

A New Initial Basis for Standard ML
(DRAFT ~~12~~ DO NOT DISTRIBUTE)

June 26, 1995

Contents

Preface	v
I Discussion	1
1 Introduction	3
1.1 Conventions and design philosophy	4
1.2 Overview	5
1.3 Things to discuss	6
1.4 Known incompatibilities with the Definition	6
2 General	10
3 Arithmetic types	11
3.1 Integers	11
3.2 Words	11
3.3 Real numbers	12
3.4 Conversions	13
3.5 Floating-point arrays	13
4 Text	14
5 Aggregates	15
5.1 Vectors	15
5.2 Arrays	15
5.3 Monomorphic aggregates	16
5.4 Lists	16

6	System interface	18
6.1	Operating system interface	18
6.2	Locale	18
6.3	Directories and paths	19
6.4	Time	19
6.5	Misc. stuff	19
7	UNIX interface	20
8	The <code>module</code> structure	21
9	The top-level environment	22
9.1	Pre-loaded modules	22
9.2	Top-level type, exception and value identifiers	22
9.3	Index identifiers	23
9.4	Overloaded identifiers	24
10	Language issues	25
10.1	Overloading	25
10.2	Literals	25
10.3	Character literals	25
10.4	Numeric literals	26
10.5	Vector literals	27
II	Manual pages	29
	Array	31
	Bool	34
	Byte	35
	Char	36
	CONVERT_INT	38
	CONVERT_REAL	39
	CONVERT_WORD	40
	Date	41
	Float	42

General	43
INTEGER	45
LargeInt	48
List	49
ListPair	53
Locale	54
MATH	56
MONO_ARRAY	58
MONO_VECTOR	61
OS	63
OS.FileSys	64
OS.Path	67
OS.Process	69
PACK_WORD	71
Real	73
String	75
StringCvt	77
Substring	79
Time	82
Timer	85
Vector	86
Word	88

III Amendment: POSIX 1003.1b-1993	91
POSIX	93
Posix.Error	94
POSIX_FLAGS	96
Posix.FileSys	97
Posix.IO	101
Posix.ProcEnv	104
Posix.Process	106
Posix.Signal	108
Posix.SysDB	110

Posix.Tty	111
---------------------	-----

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., `list`, `vector`, 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 Huelshbergen	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.
- Amendments for operating specific APIs.

Table 1.1: List of required generic signatures

Signature	Description
<code>int2int</code>	Conversions between two integer representations.
<code>real2real</code>	Conversions between two real representations.
<code>uint2uint</code>	Conversions between two unsigned representations.
<code>IEEEFP</code>	Generic IEEE floating-point module interface
<code>intmod</code>	Generic integer module interface.
<code>mathlib</code>	Generic math library interface.
<code>monomorph</code>	Mutable monomorphic arrays.
<code>monomorphVec</code>	Immutable monomorphic vectors.
<code>OS</code>	Generic interface to basic operating system features
<code>realnum</code>	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:

[[Are the `Vec` and `Vec2` structures aliases, or abstract?]]

1.3 Things to discuss

Packing/unpacking values

[illegible]

1.4 Known incompatibilities with the Definition

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

- The types of `int` and `long`.
- 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
<code>mutable</code>	<code>mutable</code>	Mutable polymorphic arrays.
<code>input</code>	<code>input</code>	Binary input/output streams and operations.
<code>output</code>	<code>output</code>	
<code>bool</code>	<code>bool</code>	Operations on booleans.
<code>word8</code>	<code>word8</code>	Conversions between Word8 and Char
<code>char</code>	<code>char</code>	Characters
<code>mutableChar</code>	<code>mutableChar</code>	Mutable arrays of characters
<code>immutableChar</code>	<code>immutableChar</code>	Immutable vectors of characters
<code>calendar</code>	<code>calendar</code>	Calendar operations
<code>types</code>	<code>types</code>	General-purpose types, exceptions and miscellaneous operations.
<code>interger</code>	<code>interger</code>	Default interger structure.
<code>lists</code>	<code>lists</code>	Utility functions on lists.
<code>lists2</code>	<code>lists2</code>	Utility functions on pairs of lists.
<code>localization</code>	<code>localization</code>	Support for localization.
<code>math</code>	<code>math</code>	Default math structure.
<code>os</code>	<code>os</code>	Basic operating system services.
<code>file</code>	<code>file</code>	File status and directory operations
<code>pathname</code>	<code>pathname</code>	Pathname operations
<code>process</code>	<code>process</code>	Simple process manipulation operations
<code>real</code>	<code>real</code>	Default real structure.
<code>strings</code>	<code>strings</code>	Utility functions on strings (cf., <code>strings</code> and <code>strings2</code>).
<code>string</code>	<code>string</code>	Basic string conversions.
<code>pieces</code>	<code>pieces</code>	Utility functions on pieces of strings.
<code>text</code>	<code>text</code>	Text input/output streams and operations.
<code>time</code>	<code>time</code>	Representation of time values
<code>timing</code>	<code>timing</code>	Timing operations
<code>immutable</code>	<code>immutable</code>	Immutable polymorphic vectors.
<code>int8</code>	<code>int8</code>	8-bit unsigned integers
<code>int8Array</code>	<code>int8Array</code>	Arrays of 8-bit unsigned integers
<code>int8Vector</code>	<code>int8Vector</code>	Vectors of 8-bit unsigned integers

Table 1.3: List of optional structures

Module	Signature	Description
<code>bool *</code>	<code>bool *</code>	Mutable arrays of booleans
<code>bool *</code>	<code>bool *</code>	Immutable vectors of booleans
<code>double</code>	<code>double</code>	Default floating-point structure.
<code>double *</code>	<code>double *</code>	Mutable arrays of default floating-point numbers.
<code>double *</code>	<code>double *</code>	Default floating-point math library.
<code>double *</code>	<code>double *</code>	Immutable vectors of default floating-point numbers.
<code>double *</code>	<code>double *</code>	Floating-point numbers (n -bits, for <code>##</code> <code>int</code> <code>[32, 64, 96, 128]</code>).
<code>double *</code>	<code>double *</code>	Mutable arrays of floating-point numbers (n -bit floats, <code>##</code> <code>int</code> <code>[32, 64, 96, 128]</code>).
<code>double *</code>	<code>double *</code>	Floating-point math library (n -bit floats, <code>##</code> <code>int</code> <code>[32, 64, 96, 128]</code>).
<code>double *</code>	<code>double *</code>	Immutable vectors of floating-point numbers (n -bit floats, <code>##</code> <code>int</code> <code>[32, 64, 96, 128]</code>).
<code>int *</code>	<code>int *</code>	n -bit, fixed precision integers
<code>int *</code>	<code>int *</code>	Arbitrary-precision integers.
<code>int *</code>	<code>int *</code>	POSIX 1003.1a binding
<code>int *</code>	<code>int *</code>	File and directory operations
<code>int *</code>	<code>int *</code>	Input/output primitives.
<code>int *</code>	<code>int *</code>	Process primitives
<code>int *</code>	<code>int *</code>	Process environment primitives
<code>int *</code>	<code>int *</code>	System database primitives
<code>int *</code>	<code>int *</code>	Terminal device primitives
<code>int *</code>	<code>int *</code>	Fixed-precision integers.
<code>int *</code>	<code>int *</code>	Unsigned machine integers
<code>int *</code>	<code>int *</code>	n -bit, unsigned machine integers
<code>int *</code>	<code>int *</code>	Mutable arrays of unsigned machine integers
<code>int *</code>	<code>int *</code>	Mutable arrays of <code>##</code> -bit unsigned machine integers
<code>int *</code>	<code>int *</code>	Immutable vectors of unsigned machine integers
<code>int *</code>	<code>int *</code>	Immutable vectors of <code>##</code> -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 `ref` because it has a strange equality property that can't be written down in a signature.

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

A number of common exceptions (`InvalidArgument`, `IOException`, `InterruptedException` and `FileNotFoundException`) are defined in `IOException`. These are the standard exceptions used by various modules to signal error conditions.

We include the exception `InterruptedException`, 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 `InterruptedException` 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 `LargeInteger` and `SmallInteger` matching the signature. Finally, a structure `Int` will be bound to either `LargeInteger` or `SmallInteger` 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 `Intn`, where n is the number of bits of precision (e.g., `Int32`).]]

3.2 Words

Words are an abstraction of the underlying hardware machine word. They represent a sequence of `WordSize` bits; an unsigned integer; and a machine-dependent encoding of the `WordSize` .

type.

The `float` 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 `floatn`, where n is the number of bits of precision (e.g., `float64`).]]

