY

structure ByteArray : BYTE_ARRAY

SIGNATURE

```
eqtype bytearray

exception Range

val maxlen      : int
val array       : (int * int) -> bytearray
val arrayoflist : int list -> bytearray
val tabulate    : (int * (int -> int)) -> bytearray
val length      : bytearray -> int
val extract     : (bytearray * int * int) -> string (* ?? *)
val sub         : (bytearray * int -> int
val update      : (bytearray * int * int) -> unit
```

DESCRIPTION

SEE ALSO

String(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 : chr -> int

val maxCharOrd : int

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

DESCRIPTION

The character type is a dense enumeration running from 0 to `maxCharOrd`, which is an implementation dependent value. For example, an ASCII-based implementation might use `255` for `maxCharOrd`. The mapping between characters and integers is provided by the following two operators:

chr *i*

returns the *i*th character. If $i < 0$ or `maxCharOrd` $< i$, 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_FLOAT — signature of floating-point conversions

SYNOPSIS

signature CONVERT_FLOAT

SIGNATURE

```
eqtype small_real
eqtype large_real

extend : small_real -> large_real
round  : large_real -> small_real
trunc  : large_real -> small_real
floor  : large_real -> small_real
ceil   : large_real -> small_real
```

DESCRIPTION

SEE ALSO

FLOAT(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,
    month : month,
    day : int,          (* 0-31 *)
    hour : int,         (* 0-23 *)
    minute : int,       (* 0-60 *)
    second : int,       (* 0-60 *)
    offset : int,
    zone : string,
    wday : weekday
  }

type timezone

val localTZ : timezone
val univTZ  : timezone

exception Date

val timeToDate : (Time.time * timezone) -> date
val dateToTime : date -> Time.time
val localTime  : Time.time -> date
```

DESCRIPTION

The `offset` field in the `date` type is the difference in seconds between the date and *Universal Coordinated Time* (UTC). This reflects both the difference in time zone, and daylight savings time.

A `timezone` is an abstract representation of a time zone. The variables `localTZ` and `univTZ` represent the local and UTC time zones, respectively. The function `timeToDate` converts a time value to a `date`, as observed in the given time zone. The function `dateToTime` does the conversion the other way, using the `offset` field to convert to UTC. These functions raise the `Date` exception, if the arguments are ill-formed. The `localTime` function does the `timeToDate` conversion, using `localTZ` as the time zone.

SEE ALSO

FmtDate(BASIS), Time(BASIS)

NAME

Float — floating-point arithmetic

SYNOPSIS

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

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

SEE ALSO

Real(BASIS), Math(BASIS)

NAME

FmtDate — Formatting of dates

SYNOPSIS

signature FMT_DATE

structure FmtDate : FMT_DATE

SIGNATURE

```
val dateToStr : Date.date -> string
val formatDate : string -> Date.date -> string
val scanDate : string -> (string * int) -> (Date.date * int)
```

DESCRIPTION

The `dateToStr` function converts a date value to a 25 character string of the form:

```
"Sun Sep 16 01:03:52 1973\n"
```

The `formatDate` and `scanDate` functions provide the function of the ANSI C routines `strftime` and `strptime`.

SEE ALSO

Date(BASIS), Locale(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 Fail of string

datatype bool = true | false
val not : bool -> bool

datatype 'a option = NONE | SOME of 'a

datatype ('a, 'b) union = INL of 'a | INR of 'b

datatype 'a list = nil | :: of ('a * 'a list)

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

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

DESCRIPTION

SEE ALSO

NAME

INTEGER — Generic signature for integer arithmetic types and operations

SYNOPSIS

signature INTEGER

SIGNATURE

```
eqtype int

exception Div
exception Overflow

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

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

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

val toDefault   : int -> Integer.int
val fromDefault : Integer.int -> int

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

DESCRIPTION

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:

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

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

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.

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

MATH — signature of mathematical library functions

SYNOPSIS

signature MATH

SIGNATURE

```
type real

exception Sqrt
exception Ln

val sqrt   : real -> real
val sin    : real -> real
val cos    : real -> real
val arctan : real -> real
val atan2  : (real * real) -> real
val exp    : real -> real
val ln     : 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; except that we have also include the `atan2` function with the following properties:

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

[[ANSI C also defines `tan`, `asin`, `acos`, `sinh`, `cosh`, `tanh`, `log10`, `pow`, `fabs`, `ldexp`, `frexp`, `modf`, and `fmod`. Also constants `pi` and `e` might be useful.]]

SEE ALSO

Real(BASIS), Float(BASIS)

NAME

MONO_ARRAY — generic signature of monomorphic array structures

SYNOPSIS

signature MONO_ARRAY

SIGNATURE

```
eqtype array
type elem
type vector

val maxlen      : int

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

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

DESCRIPTION

This is the generic signature of monomorphic arrays (e.g., ByteArray). The 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. 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* or $i + (n - 1)$ is out of bounds.

