Appendix B: The Standard ML Basis Library

10

Following is a subset of the Standard ML Basis Library. The Basis Library is covered in more detail at http://www.standardml.org/Basis. Documentation for these structures is found in this appendix.

- Bool
- Int
- Real
- Char
- String
- List
- Array
- TextIO

Other structures exist on the Basis website. The descriptions provided here may be helpful as well. Each function, along with its signature, is provided for each of the structures listed in this appendix.

10.1 The Bool Structure

This is the signature of the functions Bool structure. In addition to the not operator, SML defines the andalso and orelse operators which implement shortcircuit logic. More information can be found at http://www.standardml.org/Basis/bool.html.

datatype bool = false | true

The bool datatype is either false or true.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_10

val not : bool -> bool

not true = false, not false = true.

val toString : bool -> string

Converts a true/false value to a string for printing or other purposes.

val fromString : string -> bool option

Converts from a string to a bool. An option is either *NONE* or *SOME val*. If the string cannot be converted to a bool (i.e. it does not contain true or false), then *NONE* is returned. Otherwise *SOME true* or *SOME false* is returned. Pattern-matching can be used to determine the return value.

val scan : (char,'a) StringCvt.reader -> (bool,'a) StringCvt.reader

This behaves like *fromString* except that the remaining character stream is returned along with the value if a bool is found in the stream.

10.2 The Int Structure

Implementing the INTEGER signature, the Int structure contains the *int* type. Integer precision is platform dependent. Normally 32-bit or 64-bit precision is available depending on the platform. More information can be found at http://www.standardml.org/Basis/integer.html.

type int

The type of integers.

val precision : Int31.int option

An option indicating the precision of integers. For instance, *SOME 31* indicating 32-bit integers from -2^{31} to $2^{31} - 1$. If the value is *NONE* it indicates arbitrary precision.

val minInt : int option

val maxInt : int option

Minimum and maximum integer values given the precision available. *NONE* if integers have arbitrary precision.

val toLarge : int -> IntInf.int

val fromLarge : IntInf.int -> int

Conversion functions from and to large integers.

val toInt : int -> Int31.int

val fromInt : Int31.int -> int

Conversion functions from and to 32-bit integers. Depending on implementation these may be identity functions.

val ~ : int -> int

Unary negation. ~6 is a negative 6.

val + : int * int -> int

val - : int * int -> int

val * : int * int -> int

val div : int * int -> int

val mod : int * int -> int

Typical integer operations. Note that *div* and *mod* are infix operators returning the integer division and remainder respectively. For instance, *6 div 4 = 1* and *6 mod 4 = 2*. These operations are infix operators.

val quot : int * int -> int

val rem : int * int -> int

These two operations reflect that most hardware implementations of integer division behave differently than the mathematical definition used by *div* and *mod* for negative integers. Consider the following.

```
- val x = ~6;
val x = ~6 : int
- x mod 4;
val it = 2 : int
- x div 4;
val it = ~2 : int
- Int.quot(x,4);
val it = ~1 : int
- Int.rem(x,4);
val it = ~2 : int
```

This shows that *mod* and *div* factor *–6* as *–2 * 4 +2* while *quot* and *rem* factor *–6* as *–1 * 4 +–2*. The mathematical definition of *mod* always results in a positive remainder. However, computer hardware often calculates using *quot* and *rem* semantics possibly resulting in faster calculations.

val min : int * int -> int

val max : int * int -> int

Maximum and minimum functions of two integers. Returns the max or min value of the pair of integers.

val abs : int -> int

Returns the absolute value.

val sign : int -> Int31.int

Returns either 1 or −1 depending on the sign of the integer.

val sameSign : int * int -> bool

True or false depending on the two integers.

val > : int * int -> bool

val >= : int * int -> bool

val < : int * int -> bool

val <= : int * int -> bool

Relational operators for the ordering of integers. These operators are infix operators.

val compare : int * int -> order

Returns one of the *order* values of *GREATER*, *LESS*, or *EQUAL* depending on the integers.

val toString : int -> string

val fromString : string -> int option

val scan : StringCvt.radix-> (char,'a) StringCvt.reader -> (int,'a) StringCvt.reader

val fmt : StringCvt.radix -> int -> string

Conversion functions for integer to string and streams. See the Bool structure for descriptions. The *StringCvt.radix* may be one of StringCvt.BIN, StringCvt.OCT, StringCvt.DEC, or StringCvt.HEX for conversion to/from their respective bases.