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 NaNs;
- IEEE-754 floating point in several sizes, with infinities and NaNs;
- 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 `float` signature, which can use infinite-precision or floating-point representations.

The (optional) structure `ConstructiveReals` (possibly the same structure as `float`) 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:

- `Short` Short precision (less than 32-bit) floating-point numbers represented as unboxed values to save time and space at the expense of accuracy.
- `Single` Single precision (32-bit) floating point.
- `Double` Double precision (64-bit) floating point.
- `Higher` Higher precision (96 or 128-bit) floating point.

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

The standard mathematical functions (e.g., `sin`, `cos`, etc.) are found in the `math` structure. For each different representation of reals (e.g., `float`, `double`, `long double`), there is an instance of the `math` structure (e.g., `math_float`, `math_double`). 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 (`int` → `long` N `int` M).
- conversions between different sizes of words (`int` → `long` N `int` M).
- conversions between different sizes of floating-point numbers (`float` → `double` N `float` M).
- conversions floating point numbers and integers (`float` → `int` N `float` M).
- conversions between words and integers (`int` → `long` N `int` M).

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

For each pair of float structures `float`, `double` (e.g., `float`, `double`, `long double`, `float`), in the system, such that `float` → `double` is a conversion, there must also be a structure `conv_float_double` matching the signature `conv_float_double`.

[[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., `round`) when the default `double` 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 structure called `float` N `array` that matches the `conv_float_double` signature.

Chapter 4

Text

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

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

We propose that the single `char` type provided by the Definition be replaced with two types: `char8_t` and `char32_t`, where the `char8_t` type is a *vector* of characters.

[[we need to think about Unicode]]

[[There should be a `string` structure with `string::find`, `string::match`, `string::substr`. `string::find` and `string::match`. We may want to add `string::find` to `string`]]

String conversions

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

```
template<typename T, typename Char, typename Traits>  
string to_string(T const& t, Char const* p, Traits const& traits)  
{  
    return string(p, traits.to_string(t, p));  
}
```

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 structure defines the integer variable `maxLength`, 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.of` takes a list of elements and makes a vector out of it, and `Vector.fill` 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 `Vector.length`), get an element (using `Vector.get`), or extract a sub-vector (using `Vector.slice`).

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.fill` operation creates an array initialized to a given value, while the `Array.ofElements` operation is used to make an array from a list. An array value can be modified using the `Array.set` operation, which replaces a given element with another value. Lastly, the `Array.toVector` operation returns a vector of the corresponding vector type.

5.3 Monomorphic aggregates

An implementation may choose to provide various implementations of the `array` and `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 `vector` implementation might use one bit per element).

Character vectors

The `string` structure defines a view of the `array` structure that matches to the `string` signature. The type `string` is the same as `array`.

Byte arrays and vectors

The `array` structure provides functions to extract strings from monomorphic arrays and vectors of `char`s. In addition, these types support additional operations for packing and unpacking larger sizes of words. These can be found in the `array` and `vector` 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:
 - ½ Proven useful
 - ½ Complicated to implement, or significantly more concise or more efficient than an equivalent combination of the other list functions.
- No gratuitous name changes.

¹Since the `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 `sys` that contains all of the operating system related interfaces. At a minimum, this structure must match the `sys` 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, formatting 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 `libpath` structure provides operations for navigating the directory hierarchy, for listing the files in a directory, and some operations on files. The `libpath` structure provides an abstract, system independent, view of pathnames.

6.4 Time

We propose three structures to support access to timing and dates: `libtime`, `libdate` and `libtimer`.

The abstract type `time_t` 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 `libtime` structure provides mechanisms to convert between the `time_t` type and various concrete representations. The `libdate` 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 `libtimer` structure provides timers for measuring both CPU and wall-clock times.

6.5 Misc. stuff

```

/*
 *  libmisc.h
 *  misc.h
 *  misc.c
 *  misc.h
 *  misc.c
 */

```

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 `PosixIO` 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, process 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:

```
let struct Old in
  let struct Old in
    let struct Old in
      user/program
    end
  end
end
```

to compile a `user/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?]]

```

-- | The top-level environment.
-- |
-- | The top-level environment is the environment in which the
-- | top-level expressions are evaluated. It contains the
-- | pre-loaded modules, the identifiers made available
-- | without qualification, the infix identifiers, and the
-- | overloading.
-- |
-- | The top-level environment is defined by the following
-- | equations:
-- |
-- |   topLevelEnv = let
-- |     in {
-- |       -- The pre-loaded modules.
-- |       preLoadedModules = ...
-- |       -- The identifiers made available without qualification.
-- |       identifiers = ...
-- |       -- The infix identifiers.
-- |       infixIdentifiers = ...
-- |       -- The overloading.
-- |       overloading = ...
-- |     }
-- |
-- | The top-level environment is used by the compiler to
-- | evaluate the top-level expressions. It is also used by
-- | the compiler to generate the code for the top-level
-- | expressions.
```

[illegible]
$$\frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} f(s) ds = \frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} g(s) ds + \frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} h(s) ds$$

Γ_1 Γ_2 Γ_3 Γ_4 Γ_5 Γ_6 Γ_7 Γ_8 Γ_9 Γ_{10} Γ_{11} Γ_{12} Γ_{13} Γ_{14} Γ_{15} Γ_{16} Γ_{17} Γ_{18} Γ_{19} Γ_{20} Γ_{21} Γ_{22} Γ_{23} Γ_{24} Γ_{25} Γ_{26} Γ_{27} Γ_{28} Γ_{29} Γ_{30} Γ_{31} Γ_{32} Γ_{33} Γ_{34} Γ_{35} Γ_{36} Γ_{37} Γ_{38} Γ_{39} Γ_{40} Γ_{41} Γ_{42} Γ_{43} Γ_{44} Γ_{45} Γ_{46} Γ_{47} Γ_{48} Γ_{49} Γ_{50} Γ_{51} Γ_{52} Γ_{53} Γ_{54} Γ_{55} Γ_{56} Γ_{57} Γ_{58} Γ_{59} Γ_{60} Γ_{61} Γ_{62} Γ_{63} Γ_{64} Γ_{65} Γ_{66} Γ_{67} Γ_{68} Γ_{69} Γ_{70} Γ_{71} Γ_{72} Γ_{73} Γ_{74} Γ_{75} Γ_{76} Γ_{77} Γ_{78} Γ_{79} Γ_{80} Γ_{81} Γ_{82} Γ_{83} Γ_{84} Γ_{85} Γ_{86} Γ_{87} Γ_{88} Γ_{89} Γ_{90} Γ_{91} Γ_{92} Γ_{93} Γ_{94} Γ_{95} Γ_{96} Γ_{97} Γ_{98} Γ_{99} Γ_{100}

၇၇၇ နေ့မှစ၍ ၇၈၀ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၇၈ နေ့မှစ၍ ၇၈၁ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၇၉ နေ့မှစ၍ ၇၈၂ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၀ နေ့မှစ၍ ၇၈၃ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၁ နေ့မှစ၍ ၇၈၄ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၂ နေ့မှစ၍ ၇၈၅ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၃ နေ့မှစ၍ ၇၈၆ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၄ နေ့မှစ၍ ၇၈၇ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်
 ၇၈၅ နေ့မှစ၍ ၇၉၀ နေ့အထိ နေ့စဉ် နေ့စဉ် နေ့စဉ်

[illegible]

The top-level environment has the following identifiers:

Draft of June 26, 1995 7:03

The following symbols are overloaded:

一
二
三
四
五
六
七
八
九
十
十一
十二
十三
十四
十五
十六
十七
十八
十九
二十
二十一
二十二
二十三
二十四
二十五
二十六
二十七
二十八
二十九
三十
三十一
三十二
三十三
三十四
三十五
三十六
三十七
三十八
三十九
四十
四十一
四十二
四十三
四十四
四十五
四十六
四十七
四十八
四十九
五十
五十一
五十二
五十三
五十四
五十五
五十六
五十七
五十八
五十九
六十
六十一
六十二
六十三
六十四
六十五
六十六
六十七
六十八
六十九
七十
七十一
七十二
七十三
七十四
七十五
七十六
七十七
七十八
七十九
八十
八十一
八十二
八十三
八十四
八十五
八十六
八十七
八十八
八十九
九十
九十一
九十二
九十三
九十四
九十五
九十六
九十七
九十八
九十九
一百

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

$\text{\texttt{\char"n}}$

where $\text{\texttt{\char"n}}$ 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 $\text{\texttt{\char"n}}$ where n is a non-negative integer literal, be recognized. Also, we will need syntax for Unicode strings.

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

`[<e> ... <e>]`

and can be used in both expressions and patterns.