SEE ALSO

MONO_VECTOR(BASIS)

NAME

MONO_VECTOR — generic signature of monomorphic vector structures

SYNOPSIS

signature MONO_VECTOR

SIGNATURE

```
type vector
type elem

val maxlen      : int
val vector      : elem list -> vector
val tabulate    : (int * (int -> elem)) -> vector
val length      : vector -> int
val sub         : (vector * int) -> elem
val extract     : (vector * int * int) -> vector
```

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* or $i + (n - 1)$ is out of bounds.

SEE ALSO

MONO_ARRAY(BASIS), Vector(BASIS)

NAME

OS — Generic interface to operating system

SYNOPSIS

signature OS

structure OS : OS

SIGNATURE

```
val osInfo : unit -> {
        archFamily : string,
        archName   : string,
        osName     : string,
        osVersion  : string
      }

type syserror
val errorName : syserror -> string

exception SysErr of {
    ml_op : string,
    os_op : string,
    reason : syserror
  }

structure FileSys : FILE_SYS
structure Path    : PATH
structure Process : PROCESS
```

DESCRIPTION

The function `osInfo` returns information about the host system. The field `archFamily` specifies the processor family; possible values include: `alpha`, `arm`, `68k`, `vax`, `mips`, `sparc`, `power`, `x86`, and `interp`. The value `interp` is reserved for interpreter based implementations. The field `archName` specifies the specific architecture; values include: `mipsel` (little-endian MIPS-1), `mipseb` (big-endian MIPS-1), `mipsel-2` (little-endian MIPS-2), `sparc-7` (SPARC version 7), etc. The `osName` field gives the name of the underlying operating system; values include: `bsd`, `irix`, `sunos`, `solaris` (version 2 and above), `os2`, `macos`, and `windows`.

The type `syserror` represents a system dependent error code; the function `errorName` returns a useful error message from a `syserror`.

The exception `SysErr` is raised by calls to low-level operating system routines.

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 : FILE_SYS
    ...
  end
```

SIGNATURE

```
type dirstream

val open_dir   : string -> dirstream
val read_dir   : dirstream -> string
val rewind_dir : dirstream -> unit
val close_dir  : dirstream -> unit

val chdir      : string -> unit
val getdir     : unit -> string
val make_dir   : string -> unit
val remove_dir : string -> unit
val is_dir     : string -> bool

val modtime : string -> Time.time
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
```

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 name in some system dependent order. The `dirstream` type represents this abstraction; the operations are:

open_dir *path*

opens the specified directory stream.

read_dir *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.

rewind_dir *ds*

rewinds the stream *ds* to the beginning.

close_dir *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.

make_dir *path*

creates the specified directory.

remove_dir *path*

removes the specified directory.

isdir *path*]

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

Several operations are provided on other files:

modtime *path*

remove *path*

rename {*new*, *old*}

access (*path*, *acl*

tests the access permissions associated with the named file.

SEE ALSO

OS(BASIS),Path(BASIS)

NAME

OS.Path — System independent interface to pathnames

SYNOPSIS

```
signature PATH

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

SIGNATURE

```
datatype path_root = REL | ABS of string

exception Path

val explodePath : string
      -> {root : path_root, arcs : string list, last : string}
val implodePath : {root : path_root, arcs : string list, last : string}
      -> string

val parent  : string
val current : string

val isValidPath : string -> bool
val isValidRoot : path_root -> bool
val isValidArc  : string -> bool
val isAbsolute  : string -> bool
val isRelative  : string -> bool

val getParent   : string -> string
val concatPath  : (string * string) -> string
val mkAbsolute  : (string * string) -> string
val mkRelative  : (string * string) -> string
val mkCanonical : string -> string

datatype path_ext = NOEXT | EXT of string

val makePath  : {prefix : string, base : string, ext : path_ext} -> string
val splitPath : string -> {prefix : string, base : string, ext : path_ext}

val root     : string -> string option
val prefix   : string -> string
val last     : string -> string
val base     : string -> string
val lastBase : string -> string
val lastExt  : string -> path_ext
```

DESCRIPTION

This is a system independent module for manipulating strings that represent paths in the directory structure. This structure supports two views of paths. The first is directory oriented, and would typically be used for file-system navigation and searches. The second view focuses on the named file, and would typically be used by applications that generate output file names from input files.

