Moscow ML Language Overview

Version 2.00 of June 2000

Sergei Romanenko, Russian Academy of Sciences, Moscow, Russia
Claudio Russo, Cambridge University, Cambridge, United Kingdom
Peter Sestoft, Royal Veterinary and Agricultural University, Copenhagen, Denmark

This is a compact reference to the language implemented by Moscow ML, a superset of Standard ML. For reference material on Standard ML, see Milner, Tofte, Harper and MacQueen: *The Definition of Standard ML*, The MIT Press 1997. For a guide to the practical use of Moscow ML, see the *Moscow ML Owner's Manual*. For a detailed description of all Moscow ML library modules, see the *Moscow ML Library Documentation*.

Contents

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

1 Moscow ML's relation to Standard ML

Moscow ML implements a proper extension of Standard ML, as defined in the 1997 *Definition of Standard ML*. This document describes the language implemented by Moscow ML, not Standard ML *per se*: users seeking an orthodox Standard ML reference should look elsewhere. Having said that, Moscow ML is specifically designed to be backwards compatible with Standard ML. Thus every valid Standard ML program should be a valid Moscow ML program, and Moscow ML may be used as if it were simply a Standard ML compiler. Any deviation from this behaviour should be reported as a bug.

2 Reserved words

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

3 Comments

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

4 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).
\u$xxxx$	The character with code $xxxx$ (4 hexadecimal digits denoting an integer 0–255).
\"	The double-quote character (`"`)
\\	The backslash character (\)
\$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`	`#""""`		

5   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 2) are excluded. This means that for example # and | are not identifiers, but ## and |=| are identifiers. There are several classes of identifiers:

vid	(value identifiers)	long
tyvar	(type variables)	
tycon	(type constructors)	long
lab	(record labels)	
strid	(structure identifiers)	long
funid	(functor identifiers)	long
modid	(module identifiers)	long
sigid	(signature identifiers)	
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 long structure identifier followed by a dot '.' :

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

- Although structure and functor identifiers reside in separate name-spaces, the syntax of structure and functor identifiers is identical. The set of identifiers *modid* ranges over the union of *strid* and *funid*; *longmodid* ranges over the union of *longstrid* and *longfunid*. Moscow ML uses type information to resolve each occurrence of a *modid* or *longmodid* to a structure or functor identifier during type checking, using the optional keyword op to resolve any remaining ambiguities. See the comments at the end of Section 10.
- Any occurrence of a structure identifier *strid* that is not bound in the current context refers to the unit implementation unitid.uo of the same name (ie. unitid = *strid*). At compile time, the unit's compiled interface unitid.ui must exist and have been compiled in *structure* mode. At link time, the unit's compiled implementation unitid.uo must exist and have been compiled in *structure* mode.
- Any occurrence of a signature identifier *sigid* that is not bound in the current context refers to the compiled unit interface unitid.ui of the same name (ie. unitid = *sigid*). The file unitid.ui must have been compiled in *structure* mode from an explicit interface unitid.sig.

6 Infixed operators

An identifier may be given infix status by the `infix` or `infixr` directive, which may occur as a declaration or specification. 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 ($n \geq 1$):

$$
\begin{array}{llll}
\texttt{infix} & \langle d \rangle & id_1 \cdots id_n & \text{left associative} \\
\texttt{infixr} & \langle d \rangle & id_1 \cdots id_n & \text{right associative} \\
\texttt{nonfix} & & id_1 \cdots id_n & \text{non-associative}
\end{array}
$$

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`. Fixity directives occurring within *dec* in a structure expression `struct` *dec* `end` are local to *dec*. Fixity directives occurring within *spec* in a signature `sig` *spec* `end` are local to *spec*.

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.

7 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:

$$
\begin{array}{llll}
xseq & ::= & x & \text{(singleton sequence)} \\
& & & \text{(empty sequence)} \\
& & (x_1, \cdots, x_n) & \text{(sequence, } n \geq 1)
\end{array}
$$

- 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.
- Moscow ML phrases that are non-compliant extensions of Standard ML syntax are marked with an bullet (•) in the margin.
- Moscow ML phrases that are non-compliant generalisations of Standard ML syntax, but have instances that comply with Standard ML, are marked with an an bullet and a number (•*N*) in the margin, where *N* refers to an explanatory comment that appears in Section 12.