Part II

Manual pages

NAME

Array ~~is~~ polymorphic mutable arrays

SYNOPSIS

signature ARRAY

structure Array : ARRAY

SIGNATURE

```

type 'a array = 'a list
type 'a array = 'a list
type 'a array = 'a list

val m : int -> 'a array -> 'a array

val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array

val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array

val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array

val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array

val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array
val array : 'a array -> 'a array -> 'a array -> 'a array -> 'a array -> 'a array

```

DESCRIPTION

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

`max_array_len`

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 `ValueError` if $n \leq 0$ or if v is not mutable.

`array(n, f)`

create an n element array whose i th element is initialized to `f(i)`. The function f is called in increasing order of i . Raises `ValueError` if $n \leq 0$ or if f is not callable.

`array(list, l)`

create an array whose elements are initialized to the elements of `l`. Raises `ValueError` if the list has more than 256 elements.

`array()`

is the unique zero-length array.

`len(arr)`

the number of elements in the array `arr`.

`arr[i]`

extracts (subscript) the i th element of array `arr`. Raises `ValueError` if $i \leq 0$ or $i \geq \text{length}(arr)$.

`arr[i, v]`

replaces the i th element of `arr` by the value v . Raises `ValueError` if $i \leq 0$ or $i \geq \text{length}(arr)$.

`arr[i, n]`

extracts the elements `arr[i : i + n]` as a vector of length n . The exception `ValueError` is raised if $i \leq 0$ or $i + n > \text{length}(arr)$.

`array(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 `ValueError` is raised if $len < 0$, or if either $si \leq 0$ or $si + len > \text{length}(src)$, or $di \leq 0$ or $di + len > \text{length}(dst)$.

More precisely, let s and d be the contents of `src` and `dst` immediately prior to the call to `array`. Then upon successful completion of the call, for $0 \leq i < len$:

$$d[di + i] = s[si + i] \quad \text{if } si + i < \text{length}(src) \\ \text{otherwise}$$

Moreover, if `src` and `dst` are different arrays, then for $0 \leq i < len$: $d[di + i]$ is the same object as $s[si + i]$.

`array (src, si, len, dst, di)`

is like `array`, except that *src* is a vector.

Note that type `array` is an equality type even if `src` is not. Thus, the `array` 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 `arr` and `arr2`, created by different calls to `array`, are always unequal, even if their elements are equal.

SEE ALSO

Vector(BASIS), MONO_ARRAY(BASIS)

BOOL(BASIS)

Initial Basis

BOOL(BASIS)

NAME

Bool operations on booleans

SYNOPSIS

signature BOOL

structure Bool : BOOL

SIGNATURE

$\text{val } \text{and} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ | $\text{val } \text{or} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$

$\text{val } \text{not} : \text{bool} \rightarrow \text{bool}$ | $\text{val } \text{xor} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$

$\text{val } \text{if_then_else} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ | $\text{val } \text{if_then} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$

$\text{val } \text{if_then_else_if_then} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ | $\text{val } \text{if_then_else_if} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$

$\text{val } \text{if_then_else_if_then_else} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ | $\text{val } \text{if_then_else_if_then_else_if} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$

DESCRIPTION

BYTE(BASIS)

Initial Basis

BYTE(BASIS)

NAME

Byte ~~is~~ unsigned 8-bit integers

SYNOPSIS

signature BYTE

structure Byte : BYTE

SIGNATURE

```

with private use of unsigned_integer;
with use of unsigned_integer;

type Byte is unsigned_integer(0..255);

with use of unsigned_integer;
with use of unsigned_integer;

type Byte is unsigned_integer(0..255);

with use of unsigned_integer;
with use of unsigned_integer;

type Byte is unsigned_integer(0..255);

with use of unsigned_integer;
with use of unsigned_integer;

type Byte is unsigned_integer(0..255);
```

DESCRIPTION

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

The function `to_unsigned_integer` 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, `to_unsigned_integer` will raise the `exception`.

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

SEE ALSO

WORD(BASIS)

NAME

Initial Basis

CHAR(BASIS)

Char 1/2 character type and operations

SYNOPSIS

signature CHAR

```
structure Char : CHAR
```

open Char

SIGNATURE

[illegible]

DESCRIPTION

The character type is a dense enumeration running from `CHAR_MIN` to `CHAR_MAX`. We require that `CHAR_MIN` be 0, and that `CHAR_MAX` be `CHAR_BIT - 1`. The actual value of `CHAR_MAX` is implementation dependent. For example, an ASCII-based implementation might use `255` for `CHAR_MAX`. The mapping between characters and integers is provided by the following two operators:

`char c` \leftarrow `CHAR_BIT` `i`

returns the *i*th character. If *i* < 0 or *i* > `CHAR_BIT - 1`, then the exception `RangeError` is raised.

`int i` \leftarrow `CHAR_BIT` `c`

returns the integer representation of the character. It should be the case that `CHAR_BIT` `CHAR_BIT` `c` = `c`, for all characters *c*.

The relational operators on characters are defined by:

$$c_1 < c_2 \iff f(c_1) < f(c_2), \quad c_1 < c_2 \iff f(c_1) < f(c_2), \quad c_1 < c_2 \iff f(c_1) < f(c_2)$$

where *f* is one of `CHAR_BIT`, `CHAR_BIT`, or `CHAR_BIT`.

SEE ALSO

String(BASIS)

CONVERT-INT(BASIS)

Initial Basis

CONVERT-INT(BASIS)

NAME

CONVERT_INT % conversions between integer types

SYNOPSIS

signature CONVERT_INT

SIGNATURE

```

// int32_t <-> int
// int32_t <-> int64_t

int <-> int32_t ; int64_t -> int32_t
int <-> int64_t ; int32_t -> int64_t
```

DESCRIPTION

SEE ALSO

INTEGER (BASIS)

ConvertReal(BASIS)

Initial Basis

ConvertReal(BASIS)

NAME

CONVERT_REAL is signature of floating-point conversions

SYNOPSIS

signature CONVERT_REAL

structure Cvt.FloatNFloatM : CONVERT_REAL

SIGNATURE

```

signature CONVERT_REAL =
  struct
    val N : real -> real
    val M : real -> real
    val ... : ...
  end

```

DESCRIPTION

This interface needs revision, but I'm not sure what the current proposal is.

SEE ALSO

FLOAT(BASIS)

ConvertWord(BASIS)	Initial Basis	ConvertWord(BASIS)
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10
11	11	11
12	12	12
13	13	13
14	14	14
15	15	15
16	16	16
17	17	17
18	18	18
19	19	19
20	20	20
21	21	21
22	22	22
23	23	23
24	24	24
25	25	25
26	26	26
27	27	27
28	28	28
29	29	29
30	30	30
31	31	31
32	32	32
33	33	33
34	34	34
35	35	35
36	36	36
37	37	37
38	38	38
39	39	39
40	40	40
41	41	41
42	42	42
43	43	43
44	44	44
45	45	45
46	46	46
47	47	47
48	48	48
49	49	49
50	50	50
51	51	51
52	52	52
53	53	53
54	54	54
55	55	55
56	56	56
57	57	57
58	58	58
59	59	59
60	60	60
61	61	61
62	62	62
63	63	63
64	64	64
65	65	65
66	66	66
67	67	67
68	68	68
69	69	69
70	70	70
71	71	71
72	72	72
73	73	73
74	74	74
75	75	75
76	76	76
77	77	77
78	78	78
79	79	79
80	80	80
81	81	81
82	82	82
83	83	83
84	84	84
85	85	85
86	86	86
87	87	87
88	88	88
89	89	89
90	90	90
91	91	91
92	92	92
93	93	93
94	94	94
95	95	95
96	96	96
97	97	97
98	98	98
99	99	99
100	100	100

Initial Basis

ConvertWord(BASIS)

NAME

CONVERT_WORD is signature of unsigned integer conversions

SYNOPSIS

signature CONVERT_WORD

SIGNATURE

[illegible]

DESCRIPTION

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

$$\begin{array}{l} \text{w} \\ w \\ W \\ n \end{array}$$

SEE ALSO

WORD(BASIS)

DATE(BASIS)

Initial Basis

DATE(BASIS)

NAME

Date %interface to local time and date information

SYNOPSIS

signature DATE

structure Date : DATE

SIGNATURE

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

type Date = { year : int; month : int; day : int; hour : int; min : int; sec : int; usec : int; basis : int; }

DESCRIPTION

This interfaces follows the ANSI C semantics. The `date` operation defines a lexical ordering using the `year`, `month`, `day`, `hour`, `min`, `sec`, and `usec` fields. The other fields are ignored.

