13 Built-in types, constructors and exceptions

The following types, constructors, and exceptions are available in the initial environment, of the interactive system as well as files compiled with the batch compiler `mosmlc` or the `compile` function.

Built-in types

Type	Values	Admits equality	Constructors and constants
`'a array`	Arrays	yes	
`bool`	Booleans	yes	`false`, `true`
`char`	Characters	yes	`#"a"`, `#"b"`, ···
`exn`	Exceptions	no	
`'a frag`	Quotation fragments	if `'a` does	`QUOTE`, `ANTIQUOTE`
`int`	Integers	yes	`241`, `0xF1`, ···
`'a list`	Lists	if `'a` does	`nil`, `::`
`'a option`	Optional results	if `'a` does	`NONE`, `SOME`
`order`	Comparisons	yes	`LESS`, `EQUAL`, `GREATER`
`real`	Floating-point numbers	yes	
`'a ref`	References	yes	`ref`
`string`	Strings	yes	
`substring`	Substrings	no	
`unit`	The empty tuple `()`	yes	
`'a vector`	Vectors	if `'a` does	
`word`	Words (31-bit)	yes	`0w241`, `0wxF1`, ···
`word8`	Bytes (8 bit)	yes	`0w241`, `0wxF1`, ···

Built-in exception constructors

```
Bind  Chr  Domain  Div  Fail  Graphic  Interrupt  Io
Match  Option  Ord  Overflow  Size  Subscript  SysErr
```

14 Built-in variables and functions

For each variable or function we list its type and meaning. Some built-in identifiers are overloaded; this is specified using *overloading classes*. For instance, an identifier whose type involves the overloading class `realint` stands for two functions: one in which `realint` (in the type) is consistently replaced by `int`, and another in which `realint` is consistently replaced by `real`. The overloading classes are:

Overloading class	Corresponding base types
`realint`	`int`, `real`
`wordint`	`int`, `word`, `word8`
`num`	`int`, `real`, `word`, `word8`
`numtxt`	`int`, `real`, `word`, `word8`, `char`, `string`

When the context does not otherwise resolve the overloading, it defaults to `int`.

Nonfix identifiers in the initial environment

id	type	effect	exception
`~`	`realint -> realint`	arithmetic negation	`Overflow`
`!`	`'a ref -> 'a`	dereference	
`abs`	`realint -> realint`	absolute value	`Overflow`
`app`	`('a -> unit) -> 'a list -> unit`	apply to all elements	
`ceil`	`real -> int`	round towards $+\infty$	`Overflow`
`chr`	`int -> char`	character with number	`Chr`
`concat`	`string list -> string`	concatenate strings	`Size`
`explode`	`string -> char list`	list of characters in string	
`false`	`bool`	logical falsehood	
`floor`	`real -> int`	round towards $-\infty$	`Overflow`
`foldl`	`('a*'b->'b)->'b->'a list->'b`	fold from left to right	
`foldr`	`('a*'b->'b)->'b->'a list->'b`	fold from right to left	
`hd`	`'a list -> 'a`	first element	`Empty`
`help`	`string -> unit`	simple help utility	
`ignore`	`'a -> unit`	discard argument	
`implode`	`char list -> string`	make string from characters	`Size`
`length`	`'a list -> int`	length of list	
`map`	`('a -> 'b) -> 'a list -> 'b list`	map over all elements	
`nil`	`'a list`	empty list	
`not`	`bool -> bool`	logical negation	
`null`	`'a list -> bool`	true if list is empty	
`ord`	`char -> int`	number of character	
`print`	`string -> unit`	print on standard output	
`real`	`int -> real`	`int` to `real`	
`ref`	`'a -> 'a ref`	create reference value	
`rev`	`'a list -> 'a list`	reverse list	
`round`	`real -> int`	round to nearest integer	`Overflow`
`size`	`string -> int`	length of string	
`str`	`char -> string`	create one-character string	
`substring`	`string * int * int -> string`	get substring $(s, first, len)$	`Subscript`
`tl`	`'a list -> 'a list`	tail of list	`Empty`
`true`	`bool`	logical truth	
`trunc`	`real -> int`	round towards 0	`Overflow`
`vector`	`'a list -> 'a vector`	make vector from list	`Size`

