Moscow ML Language Overview

Version 1.31 of 15 October 1995

This is a compact reference to the language implemented by Moscow ML, a subset of Standard ML. The definitive reference on Standard ML is Milner, Tofte and Harper: *The Definition of Standard ML*, MIT Press 1990, which presents also the semantics.

Although the design of the new SML Standard Library is still in progress, Moscow ML already implements some parts of it, and the long-term intention is to adhere to the SML Standard Library as closely as possible. Currently this implementation must be considered non-authoritative.

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

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  then  type  val  with  withtype  while
(  )  [  ]  {  }  ,  :  ;  ...  _  |  =  =>  ->  #
```

2 Special constants

Integer constants

Examples:	`0`	`~0`	`4`	`~04`	`999999`
Non-examples:	`0.0`	`~0.0`	`4.0`	`1E0`	

Real constants

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

String constants

A string constant is a sequence, between quotes (`"`), of zero or more printable characters (i.e., ASCII code 33–126), spaces, or escape sequences. An escape sequence starts with the escape character `\` and stands for a character sequence:

`\n`	A single character interpreted by the system as end-of-line.
`\t`	Tabulator.
`\^`*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 ASCII code *ddd* (3 decimal digits denoting an integer in the interval [0,255]).
`\"`	`"`
`\\`	`\`
`\`*f* ··*f*`\`	This sequence is ignored, where *f* ··*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`	`#"""`		

3 Comments

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

4 Identifiers

An identifier must have one of these forms:

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

In either case, however, reserved words are excluded. This means that for example `#` and `|` are not identifiers, but `##` and `|=|` are identifiers.

There are six 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 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
		⟨`op`⟩ *longvar*	value variable
		⟨`op`⟩ *longcon*	value constructor
		⟨`op`⟩ *longexcon*	exception constructor
		`{` ⟨*exprow* ⟩`}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		`(`exp_1`,` $\cdots$ `,` exp_n`)`	n-tuple, $n \geq 2$
		`[`exp_1`,` $\cdots$ `,` exp_n`]`	list, $n \geq 0$
		`#[`exp_1`,` $\cdots$ `,` exp_n`]`	vector, $n \geq 0$
		`(`exp_1`;` $\cdots$ `;` exp_n`)`	sequence, $n \geq 2$
		`let` *dec* `in` exp_1`;` $\cdots$ `;` 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` *valbind*	value declaration
		`fun` *fvalbind*	function declaration
		`type` *typbind*	type declaration
		`datatype` *datbind* $\langle$ `withtype` *typbind* $\rangle$	datatype declaration
		`abstype` *datbind* $\langle$ `withtype` *typbind* $\rangle$ `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 $\langle$`;`$\rangle$ dec_2	sequential declaration
		`infix` $\langle d \rangle$ $id_1 \cdots id_n$	infix (left) directive, $n \geq 1$
		`infixr` $\langle d \rangle$ $id_1 \cdots id_n$	infix (right) directive, $n \geq 1$
		`nonfix` $id_1 \cdots id_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* `=` *exp* $\langle$ `and` *valbind* $\rangle$	value binding
		`rec` *valbind*	recursive binding
fvalbind	::=	$\langle$`op`$\rangle$ *var* $atpat_{11} \cdots atpat_{1n}$ $\langle$`:ty`$\rangle$ `=` exp_1	$m, n \geq 1$
		`|` $\langle$`op`$\rangle$ *var* $atpat_{21} \cdots atpat_{2n}$ $\langle$`:ty`$\rangle$ `=` exp_2	
		`|` $\cdots$	
		`|` $\langle$`op`$\rangle$ *var* $atpat_{m1} \cdots atpat_{mn}$ $\langle$`:ty`$\rangle$ `=` exp_m	
		$\langle$ `and` *fvalbind* $\rangle$	
typbind	::=	*tyvarseq* *tycon* `=` *ty* $\langle$ `and` *typbind* $\rangle$	
datbind	::=	*tyvarseq* *tycon* `=` *conbind* $\langle$ `and` *datbind* $\rangle$	
conbind	::=	$\langle$`op`$\rangle$ *con* $\langle$`of` *ty*$\rangle$ $\langle$ `|` *conbind* $\rangle$	
exbind	::=	$\langle$`op`$\rangle$ *excon* $\langle$`of` *ty*$\rangle$ $\langle$ `and` *exbind* $\rangle$	
		$\langle$`op`$\rangle$ *excon* `=` $\langle$`op`$\rangle$ *longexcon* $\langle$ `and` *exbind* $\rangle$	

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.

