Moscow ML Language Overview

Version 1.43 of April 1998

Sergei Romanenko
Russian Academy of Sciences
Moscow, Russia

Peter Sestoft
Royal Veterinary and Agricultural University
Copenhagen, Denmark

This is a compact reference to the language implemented by Moscow ML, a subset of Standard ML. For reference material on Standard ML, see Milner, Tofte, Harper and MacQueen: *The Definition of Standard ML*, The MIT Press 1997. Moscow ML implements parts of the SML Basis Library. Section 14 of this manual lists all structure, type, constructor, value, function, and exception identifiers defined by Moscow ML.

Acknowledgement: The present document owes a lot to the *Definition*.

Contents

The Moscow ML home page is `http://www.dina.kvl.dk/~sestoft/mosml.html`

1 Reserved words

```
abstype  and  andalso  as  case  do  datatype  else
end  exception  fn  fun  handle  if  in  infix  infixr  let
local  nonfix  of  op  open  orelse  raise  rec  sig  signature
struct  structure  then  type  val  with  withtype  while
(  )  [  ]  {  }  ,  :  :>  ;  ...  _  |  =  =>  ->  #
```

2 Comments

A comment is any character sequence within comment brackets `(*` and `*)` in which comment brackets are properly nested.

3 Special constants

Integer constants

Examples:	`0`	`~0`	`4`	`~04`	`999999`	`0xFFFF`	`~0x1ff`
Non-examples:	`0.0`	`~0.0`	`4.0`	`1E0`	`-317`	`0XFFFF`	`-0x1ff`

Real constants

Examples:	`0.7`	`~0.7`	`3.32E5`	`3E~7`	`~3E~7`	`3e~7`	`~3e~7`
Non-examples:	`23`	`.3`	`4.E5`	`1E2.0`	`1E+7`	`1E-7`	

Word constants

Examples:	`0w0`	`0w4`	`0w999999`	`0wxFFFF`	`0wx1ff`	
Non-examples:	`0w0.0`	`~0w4`	`-0w4`	`0w1E0`	`0wXFFFF`	`0WxFFFF`

String constants

A string constant is a sequence, between quotes ("), of zero or more printable characters, spaces, or escape sequences. An escape sequence starts with the escape character \ and stands for a character sequence:

`\a`	A single character interpreted by the system as alert (BEL, ASCII 7).
`\b`	Backspace (BS, ASCII 8).
`\t`	Horisontal tab (HT, ASCII 9).
`\n`	Linefeed, also known as newline (LF, ASCII 10).
`\v`	Vertical tab (VT, ASCII 11).
`\f`	Form feed (FF, ASCII 12).
`\r`	Carriage return (CR, ASCII 13).
`\^`c	The control character c, where c may be any character with ASCII code 64–95 (`@` to `_`). The ASCII code of `\^`c is 64 less than that of c.
`\`ddd	The character with code ddd (3 decimal digits denoting an integer 0–255).
`\"`	"
`\\`	\
`\`$f\cdots f$`\`	This sequence is ignored, where $f\cdots f$ stands for a sequence of one or more formatting characters (such as space, tab, newline, form-feed).

Character constants

A character constant consists of the symbol # immediately followed by a string constant of length one.

Examples:	`#"a"`	`#"\n"`	`#"\^Z"`	`#"\255"`	`#"\""`
Non-examples:	`# "a"`	`#c`	`#"""`		

4 Identifiers

- **alphanumeric:** a sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime;
- **symbolic:** any non-empty sequence of the following symbols:
  `! % & $ # + - / : < = > ? @ \ ~ ‘ ^ | *`

Reserved words (Section 1) are excluded. This means that for example `#` and `|` are not identifiers, but `##` and `|=|` are identifiers. There are several classes of identifiers:

var	(value variables)	long
con	(value constructors)	long
excon	(exception constructors)	long
tyvar	(type variables)	
tycon	(type constructors)	long
lab	(record labels)	
unitid	(unit identifiers)	

- A type variable `'a` is an alphanumeric identifier starting with a prime.
- A label lab is an identifier, or a positive integral numeral `1 2 3` ... not starting with `0`.
- For each identifier class X marked 'long' above there is a class longX of long identifiers, which may have a qualifier consisting of a unit identifier followed by a dot '.' :

longx	::=	*x*	identifier
		unitid . *x*	qualified identifier

5 Infixed operators

An identifier may be given infix status by the `infix` or `infixr` directive, which may occur as a declaration. If identifier *id* has infix status, then exp_1 *id* exp_2 may occur, in parentheses if necessary, wherever the application *id*`(`exp_1`,` exp_2`)` or *id*`{1=`exp_1`,2=`exp_2`}` would otherwise occur. Infix identifiers in patterns are analogous. On the other hand, an occurrence of a qualified identifier, or any identifier prefixed by `op`, is treated as non-infixed. The form of the fixity directives is as follows ($n \geq 1$):

`infix`	$\langle d \rangle$	$id_1 \cdots id_n$	left associative
`infixr`	$\langle d \rangle$	$id_1 \cdots id_n$	right associative
`nonfix`		$id_1 \cdots id_n$	non-associative

where $\langle d \rangle$ is an optional decimal digit d indicating binding precedence. A higher value of d indicates tighter binding; the default is `0`. Fixity directives are subject to the usual scope rules governing visibility of identifiers declared inside `let` and `local`.