In the first view, a path is abstractly viewed as a sequence of *arcs*, where the first arc specifies the *root* of the path, which is represented by the `path_root` datatype. If the root is `REL`, then the path is said to be *relative*; otherwise it is said to be *absolute*, and the argument to `ABS` specifies the root (e.g., `"/"` on UNIX file systems). The other arcs in the path are represented by strings. The various operations on paths are defined as follows:

`explodePath` *p*

decomposes the path into a list of arcs. For relative paths, the root will be `REL`. If the path is valid, then the roots and arcs will be.

`implodePath` {*root*, *arcs*, *last*}

composes a path from a root, list of arcs and last arc. If the root and arcs are valid, then the path will be valid.

`parent`

is the special arc name that designates the parent directory (e.g., in UNIX this is `".."`).

`current`

is the special arc name that designates the current directory (e.g., in UNIX this is `"."`).

`isValidPath` *p*

returns `true`, if the pathname *p* is a valid pathname for the host operating system.

`isValidRoot` *arc*

returns `true`, if the arc name *arc* is a valid root directory name for the host operating system.

`isValidArc` *arc*

returns `true`, if the arc name *arc* is a valid arc name for the host operating system.

`isAbsolute` *p*

returns `true`, if the pathname *p* is absolute.

`isRelative` *p*

returns `true`, if the pathname *p* is relative.

`getParent` *p*

returns the path of the parent of *p*; if *p* does not have a parent, then the exception `Path` is raised.

`concatPath (`*p1*`,` *p2*`)`

returns the path formed by concatenating *p1* and *p2*. If *p2* is not a relative path, then the exception `Path` is raised.

`mkAbsolute (`*p1*`,` *p2*`)`

returns *p1* if it is absolute; otherwise, it returns an absolute path that is formed by concatenating *p2* and *p1* (i.e., the absolute p corresponding to *p1* with respect to *p2*). If *p2* is not absolute, and even if *p1* is, the exception `Path` is raised.

`mkRelative (`*p1*`,` *p2*`)`

returns *path1* if it is not absolute; otherwise it returns an equivalent path relative to *path2*. If *path2* is not absolute, and even if *path1* is, the exception `Path` is raised.

`mkCanonical` *p*

returns a canonical version of the path *p*. Redundant occurrences of the *current* arc and redundant arc separators are removed. Occurrences of the *parent* arc are folded in, if possible, or else moved to the front of the path. The canonical path will never be the empty string; any empty path is converted to the current directory path. If the path has a trailing arc separator (i.e., the last arc is empty), it is preserved.

The second view of paths divides a path into *prefix*, *base*, and *extension* parts. The prefix specifies the directory that holds the file, and the base and extension comprise the file name.

`makePath {`*prefix*`,` *base*`,` *ext*`}`

creates a path from a prefix, base and extension.

`splitPath` *p*

splits a pathname into prefix, base and extension.

`root` *p*

returns the root of *p*, if it is absolute, or else `NONE`.

`prefix` *p*

returns the prefix of *p* (everything upto the last arc name).

`last` *p*

returns the last arc name in *p*; if *p* consists of only a root arc, then it returns the empty string.

`base` *p*

returns the pathname *p* with itslast arc name replaced by its base.

`lastBase` *p*

is equivalent to `#base(splitPath` *p*`)`.

`lastExt` *p*

is equivalent to `#ext(splitPath` *p*`)`.

SEE ALSO

OS(BASIS)

NAME

OS.Process — System independent interface to process primitives

SYNOPSIS

```
signature PROCESS

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

SIGNATURE

```
val exit : int -> 'a

val system : string -> syserror option
```

DESCRIPTION

SEE ALSO

OS(BASIS)

NAME

POSIX — POSIX 1003.1 binding

SYNOPSIS

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

SIGNATURE