Infixed identifiers in the initial environment

id	type	effect	exception
Infix precedence 7:			
`/`	`real * real -> real`	floating-point quotient	`Div, Overflow`
`div`	`wordint * wordint -> wordint`	quotient (round towards $-\infty$)	`Div, Overflow`
`mod`	`wordint * wordint -> wordint`	remainder (of `div`)	`Div, Overflow`
`*`	`num * num -> num`	product	`Overflow`
Infix precedence 6:			
`+`	`num * num -> num`	sum	`Overflow`
`-`	`num * num -> num`	difference	`Overflow`
`^`	`string * string -> string`	concatenate	`Size`
Infix precedence 5:			
`::`	`'a * 'a list -> 'a list`	cons onto list (R)	
`@`	`'a list * 'a list -> 'a list`	append lists (R)	
Infix precedence 4:			
`=`	`''a * ''a -> bool`	equal to	
`<>`	`''a * ''a -> bool`	not equal to	
`<`	`numtxt * numtxt -> bool`	less than	
`<=`	`numtxt * numtxt -> bool`	less than or equal to	
`>`	`numtxt * numtxt -> bool`	greater than	
`>=`	`numtxt * numtxt -> bool`	greater than or equal to	
Infix precedence 3:			
`:=`	`'a ref * 'a -> unit`	assignment	
`o`	`('b->'c) * ('a->'b) -> ('a->'c)`	function composition	
Infix precedence 0:			
`before`	`'a * 'b -> 'a`	return first argument	

Built-in functions available only in the interactive system (unit `Meta`)

id	type	effect	exception
`compile`	`string -> unit`	compile unit (U.sig or U.sml) (in *structure* mode)	`Fail`
`compileStructure`	`string list -> string -> unit`	In context $U_1, \ldots, U_n$, compile unit (U.sig or U.sml) (in *structure* mode)	`Fail`
`compileToplevel`	`string list -> string -> unit`	In context $U_1, \ldots, U_n$, compile unit (U.sig or U.sml) (in *toplevel* mode)	`Fail`
`conservative`	`unit -> unit`	deprecate all Moscow ML extensions	
`installPP`	`(ppstream->'a->unit) -> unit`	install prettyprinter	
`liberal`	`unit -> unit`	accept all Moscow ML extensions	
`load`	`string -> unit`	load unit U and any units it needs	`Fail`
`loaded`	`unit -> string list`	return list of loaded units	
`loadOne`	`string -> unit`	load unit U (only)	`Fail`
`loadPath`	`string list ref`	search path for `load`, `loadOne`, `use`	
`orthodox`	`unit -> unit`	reject any Moscow ML extensions	
`printVal`	`'a -> 'a`	print value on `stdOut`	
`printDepth`	`int ref`	limit printed data depth	
`printLength`	`int ref`	limit printed list and vector length	
`quietdec`	`bool ref`	suppress prompt and responses	
`quit`	`unit -> unit`	quit the interactive system	
`quotation`	`bool ref`	permit quotations in source code	
`system`	`string -> int`	execute operating system command	
`use`	`string -> unit`	read declarations from file	
`valuepoly`	`bool ref`	adopt value polymorphism	
`verbose`	`bool ref`	permit feedback from `compile`	

- The Moscow ML Owner's Manual describes how to use `compile`, `compileStructure`, `compileToplevel` and `load` to perform separate compilation, and how to use quotations. Evaluating `load` U automatically loads any units needed by U, and does nothing if U is already loaded; whereas `loadOne` U fails if any unit needed by U is not loaded, or if U is already loaded. The `loadPath` variable determines where `load`, `loadOne`, and `use` will look for files. The commands `orthodox`, `conservative` and `liberal` cause Moscow ML to enforce, monitor or ignore compliance to Standard ML.