SEE ALSO

FmtDate(BASIS), Time(BASIS)

FLOAT(BASIS)

Float 1/2 Coating-point arithmetic

structure Float^{float} : FLOAT *etc.*

החלטת המועצה המקומית להקים את המועצה הירוקה, תכונה כהחלטת ממשל, ויש להבחין בין החלטות ממשל לבין החלטות ממשלה. החלטות ממשל אינן מחייבות את הממשלה, ואילו החלטות ממשלה מחייבות את הממשלה. החלטות ממשל אינן מחייבות את הממשלה, ואילו החלטות ממשלה מחייבות את הממשלה.

[[If we assume IEEE representations, then do we need $\text{round}()$?]]

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

Real(BASIS), Math(BASIS)

NAME

General Basic definitions used in the pervasive environment

SYNOPSIS

signature GENERAL

structure General : GENERAL

open General

SIGNATURE

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

module type

NAME _____

INTEGER 16 Generic signature for integer arithmetic types and operations

SYNOPSIS

signature INTEGER

```
structure Integer : INTEGER
```

```
structure SmallInt : INTEGER (optional)
```

```
structure LargeInt : LARGE_INT (optional)
```

```
structure Int32 : INTEGER etc.
```

SIGNATURE

[illegible]

[illegible][illegible]

The musical score for 'The Rose Tree' is presented in two systems. The first system consists of a vocal line and a piano accompaniment. The vocal line begins with a treble clef and a key signature of one flat (B-flat). The piano accompaniment starts with a bass clef and a key signature of one flat. The second system continues the vocal line and piano accompaniment. The vocal line features a series of eighth and sixteenth notes, while the piano accompaniment provides a harmonic foundation with chords and moving lines in both hands. The score is written in a standard musical notation style, with notes, rests, and bar lines clearly visible.

$$2\pi \int_0^1 \frac{1}{\sqrt{1-x^2}} dx = 2\pi \left[\arcsin x \right]_0^1 = 2\pi \left(\frac{\pi}{2} - 0 \right) = \pi^2$$

42 22 43 16 44 17 45 18 46 19 47 20 48 21 49 22 50 23 51 24 52 25 53 26 54 27 55 28 56 29 57 30 58 31 59 32 60 33 61 34 62 35 63 36 64 37 65 38 66 39 67 40 68 41 69 42 70 43 71 44 72 45 73 46 74 47 75 48 76 49 77 50 78 51 79 52 80 53 81 54 82 55 83 56 84 57 85 58 86 59 87 60 88 61 89 62 90 63 91 64 92 65 93 66 94 67 95 68 96 69 97 70 98 71 99 72 100 73 101 74 102 75 103 76 104 77 105 78 106 79 107 80 108 81 109 82 110 83 111 84 112 85 113 86 114 87 115 88 116 89 117 90 118 91 119 92 120 93 121 94 122 95 123 96 124 97 125 98 126 99 127 100 128 101 129 102 130 103 131 104 132 105 133 106 134 107 135 108 136 109 137 110 138 111 139 112 140 113 141 114 142 115 143 116 144 117 145 118 146 119 147 120 148 121 149 122 150 123 151 124 152 125 153 126 154 127 155 128 156 129 157 130 158 131 159 132 160 133 161 134 162 135 163 136 164 137 165 138 166 139 167 140 168 141 169 142 170 143 171 144 172 145 173 146 174 147 175 148 176 149 177 150 178 151 179 152 180 153 181 154 182 155 183 156 184 157 185 158 186 159 187 160 188 161 189 162 190 163 191 164 192 165 193 166 194 167 195 168 196 169 197 170 198 171 199 172 200 173 201 174 202 175 203 176 204 177 205 178 206 179 207 180 208 181 209 182 210 183 211 184 212 185 213 186 214 187 215 188 216 189 217 190 218 191 219 192 220 193 221 194 222 195 223 196 224 197 225 198 226 199 227 200 228 201 229 202 230 203 231 204 232 205 233 206 234 207 235 208 236 209 237 210 238 211 239 212 240 213 241 214 242 215 243 216 244 217 245 218 246 219 247 220 248 221 249 222 250 223 251 224 252 225 253 226 254 227 255 228 256 229 257 230 258 231 259 232 260 233 261 234 262 235 263 236 264 237 265 238 266 239 267 240 268 241 269 242 270 243 271 244 272 245 273 246 274 247 275 248 276 249 277 250 278 251 279 252 280 253 281 254 282 255 283 256 284 257 285 258 286 259 287 260 288 261 289 262 290 263 291 264 292 265 293 266 294 267 295 268 296 269 297 270 298 271 299 272 300 273 301 274 302 275 303 276 304 277 305 278 306 279 307 280 308 281 309 282 310 283 311 284 312 285 313 286 314 287 315 288 316 289 317 290 318 291 319 292 320 293 321 294 322 295 323 296 324 297 325 298 326 299 327 300 328 301 329 302 330 303 331 304 332 305 333 306 334 307 335 308 336 309 337 310 338 311 339 312 340 313 341 314 342 315 343 316 344 317 345 318 346 319 347 320 348 321 349 322 350 323 351 324 352 325 353 326 354 327 355 328 356 329 357 330 358 331 359 332 360 333 361 334 362 335 363 336 364 337 365 338 366 339 367 340 368 341 369 342 370 343 371 344 372 345 373 346 374 347 375 348 376 349 377 350 378 351 379 352 380 353 381 354 382 355 383 356 384 357 385 358 386 359 387 360 388 361 389 362 390 363 391 364 392 365 393 366 394 367 395 368 396 369 397 370 398 371 399 372 400 373 401 374 402 375 403 376 404 377 405 378 406 379 407 380 408 381 409 382 410 383 411 384 412 385 413 386 414 387 415 388 416 389 417 390 418 391 419 392 420 393 421 394 422 395 423 396 424 397 425 398 426 399 427 400 428 401 429 402 430 403 431 404 432 405 433 406 434 407 435 408 436 409 437 410 438 411 439 412 440 413 441 414 442 415 443 416 444 417 445 418 446 419 447 420 448 421 449 422 450 423 451 424 452 425 453 426 454 427 455 428 456 429 457 430 458 431 459 432 460 433 461 434 462 435 463 436 464 437 465

[illegible]

$\overline{TT} \quad \overline{T\bar{T}} \quad T\bar{T} \quad S\bar{S} \quad \overline{TT} \quad T\bar{T} \quad \downarrow \quad \downarrow$

[illegible]

[illegible][illegible][illegible]














































































三 四 五 六 七 八 九 十 十一 十二 十三 十四 十五 十六 十七 十八 十九 二十 二十一 二十二 二十三 二十四 二十五 二十六 二十七 二十八 二十九 三十 三十一 三十二 三十三 三十四 三十五 三十六 三十七 三十八 三十九 四十 四十一 四十二 四十三 四十四 四十五 四十六 四十七 四十八 四十九 五十 五十一 五十二 五十三 五十四 五十五 五十六 五十七 五十八 五十九 六十 六十一 六十二 六十三 六十四 六十五 六十六 六十七 六十八 六十九 七十 七十一 七十二 七十三 七十四 七十五 七十六 七十七 七十八 七十九 八十 八十一 八十二 八十三 八十四 八十五 八十六 八十七 八十八 八十九 九十 九十一 九十二 九十三 九十四 九十五 九十六 九十七 九十八 九十九 一百





















INTEGER(BASIS)

Last change: February 6, 1995

INTEGER(BASIS)

Initial Basis

INTEGER(BASIS)

$$0 \leq x_i \leq 1 \text{ or } x_i = 0$$
$$0 \leq x_i \leq 1$$

The operators min , max , abs , and sign raise Overflow if their second argument is zero. If the second argument is nonzero but the result is too large to be representable, Overflow is raised.

$\text{sign}(i)$
returns -1 , if $i < 0$; and 1 , if $i > 0$.

$\text{sign}(i) \text{sign}(j)$
returns true, if i and j have the same sign.

SEE ALSO

LargeInt(BASIS)

LargeInt Arbitrary-precision integer structure

```
signature LARGE_INT
structure LargeInt : LARGE_INT
```

[illegible]

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

[illegible]

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

INTEGER(BASIS)

described below; some of these may raise the `IndexError` exception when applied to `list`.

`len(l)`

returns `len(l)`, if the list `l` is `list`.

`l[0]`

returns the first item of the list `l`; it raises `IndexError` when applied to `list`.

`l[1:]`

returns the all but the first item of the list `l`; it raises `IndexError` when applied to `list`.

`l[-1]`

returns the last item of the list `l`; it raises `IndexError` when applied to `list`.