Patterns

atpat	::=	`_`	wildcard
		scon	special constant (see Section 2)
		⟨`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

Type expressions

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

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.

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 Bodies (file U.sml)

unitbody ::= *topdec* `;` ⟨ *unitbody* ⟩

Unit Signatures (file U.sig)

uspec	::=	`val` *valdesc*	value specification
		`type` *typdesc*	abstract type
		`type` *typbind*	type abbreviation
		`eqtype` *typdesc*	abstract equality type
		`datatype` *datbind*	datatype
		`datatype` *datbind* `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

- 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.

10 Built-in types, constructors and exceptions

This section lists all identifiers 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
`'a array`	Arrays	yes	
`bool`	Booleans	yes	`false`, `true`
`char`	Characters	yes	`#"a"`, `#"b"`, $\cdots$
`exn`	Exceptions	no	
`'a frag`	Quotation fragments	if `'a` does	`QUOTE`, `ANTIQUOTE`
`instream`	Input streams	no	
`int`	Integers	yes	
`'a list`	Lists	if `'a` does	`nil`, `::`
`'a option`	Optional results	if `'a` does	`NONE`, `SOME`
`ordering`	Comparisons	yes	`LESS`, `EQUAL`, `GREATER`
`outstream`	Output streams	no	
`real`	Floating-point numbers	yes	
`'a ref`	References	yes	`ref`
`string`	Strings	yes	
`unit`	The empty tuple `()`	yes	
`'a vector`	Vectors	if `'a` does	

Built-in exception constructors

```
Bind  Chr  Div  Fail  Graphic_failure  Interrupt  Io
Ln  Match  Ord  Overflow  Sqrt  Size  Subscript  Trig
```

11 Built-in variables and functions

For each variable we list its type, effect, and any exception it may raise.

An identifier with type involving `num` stands for two functions: one in which `num` (in the type) is replaced by `int`, and another in which `num` is replaced by `real`. Similarly for `cirs`, which stands for `char`, `int`, `real`, or `string`. An explicit type constraint will be needed if the surrounding text does not determine the type of a particular occurrence.

Nonfix identifiers in the initial environment

id	type	effect	exception
`~`	`num -> num`	arithmetic negation	`Overflow`
`!`	`'a ref -> 'a`	dereference	
`abs`	`num -> num`	absolute value	`Overflow`
`app`	`('a -> unit) -> 'a list -> unit`	apply to all elements	
`arctan`	`real -> real`	arc tangent (to radians)	
`chr`	`int -> char`	character with number	`Chr`
`concat`	`string list -> string`	concatenate strings	`Size`
`cos`	`real -> real`	cosine (from radians)	`Fail`
`exp`	`real -> real`	exponential function	`Overflow`
`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	
`makestring`	`cirs -> string`	convert to string	
`ln`	`real -> real`	natural logarithm	`Ln`
`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	
`real`	`int -> real`	`int` to `real`	
`ref`	`'_a -> '_a ref`	create reference value	
`rev`	`'a list -> 'a list`	reverse list	
`sin`	`real -> real`	sine (from radians)	`Fail`
`size`	`string -> int`	length of string	
`sqrt`	`real -> real`	square root	`Sqrt`
`str`	`char -> string`	create one-character string	
`tl`	`'a list -> 'a list`	tail of list	`Empty`
`true`	`bool`	logical truth	
`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	int * int -> int	quotient (round towards $-\infty$)	Div, Overflow
mod	int * int -> int	remainder (of div)	Div, Overflow
quot	int * int -> int	quotient (round towards 0)	Div, Overflow
rem	int * int -> int	remainder (of quot)	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	
<	cirs * cirs -> bool	less than	
<=	cirs * cirs -> bool	less than or equal to	
>	cirs * cirs -> bool	greater than	
>=	cirs * cirs -> 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 * unit -> ´a	discard second 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
print	´a -> ´a	print value on std_out	
printDepth	int ref	limit printed data depth	
printLength	int ref	limit printed list and vector length	
quit	unit -> ´a	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	
verbose	bool ref	permit feedback from compile	

- These functions are not part of the SML Standard Library.
