1
Introduction

This document describes the Standard ML Basis Library. The Library provides an extensive collection of basic types and functions for the Standard ML (SML) language, as described by the Definition of Standard ML (Revised) [MTHM97]. The goals of the Basis library are to:

- serve as the basic toolkit for the SML programmer, whether novice or professional;
- focus attention on the attractiveness of SML as a language for programming in a wide variety of domains, e.g., systems programming;
- replace the many incompatible general-purpose libraries currently available.

The original definition of the Standard ML language [MTH90] was published in 1990, for which reason we refer to it as SML'90. The Definition specified an *initial basis*, i.e., a set of primitive types such as `int` and `string` along with some related operations, which was used to define various derived forms and special constants. Though adequate for the purpose of language specification, it was too limited for programming applications. In response, most implementations of the language extended the basis with large collections of generic libraries. With the libraries coming from different sources, they tended to be incompatible, even when implementing the same abstract types and functions. The result was that, despite the standardization of the language, any significant SML program could be compiled on multiple implementations only if the programmer were willing to provide portable libraries that relied only on the initial basis.

The SML Basis Library is a rich collection of general-purpose modules, which can serve as the foundation for applications programming or for more domain-specific libraries. It provides most of the basic types and operations expected by a working programmer and specifies that anyone using SML can expect to find them in any implementation.

Some goals in designing the Library worked toward its expansion. One, suggested above, was the desire for the Library to be "complete enough." If using a type provided by the Library, the programmer should be able to look in the defining structure and find the right function or, at least, the functions needed to build the desired function easily. In addition, the Library attempts to provide similar functions in similar contexts. Thus, the traditional `app` function for lists, which applies a function to each member of a list, has also been provided for arrays and vectors.

An opposite design force has been the desire to keep the Basis library small. In general, a function has been included only if it has clear or proven utility, with additional emphasis on those that are complicated to implement, require compiler or runtime system support, or are more concise or efficient than an equivalent combination of other functions. Some exceptions were made for historical reasons or for perceived user convenience.

The SML language has the rare property of being a practical, general-purpose programming language possessing a well-defined, indeed formal, semantics. Following in this spirit, some SML-based libraries, e.g., CML [Rep99], build on this precision by supplying their own formal semantics. Although we viewed this goal as beyond what we could provide for the Basis library, we still felt very strongly that the functions included here should be defined as precisely and clearly as possible. In some cases, we have defined the meaning of basis functions via reference implementations. We want SML programs to be *deterministic* (aside from their interaction with the external world), and so we specify the traversal order for higher-level functions such as `List.map`. The description of a function provides the dynamic constraints on the arguments, such as that an integer index into an array must be less than the length of the array, and relates what happens when a function invocation violates these constraints, typically the raising of a particular exception. We have tried to stipulate completely the format of return values, so that, when a type's representation is visible, the programmer will know what to expect concretely, not just abstractly. We have avoided unspecified or implementation-dependent results whenever possible. Some functions were excluded from the Library because we could not provide a clean specification for the function's behavior.

1.1 Design rules and conventions

In designing the Library, we have tried to follow a set of stylistic rules to make library usage consistent and predictable, and to preclude certain errors. These rules are not meant to be prescriptive for the programmer using or extending the Library. On the other hand, although the Library itself flouts the conventions on occasion, we feel the rules are reasonable and helpful and would encourage their use.

1.1.1 Orthographic conventions

We use the following set of spelling and capitalization conventions. Some of these conventions, e.g., the capitalization of value constructors, seem to be widely accepted in the user community. Other decisions were based less on a dominant style or a compelling reason than on compromise and the need for consistency and some sense of good taste.

The conventions we use are

- Alphanumeric value identifiers are in mixed-case, with a leading lowercase letter; e.g., `map` and `openIn`.
- Type identifiers are all lowercase, with words separated by underscores; e.g., `word` and `file_desc`.
- Signature identifiers are in all capitals, with words separated by underscores; e.g., `PACK_WORD` and `OS_PATH`. We refer to this convention as the *signature* convention.
- Structure and functor identifiers are in mixed-case, with initial letters of words capitalized; e.g., `General` and `WideChar`. We refer to this convention as the *structure* convention.
- Alphanumeric datatype constructors follow the signature convention; e.g., `SOME`, `A_READ`, and `FOLLOW_ALL`. In certain cases, where external usage or aesthetics dictates otherwise, the structure convention can be used. Within the Basis library, the only use of the latter convention occurs with the months and weekdays in `Date`, e.g., `Jan` and `Mon`. The only exceptions to these rules are the identifiers `nil`, `true`, and `false`, where we bow to tradition.
- Exception identifiers follow the structure convention; e.g., `Domain` and `SysErr`.