`l[i]`

returns the i th element of the list `l` counting from zero. If $i < 0$ or $i \geq \text{len}(l)$, then the exception `IndexError` is raised.

`l[:i]`

Returns the first i elements of the list `l`. If $i < 0$ or $i \geq \text{len}(l)$, then the exception `IndexError` is raised.

`l[i:]`

Returns the tail of the list `l` starting at the i th element (i.e., it drops the first i elements). If $i < 0$ or $i \geq \text{len}(l)$, then the exception `IndexError` is raised.

`len(l)`

returns the number of elements in the list `l`.

`l.reverse()`

reverses the order of the elements of `l`.

`l1 += l2`

appends the elements of list `l2` onto the end of `l1`.

`l1 + l2`

concatenates a list of lists.

`l1 + l2`

returns `l1 + l2`.

`map(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 `list`.

$\text{map } f \ l$

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

$\text{map } f \ l$

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 undefined where it is not defined.

$\text{find } pred \ l$

returns the leftmost element of the list l that satisfies the predicate $pred$; it returns undefined , 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 $pred$).

$\text{filter } pred \ l$

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 .

$\text{partition } pred \ l$

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 .

$\text{fold } f \ init \ l$

computes $f(e_n, f(e_{n-1}, \dots, f(e_1, init) \dots))$, where the e_i are the elements of l . Note that f is applied to the elements in left-to-right order.

$\text{foldr } f \ init \ l$

computes $f(e_1, f(e_2, \dots, f(e_n, init) \dots))$, where the e_i are the elements of l . Note that f is applied to the elements in right-to-left order.

$\text{exists } pred \ l$

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.

$\text{all } pred \ l$

returns true , if all elements of the list l satisfy the predicate $pred$. It is equivalent to $\text{not } (\text{exists } (\text{not } \circ pred) \ l)$.

$\text{unfold } n \ f$

generates the list $[f \ e_n, f \ e_{n-1}, \dots, f \ e_1]$. The function f is applied in

LIST(BASIS)

Initial Basis

LIST(BASIS)

left-to-right (increasing index) order. If $\text{len}(\text{Initial Basis}) < 0$, then the exception `ValueError` is raised.

SEE ALSO

`General(Initial Basis)`, `ListPair(Initial Basis)`

ListPair 16 operations on pairs of lists and lists of pairs

```
signature LIST_PAIR
structure ListPair : LIST_PAIR
```

[illegible]

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

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.

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

is equivalent to $\mathbb{M}_{\text{LTL}} \models f \iff \mathbb{M}_{\text{LTL}} \models l1 \vee l2$.

is equivalent to $\|f\|_{l_1, l_2} \leq \|f\|_{l_1, l_2}$.

is equivalent to $\mathbb{M}_{\text{pred}} \vdash_{\text{pred}} \text{ll}, \text{ll2}$.

is equivalent to $\text{III} \vdash_{\text{L}} \text{pred} \text{II}, \text{II}.$

List(Initial Basis)

Locale(BASIS)

Initial Basis

Locale(BASIS)

[illegible]

DESCRIPTION

This is not the most recent version of this interface.

SEE ALSO

CAVEATS

MATH signature of mathematical library functions

signature MATH

[illegible][illegible]

The $\mathbb{R}^{\mathbb{R}^{\mathbb{R}}}$ structure is a substructure of the structures matching the $\mathbb{R}^{\mathbb{R}^{\mathbb{R}}}$ signature. The square root, exponential, and trigonometric functions are the same as those in the Definition, but we have added additional standard functions:

 Springer

The constant `epsilon` in the full precision of the given real type.

247

The constant `inf` in the full precision of the given real type.




returns $\overline{\text{value}}$, for $\text{value} \neq 0$. If $\text{value} = 0$, then the exception `ValueError` is raised.

 \mathbb{R}^n 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 `ValueError` is raised.

`asin(x)`

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

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

`exp(x)`

returns e^x .

`exp2(x)`

returns 2^x .

`log(x)`

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

`log10(x)`

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

SEE ALSO

Real(BASIS), Float(BASIS)

SIGNATURE

[illegible]

Last change: June 8, 1995

`array.length`

is the maximum length supported for arrays of this type.

`array(n, v)`

creates an array of n elements initialized to v . This raises the `Exception` if n is either too large (`array.length`) or negative.

`array(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 `Exception` if n is either too large (`array.length`) or negative.

`array.from(l)`

creates an array from the list of elements l . This raises the `Exception` if the l has more than `array.length` elements. The zero-length array created by `array.from()` is unique.

`array.length(arr)`

returns the length of the array arr .

`array(arr, i)`

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

`array(arr, i, v)`

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

`array(arr, i, n)`

extracts a vector of length n from the array arr , starting with the i th element. The exception `Exception` is raised if $i < 0$ or $i + n > array.length$.

`array.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 `Exception` is raised if $len < 0$, or if either $si < 0$ or $di < 0$, or $si + len > src.length$, or $di + len > dst.length$.

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

$$dst[di + i] = src[si + i] \quad \text{if } si + i < src.length \\ \text{otherwise}$$

Moreover, if src and dst are different arrays, then for $0 \leq i < len$:

MONO-ARRAY(BASIS)

Initial Basis

MONO-ARRAY(BASIS)

`MONO_ARRAY(BASIS, src, si, len, dst, di)`

is like `MONO_ARRAY(BASIS)`, except that *src* is a vector.

SEE ALSO

`Array(BASIS)`, `MONO_VECTOR(BASIS)`

NAME

MONO_VECTOR $\{t\}$ generic signature of monomorphic vector structures

SYNOPSIS

signature MONO_VECTOR

SIGNATURE

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

$\text{MONO_VECTOR}(t)$

DESCRIPTION

This is the generic signature of monomorphic vectors (e.g., CharVector). The type $\text{MONO_VECTOR}(t)$ is the monomorphic vector type, which is indexed from 0. The type $\text{MONO_VECTOR}(t)$ is the element type, and the type $\text{MONO_VECTOR}(t)$ is the type of the corresponding immutable vectors of the $\text{MONO_VECTOR}(t)$ type. The other members of the structure are:

$\text{MONO_VECTOR}(t)$

is the maximum length supported for vectors of this type.

$\text{MONO_VECTOR}(t)$

creates an vector from the list of elements l . This raises the $\text{MONO_VECTOR}(t)$ exception, if the l has more than $\text{MONO_VECTOR}(t)$ elements.

$\text{MONO_VECTOR}(t)$

creates an vector of n elements, where the i th element is initialized to $\text{MONO_VECTOR}(t)$. The function f is called in increasing order of i . This raises the $\text{MONO_VECTOR}(t)$ exception, if n is either too large ($\text{MONO_VECTOR}(t)$) or negative.

`len(vec)`

returns the length of the vector *vec*.

`vec[i]`

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

`vec[i:n]`

extracts a vector of length *n* from the vector *vec*, starting with the *i*th element. The exception `IndexError` is raised if `i < 0` or `i > len(vec)`.

`vec + vl`

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

SEE ALSO

`MONO_ARRAY(BASIS)`, `Vector(BASIS)`

OS Generic interface to operating system

```
signature OS
structure OS : OS
```

[illegible]

The type `errno_t` represents a system dependent error code; the function `strerror_s` returns a useful error message from a `errno_t`, while the function `strerror` returns the name used by the system for the error code. For example on UNIX systems, applying `strerror_s` to the `ENOENT` error code might return "`File not found`", while `strerror` would return "`no such file or directory`". The exception `std::system_error` is the general exception used by the system interfaces.

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

NAME

OS.FileSys **system independent file-system operations**

SYNOPSIS

```

#include <os.filesys.h>

#include <os.filesys.h>
#include <os.filesys.h>
...
#include <os.filesys.h>
...