10.3 The Real Structure

Real numbers in Standard ML, and any other programming language, are approximations for Real numbers in Mathematics. They are always precisely the same.

The Real numbers of Standard ML conform to the underlying architecture's implementation of double precision floating point numbers. Typically, this standard is attributed to the IEEE. More information on Standard ML Reals can be found at http://www.standardml.org/Basis/real.html.

type real

The type of Real numbers. Type *real* are approximations of Real numbers.

val pi : real

val e : real

Constant values for convenience for *pi* and *e*. *e* is the base of natural log values, *ln e = 1*.

val Math.sqrt : real -> real

The square root of a non-negative real yields a real. For negative numbers it yields *nan* which stands for *Not A Number*.

val Math.sin : real -> real

val Math.cos : real -> real

val Math.tan : real -> real

val Math.asin : real -> real

val Math.acos : real -> real

val Math.atan : real -> real

val Math.atan2 : real * real -> real

Various trigonometric functions.

val Math.exp : real -> real

This raises *e* to the specified power.

val Math.pow : real * real -> real

Raises the first argument to the power specified by the second argument.

val Math.ln : real -> real

val Math.log10 : real -> real

Natural and log base 10 functions.

val Math.sinh : real -> real

val Math.cosh : real -> real

val Math.tanh : real -> real

Hyperbolic functions.

val radix : int

The base used in the floating point representation, either 2 or 10.

val precision : int

The number of digits in the mantissa in the base specified by radix.

val maxFinite : real

val minPos : real

val minNormalPos : real

val posInf : real

val negInf : real

Various constant values.

val + : real * real -> real

val - : real * real -> real

val * : real * real -> real

val / : real * real -> real

Normal binary operations. These operators are infix operators.

val *+ : real * real * real -> real

val *- : real * real * real -> real

Multiply by a factor and add a term as in *+(6.0, 5.0, 3.0) which yields 33.0.

val ~ : real -> real

Unary negation.

val abs : real -> real

Absolute value.

val min : real * real -> real

val max : real * real -> real

Binary max and min.

val sign : real -> int

Returns −1 or 1 depending on the sign.

val signBit : real -> bool

True if negative and false otherwise.

val sameSign : real * real -> bool

True if both have same sign.

val copySign : real * real -> real

The result is the first argument with the sign of the second argument.

val compare : real * real -> order

val compareReal : real * real -> IEEEReal.real_order

Returns *GREATER*, *LESS*, or *EQUAL* depending on how the first argument compares to the second. The compareReal has slightly different semantics for unordered real numbers (i.e. *nan*) returning IEEEReal.UNORDERED in those cases.

val < : real * real -> bool

val <= : real * real -> bool

val > : real * real -> bool

val >= : real * real -> bool

val == : real * real -> bool

val != : real * real -> bool

val ?= : real * real -> bool

Binary relational operators. These are infix operators.

val unordered : real * real -> bool

Returns true if one is *nan*.

val isFinite : real -> bool

val isNan : real -> bool

val isNormal : real -> bool

Tests for real values.

val class : real -> IEEEReal.float_class

Returns the IEEE class to which the real belongs.

val fmt : StringCvt.realfmt -> real -> string

val toString : real -> string

val fromString : string -> real option

val scan : (char,'a) StringCvt.reader -> (real,'a) StringCvt.reader

Various real to string or stream conversion functions. See int or bool for details on these functions.

val toManExp : real -> {exp:int, man:real}

val fromManExp : {exp:int, man:real} -> real

val split : real -> {frac:real, whole:real}

val realMod : real -> real

val rem : real * real -> real

Mantissa, exponent and fractional part functions.

val checkFloat : real -> real

Determines if it is a proper real number (not *nan* or *inf*). If it is proper, it returns the argument, otherwise an exception is raised.

val floor : real -> int

val ceil : real -> int

val trunc : real -> int

val round : real -> int

val realFloor : real -> real

val realCeil : real -> real

val realTrunc : real -> real

val realRound : real -> real

Various truncation and rounding functions.

val toInt : IEEEReal.rounding_mode -> real -> int

val toLargeInt : IEEEReal.rounding_mode -> real -> IntInf.int

val fromInt : int -> real

val fromLargeInt : IntInf.int -> real

val toLarge : real -> Real64.real

val fromLarge : IEEEReal.rounding_mode -> Real64.real -> real

val toDecimal : real -> IEEEReal.decimal_approx

val fromDecimal : IEEEReal.decimal_approx -> real

Numeric conversion functions.