These conventions concerning variable and constructor names, if followed consistently, can be used by a compiler to aid in detecting the subtle error in which a constructor is misspelled in a pattern-match and is thus treated as a variable binding. Some implementations may provide the option of enforcing these conventions by generating warning messages.

1.1.2 Naming

Similar values should have similar names, with similar type shapes, following the conventions outlined above. For example, the function `Array.app` has the type:

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

which has the same shape as `List.app`. Names should be meaningful but concise. We have broken this rule, however, in certain instances where previous usage seemed compelling. For example, we have kept the name `app` rather than adopt `apply`. More dramatically, we have purposely kept most of the traditional Unix names in the optional `Posix` modules, to capitalize on the familiarity of these names and the available documentation.

1.1.3 Comparisons

If a type *ty*, such as `int` or `string`, has a standard or obvious linear order, the defining structure should define the expected relational operators >, >=, <, and <=, plus a comparison function

```
val compare : ty * ty -> order
```

(where `order` is defined in the `General` structure and has the constructors `LESS`, `EQUAL`, `GREATER`). In all cases, the expected relationships should hold between these functions. For example, we have `x > y = true` if and only if `compare(x, y) = GREATER`. If, in addition, *ty* is an equality type, we assume that the operators = and <> satisfy the usual relationships with `compare` and the relational operators. For example, if `x = y`, then `compare(x,y) = EQUAL`. Note that these assumptions are not quite true for `real` values; see the `REAL` signature for more details.

For reasons of style and simplicity, we have in general attempted to avoid equality types except where tradition or convenience dictated otherwise. Most of the equality types are abstractions of integral values. We prefer to keep equality evaluation explicit, usually by an associated `compare` function.

Certain abstract types, e.g., `OS.FileSys.file_id`, provide a `compare` function even though elements of the type do not possess an inherent linear order. These functions are useful in maintaining and searching sets of these elements in, for example, ordered binary trees.

1.1.4 Conversions

Most structures defining a type provide conversion functions to and from other types. When unambiguous, we use the naming convention `to`*T* and `from`*T*, where *T* is some version of the name of the other type. For example, in `WORD`, we have

```
val fromInt : Int.int -> word
val toInt : word -> Int.int
```

If this naming is ambiguous (e.g., a structure defines multiple types that have conversions from integers), we use the convention *T*`From`*TT* and *T*`To`*TT*. For example, in `POSIX_PROC_ENV`, we have

```
val uidToWord : uid -> SysWord.word
val gidToWord : gid -> SysWord.word
```

There should be conversions to and from strings for most types. Following the convention above, these functions are typically called `toString` and `fromString`. Usually, modules provide additional string conversion functions that allow more control over format and operate on an abstract character stream. These functions are called `fmt` and `scan`. The input accepted by `fromString` and `scan` consists of printable ASCII characters. The output generated by `toString` and `fmt` consists of printable ASCII characters. Additional discussion of string scanning and formatting can be found in Chapter 5.

We adopt the convention that conversions from strings should be forgiving, allowing initial whitespace and multiple formats and ignoring additional terminating characters. In addition, for basic types, scanning functions should accept legal SML literals. On the other hand, we have tried to specify conversions to strings precisely. Formatting functions should, whenever possible, produce strings that are compatible with SML literal syntax, but without certain annotations. For example, `String.toString` produces a valid SML string constant, but without the enclosing quotes, and `Word.toString` produces a word constant without the `"0wx"` prefix.

1.1.5 Characters and strings

The revised SML definition [MTHM97] introduces a `char` type and syntax for character literals. The SML Basis Library provides support for both `string` and `char` types, where the `string` type is a *vector* of characters. In addition, we define the optional types `WideString.string` and `WideChar.char`, in which the former is again a vector of the latter, for handling character sets more extensive than Latin-1.

The SML'90 Basis Library did not provide a character type. To manipulate characters, programmers used integers corresponding to the character's code. This situation was unsatisfactory for several reasons: there were no symbolic names for single characters in patterns, character to string conversions required unnecessary range checks, and there was no provision for the extended character sets necessary for international use. Alternatively, programmers could use strings of length one to represent characters, which was less efficient and could not be enforced by the type system.

1.1.6 Operating system interfaces

The Library design probably has the least freedom concerning access to an implementation's operating system (OS). To allow code written using the Library to be as portable as possible, we have adopted standard interfaces, either *de jure* or *de facto*. We have layered the structures dealing with the OS. The facilities encapsulated in the `OS` structure and the I/O modules represent models common to most current operating systems;

code restricted to these should be generally portable. Structures such as `Unix` are more OS-specific but still general enough to be available on a variety of systems. Structures such as the `Posix` structure provide an interface to a particular OS in detail; it would be unlikely that an implementation would provide this structure unless the underlying OS were a version of POSIX. Finally, an implementation may choose to provide additional structures containing bindings for all of the types and functions supported by a given OS.