- The Moscow ML Owner's Manual describes how to use the functions `compile` and `load` to perform separate compilation. It also explains the use of 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.

Built-in values and functions for input/output (unit `BasicIO`)

id	type	effect
`close_in`	`instream -> unit`	close input stream
`close_out`	`outstream -> unit`	close output stream
`end_of_stream`	`instream -> bool`	true if at end of stream
`exit`	`int -> 'a`	terminate ML, flush output
`flush_out`	`outstream -> unit`	flush output to consumer
`input`	`instream * int -> string`	input at most n chars
`inputc`	`instream -> int -> string`	same
`input_line`	`instream -> string`	read up to (and including) next end of line
`lookahead`	`instream -> string`	get next char non-destructively
`open_append`	`string -> outstream`	open file for appending to it
`open_in`	`string -> instream`	open file for input
`open_out`	`string -> outstream`	open file for output
`output`	`outstream * string -> unit`	write to output stream
`outputc`	`outstream -> string -> unit`	same
`say`	`string -> unit`	print to `std_out` and flush it
`std_err`	`outstream`	standard error output stream
`std_in`	`instream`	standard input stream
`std_out`	`outstream`	standard output stream

- These functions are not part of the SML Standard Library.
- On error, these will raise exception `Io` with an explanatory string argument.
- `input(`is`, `n`)` returns a string containing the n first characters of is, also removing them from is. If only $k < n$ characters are available on is, and it is terminated, then those k characters are returned; otherwise no result is returned until n characters are available or until is gets closed.
- `lookahead(`is`)` returns a 1-character string s containing the first character from is, without removing it. If no character is available, and the stream is closed, the empty string is returned; otherwise no result is returned until a character is available or until is gets closed.
- `open_out(`f`)` deletes any existing contents of the file named f and returns a new outstream associated with f.
- `open_append(`f`)` returns a new outstream associated with file f; any output to that stream gets appended to the file f.

12 The preloaded library units

For a listing and description of all libraries, see file `README` and the signature files in directory `mosml/lib`, or type `help "lib";` while inside a Moscow ML session.

Here we mention only those SML Standard Library units which are preloaded by default: `Array`, `Char`, `List`, `String`, and `Vector`.

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

- Notation above: functional arguments are called f, except that arguments of type `´a -> bool` are called p. Arguments of type `int` are called n, and list-type arguments are called xs and ys.
- Functions `hd`, `tl`, and `last` may raise exception `Empty`; functions `drop`, `nth`, and `take` may raise exception `Subscript`.
- For a more detailed description, see file `mosml/lib/List.sig`, or type `help "List";`. 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`.

Character manipulation functions (unit `Char`)

id	type	effect	exception
`chr`	`int -> char`	from ASCII code to character	`Chr`
`compare`	`char * char -> ordering`	compare characters (ASCII)	
`contains`	`string -> char -> bool`	contained in string	
`isAlpha`	`char -> bool`	alphabetic	
`isAlphaNum`	`char -> bool`	alphanumeric	
`isAscii`	`char -> bool`	seven-bit ASCII character	
`isDigit`	`char -> bool`	decimal digit	
`isGraph`	`char -> bool`	printable and visible	
`isHexDigit`	`char -> bool`	hexadecimal digit	
`isLower`	`char -> bool`	lower case alphabetic	
`isPrint`	`char -> bool`	printable (including space)	
`isSpace`	`char -> bool`	space and lay-out (HT, CR, LF, VT, FF)	
`isUpper`	`char -> bool`	upper case alphabetic	
`maxChar`	`char`	last character (in `<=` ordering)	
`maxOrd`	`int`	largest character code	
`minChar`	`char`	first character (in `<=` ordering)	
`notContains`	`string -> char -> bool`	not in string	
`ord`	`char -> int`	from character to ASCII code	
`pred`	`char -> char`	preceding character	`Chr`
`succ`	`char -> char`	succeding character	`Chr`
`toLower`	`char -> char`	convert to lower case	
`toUpper`	`char -> char`	convert to upper case	

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

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
`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)

- Notation: functional arguments are called f, and arguments of type `int` are called n.