Mixed left-associative operators of the same precedence associate to the left, mixed right-associative operators of the same precedence associate to the right, and it is illegal to mix left- and right-associative operators of the same precedence.

6 Notational conventions used in the grammar

- Each syntax class is defined by a list of alternatives, one alternative on each line. An empty phrase is represented by an empty line.
- The brackets $\langle$ and $\rangle$ enclose optional phrases.
- For any syntax class X (over which x ranges) we define the syntax class Xseq (over which *xseq* ranges) as follows:

xseq	::=	*x*	(singleton sequence)
			(empty sequence)
		`(`x_1`,` $\cdots$ `,` x_n`)`	(sequence, $n \geq 1$)

- Alternative phrases are listed in order of decreasing precedence.
- L and R indicate left and right association.
- The syntax of types binds more tightly than that of expressions.
- Each iterated construct (e.g. *match*) extends as far to the right as possible. Hence a `case` inside a `case`, `fn`, or `fun` may have to be enclosed in parentheses.

7 Grammar for the Standard ML Core language

Expressions and matches

exp	::=	*infexp*	
		exp `:` *ty*	type constraint (L)
		exp_1 `andalso` exp_2	sequential conjunction
		exp_1 `orelse` exp_2	sequential disjunction
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`if` exp_1 `then` exp_2 `else` exp_3	conditional
		`while` exp_1 `do` exp_2	iteration
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function expression
infexp	::=	*appexp*	
		$infexp_1$ *id* $infexp_2$	infixed application
appexp	::=	*atexp*	
		appexp atexp	application
atexp	::=	*scon*	special constant (see Section 3)
		⟨`op`⟩ *longvar*	value variable
		⟨`op`⟩ *longcon*	value constructor
		⟨`op`⟩ *longexcon*	exception constructor
		`{` ⟨ *exprow* ⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		(*exp*$_1$`,` ⋯ `,` *exp*$_n$)	n-tuple, $n \geq 2$
		`[`*exp*$_1$`,` ⋯ `,` *exp*$_n$`]`	list, $n \geq 0$
		`#[`*exp*$_1$`,` ⋯ `,` *exp*$_n$`]`	vector, $n \geq 0$
		(*exp*$_1$`;` ⋯ `;` *exp*$_n$)	sequence, $n \geq 2$
		`let` *dec* `in` *exp*$_1$`;` ⋯ `;` *exp*$_n$ `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow* ⟩	expression row
match	::=	*mrule* ⟨ `|` *match* ⟩	
mrule	::=	*pat* `=>` *exp*	

Declarations and Bindings

dec	::=	`val` *tyvarseq valbind*	value declaration
		`fun` *tyvarseq fvalbind*	function declaration
		`type` *typbind*	type declaration
		`datatype` *datbind* ⟨ `withtype` *typbind* ⟩	datatype declaration
		`abstype` *datbind* ⟨ `withtype` *typbind* ⟩ `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $unitid_1 \cdots unitid_n$	open declaration, $n \geq 1$
			empty declaration
		dec_1 ⟨`;`⟩ dec_2	sequential declaration
		`infix` ⟨*d*⟩ $id_1 \cdots id_n$	infix (left) directive, $n \geq 1$
		`infixr` ⟨*d*⟩ $id_1 \cdots id_n$	infix (right) directive, $n \geq 1$
		`nonfix` $id_1 \cdots id_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* `=` *exp* ⟨ `and` *valbind* ⟩	value binding
		`rec` *valbind*	recursive binding
fvalbind	::=	⟨`op`⟩ *var* $atpat_{11} \cdots atpat_{1n}$ ⟨`:ty`⟩ `=` exp_1	$m, n \geq 1$
		`|` ⟨`op`⟩ *var* $atpat_{21} \cdots atpat_{2n}$ ⟨`:ty`⟩ `=` exp_2	
		`|` ⋯	
		`|` ⟨`op`⟩ *var* $atpat_{m1} \cdots atpat_{mn}$ ⟨`:ty`⟩ `=` exp_m ⟨ `and` *fvalbind* ⟩	
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨ `and` *typbind* ⟩	
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨ `and` *datbind* ⟩	
conbind	::=	⟨`op`⟩ *con* ⟨`of` *ty*⟩ ⟨ `|` *conbind* ⟩	
exbind	::=	⟨`op`⟩ *excon* ⟨`of` *ty*⟩ ⟨ `and` *exbind* ⟩	
		⟨`op`⟩ *excon* `=` ⟨`op`⟩ *longexcon* ⟨ `and` *exbind* ⟩	