```

SIGNATURE

```

#include <os.filesys.h>

/*
 * 1. os.filesys.h
 * 2. os.filesys.h
 * 3. os.filesys.h
 * 4. os.filesys.h
 * 5. os.filesys.h
 * 6. os.filesys.h
 * 7. os.filesys.h
 * 8. os.filesys.h
 * 9. os.filesys.h
 * 10. os.filesys.h
 * 11. os.filesys.h
 * 12. os.filesys.h
 * 13. os.filesys.h
 * 14. os.filesys.h
 * 15. os.filesys.h
 * 16. os.filesys.h
 * 17. os.filesys.h
 * 18. os.filesys.h
 * 19. os.filesys.h
 * 20. os.filesys.h
 * 21. os.filesys.h
 * 22. os.filesys.h
 * 23. os.filesys.h
 * 24. os.filesys.h
 * 25. os.filesys.h
 * 26. os.filesys.h
 * 27. os.filesys.h
 * 28. os.filesys.h
 * 29. os.filesys.h
 * 30. os.filesys.h
 * 31. os.filesys.h
 * 32. os.filesys.h
 * 33. os.filesys.h
 * 34. os.filesys.h
 * 35. os.filesys.h
 * 36. os.filesys.h
 * 37. os.filesys.h
 * 38. os.filesys.h
 * 39. os.filesys.h
 * 40. os.filesys.h
 * 41. os.filesys.h
 * 42. os.filesys.h
 * 43. os.filesys.h
 * 44. os.filesys.h
 * 45. os.filesys.h
 * 46. os.filesys.h
 * 47. os.filesys.h
 * 48. os.filesys.h
 * 49. os.filesys.h
 * 50. os.filesys.h
 * 51. os.filesys.h
 * 52. os.filesys.h
 * 53. os.filesys.h
 * 54. os.filesys.h
 * 55. os.filesys.h
 * 56. os.filesys.h
 * 57. os.filesys.h
 * 58. os.filesys.h
 * 59. os.filesys.h
 * 60. os.filesys.h
 * 61. os.filesys.h
 * 62. os.filesys.h
 * 63. os.filesys.h
 * 64. os.filesys.h
 * 65. os.filesys.h
 * 66. os.filesys.h
 * 67. os.filesys.h
 * 68. os.filesys.h
 * 69. os.filesys.h
 * 70. os.filesys.h
 * 71. os.filesys.h
 * 72. os.filesys.h
 * 73. os.filesys.h
 * 74. os.filesys.h
 * 75. os.filesys.h
 * 76. os.filesys.h
 * 77. os.filesys.h
 * 78. os.filesys.h
 * 79. os.filesys.h
 * 80. os.filesys.h
 * 81. os.filesys.h
 * 82. os.filesys.h
 * 83. os.filesys.h
 * 84. os.filesys.h
 * 85. os.filesys.h
 * 86. os.filesys.h
 * 87. os.filesys.h
 * 88. os.filesys.h
 * 89. os.filesys.h
 * 90. os.filesys.h
 * 91. os.filesys.h
 * 92. os.filesys.h
 * 93. os.filesys.h
 * 94. os.filesys.h
 * 95. os.filesys.h
 * 96. os.filesys.h
 * 97. os.filesys.h
 * 98. os.filesys.h
 * 99. os.filesys.h
 * 100. os.filesys.h
 */

```

DESCRIPTION

The `os.filesys.h` 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 `DirectoryStream` type represents this abstraction; the operations are:

`DirectoryStream(path)`

opens the specified directory stream.

`DirectoryStream(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.

`DirectoryStream(ds)`

rewinds the stream `ds` to the beginning.

`DirectoryStream(ds)`

closes the stream `ds`.

In addition to directory streams, the `os.path` structure provides operations for navigating the directory hierarchy:

`os.chdir(path)`

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

`os.getcwd()`

returns the current working directory.

`os.mkdir(path)`

creates the specified directory.

`os.rmdir(path)`

removes the specified directory.

`os.path.isdir(path)`

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

The interface provides operations for canonicalizing pathnames:

`os.path.realpath(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 `OSError` exception is raised.

`canonicalize(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 `full_path(path)`. 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 `FileError` exception is raised.

Several operations are provided on other files:

`chmod(path, mode)`

`chown(path, uid, gid)`

sets the file access and modification time (as returned by `stat(path)`) 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 `FileError` exception is raised with `errno` set to the string "`FileError: [errno] [path]`". 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)`

`test(path, acl)`

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

`temp_path(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., `/tmp` on UNIX systems).

SEE ALSO

OS(BASIS),Path(BASIS)

OS.PATH(BASIS)

Initial Basis

OS.PATH(BASIS)

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 **S** System independent interface to process primitives

SYNOPSIS

```

#include <OS.Process.h>

OS.Process OS_ProcessCreate(
    . . .
    OS_ProcessAttributes *pAttributes,
    . . .
    OS_Status *pStatus);

```

SIGNATURE

```

OS_Status OS_ProcessCreate(
    OS_ProcessAttributes *pAttributes,
    OS_Status *pStatus);

OS_Status OS_ProcessWait(
    OS_ProcessAttributes *pAttributes,
    OS_Status *pStatus);

OS_Status OS_ProcessWaitFor(
    OS_ProcessAttributes *pAttributes,
    OS_Status *pStatus);

OS_Status OS_ProcessWaitFor(
    OS_ProcessAttributes *pAttributes,
    OS_Status *pStatus);

OS_Status OS_ProcessWaitFor(
    OS_ProcessAttributes *pAttributes,
    OS_Status *pStatus);

```

DESCRIPTION

`OS_ProcessCreate`

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

`OS_ProcessWait`

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

`OS_ProcessWaitFor` *cmd*

executes the command *cmd* as a sub-process of the calling SML program. The call to `OS_ProcessWaitFor` 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.

`exit` `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 `0` for successful termination, and is `non-zero` in the case of errors.

`exit` `without`

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

COMMENT: the exit actions could have type `int` to allow them to test the return code.

SEE ALSO

OS(BASIS)

NAME

PACK_WORD in 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

[illegible]

DESCRIPTION

The `std::byte` 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 `std::byte` 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).

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

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

 $\|vec_i\|$

this extracts the $\text{min}(\text{len}(\text{arr}) - i, \text{len}(\text{arr}) - i - 1)$ bytes starting at index i $\text{min}(\text{len}(\text{arr}) - i, \text{len}(\text{arr}) - i - 1)$.

$$\left\| \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \right\|_{vec, i} = \left\| \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \right\|_{vec, i} + \left\| \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \right\|_{vec, i}$$

this extracts and sign extends the SHA256_BLOCK_SIZE bytes starting at index $i \cdot \text{SHA256_BLOCK_SIZE}$.

arr. i

this extracts the $\text{min_len} \times \text{max_len}$ bytes starting at index $i \times \text{min_len}$.

arr, i

this extracts and sign extends the int_{256} bytes starting at index i .

PACK-WORD(BASIS)

Initial Basis

PACK-WORD(BASIS)

$\text{pack_word}(basis, i, w)$

SEE ALSO

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

REAL(BASIS)

Initial Basis

REAL(BASIS)

DESCRIPTION

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

`REAL(r)`

returns -1 , if $r \leq 0$; and 1 , if $r > 0$.

`REAL(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

val concat : string * string -> string

val m : int * int -> string

val s : string -> int

val sub : string * int -> string

val sub2 : string * int * int -> string

val sub3 : string * int * int * int -> string

val sub4 : string * int * int * int * int -> string

val sub5 : string * int * int * int * int * int -> string

val sub6 : string * int * int * int * int * int * int -> string

val sub7 : string * int * int * int * int * int * int * int -> string

val sub8 : string * int * int * int * int * int * int * int * int -> string

val sub9 : string * int * int * int * int * int * int * int * int * int -> string

val sub10 : string * int * int * int * int * int * int * int * int * int * int -> string

val sub11 : string * int * int * int * int * int * int * int * int * int * int * int -> string

val sub12 : string * int * int * int * int * int * int * int * int * int * int * int * int -> string

val sub13 : string * int * int * int * int * int * int * int * int * int * int * int * int * int -> string

val sub14 : string * int * int * int * int * int * int * int * int * int * int * int * int * int * int -> string

val sub15 : string * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int -> string

val sub16 : string * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int -> string

val sub17 : string * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int * int -> string

DESCRIPTION

Strings are finite sequences of upto 255 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 $0 \leq i \leq \text{length}(s) - n$.

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 `String.Subscript_out_of_range` 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 `STRING_ERROR` is raised.

concat *sl*

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

sl* & *s2

returns the concatenation of *sl* 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 (map str cl)`.

explode *s*

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

```
explode "abc" => ['a','b','c']
```

```
implode ['a','b','c'] => "abc"
```

```
concat (map str ['a','b','c']) => "abc"
```

```
concat (explode "abc" & "d") => "abcd"
```

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

```
val ~> : char list -> string
val ~> : string -> char list
```

```
val ~> : char list -> string
val ~> : string -> char list
val ~> : char list -> string
val ~> : string -> char list
```

```
val ~> : char list -> string
val ~> : string -> char list
val ~> : char list -> string
val ~> : string -> char list
val ~> : char list -> string
val ~> : string -> char list
```

```
val ~> : char list -> string
val ~> : string -> char list
val ~> : char list -> string
val ~> : string -> char list
val ~> : char list -> string
val ~> : string -> char list
```

```
val ~> : char list -> string
val ~> : string -> char list
```

```
val ~> : char list -> string
```

```
val ~> : char list -> string
val ~> : string -> char list
```

DESCRIPTION

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

```
let rec scan : char list -> string
  = fun chars ->
    let rec loop : char list -> string
      = fun chars ->
        if chars = [] then ""
        else let (head, tail) = List.split_at 1 chars
              in head ^ loop tail
    loop chars
```

`string_cvt 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.