- Functions `fromList`, `tabulate`, and `concat` may raise exception `Size`, and `sub` and `extract` may raise `Subscript`.
- For a more detailed description, see file `mosml/lib/Vector.sig`, or type `help "Vector";` inside Moscow ML.

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
`array0`	`´a array`	empty array
`copy`	`{src:´a array, si:int,` ` dst:´a array, di:int, len:int} -> unit`	copy subarray to array
`copyv`	`{src:´a vector, si:int, dst:´a array, di:int, len:int}`	
`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)

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

String manipulation functions (unit `String`)

id	type	effect
`^`	`string * string -> string`	concatenate strings
`collate`	`(char*char->ordering)->string*string->ordering`	compare strings
`compare`	`string * string -> ordering`	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
`implode`	`char list -> string`	string from character list
`maxLen`	`int`	maximal length of a string
`size`	`string -> int`	length of string
`str`	`char -> string`	make one-character string
`sub`	`string * int -> ´a`	n'th character (0-based)
`substring`	`string * int * int -> string`	get substring $(s, first, len)$
`tokens`	`(char -> bool) -> string -> string list`	find (non-empty) tokens
`translate`	`(char -> string) -> string -> string`	apply f and concatenate

- Notation: functional arguments are called f, and arguments of type `string` are called s.
- Functions `^`, `concat`, `implode`, and `translate` may raise exception `Size`, and `sub` and `substring` may raise `Subscript`.
- For a more detailed description, see file `mosml/lib/String.sig`, or type `help "String";`. The `String` unit is loaded but not opened in the initial environment.

13 Relation to Standard ML

Moscow ML implements the full Standard ML Core language, and extends it with

- the type `char` of characters, and character constants such as `#"a"`;
- the type `'a vector` of immutable vectors of elements, vector expressions `#[`exp_1`, ···, `exp_n`]`, and vector patterns `#[`pat_1`, ···, `pat_n`]`;
- the type `'a array` of mutable arrays;
- quotations and antiquotations such as `` `record ^fields end` ``.

Moscow ML has a simple module system, which deviates from the Standard ML Modules language as follows:

- There are no functors (and hence no sharing constraints), and no nested structures.
- Signatures may contain type abbreviations. In a signature, no type, value, or exception constructor may be specified twice at top-level; a `local` specification can be used only to restrict the scope of `open` specifications and type abbreviations; and an `open` specification can appear only inside `local`.
- Signature constraints are opaque: only the information given in the signature U.sig can be used when compiling another unit which uses unit U.

Moscow ML implements the new SML Standard Library as closely as is possible, given the limitations of the Moscow ML module system.

The SML Standard Library deviates from the *Definition* in the following respects:

- There is a new type `char` of characters.
- Functions `chr` and `ord` produce resp. consume characters, not strings.
- Functions `explode` and `implode` produce resp. consume lists of characters, not lists of strings;
- The arithmetic exceptions `Abs`, `Diff`, `Div`, `Exp`, `Floor`, `Mod`, `Neg`, `Prod`, `Quot`, and `Sum` have been replaced by the two exceptions `Overflow` and `Div`, meaning 'result too big' and 'divisor is zero'.
- The identifier `@` associates to the right.
- The resolution of mixed infix operators of the same precedence has been changed: mixed left-associative operators associate to the left, mixed right-associative operators associate to the right, and mixed left- and right-associative operators are illegal. This permits expressions such as `[1] @ 2 :: [3]` and `1 :: [2] @ [3]`.

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