10.4 The Char Structure

The following functions are part of the Char structure for the *char* type. The *char* type is separate from the *string* type, covered in the next section. More information can be found at http://www.standardml.org/Basis/char.html.

type char

The character type.

val chr : int -> char

val ord : char -> int

Conversion from and to ASCII values.

val minChar : char

val maxChar : char

val maxOrd : int

Various constants.

val pred : char -> char

val succ : char -> char

Moves through ASCII values.

val < : char * char -> bool

val <= : char * char -> bool

val > : char * char -> bool

val >= : char * char -> bool

Infix relational operators.

val compare : char * char -> order

See other compare functions for a description of the order type.

val scan : (char,'a) StringCvt.reader -> (char,'a) StringCvt.reader

val fromString : String.string -> char option

val toString : char -> String.string

val fromCString : String.string -> char option

val toCString : char -> String.string

Various conversion functions to and from strings.

val contains : string -> char -> bool

val notContains : string -> char -> bool

String search functions.

val isLower : char -> bool

val isUpper : char -> bool

val isDigit : char -> bool

val isAlpha : char -> bool

val isHexDigit : char -> bool

val isAlphaNum : char -> bool

val isPrint : char -> bool

val isSpace : char -> bool

val isPunct : char -> bool

val isGraph : char -> bool

val isCntrl : char -> bool

val isAscii : char -> bool

Character test functions.

val toUpper : char -> char

val toLower : char -> char

Upper and lowercase conversion functions.

10.5 The String Structure

This is the String structure providing functions that operate on strings. Strings are not the same as characters. A string can be exploded into a list of characters, but strings are separate objects from character values. More information can be found at http://www.standardml.org/Basis/string.html.

type string

Character sequences fall under the *string* type in Standard ML. However, strings are NOT lists of characters. There are functions given here to explode and implode a string to and from a list of characters.

val maxSize : int

Maximum string size.

val size : string -> int

Current size of a string.

val sub : string * int -> char

String subscript operator.

val str : char -> string

Convert char to string.

val extract : string * int * int option -> string

val substring : string * int * int -> string

A couple of substring operations. Extract's third argument is either *SOME x* where *x* is the ending lcoation+1 for the substring, or *NONE* to have extract extend to the the end of the string.

val ^ : string * string -> string

Binary string concatenation.

val concat : string list -> string

N-ary string concatenation.

val concatWith : string -> string list -> string

A variation on the other two concatenation operations.

val implode : char list -> string

val explode : string -> char list

Conversion to/from a list of characters to a string. These are useful when writing recursive string functions.

val map : (char -> char) -> string -> string

This is a higher order function that applies a character to character function to each character of a string and returns the string of collected results.

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

Same as map above, but applies a character to string function to each character returning the string of collected strings.

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

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

These two functions return tokens from a string. The char to bool function defines the delimiters of tokens. In other words the first argument is a function that returns true when white space is encountered. The *tokens* function always returns a non-empty token, the *fields* function may return empty tokens.

val isPrefix : string -> string -> bool

val isSubstring : string -> string -> bool

val isSuffix : string -> string -> bool

These are substring dectecting functions.

val compare : string * string -> order

Returns one of *GREATER*, *LESS*, or *EQUAL* depending on the two values being compared.

val collate : (char * char -> order) -> string * string -> order

Compares two strings lexicographically according to the provided character ordering.

val < : string * string -> bool

val <= : string * string -> bool

val > : string * string -> bool

val >= : string * string -> bool

Four infix, normal lexicographical comparisons.

val toString : string -> String.string

Replaces non-printing characters with SML escape character sequences.

val scan : (char,'a) StringCvt.reader -> (string,'a) StringCvt.reader

val fromString : String.string -> string option

val toCString : string -> String.string

val fromCString : String.string -> string option

Various string conversion functions and stream reading functions.

10.6 The List Structure

This is the List structure for the *list* polymorphic datatype in SML. More information can be found at http://www.standardml.org/Basis/list.html.