`string_scan 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 `string_scan` is returned. If *s* starts with an escape character code that is out of range, the `string_scan` exception is raised.

SEE ALSO

String(BASIS)

NAME

Substring & substring manipulations

SYNOPSIS

signature SUBSTRING

structure Substring : STRING

SIGNATURE

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

$\text{Substring} : \text{string} \rightarrow \text{string}$

```

def substring(s, i, n):
    """Return the substring of s starting at i and ending at i+n-1.
    If i is negative, it is treated as i+len(s). If i+n is greater than
    len(s), it is treated as len(s)."""
    i = max(0, min(i, len(s)))
    n = max(0, min(n, len(s) - i))
    return s[i:i+n]

def substr(s, i, n):
    """Return the substring of s starting at i and ending at i+n-1.
    If i is negative, it is treated as i+len(s). If i+n is greater than
    len(s), it is treated as len(s)."""
    i = max(0, min(i, len(s)))
    n = max(0, min(n, len(s) - i))
    return s[i:i+n]

def substr(s, i, n):
    """Return the substring of s starting at i and ending at i+n-1.
    If i is negative, it is treated as i+len(s). If i+n is greater than
    len(s), it is treated as len(s)."""
    i = max(0, min(i, len(s)))
    n = max(0, min(n, len(s) - i))
    return s[i:i+n]

```

DESCRIPTION

A *substring* is an abstract representation of a contiguous subsequence of a string; we can think of a substring as a triple $\llbracket s, i, n \rrbracket$, 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 $\llbracket s, i, n \rrbracket$ to refer to an abstract substring. We say that a substring $\llbracket s, i, n \rrbracket$ is *valid*, if $0 \leq i < n \leq |s|$. The functions for creating substrings check validity, and the substring operators all preserve validity. This allows efficient implementations that can avoid bounds checking.

`concrete_substring(s, i, n)`

returns the concrete representation of the substring; i.e., the triple $\llbracket s, i, n \rrbracket$.

`extract_substring(s, i, n)`

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

`valid_substring(s, i, n)`

Returns the substring $\llbracket s, i, n \rrbracket$, if it is valid. Otherwise, it raises the `ValueError` exception. This function may also raise `IndexError`, if `i` is not representable as an `int`.

`entire_substring(s)`

returns a substring covering the entire string s .

`empty_substring(ss)`

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

`first_substring(ss)`

returns `ss[0]`, if `ss` is empty, otherwise it returns the first character in the substring and the rest of the substring.

`first_substring(ss)`

returns `ss[0]`, if `ss` is empty, otherwise it returns the first character in the substring.

`trim_substring(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 $k \leq 0$, then the `ValueError` exception is raised.

`ss[k:]`

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 `IndexError` exception is raised.

`ss[s, i, n, j]`

returns `ss[s:j]`, if $0 \leq i \leq n$. Otherwise the `IndexError` exception is raised.

`len(ss)`

returns n .

SEE ALSO

`Char(BASIS)`, `String(BASIS)`

TIME(BASIS)

Initial Basis

TIME(BASIS)

NAME

Time Representation of time values

SYNOPSIS

signature TIME

structure Time : TIME

SIGNATURE

$\text{val } \text{zero} : \text{TIME}$

$\text{val } \text{now} : \text{TIME}$

$\text{val } \text{now_diff} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel_abs_rel_abs_rel_abs_rel} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_abs_rel_abs_rel_abs_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

$\text{val } \text{now_diff_rel_abs_rel_abs_rel_abs_rel_abs_rel_abs_rel_abs} : \text{TIME} \rightarrow \text{TIME}$

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).

`zero`

is the time representation of zero (e.g., `now - now`).

`timeval sec timeval r`

converts a real number representing seconds to a time value. If `sec` is 0, then the exception `timeval` is raised.

`timeval timeval t`

If this is not representable as an `int64_t`, then the `timeval` exception is raised.

`timeval sec sec`

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

`timeval 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 too large to be represented as an `int64_t`, then the `timeval` exception is raised.

`timeval sec sec`

`timeval t`

`timeval sec sec`

`timeval t`

`t1 + t2`

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

`t1 - t2`

subtracts the time value `t2` from `t1`. If `t1 < t2`, then the `timeval` 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`.

`time()`

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

`time 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` is 0, then no decimal digits are reported.

`time charSrc`

`time t`

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

`time 3`.

`time s`

This converts the string *s* to a time value; it returns `0`, if *s* is not valid, and raises `ValueError` if *s* is too large. It is equivalent to: `time(0) + float(s)`.

SEE ALSO

Date(BASIS), Timer(BASIS)

TIMER(BASIS)

Initial Basis

TIMER(BASIS)

NAME

Timer ~~is~~ Interface to system timers

SYNOPSIS

signature TIMER

structure Timer : TIMER

SIGNATURE

```
type timer =
  | CPU_TIMER
  | WALL_TIMER

type timer_value =
  | CPU_TIMER
  | WALL_TIMER

type timer_status =
  | CPU_TIMER
  | WALL_TIMER

type timer_error =
  | CPU_TIMER
  | WALL_TIMER

type timer_result =
  | CPU_TIMER
  | WALL_TIMER

type timer_status =
  | CPU_TIMER
  | WALL_TIMER

type timer_error =
  | CPU_TIMER
  | WALL_TIMER

type timer_result =
  | CPU_TIMER
  | WALL_TIMER
```

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 ~~is~~ immutable polymorphic vectors

SYNOPSIS

signature VECTOR

structure Vector : VECTOR

SIGNATURE

`type 'a t = 'a list`

`val m : 'a t -> 'a t`

`val cons : 'a -> 'a t -> 'a t`

`val hd : 'a t -> 'a`

`val tail : 'a t -> 'a t`

`val cons2 : 'a -> 'a t -> 'a t`

`val cons3 : 'a -> 'a t -> 'a t`

`val cons4 : 'a -> 'a t -> 'a t`

`val cons5 : 'a -> 'a t -> 'a t`

`val cons6 : 'a -> 'a t -> 'a t`

`val cons7 : 'a -> 'a t -> 'a t`

`val cons8 : 'a -> 'a t -> 'a t`

`val cons9 : 'a -> 'a t -> 'a t`

`val cons10 : 'a -> 'a t -> 'a t`

DESCRIPTION

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

`val max_length : int`

is the maximum length supported for polymorphic vectors.

`val of_list : 'a list -> 'a t`

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

`val of_array : 'a array -> 'a t`

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

`val length : 'a t -> int`

returns the length of the vector *vec*.

```
vec[i]
```

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

```
vec[i:n]
```

extracts a vector of length n from the vector *vec*, starting with the i th element. The exception `IndexError` is raised if $i < 0$ or $n < 0$ or $i > \text{len}(vec)$.

```
v+l
```

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

SEE ALSO

Array(BASIS), MONO_VECTOR(BASIS)

WORD(BASIS)

Initial Basis

WORD(BASIS)

NAME

Word ~~to~~ unsigned integers

SYNOPSIS

signature WORD

structure Word : WORD

structure Word n : WORD

SIGNATURE

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

val ~~to~~ : int list -> word

DESCRIPTION

The ~~to~~ type represents integers modulo 2^{64} , where ~~to~~ is a 64-bit integer.

If the structure $\text{Basis}(n)$ is present, then

$$\text{Basis}(n) = \{ \text{basis}_i \mid 0 \leq i < n \}$$

Also, if there are both $\text{Basis}(n)$ and $\text{Basis}(n)$ structures present, then

$$\text{Basis}(n) = \{ \text{basis}_i \mid 0 \leq i < n \}$$

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

$$\text{bitwise}(x, y) = \sum_{i=0}^{n-1} 2^i \cdot (x_i \oplus y_i) \bmod 2^n,$$

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

$\text{Basis}(n)$ i

yields a word x_i representing $i \bmod 2^n$. Cannot raise $\text{Basis}(n)$.

$\text{Basis}(n)$ w

Returns a the smallest nonnegative integer i such that $\text{Basis}(n)(i) = w$, if w is representable as an i . Otherwise, returns the negative integer i of smallest absolute value such that $\text{Basis}(n)(i) = w$, if w is representable as an i . Otherwise, raises $\text{Basis}(n)$.