```
datatype syserror
  = E2BIG | EACCES | EAGAIN | EBADF | EBUSY | ECHILD | EDEADLK
  | EDOM | EEXIST | EFAULT | EFBIG | EINTR | EINVAL | EIO
  | EISDIR | EMFILE | EMLINK | ENAMETOOLONG | ENFILE | ENODEV
  | ENOENT | ENOEXEC | ENOLOCK | ENOMEM | ENOSPC | ENOSYS
  | ENOTDIR | ENOTEMPTY | ENOTTY | ENXIO | EPERM | EPIPE
  | ERANGE | EROFS | ESPIPE | ESRCH | EXDEV
  | EOTHER of int

val errorName : syserror -> string

structure Process : POSIX_PROCESS
structure ProcEnv : POSIX_PROC_ENV
structure FileSys : POSIX_FILE_SYS
structure IO      : POSIX_IO
structure TTY     : POSIX_TTY
structure SysDB   : POSIX_SYS_DB
  sharing type Process.pid = ProcEnv.pid = TTY.pid
      and type FileSys.offset = PrimIO.offset
      and type ProcEnv.file_desc = FileSys.file_desc
             = PrimIO.file_desc = TTY.file_desc
      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 1003.1-1990 standard (with some 1003.1a extensions). The datatype `syserror` represents the POSIX system error codes. The constructor `EOTHER` is for error codes not covered by the POSIX standard. The function `errorName` maps an error code to an error message (e.g., `errorName(ENOENT)` might return the string `"No such file or directory"`). The organization of the `POSIX` structure follows that of the standard; each substructure corresponds to a different section in standard.

SEE ALSO

POSIX.Process(BASIS), POSIX.ProcEnv(BASIS), POSIX.FileSys(BASIS), POSIX.PrimIO(BASIS), POSIX.TTY(BASIS), POSIX.SysDB(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

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

eqtype mode
datatype open_mode = O_RDONLY | O_WRONLY | O_RDWR
datatype open_flag
  = O_APPEND
  | O_CREAT of mode
  | O_EXCL
  | O_NOCTTY
  | O_NONBLOCK
  | O_TRUNC

val openf    : (string * open_mode * open_flag list) -> file_desc
val umask    : mode -> mode
val link     : {old : string, new : string} -> unit
val mkdir    : string * mode -> unit
val mkfifo   : string * mode -> unit
```

```
val unlink   : string -> unit
val rmdir    : string -> unit
val rename   : {old : string, new : string} -> unit
val symlink  : {old : string, new : string} -> unit (* POSIX 1003.1a *)
val readlink : string -> string    (* POSIX 1003.1a *)

eqtype dev
eqtype ino
eqtype nlink
eqtype offset

datatype file_type
  = DIR     (* directory *)
  | CHR     (* character special file *)
  | BLK     (* block special file *)
  | REG     (* regular file *)
  | FIFO    (* pipe or fifo file *)
  | LINK    (* symbolic link (POSIX 1003.1a) *)
  | SOCK    (* socket (not POSIX) *)

type stat = {
         ftype : file_type,
         mode  : mode,
         ino   : ino,
         dev   : dev,
         nlink : nlink,
         uid   : uid,
         gid   : gid,
         size  : offset option,
         atime : Time.time,
         mtime : Time.time,
         ctime : Time.time
       }

val stat : string -> stat
val lstat : string -> stat     (* POSIX 1003.1a *)
val fstat : file_desc -> stat

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

val chmod  : (string * mode) -> unit
val chown  : (string * uid * gid) -> unit
val fchown : (file_desc * uid * gid) -> unit (* POSIX 1003.1a *)

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

val pathconf  : (string * string) -> int
val fpathconf : (file_desc * string) -> int
```

DESCRIPTION

These are the operations described in Section 5 of the IEEE Std 1003.1-1990.

SEE ALSO

Posix(BASIS)

NAME

Posix.IO — POSIX compliant interface to primitive I/O operations

SYNOPSIS