15 List of all library modules

A table of Mosml ML's predefined library modules is given on page 20. The status of each module is indicated as follows:

`S`	:	the module belongs to the SML Basis Library.
`D`	:	the module is preloaded by default.
`F`	:	the module is loaded when option `-P full` is specified.
`N`	:	the module is loaded when option `-P nj93` is specified.
`O`	:	the module is loaded when option `-P sml90` is specified.

To find more information about the Moscow ML library:

- Typing `help "lib";` in a `mosml` session gives a list of all library modules.
- Typing `help "`*module*`";` in a `mosml` session gives information about library module *module*.
- Typing `help "`*id*`";` in a `mosml` session gives information about identifier *id*, regardless which library module(s) it is defined in.
- In your Moscow ML installation, consult the library documentation (in printable format):

  ```
  mosml/doc/mosmllib.ps
  mosml/doc/mosmllib.pdf
  ```

- In your Moscow ML installation, you may find the same documentation in HTML-format at

  ```
  mosml/doc/mosmllib/index.html
  ```

- On the World Wide Web the same pages are online at

  ```
  http://www.dina.kvl.dk/~sestoft/mosmllib/index.html
  ```

  If you do not have the HTML pages, you may download them from the Moscow ML home page.

Library module	Description	Status
Array	Mutable polymorphic arrays	SDF
Array2	Two-dimensional arrays	S
Arraysort	Array sorting (quicksort)	
BasicIO	Input-output as in SML'90	DF
Binarymap	Binary tree implementation of finite maps	
Binaryset	Binary tree implementation of finite sets	
BinIO	Binary input-output streams (imperative)	S F
Bool	Booleans	S F
Byte	Conversions between Word8 and Char	S F
Callback	Registering ML values for access from C code	SDF
Char	Characters	SDF
CharArray	Mutable arrays of characters	S F
CharVector	Immutable character vectors (that is, strings)	S F
CommandLine	Program name and arguments	S F
Date	From time points to dates and vice versa	S F
Dynarray	Dynamic arrays	
Dynlib	Dynamic linking with C	
FileSys	File system interface	S F
Gdbm	Persistent hash tables of strings (GNU gdbm)	
Gdimage	Generation of PNG images (Boutell's GD package)	
General	Various top-level primitives	SD
Help	On-line help	DFNO
Int	Integer arithmetic and comparisons	S F
Intmap	Finite maps from integers	
Intset	Finite sets of integers	
List	Lists	SDFNO
ListPair	Pairs of lists	S F
Listsort	List sorting (mergesort)	
Location	Error reporting for lexers and parsers	
Math	Trigonometric and transcendental functions	S F
Meta	Functions specific to the interactive system	
Mosml	Various Moscow ML utilities	F
Mosmlcgi	Utilities for writing CGI programs	
Mosmlcookie	manipulating cookies in CGI programs	
Msp	Utilities for efficiently generating HTML code	
Mysql	Interface to the MySQL database server	
NJ93	Top-level compatibility with SML/NJ 0.93	N
OS	Operating system interface	S F
Option	Partial functions	SDFNO
Path	File pathnames	S F
Polygdbm	Polymorphic persistent hash tables (GNU gdbm)	
Polyhash	Polymorphic hash tables	
Postgres	Interface to the PostgreSQL database server	
PP	General prettyprinters	F
Process	Process interface	S F
Random	Generation of pseudo-random numbers	
Real	Real arithmetic and comparisons	S F
Regex	Regular expressions as in POSIX 1003.2	
Signal	Unix signals	S
SML90	Top-level compatibility with 1990 Definition	S O
Socket	Interface to sockets	
Splaymap	Splay-tree implementation of finite maps	
Splayset	Splay-tree implementation of finite sets	
String	String utilities	SDF
StringCvt	Conversion to and from strings	S F
Substring	Scanning of substrings	S F
TextIO	Text input-output streams (imperative)	SDF
Time	Time points and durations	S F
Timer	Timing operations	S F
Unix	Starting concurrent subprocesses under Unix	S
Vector	Immutable vectors	SDF
Weak	Arrays of weak pointers	
Word	Unsigned 31-bit integers ('machine words')	S F
Word8	Unsigned 8-bit integers (bytes)	S F
Word8Array	Mutable arrays of unsigned 8-bit integers	S F
Word8Vector	Immutable vectors of unsigned 8-bit integers	S F