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

A list is formed from an element and a list. It is a recursive data structure with O(n) access to any element of the list. This should not be confused with an array that provides O(1) element access. The :: is called *cons* and stands for list construction or constructor. It forms a list from an element, *e*, and a list, *lst* as in *e::lst*. The *nil* keyword is used to represent an empty list. Writing *[]* is equivalent to *nil* in Standard ML. Lists in Standard ML must be homogenous, containing all the same type of elements.

exception Empty

Raised as necessary by various functions should an empty list be used as an argument. Not raised unless necessary.

val null : 'a list -> bool

Returns true if the given list is empty.

val hd : 'a list -> 'a

val tl : 'a list -> 'a list

hd e::lst returns *e* while *tl e::lst* returns lst. *hd* is short for head of the list and *tl* is short for tail of the list.

val last : 'a list -> 'a

Returns the last element of the given list. Raise *Empty* if given an empty list.

val getItem : 'a list -> ('a * 'a list) option

Returns SOME of the head and tail of a list or *NONE* if the list is empty. Calling *getItem (e::lst)* returns *SOME (e,lst)*.

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

Returns the nth item of the list (zero based) and raise *Subscript* if the list is too short.

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

Returns the first *i* elements of a list given a list and *i*. Raises *Subscript* if the list is too short.

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

Returns the rest of a list after the first *i* elements. Raises *Subscript* if the list is too short.

val length : ‘a list -> int

Returns the length of a list.

val rev : ‘a list -> ‘a list

Returns the reverse of a list.

val @ : ‘a list * ‘a list -> ‘a list

This is list concatenation, not to be confused with :: which is list construction. This is an infix operator. So *[1, 2, 3]@[4, 5, 6]* is legal and so is *1 ::[2, 3, 4, 5, 6]* which both yield the same result.

val concat : ‘a list list -> ‘a list

This takes a list of lists of all the same element and concatenates each of the lists together returning one big list of all the elements.

val revAppend : ‘a list * ‘a list -> ‘a list

Reverses the first list and appends it to the second.

val app : (‘a -> unit) -> ‘a list -> unit

This function applies the first argument, a function with a side-effect, to each element of a list. The *unit* type is another name for the empty tuple (i.e. *()*) which is the return type of many functions that have side-effects.

val map : (‘a -> ‘b) -> ‘a list -> ‘b list

The map function applies a function to each element of a list, building a new list of all the results.

val mapPartial : (‘a -> ‘b option) -> ‘a list -> ‘b list

This is like *map* except that if *NONE* is returned by the function, it is omitted from the resulting list. Only values of *SOME val* are included in the final result.

val find : (‘a -> bool) -> ‘a list -> ‘a option

Given a predicate function and a list, the *find* function returns either *SOME val* for the found value or *NONE* indicating the predicate did not return true for any element of the list.

val filter : (‘a -> bool) -> ‘a list -> ‘a list

This function returns a new list of all elements of the list that satisfy the provided predicate function.

val partition : (‘a -> bool) -> ‘a list -> ‘a list * ‘a list

This function returns a tuple where the first list consists of all elements that satisfy the predicate function and the second is comprised of the elements that did not satisfy the predicate.

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

This function applies a provided function to each element and an initial value, folding all the results into one finals result. This function is called *foldr* because it is right-associative. Here is an example of calling *foldr*.

```
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
```

The use of *op* - in the example transforms the infix - operator to a prefix function. The example computed (1 – (2 – (3 – (4 – 0)))). If the list is empty then the initial value, the second argument, is returned.

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

This function is the left-associative analog of *foldr* meaning that the initial value is applied along with the first element of the list and that result applied along with the second element of the list and so on. For example,

```
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```

The example computed (4 – (3 – (2 – (1 – 0)))). If the list is empty, then the initial value, the second argument, is returned.

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

Given a predicate function, *exists* returns true if the predicate function evaluates to true for at least one element of the list.

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

Given a predicate function, *all* returns true if the predicate function evaluates to true for all elements of the list.

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

Builds a list of *n* elements. The *n* is the first argument to *tabulate*. The each element is generated by passing one of 0 to *n-1* to the second argument, a function. Raises *Size* if there are less than *n* elements in the list.