```
signature POSIX_IO

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

SIGNATURE

```
eqtype file_desc
eqtype offset

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 read : (file_desc * int) -> string
val readbuf : {
        fd : file_desc, nbytes : int, buf : ByteArray.bytearray, start : int
      } -> int
val write : (file_desc * int * string) -> int
val writebuf : {
        fd : file_desc, nbytes : int, buf : ByteArray.bytearray, start : int
      } -> int

datatype whence = SEEK_SET | SEEK_CUR | SEEK_END

datatype fd_flag = FD_CLOEXEC
datatype file_status = FS_APPEND | FS_NONBLOCK
datatype open_mode = O_RDONLY | O_WRONLY | O_RDWR

val fcntl_DUPFD : old : file_desc, new : file_desc -> unit
val fcntl_GETFD : file_desc -> fd_flag list
val fcntl_SETFD : (file_desc * fd_flag list) -> unit
val fcntl_GETFL : file_desc -> (file_status list * open_mode)
val fcntl_SETFL : (file_desc * file_status list) -> unit

datatype lock_type = F_RDLCK | F_WRLCK | F_UNLCK
type flock = {
         l_type : lock_type,
         l_whence : whence,
         l_start : offset,
         l_len : offset,
         l_pid : pid option
       }
val fcntl_GETLK : (file_desc * flock) -> flock
val fcntl_SETLK : (file_desc * flock) -> flock
val fcntl_SETLKW : (file_desc * flock) -> flock

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

DESCRIPTION

These are the operations described in Section 6 of the IEEE Std 1003.1-1990.

SEE ALSO

Posix(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 uid
eqtype gid
eqtype pid
eqtype file_desc

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 setpgrp : pid : pid option, pgid : pid -> unit

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

val time : unit -> Time.time

val times : unit -> {
        utime : Time.time,    (* user time of process *)
        stime : Time.time,    (* system time of process *)
        cutime : Time.time,   (* user time of terminated child processes *)
        cstime : Time.time    (* system time of terminated child processes *)
      }

val getenv : string -> string option

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

val sysconf : string -> int
```

DESCRIPTION

These are the operations described in Section 4 of the IEEE Std 1003.1-1990.

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 pid

val fork : unit -> pid option

val exec : string * string list -> int
val exece : string * string list * string list -> int
val execp : string * string list -> int

datatype posix_signal
  = SIGABRT | SIGALRM | SIGFPE | SIGHUP | SIGILL | SIGINT | SIGKILL
  | SIGPIPE | SIGQUIT | SIGSEGV | SIGTERM | SIGUSR1 | SIGUSR2
  | SIGCHLD | SIGCONT | SIGSTOP | SIGTSTP | SIGTTIN | SIGTTOU
  | SIGOTHER of int

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

datatype exit_status
  = W_EXITED                  (* Why not W_EXITSTATUS 0? *)
  | W_EXITSTATUS of int
  | W_SIGNALED of posix_signal
  | W_STOPPED of posix_signal

val wait : unit -> pid * exit_status
val waitpid : waitpid_arg * bool -> pid * exit_status
val waitpid_nh : waitpid_arg * bool -> (pid * exit_status) option

val exit : int -> 'a

val kill : pid * posix_signal -> unit

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

DESCRIPTION

These are the operations described in Section 3 of the IEEE Std 1003.1-1990.

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

type passwd = {
         name : string,
         uid : uid,
         gid : gid,
         home_dir : string,
         shell : string
       }
type group = {
         name : string,
         gid : gid,
         members : string list
       }

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

DESCRIPTION

These are the operations described in Section 9 of the IEEE Std 1003.1-1990.

SEE ALSO

Posix(BASIS)

NAME

Posix.Tty — operations on terminal devices

SYNOPSIS

```
signature POSIX_DEVICE

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)

NAME

PrimIO — Primitive input/output operations

SYNOPSIS

signature PRIM_IO

structure PrimIO : PRIM_IO

SIGNATURE

```
exception Io of {
    ml_op    : string,
    filename : string,
    os_op    : string,
    reason   : OS.syserror
  }

datatype 'a wr = Wr of {
    name     : string,
    write    : string -> unit,
    putc     : char -> unit
    seek     : int -> unit,
    index    : unit -> int,
    flush    : unit -> unit,
    close    : unit -> unit,
    closed   : unit -> bool,
    buffered : unit -> bool,
    seekable : unit -> bool,
    ext      : 'a
  }

datatype 'a rd = Rd of {
    name     : string,
    read     : int -> string,
    getc     : unit -> char option,
    peek     : unit -> char option,
    avail    : unit -> int,
    seek     : int -> unit,
    index    : unit -> int,
    close    : unit -> unit,
    eof      : unit -> bool,
    closed   : unit -> bool,
    buffered : unit -> bool,
    seekable : unit -> bool,
    ext      : 'a
  }

type instream
type outstream
```

```
val mkInstream  : 'a rd -> instream
val mkOutstream : 'a wr -> outstream

val mkReader : instream -> unit rd
val mkWriter : outstream -> unit wr

val close_in      : instream -> unit
val input         : (instream * int) -> string
val inputc        : instream -> int -> string
val input_line    : instream -> string
val lookahead     : instream -> string
val end_of_stream : instream -> string
val clear_eof     : instream -> unit

val close_out : outstream -> unit
val output    : (outstream * string) -> unit
val outputc   : outstream -> string -> unit
val flush_out : outstream -> unit
```

DESCRIPTION

SEE ALSO

IO(BASIS)

NAME

Real — generic interface to real arithmetic

SYNOPSIS

signature REAL

structure Real : REAL

SIGNATURE

```
type real

exception Div
exception Overflow

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

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
```

DESCRIPTION

SEE ALSO

Math(BASIS)

NAME

String — basic operations on strings

SYNOPSIS

signature STRING

structure String : STRING

SIGNATURE

```
eqtype string

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

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

DESCRIPTION

Strings are finite sequences of characters.