Note: In the *fvalbind* form above, if *var* has infix status then either `op` must be present, or *var* must be infixed. Thus, at the start of any clause, `op` *var* `(`*atpat*`,` *atpat′*`)` may be written `(`*atpat var atpat′*`)`. The parentheses may be dropped if '`:`*ty*' or '`=`' follows immediately.

Type expressions

ty	::=	*tyvar*	type variable
		`{` ⟨ *tyrow* ⟩ `}`	record type expression
		tyseq longtycon	type construction
		ty_1 `*` ⋯ `*` ty_n	tuple type, $n \geq 2$
		ty_1 `->` ty_2	function type expression
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow* ⟩	type-expression row

Patterns

atpat	::=	`_`	wildcard
		scon	special constant (see Section 3)
		⟨`op`⟩ *var*	variable
		⟨`op`⟩ *longcon*	constructor
		⟨`op`⟩ *longexcon*	exception constructor
		`{` ⟨ *patrow* ⟩ `}`	record
		`()`	0-tuple
		$($*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$$)$	n-tuple, $n \geq 2$
		`[`*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$`]`	list, $n \geq 0$
		`#[`*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$`]`	vector, $n \geq 0$
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow* ⟩	pattern row
		lab ⟨`:`*ty*⟩ ⟨ `as` *pat* ⟩ ⟨ `,` *patrow* ⟩	label as variable
pat	::=	*atpat*	atomic pattern
		⟨`op`⟩ *longcon atpat*	value construction
		⟨`op`⟩ *longexcon atpat*	exception construction
		pat$_1$ *con pat*$_2$	infixed value construction
		pat$_1$ *excon pat*$_2$	infixed exception construction
		pat `:` *ty*	typed
		⟨`op`⟩ *var* ⟨`:`*ty*⟩ `as` *pat*	layered

Syntactic restrictions

- No pattern may bind the same *var* twice. No expression row, pattern row or type row may bind the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value constructors within a *datbind*.
- In the left side *tyvarseq tycon* of any *typbind* or *datbind*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring within the right side must occur in *tyvarseq*.
- For each value binding *pat* `=` *exp* within `rec`, *exp* must be of the form `fn` *match*, possibly enclosed in parentheses, and possibly constrained by one or more type expressions.
- No *datbind* or *exbind* may bind `true`, `false`, `it`, `nil`, `::`, or `ref`.

8 Interactive sessions

An expression *exp* which occurs grammatically at top-level in an interactive session is taken to be an abbreviation for the declaration

```
val it = exp
```

This convention applies to interactive sessions only. In a batch-compiled unit, write `val it = exp` or `val _ = exp` etc.

9 Grammar for the Moscow ML module language

The Moscow ML module language is much simplified compared to the full Standard ML Modules language.

The equivalent of a Standard ML structure is a unit. A unit U consists of a unit interface in file U.sig (corresponding to an SML signature) and a unit body in file U.sml (corresponding to an SML structure). The unit signature can be left out.

Unit Body (in file unitid.sml)

unitbody	::=	`structure` *unitid* `=` `struct` *dec* `end`	structure
		`structure` *unitid* `:>` *unitid* `=` `struct` *dec* `end`	structure with signature
		dec	structure (old syntax)

Unit Signature (in file unitid.sig)

unitsig	::=	`signature` *unitid* `=` `sig` *uspec* `end`	named signature
		uspec	signature (old syntax)
uspec	::=	`val` *valdesc*	value specification
		`type` *typdesc*	abstract type
		`type` *typbind*	type abbreviation
		`eqtype` *typdesc*	abstract equality type
		`datatype` *datdesc*	datatype
		`datatype` *datdesc* `withtype` *typbind*	datatype with typbind
		`exception` *exdesc*	exception
		`local` *lspec* `in` *uspec* `end`	local specifications
			empty
		uspec ⟨`;`⟩ *uspec*	sequential
lspec	::=	`open` $unitid_1 \cdots unitid_n$	open other units
		`type` *typbind*	type abbreviation
		`local` *lspec* `in` *lspec* `end`	local specifications
			empty
		lspec ⟨`;`⟩ *lspec*	sequential
valdesc	::=	*var* `:` *ty* ⟨ `and` *valdesc* ⟩	value description
typdesc	::=	*tyvarseq tycon* ⟨ `and` *typdesc* ⟩	type constructor description
datdesc	::=	*tyvarseq tycon* `=` *condesc* ⟨ `and` *datdesc* ⟩	datatype description
condesc	::=	*con* ⟨`of` *ty*⟩ ⟨ `|` *condesc* ⟩	constructor description
exdesc	::=	*excon* ⟨`of` *ty*⟩ ⟨ `and` *exdesc* ⟩	exception constructor description