```
- List.tabulate(5,fn x => x + 1);
val it = [1,2,3,4,5] : int list
```

val collate : ('a * 'a -> order) -> 'a list * 'a list -> order

This performs a lexicographical comparison of two lists according to the provided ordering function for each element of the lists. Returns one of *LESS*, *GREATER*, or *EQUAL*.

10.7 The Array Structure

Arrays are mutable sequences that provide O(1) lookup and assignment complexities. Lists are immutable and provide O(n) lookup time. Lists are immutable so item assignment is not possible in a list. Since arrays are mutable, many of the functions on arrays return *unit* the type of *()* which is used as the return type of mutating functions in Standard ML. More information can be found at http://www.standardml.org/Basis/array.html.

type ‘a array

Arrays must be homogeneous in Standard ML, comprised of all the same type of elements.

val maxLen : int

Maximum size of an array.

val array : int * ‘a -> ‘a array

Build an array with size *n*, the first argument, and all elements initialized to the value of *a*, the second argument.

val fromList : ‘a list -> ‘a array

Build an array from a list.

val tabulate : int * (int -> ‘a) -> ‘a array

See List.tabulate.

val length : ‘a array -> int

The length of an array.

val sub : ‘a array * int -> ‘a

The O(1) element retrieval operation not provided by lists in Standard ML.

val update : ‘a array * int * ‘a -> unit

The array element assignment operation, a O(1) mutating operation.

val vector : ‘a array -> ‘a vector

Builds a vector from an array.

val copy : {di:int, dst:’a array, src:’a array} -> unit

val copyVec : {di:int, dst:’a array, src:’a vector} -> unit

Copy utility functions.

val appi : (int * ‘a -> unit) -> ‘a array -> unit

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

Applies a function to an array. The first supplies the function with *i* provided as the first argument where *i* is the index of the element. The second applies the function to each element of the vector without knowledge of its location. The function applied would have some side-effect.

val modifyi : (int * 'a -> 'a) -> 'a array -> unit

val modify : ('a -> 'a) -> 'a array -> unit

Applies a function to an array. The first supplies the function with *i* provided as the first argument where *i* is the index of the element. The second applies the function to each element of the vector without knowledge of its location. The function applied results in a value that replaces the value in the array at the same location.

val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a array -> 'b

val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a array -> 'b

val foldl : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b

val foldr : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b

The fold equivalents (see List.fold functions) for arrays. The foldli and foldri functions provide the index of the value in addition to the value at each element of the array.

val findi : (int * 'a -> bool) -> 'a array -> (int * 'a) option

val find : ('a -> bool) -> 'a array -> 'a option

val exists : ('a -> bool) -> 'a array -> bool

val all : ('a -> bool) -> 'a array -> bool

val collate : ('a * 'a -> order) -> 'a array * 'a array -> order

All similar to List functions. See the List equivalents for explanations.

10.8 The TextIO Structure

This is a subset of the entire TextIO structure. Detailed descriptions of all functions can be found on the Basis Library website at http://www.standardml.org/Basis/text-io.html.

type instream

type outstream

Standard ML supports stream operations for both input and output streams.

val input : instream -> vector

val input1 : instream -> elem option

val inputN : instream * int -> vector

val inputAll : instream -> vector

These are blocking input functions. The *input* returns an empty vector if the input stream is closed, otherwise returning one or more items in the stream. The *input1* reads just one element from the stream and returns *NONE* if the input stream is closed. The *inputN* returns at most *n* items. The *inputAll* returns everything up to the end of stream.

val canInput : instream * int -> int option

val lookahead : instream -> elem option

These two functions look at the state of the stream. They are useful in making input decisions.

val closeIn : instream -> unit

val endOfStream : instream -> bool

The *closeIn* function closes a stream and *endOfStream* closes the given stream.

val output : outstream * vector -> unit

val output1 : outstream * elem -> unit

Writes all elements of a vector and one element, respectively, to a stream.

val flushOut : outstream -> unit

val closeOut : outstream -> unit

Before input is read, it may be necessary to flush output if a prompt is printed for instance. Otherwise, the prompt may not appear on the screen. The *closeOut* function closes an output stream.

val inputLine : instream -> string option

Reads an input *line* and returns either *SOME line* or *NONE*.

val outputSubstr : outstream * substring -> unit

Writes a substring.

val openIn : string -> instream

Opens an input stream for reading. The argument is a filename.

val openString : string -> instream

Opens a string stream for reading.

val openOut : string -> outstream

Opens an output stream for writing. The argument is a filename.

val openAppend : string -> outstream

Opens an output stream for writing. The argument is a filename. If the file exists, the data written will be appended to the end of the file.

val stdIn : instream

val stdOut : outstream

val stdErr : outstream

These are the names of the default input, output, and error streams supplied with every program. They are precreated objects.

val print : string -> unit

Prints to standard output the given string.

val scanStream : ((elem,StreamIO.instream) StringCvt.reader -> instream -> ('a,StreamIO.instream) StringCvt.reader) -> instream -> 'a option

Uses a stream and converts it to an imperative stream where conversions can be done while reading input. See the Basis Library for a more complete description of how this works.