When two structures both define a type that has the same meaning, e.g., both the `OS` and `Posix.Error` structures define a `syserror` type, then these types should be equivalent or there should be an effective way to map between the types. Implementations should preserve lower-level information. For example, on a POSIX system, if a program calls `OS.Process.exit(st)`, where `Posix.Process.fromStatus(st)` evaluates to `Posix.Process.W_EXITSTATUS(v)`, then the program should exit using the call `Posix.Process.exit(v)`.

Most OS interfaces involve certain specified values, such as the platform IDs specified in `Windows.Config` or the error values given in `Posix.Error`. Any static list is bound to be incomplete, either due to changes over time or from variations among implementations. To allow for differences and extensibility, the Library typically does not use a datatype for these types; it allows implementations to generate values not specified in the signature; and, when possible, it will allow the programmer access to the primitive representation of such values. Conforming implementations of the SML Basis Library can not extend the signatures given by the Library.

1.1.7 Miscellany

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

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

It is also recommended that implementations generate a warning message when an expression on the left-hand side of a semicolon in a sequence expression (i.e., e_1 in $(e_1; e_2)$) does not have `unit` type.

The use and need for exceptions should be limited. If possible, the type of an argument should prevent an exceptional condition from arising. Thus, rather than have `Posix.FileSys.openf` return an `int` value, as its analogue does in C, the Library uses the `Posix.FileSys.file_desc` type. In cases where multiple function arguments have the same type, opening the possibility that the programmer may switch them, the Library employs SML records, as with `OS.FileSys.rename`. The avoidance of exceptions is particularly apparent in functions that parse character input to create a value, which uniformly return a value of `NONE` to indicate incorrect input rather than raising exceptions.

If a curried function can raise an exception, the exception should be raised as soon as sufficient arguments are available to determine that the exception should be raised. Thus, given a function

```
val gen : int -> string -> string
```

that raises an exception if the first argument is negative, the evaluation of `gen ~1` should trigger the exception.

SML is a value-oriented language which discourages the gratuitous use of state. Thus, the Library tries to minimize the use of state. In particular, we note that, although the Library allows imperative-style input, it provides stream-based input with unbounded lookahead and many of the routines for converting characters to values work most naturally in this style.

The language does allow functions to have side effects. To take this fact into account, the Library requires that the implementations of higher-order functions over aggregates invoke their function-valued arguments at most once per element.

Whenever possible, structures specified as signature instances are matched opaquely, so that all types are abstract unless explicitly specified in the signature or any associated **`where type`** clauses.

1.2 Documentation conventions

This section describes the conventions used in this document. These include the layout of the manual pages, notational conventions, and liberties that we have taken with the SML syntax and semantics to make the specification clearer.

When applicable in multiple contexts, some information is repeated. We felt it was better to accept some redundancy rather than to force the reader to glean information scattered all over the document.

1.2.1 Organization of the manual pages

The bulk of the SML Basis Library specification consists of *manual pages* specifying the signature and semantics of the Basis modules. These pages are organized in alphabetical order. Each manual page typically describes a single signature and the structure or structures that implement it.

A manual page, which is typically comprised of several physical pages, is organized into at most five sections. The first of these is the “Synopsis,” which lists the signature, structure, and functor names covered by the manual page. With the exception of manual pages that cover functors, which can have both argument and result signatures, the synopsis consists of a signature name and the names of the structures matching the signature. The second section is the "Interface," which gives the SML specifications that form the body of the signature. Functor specifications can have multiple interface sections, since there are both argument and result signatures involved. Following the interface part is the "Description," which consists of detailed descriptions of each of the SML specifications listed in the interface. Some manual pages follow the description section with a "Discussion" section that covers broader aspects of the interface. Finally, there is the "See also" section, which gives cross references to related manual pages.

1.2.2 Terminology and notation

For functions that convert from strings to primitive types, we often use a standard regular expression syntax to describe the accepted input language.

Regular Expression	**Meaning**
$re^?$	An optional instance of re.
re^*	A sequence of zero or more instances of re.
re^+	A sequence of one or more instances of re.
$re_1 re_2$	An instance of re_1 followed by an instance of re_2.
$re_1 \mid re_2$	An instance of either re_1 or re_2.

In a regular expression, a character in `teletype` font represents itself. As a shorthand, we write [abc] for a | b | c. A choice of consecutive characters (in the ASCII ordering) can be specified using a range notation, e.g., [a−c].