16 The preloaded library modules

The following libraries are preloaded by default: `Array`, `Char`, `List`, `String`, `TextIO`, and `Vector`. To load any other library *lib*, evaluate `load "`*lib*`"` in the interactive system.

Notation in the tables below

f	functional argument
n	integer
p	predicate of type (`'a -> bool`)
s	string
xs, *ys*	lists

List manipulation functions (module `List`)

id	type	effect
`@`	`'a list * 'a list -> 'a list`	append
`all`	`('a -> bool) -> 'a list -> bool`	if *p* true of all elements
`app`	`('a -> unit) -> 'a list -> unit`	apply *f* to all elements
`concat`	`'a list list -> 'a list`	concatenate lists
`drop`	`'a list * int -> 'a list`	drop *n* first elements
`exists`	`('a -> bool) -> 'a list -> bool`	if *p* true of some element
`filter`	`('a -> bool) -> 'a list -> 'a list`	the elements for which *p* is true
`find`	`('a -> bool) -> 'a list -> 'a option`	first element for which *p* is true
`foldl`	`('a * 'b -> 'b) -> 'b -> 'a list -> 'b`	fold from left to right
`foldr`	`('a * 'b -> 'b) -> 'b -> 'a list -> 'b`	fold from right to left
`hd`	`'a list -> 'a`	first element
`last`	`'a list -> 'a`	last element
`length`	`'a list -> int`	number of elements
`map`	`('a -> 'b) -> 'a list -> 'b list`	results of applying *f* to all elements
`mapPartial`	`('a -> 'b option) -> 'a list -> 'b list`	list of the non-`NONE` results of *f*
`nth`	`'a list * int -> 'a`	*n*'th element (0-based)
`null`	`'a list -> bool`	true if list is empty
`partition`	`('a->bool)->'a list->'a list*'a list`	compute (true for *p*, false for *p*)
`rev`	`'a list -> 'a list`	reverse list
`revAppend`	`'a list * 'a list -> 'a list`	compute (`rev` *xs*) `@` *ys*
`tabulate`	`int * (int -> 'a) -> 'a list`	compute [*f*`(0)`,$\cdots$,*f*`(n-1)`]
`take`	`'a list * int -> 'a list`	take *n* first elements
`tl`	`'a list -> 'a list`	tail of list

- For a more detailed description, type `help "List";` or see file `mosml/lib/List.sig`. The `List` module is loaded and partially opened in the initial environment, making the following functions available: `@`, `app`, `foldl`, `foldr`, `hd`, `length`, `map`, `null`, `rev`, `tl`.

Built-in values and functions for text-mode input/output (module `TextIO`)

id	type	effect
`closeIn`	`instream -> unit`	close input stream
`closeOut`	`outstream -> unit`	close output stream
`endOfStream`	`instream -> bool`	true if at end of stream
`flushOut`	`outstream -> unit`	flush output to consumer
`input`	`instream -> string`	input some characters
`input1`	`instream -> char option`	input one character
`inputN`	`instream * int -> string`	input at most *n* characters
`inputAll`	`instream -> string`	input all available characters
`inputLine`	`instream -> string`	read up to (and including) next end of line
`inputNoBlock`	`instream -> string option`	read, if possible without blocking
`lookahead`	`instream -> char option`	get next char non-destructively
`openAppend`	`string -> outstream`	open file for appending to it
`openIn`	`string -> instream`	open file for input
`openOut`	`string -> outstream`	open file for output
`output`	`outstream * string -> unit`	write string to output stream
`output1`	`outstream * char -> unit`	write character to output stream
`print`	`string -> unit`	write to standard output
`stdErr`	`outstream`	standard error output stream
`stdIn`	`instream`	standard input stream
`stdOut`	`outstream`	standard output stream