$\text{Basis}(n)$ w

If $w \bmod 2^n = w \bmod 2^{n-1}$, returns the smallest nonnegative integer i such that $\text{Basis}(n)(i) = w$.

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

If no such i is representable, raises $\text{Basis}(n)$.

$\text{Basis}(n)$ x, y

returns the bitwise or of x and y . That is, $\text{Basis}(n)(x \vee y) = \text{bitwise}(x, y) \bmod 2^n$.

$\text{Basis}(n)$ x, y

bitwise exclusive-or, that is $\text{Basis}(n)(x \oplus y) = \text{bitwise}(x, y) \bmod 2^n$.

$\text{Basis}(n)$ x, y

bitwise and, that is $\text{Basis}(n)(x \wedge y) = \text{bitwise}(x, y) \bmod 2^n$.

$\text{Basis}(n)$ w

returns the bitwise complement of w , that is $\text{Basis}(n)(\neg w) = \text{bitwise}(\neg w, 0) \bmod 2^n$.

WORD(BASIS)

Initial Basis

WORD(BASIS)

 $\text{WORD}(BASIS) \ll w, k$

shifts k left bits; or shifts right if k is negative. $\text{WORD}(BASIS) \ll w, k = \text{WORD}(BASIS \bmod 2^{32}) \cdot 2^k \bmod 2^{32}$.

 $\text{WORD}(BASIS) \gg w, k$

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

$$\begin{aligned} \text{WORD}(BASIS) \gg w, k &= \text{WORD}(BASIS) \gg w, k \text{ if } \text{WORD}(BASIS) \bmod 2^{32} \gg w \bmod 2^{32-1} \text{ or } k < 0 \\ \text{WORD}(BASIS) \gg w, k &= -\text{WORD}(BASIS) \ll w, k \text{ otherwise} \end{aligned}$$
 $\text{WORD}(BASIS) \ll w1, w2$

returns $\text{WORD}(BASIS) \ll w1 \ll w2 \bmod 2^{32}$.

 $\text{WORD}(BASIS) - \ll w1, w2$

returns $\text{WORD}(BASIS) - \text{WORD}(BASIS) \ll w1 \ll w2 \bmod 2^{32}$.

 $\text{WORD}(BASIS) \gg w1, w2$

returns $\text{WORD}(BASIS) \gg w1 \gg w2 \bmod 2^{32}$.

 $\text{WORD}(BASIS) \div \text{WORD}(BASIS) \ll x, y$

Unsigned division: returns $\left\lfloor \frac{x}{y} \right\rfloor$, where $x \bmod 2^{32} = 0$ or $x \bmod 2^{32} \neq 0$ and $y \bmod 2^{32} \neq 0$. Raises the exception if y is 0.

 $\text{WORD}(BASIS) \div \text{WORD}(BASIS) \gg x, y$

returns $\left\lfloor \frac{x}{y} \right\rfloor \bmod 2^{32}$. Raises the exception if y is 0.

SEE ALSO

Byte(BASIS), Int(BASIS), SmallInt(BASIS), CONVERT WORD(BASIS)

Part III

Amendment: POSIX 1003.1b-1993

SYNOPSIS

SIGNATURE

94 Last change: March 17, 1995

POSIX-ERROR(BASIS)

[illegible]

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 `errno` values correspond to the basic errors defined in the POSIX standard (cf. Section 2.4 of IEEE Std 1003.1b-1993). The function `strerror` maps an error code to an error message (e.g., `errno` might return the string "No such file or directory"). The `errno` and `strerror` 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.

Posix(BASIS)

NAME

POSIX_FLAGS POSIX bit flags interface

SYNOPSIS

```
#include <unistd.h>
#include <sys/_types.h>
```

SIGNATURE

```
int posix_flag_t;
```

```
int posix_flag_t posix_flag_t | posix_flag_t;
```

```
int posix_flag_t posix_flag_t & posix_flag_t;
```

```
int posix_flag_t posix_flag_t & posix_flag_t;
```

```
int posix_flag_t posix_flag_t & posix_flag_t;
```

```
int posix_flag_t posix_flag_t & posix_flag_t;
```

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 `posix_flag_t` forms the union of all the flags set in its argument list. The call `posix_flag_t posix_flag_t` returns true if all the flags set in `posix_flag_t` are also set in `posix_flag_t`, i.e., `posix_flag_t` is a subset of `posix_flag_t`. The call `posix_flag_t posix_flag_t` returns true if any flag set in `posix_flag_t` is also set in `posix_flag_t`, i.e., the intersection of `posix_flag_t` and `posix_flag_t` is non-empty. The `posix_flag_t` and `posix_flag_t` 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)

Posix.FileSys for operations on the file system

[illegible][illegible]

POSIX-FILE-SYS(BASIS)

Initial Basis

POSIX-FILE-SYS(BASIS)

[illegible][illegible][illegible]

26. 779.111

[illegible][illegible][illegible]

functions `chmod`, `chown`, `lchmod` and `lchown` and the types `mode_t` and `uid_t`. The substructure `stat` implements the standard POSIX permission bits. Here also, the functions `stat` and `lstat` allow access to the underlying arithmetic representation. The functions `statvfs`, `fstatvfs`, `statfs` and `fstatfs` are provided as part of the POSIX standard 1003.1a, although this has not been officially accepted as yet. The functions `statfs` and `fstatfs` return `-1` if the corresponding value is unbounded.

SEE ALSO

Posix(BASIS), POSIX_FLAGS(BASIS)

Posix.IO 1.6 basic I/O operations

[illegible][illegible]

[illegible]

DESCRIPTION

This structure provides the primitive POSIX I/O operations, as described in Section 6 of IEEE Std 1003.1b-1993. The functions `fcntl`, `fcntl64`, `fstat`, `fstat64`, `fstatfs`, `fstatfs64`, `fstatvfs` and `fstatvfs64` correspond to calls to the POSIX `fcntl` function with the commands `F_GETFL`, `F_SETFL`, `F_GETFD`, `F_SETFD`, `F_GETFRCH`, `F_SETFRCH`, `F_GETLK` and `F_SETLK`, respectively. The substructure `fcntl` implements sets of file descriptor flags, the only POSIX required value being `O_RDONLY` corresponding to the C constant `O_RDONLY`. Similarly, the substructure `stat` implements sets of file status flags, with the supplied values `STAT_RDONLY`, `STAT_WRITE` and `STAT_APPEND` corresponding to the POSIX defined C constants `STAT_RDONLY`, `STAT_WRITE`, `STAT_APPEND` and `STAT_APPEND`, respectively.

POSIX-IO(BASIS)

Initial Basis

POSIX-IO(BASIS)

SEE ALSO

Posix(BASIS), POSIX_FLAGS(BASIS)

[illegible]

DESCRIPTION

This structure encapsulates the POSIX operations on the process environment, as described in Section 4 of IEEE Std 1003.1b-1993. The `getenv`, `setenv`, `unsetenv`, `putenv` and `clearenv` functions provide access to the underlying arithmetic representation of `char*` and `int` values. The `raise` raises an exception if the corresponding feature is not supported by the underlying operating system.

SEE ALSO

Posix(BASIS)

POSIX-SIGNAL(BASIS)

Initial Basis

POSIX-SIGNAL(BASIS)

SEE ALSO

Posix(BASIS)

POSIX-TTY(BASIS)

Initial Basis

POSIX-TTY(BASIS)

1. $\text{tty_open}(\text{device}, \text{mode})$ - Open the terminal device `device` in the mode specified by `mode`.
2. $\text{tty_close}(\text{tty}$) - Close the terminal device `tty`.
3. $\text{tty_setopt}(\text{tty}, \text{opt_name}, \text{opt_val})$ - Set the option `opt_name` to the value `opt_val` for the terminal device `tty`.
4. $\text{tty_getopt}(\text{tty}, \text{opt_name}, \text{opt_val})$ - Get the option `opt_name` for the terminal device `tty`.
5. $\text{tty_flush}(\text{tty})$ - Flush the terminal device `tty`.
6. $\text{tty_write}(\text{tty}, \text{buf}, \text{count})$ - Write `count` bytes from the buffer `buf` to the terminal device `tty`.
7. $\text{tty_read}(\text{tty}, \text{buf}, \text{count})$ - Read `count` bytes from the terminal device `tty` into the buffer `buf`.
8. $\text{tty_isatty}(\text{fd})$ - Determine if the file descriptor `fd` is a terminal device.
9. $\text{tty_termios}(\text{tty})$ - Get the terminal control settings for the terminal device `tty`.
10. $\text{tty_settermios}(\text{tty}, \text{termios})$ - Set the terminal control settings for the terminal device `tty` to the `termios` structure.

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. 147-157.