When specifying the meaning of certain operations, we will sometimes use various standard mathematical notations in the SML Basis Library specification, implicitly mapping SML values to values in the domain of natural numbers, integers, or real numbers. The intended meaning should be clear from the context. The greatest integer less than or equal to a real-valued expression r (i.e., the *floor* of r) is written $\lfloor r \rfloor$ and, likewise, the least integer greater than or equal to a real-valued expression r (i.e., the *ceiling* of r) is written $\lceil r \rceil$. For integers i and m, the notation $i \pmod{m}$ is the remainder $i - m\lfloor \frac{i}{m} \rfloor$. The notation $[a, b]$ denotes the range of values x where $a \leq x \leq b$ in some corresponding domain. The sign function, $sgn(x)$, returns -1, 1, or 0, depending on whether x is negative, positive, or equal to zero, respectively.

For sequence types (i.e., lists, vectors, and arrays), we use the terminology *left-to-right* to mean the order of increasing index and *right-to-left* to mean the order of decreasing index. Sequences are indexed from zero. If seq is a value of a sequence type, then $|seq|$ means the number of elements, or length, of seq and $seq[i..j]$ is the subsequence of seq consisting of the items whose indices are in the range $[i, j]$, where $0 \leq i \leq j < |seq|$.

For the numeric types, we define generic signatures that are implemented at multiple precisions. For example, an implementation might provide `Int32` for 32-bit integers and `Int64` for 64-bit integers. Rather than list a sample of possible structures, we use the notation `Int`*`N`* to specify the *family* of structures that implement the `INTEGER` signature, where *`N`* specifies the number of bits in the representation.

Syntactic and semantic liberties

We have taken a few liberties with the syntax and semantics of SML in the SML Basis Library specification. In a couple of cases, we define interfaces that include a signature, which overlaps with other specifications in the interface. For example, the `TEXT_IO` signature has a `StreamIO` substructure and also includes the `IMPERATIVE_IO` signature, which itself specifies a `StreamIO` substructure. The intention is clear: the two `StreamIO` substructures are intended to be identified while matching the more detailed signature, in this case `TEXT_STREAM_IO`. While these violate the rules for well-formed signatures in SML, they avoid useless redundancy in the documentation.

Another place where we depart from strict SML semantics is in **`where type`** specifications. These specifications imply a dependency from the module with the specification to the module that defines the type. We want to allow implementations freedom in how they organize the Basis library modules, so when there is not a clear ordering of the modules we attach symmetric **`where type`** specifications to both modules. For example, the `String.string` and `CharVector.vector` types are equal, so we attach **`where type`** specifications to both the `String` and `CharVector` modules. Implementations are free to define one of these types in terms of the other or to define both in terms of some third type.

The “Description” section of the manual pages sometimes provides a prototypical use of a function when describing the function. When part of the argument of a function is a record type, the prototype does not use the correct record construction syntax, but rather the pattern-matching syntax as might be used in the definition of the function. For example, the description of the `Posix.IO.dup2` function has the prototype usage `dup2 {`*`old`*`,`*`new`*`}` rather than the syntactically correct `dup2 {old=`*`old`*`,new=`*`new`*`}`.

Finally, for functions returning an `option` type, in the cases when it actually returns a value as `SOME(`*`v`*`)`, the description will usually just say that the function returns *`v`*, with the `SOME` wrapper being understood.

2
Library modules

The Basis Library is organized using the SML module system. Almost every type, exception constructor, and value belongs to some structure. Although some identifiers are also bound in the initial top-level environment, we have attempted to keep the number of top-level identifiers small. Infix declarations and overloading are specified for the top-level environment.

We view the signature and structure names used below as being reserved. For an implementation to be conforming, any module it provides that is named in the SML Basis Library must exactly match the description specified in the Library. For example, the `Int` structure provided by an implementation should not match a superset of the `INTEGER` signature. Furthermore, an implementation may not introduce any new non-module identifiers into the top-level environment. If an implementation provides any types, values, or exceptions not described in the SML Basis Library, they must be encapsulated in structures whose names are not used by the SML Basis Library.

Some structures have signatures that refer to types that will belong to another structure. Rather than include the other structure as a substructure, we have chosen to rebind just the necessary types. It was felt that this policy makes the code easier to reorganize in large systems. Explicit connections between structures are specified by sharing constraints in the language, or by descriptions in the text.

2.1 Required modules

We have divided the modules into *required* and *optional* categories. Any conforming implementation of SML Basis Library must provide implementations of all of the required modules.

Many of the structures are variations on some generic module (e.g., single and double-precision floating-point numbers). Table 2.1 gives a list of the required generic signatures. A system will typically provide multiple implementations of some of these signatures; it is assumed that multiple implementations are allowed for all of them. Generic