- For a more detailed description, see file `mosml/lib/TextIO.sig`, or type `help "TextIO";`.
- For the corresponding structure `BinIO` for binary (untranslated) input and output, see `help "BinIO"`.

String manipulation functions (module `String`)

id	type	effect
`^`	`string * string -> string`	concatenate strings
`collate`	`(char*char->order)->string*string->order`	compare strings
`compare`	`string * string -> order`	compare strings
`concat`	`string list -> string`	concatenate list of strings
`explode`	`string -> char list`	character list from string
`extract`	`string * int * int option -> string`	get substring or tail
`fields`	`(char -> bool) -> string -> string list`	find (possibly empty) fields
`fromCString`	`string -> string option`	parse C escape sequences
`fromString`	`string -> string option`	parse ML escape sequences
`implode`	`char list -> string`	string from character list
`isPrefix`	`string -> string -> bool`	prefix test
`map`	`(char -> char) -> string -> string`	map over characters
`maxSize`	`int`	maximal size of a string
`size`	`string -> int`	length of string
`str`	`char -> string`	make one-character string
`sub`	`string * int -> char`	*n*'th character (0-based)
`substring`	`string * int * int -> string`	get substring $(s, first, len)$
`toCString`	`string -> string`	make C escape sequences
`toString`	`string -> string`	make ML escape sequences
`tokens`	`(char -> bool) -> string -> string list`	find (non-empty) tokens
`translate`	`(char -> string) -> string -> string`	apply f and concatenate

- In addition, the overloaded comparison operators `<`, `<=`, `>`, `>=` work on strings.
- For a more detailed description, see file `mosml/lib/String.sig`, or type `help "String";`.

Vector manipulation functions (module `Vector`)

Type `'a vector` is the type of one-dimensional, immutable, zero-based constant time access vectors with elements of type `'a`. Type `'a vector` admits equality if `'a` does.

id	type	effect
`app`	`('a -> unit) -> 'a vector -> unit`	apply *f* left-right
`appi`	`(int * 'a -> unit) -> 'a vector * int * int option -> unit`	
`concat`	`'a vector list -> 'a vector`	concatenate vectors
`extract`	`'a vector * int * int option -> 'a vector`	extract a subvector or tail
`foldl`	`('a * 'b -> 'b) -> 'b -> 'a vector -> 'b`	fold *f* left-right
`foldli`	`(int * 'a * 'b -> 'b) -> 'b -> 'a vector*int*int option -> 'b`	
`foldr`	`('a * 'b -> 'b) -> 'b -> 'a vector -> 'b`	fold *f* right-left
`foldri`	`(int * 'a * 'b -> 'b) -> 'b -> 'a vector*int*int option -> 'b`	
`fromList`	`'a list -> 'a vector`	make vector from the list
`length`	`'a vector -> int`	length of the vector
`maxLen`	`int`	maximal vector length
`sub`	`'a vector * int -> 'a`	*n*'th element (0-based)
`tabulate`	`int * (int -> 'a) -> 'a vector`	vector of $f(0),\cdots,f(n\text{-}1)$

- For a more detailed description, type `help "Vector";` or see file `mosml/lib/Vector.sig`.