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 ?? is raised.

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

returns an *n* character substring starting at the *i*th character of *s*. (exceptions ???)

isSubstring (*s1***,** *i***,** *s2***)**

returns true if *s1* is a substring of *s2* starting at position *i*.

concat *sl*

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

*s1*ˆ*s2*

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.

SEE ALSO

Char(BASIS)

NAME

Time — Representation of time values

SYNOPSIS

signature TIME

structure Time : TIME

SIGNATURE

```
datatype time = TIME of {sec : Integer.int, usec : Integer.int}

val addTime : (time * time) -> time
val subTime : (time * time) -> time
val earlier : (time * time) -> bool

val timeOfDay : unit -> time
```

DESCRIPTION

SEE ALSO

Timer(BASIS)

CAVEATS

We may want to support nano-second granularity.

NAME

Timer — Interface to system timer

SYNOPSIS

signature TIMER

structure Timer : TIMER

SIGNATURE

```
type timer

val timer0 : timer
val startTimer : unit -> timer
val checkTimer : timer -> {usr : time, sys : time, gc : time}
```

DESCRIPTION

timer0

is a timer started at system start-up.

startTimer ()

starts a new timer.

checkTimer *timer*

returns the current values of a timer.

SEE ALSO

Time(BASIS)

CAVEATS

NAME

Vector — immutable polymorphic vectors

SYNOPSIS

signature VECTOR

structure Vector : VECTOR

SIGNATURE

```
eqtype 'a vector

val maxlen   : int

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

val extract  : ('a vector * int * int) -> 'a vector
val length   : 'a vector -> int
val sub      : ('a vector * int) -> 'a
```

DESCRIPTION

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

`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* or $i + (n - 1)$ is out of bounds.

SEE ALSO

Array(BASIS), MONO_VECTOR(BASIS)

NAME

Word — unsigned machine integers

SYNOPSIS

signature WORD
structure Word : WORD

SIGNATURE

```
eqtype word

val wordSize  : int

val wordToInt : word -> int
val intToWord : int -> word

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

val lshift : word * int -> word
val rshift : word * int -> word

val alshift : word * int -> word
val arshift : word * int -> word

val plus : word * word -> word
val minus : word * word -> word
val times : word * word -> word
val divide : word * word -> word
val mod : word * word -> word
```

DESCRIPTION

The `word` type represents a sequence of `wordSize` bits, indexed from least significant to most significant. Words can also be viewed as a machine dependent encoding of finite precision integers (e.g., 2's complement on most machines). 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)
```

wordToInt *w*

returns the integer that the word encodes.

intToWord *i*

returns the word that encodes the given integer value. If the `int` type is arbitrary precision, then this may cause the `Overflow` exception to be raised.

orb (*w1***,** *w2***)**

returns the bitwise or of *w1* and *w2*.

xorb (*w1***,** *w2***)**

returns the bitwise exclusive-or of *w1* and *w2*.

andb (*w1***,** *w2***)**

returns the bitwise and of *w1* and *w2*.

notb *w*

returns the bitwise complement of *w*.

plus (*w1***,** *w2***)**

returns $(w1 + w2) \bmod 2^{\texttt{wordSize}}$.

minus (*w1***,** *w2***)**

returns $(w1 - w2) \bmod 2^{\texttt{wordSize}}$.

times (*w1***,** *w2***)**

returns $(w1 \times w2) \bmod 2^{\texttt{wordSize}}$.

divide (*w1***,** *w2***)**

returns $\left\lfloor \frac{w1}{w2} \right\rfloor$. Raises the `Div` exception if `w2` is `0`.

mod (*w1***,** *w2***)**

returns $w1 - w2 \times \left\lfloor \frac{w1}{w2} \right\rfloor$. Raises the `Div` exception if `w2` is `0`.

[[shift operations should have Modula-3 semantics]]

SEE ALSO

Int(BASIS), SmallInt(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.