Syntactic restrictions

- In Moscow ML, the `unitid`, if specified in a unit signature or unit body file, must agree with the filename. The name of the constraining signature, if any, must equal that of the structure.
- No specification *valdesc*, *typdesc*, *typbind*, *datdesc*, or *exdesc* may describe the same identifier twice; this applies also to value constructors within a *datdesc*.
- In the left side *tyvarseq tycon* in any *typdesc*, *typbind*, or *datdesc*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring within the right side of a *typbind* or *datdesc* must occur in *tyvarseq*.
- No type, value, or exception may be specified twice at top-level in a signature.
- No *datdesc* or *exdesc* may specify `true`, `false`, `it`, `nil`, `::`, or `ref`.

10 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_failure Interrupt
Io  Match  Option  Ord  Overflow  Size  Subscript
```

11 Built-in variables and functions

For each variable we list its type, effect, and any exception it may raise. 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	
`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)	`Fail`
`installPP`	`(ppstream->'a->unit)->unit`	install prettyprinter	
`load`	`string -> unit`	load unit U and any units it needs	`Fail`
`loadOne`	`string -> unit`	load unit U (only)	`Fail`
`loadPath`	`string list ref`	search path for `load`, `loadOne`, `use`	
`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`	

- These functions are not part of the SML Basis Library.
- The Moscow ML Owner's Manual describes how to use `compile` 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.

12 List of all library units

Mosml ML's predefined library units are shown in Figure 1 below. The status of each unit is indicated according to this table:

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

More information about the Moscow ML library:

- The index in Section 14 (page 17) lists all identifiers defined in Moscow ML library units.
- Typing `help "lib";` in a `mosml` session gives a list of all library units.
- Typing `help "`*unit*`";` in a `mosml` session gives information about library unit *unit*.
- Typing `help "`*id*`";` in a `mosml` session gives information about identifier *id*, regardless which library unit(s) it is defined in.
- On the Internet the same documentation is online at

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

  Also, you may download the HTML files documenting the Moscow ML library from the Moscow ML home page and install them locally for faster browsing.

Library unit	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
`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 (Linux+Solaris)	
`FileSys`	File system interface	S F
`Gdbm`	Persistent hash tables of strings (GNU gdbm)	
`General`	Various top-level primitives	SD
`Graphics`	Graphics primitives (DOS only)	
`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	
`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	
`PP`	General prettyprinters	F
`Process`	Process interface	S F
`Random`	Generation of pseudo-random numbers	
`Real`	Real arithmetic and comparisons	S F
`SML90`	Top-level compatibility with 1990 Definition	S O
`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
`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

Figure 1: Predefined library units

13 The preloaded library units

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 (unit `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

- Functions `hd`, `tl`, and `last` may raise exception `Empty`; functions `drop`, `nth`, and `take` may raise exception `Subscript`.
- For a more detailed description, type `help "List";` or see file `mosml/lib/List.sig`. The `List` unit 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 (unit `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"`.
- On error, these functions will raise exception `Io` with an explanatory argument.

String manipulation functions (unit `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
`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

- Functions `^`, `concat`, `implode`, and `translate` may raise exception `Size`, and `sub` and `substring` may raise `Subscript`.
- 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 (unit `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`-1)`

- Functions `fromList`, `tabulate`, and `concat` may raise exception `Size`, and `sub` and `extract` may raise `Subscript`.
- For a more detailed description, type `help "Vector";` or see file `mosml/lib/Vector.sig`.

Array manipulation functions (unit `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`-1)`
`update`	`'a array * int * 'a -> unit`	set n'th element (0-based)

- Functions `array`, `tabulate`, and `fromList` may raise exception `Size`, and `sub`, `update`, `extract`, and `copy` may raise `Subscript`.
- For a more detailed description, type `help "Array";` or see file `mosml/lib/Array.sig`. The `Array` unit is loaded but not opened in the initial environment.

Character manipulation functions (unit `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` unit is loaded and partially opened in the initial environment, making the functions `chr` and `ord` available.

14 Moscow ML library index

For each identifier, we list its kind (constructor, exception, structure, type, or value) and the structure in which it is defined.

Help on any identifier *id* can be obtained by typing `help "`*id*`";` in a `mosml` session.