Array manipulation functions (module `Array`)

Type `'a array` is the type of one-dimensional, mutable, zero-based constant time access arrays with elements of type `'a`. Type `'a array` admits equality regardless whether `'a` does.

id	type	effect
`app`	`('a -> unit) -> 'a array -> unit`	apply *f* left-right
`appi`	`(int * 'a -> unit) -> 'a array * int * int option -> unit`	
`array`	`int * 'a -> 'a array`	create and initialize array
`copy`	`{src : 'a array, si : int, len : int option, dst : 'a array, di : int} -> unit`	copy subarray to subarray
`copyVec`	`{src : 'a vector, si : int, len : int option, dst : 'a array, di : int} -> unit`	copy subvector to subarray
`extract`	`'a array * int * int option -> 'a vector`	extract subarray to vector
`foldl`	`('a * 'b -> 'b) -> 'b -> 'a array -> 'b`	fold left-right
`foldli`	`(int * 'a * 'b -> 'b) -> 'b -> 'a array * int * int option -> 'b`	
`foldr`	`('a * 'b -> 'b) -> 'b -> 'a array -> 'b`	fold right-left
`foldri`	`(int * 'a * 'b -> 'b) -> 'b -> 'a array * int * int option -> 'b`	
`fromList`	`'a list -> 'a array`	make array from the list
`length`	`'a array -> int`	length of the array
`maxLen`	`int`	maximal array length
`modify`	`('a -> 'a) -> 'a array -> unit`	apply *f* and update
`modifyi`	`(int * 'a -> 'a) -> 'a array * int * int option -> unit`	
`sub`	`'a array * int -> 'a`	*n*'th element (0-based)
`tabulate`	`int * (int -> 'a) -> 'a array`	array of $f(0),\cdots,f(n\text{-}1)$
`update`	`'a array * int * 'a -> unit`	set *n*'th element (0-based)

- For a more detailed description, type `help "Array";` or see file `mosml/lib/Array.sig`. The `Array` module is loaded but not opened in the initial environment.

Character manipulation functions (module `Char`)

id	type	effect	exception
`chr`	`int -> char`	from character code to character	`Chr`
`compare`	`char * char -> order`	compare character codes	
`contains`	`string -> char -> bool`	contained in string	
`fromCString`	`string -> char option`	parse C escape sequence	
`fromString`	`string -> char option`	parse SML escape sequence	
`isAlpha`	`char -> bool`	alphabetic ASCII character	
`isAlphaNum`	`char -> bool`	alphanumeric ASCII character	
`isAscii`	`char -> bool`	seven-bit ASCII character	
`isCntrl`	`char -> bool`	ASCII control character	
`isDigit`	`char -> bool`	decimal digit	
`isGraph`	`char -> bool`	printable and visible ASCII	
`isHexDigit`	`char -> bool`	hexadecimal digit	
`isLower`	`char -> bool`	lower case alphabetic (ASCII)	
`isPrint`	`char -> bool`	printable ASCII (including space)	
`isPunct`	`char -> bool`	printable, but not space or alphanumeric	
`isSpace`	`char -> bool`	space and lay-out (HT, CR, LF, VT, FF)	
`isUpper`	`char -> bool`	upper case alphabetic (ASCII)	
`maxChar`	`char`	last character (in `<=` order)	
`maxOrd`	`int`	largest character code	
`minChar`	`char`	first character (in `<=` order)	
`notContains`	`string -> char -> bool`	not in string	
`ord`	`char -> int`	from character to character code	
`pred`	`char -> char`	preceding character	`Chr`
`succ`	`char -> char`	succeding character	`Chr`
`toLower`	`char -> char`	convert to lower case (ASCII)	
`toCString`	`char -> string`	make C escape sequence	
`toString`	`char -> string`	make SML escape sequence	
`toUpper`	`char -> char`	convert to upper case (ASCII)	

- In addition, the overloaded comparison operators `<`, `<=`, `>`, `>=` work on the `char` type.
- For a more detailed description, type `help "Char";` or see file `mosml/lib/Char.sig`. The `Char` module is loaded and partially opened in the initial environment, making the functions `chr` and `ord` available.