8 Grammar for the Moscow 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 4)
		⟨`op`⟩ *longvid*	value identifier
		{ ⟨ *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, $n \geq 1$
		`[` `structure` *modexp* `as` *sigexp* `]`	structure package •
		`[` `functor` *modexp* `as` *sigexp* `]`	functor package •
		`(` *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	
		`datatype` *tycon* `=` `datatype` *tyconpath*	datatype replication	
		`abstype` *datbind* ⟨ `withtype` *typbind* ⟩	abstype declaration	
		`with` *dec* `end`		
		`exception` *exbind*	exception declaration	
		`local` dec_1 `in` dec_2 `end`	local declaration	
		`open` $longstrid_1 \cdots longstrid_n$	open declaration, $n \geq 1$	
		`structure` *strbind*	structure declaration	•1
		`functor` *funbind*	functor declaration	•2
		`signature` *sigbind*	signature declaration	•2
			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		
		| $\cdots$		
		| ⟨`op`⟩ *var* $atpat_{m1} \cdots atpat_{mn}$ ⟨`:ty`⟩ `=` exp_m		
		⟨ `and` *fvalbind* ⟩		
typbind	::=	*tyvarseq* *tycon* `=` *ty* ⟨ `and` *typbind* ⟩		•3
datbind	::=	*tyvarseq* *tycon* `=` *conbind* ⟨ `and` *datbind* ⟩		•3
conbind	::=	⟨`op`⟩ *vid* ⟨`of` *ty*⟩ ⟨ | *conbind* ⟩		
exbind	::=	⟨`op`⟩ *vid* ⟨`of` *ty*⟩ ⟨ `and` *exbind* ⟩		
		⟨`op`⟩ *vid* `=` ⟨`op`⟩ *longvid* ⟨ `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

tyconpath	::=	*longtycon*	long type constructor	
		longtycon `where` *strid* `=` *modexp*	type constructor projection	•
ty	::=	*tyvar*	type variable	
		`{` ⟨ *tyrow* ⟩ `}`	record type expression	
		tyseq *tyconpath*	type construction	
		ty_1 `*` $\cdots$ `*` ty_n	tuple type, $n \geq 2$	
		ty_1 `->` ty_2	function type expression	
		`[` *sigexp* `]`	package type expression	•
		`(` *ty* `)`		
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow* ⟩	type-expression row	

Patterns

atpat	::=	_	wildcard
		scon	special constant (see Section 4)
		⟨op⟩ *longvid*	value identifier
		{ ⟨ *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⟩ *longvid* *atpat*	constructed value
		pat_1 *vid* pat_2	infixed value 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. Moscow ML requires that any *tyvar* occurring within the right side is in scope (either explictly or implicitly), but not necessarily in *tyvarseq* (cf. Section 12, restriction 3).
- 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 *valbind*, *datbind*, or *exbind* may bind true, false, nil, ::, or ref. No *datbind* or *exbind* may bind it.

9 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.

10   Grammar for the Moscow ML Modules language

The Moscow ML Modules language is a superset of the full Standard ML Modules language.

Module expressions

modexp	::=	*appmodexp*		
		modexp : *sigexp*	transparent constraint (L)	
		modexp :> *sigexp*	opaque constraint (L)	
		functor (*modid* : *sigexp*) => *modexp*	generative functor	•
		functor *modid* : *sigexp* => *modexp*	applicative functor	•
		rec (*strid* : *sigexp*) *modexp*	recursive structure	•
appmodexp	::=	*atmodexp*		
		appmodexp *atmodexp*	functor application	•4
atmodexp	::=	struct *dec* end	basic	
		⟨op⟩ *longmodid*	module identifier	
		let *dec* in *modexp* end	local declaration	
		(*dec*)	abbreviated structure	•
		(*modexp*)		

Module bindings

strbind	::=	*strid* ⟨ *con* ⟩ = *modexp* ⟨ and *strbind* ⟩	structure binding	
		strid as *sigexp* = *exp* ⟨ and *strbind* ⟩	package binding	•
funbind	::=	*funid* arg_1 ··· arg_n ⟨ *con* ⟩ = *modexp*		
		⟨ and *funbind* ⟩	functor binding, $n \geq 0$	•5
		funid (*spec*) ⟨ *con* ⟩ = *modexp*		
		⟨ and *funbind* ⟩	abbreviated generative binding	
		funid as *sigexp* = *exp* ⟨ and *funbind* ⟩	package binding	•
sigbind	::=	*sigid* = *sigexp* ⟨ and *sigbind* ⟩	signature binding	
con	::=	: *sigexp*	transparent constraint	
		:> *sigexp*	opaque constraint	
arg	::=	(*modid* : *sigexp*)	argument of generative functor	
		modid : *sigexp*	argument of applicative functor	•

Signature expressions

sigexp	::=	sig *spec* end	basic	
		sigid	signature identifier	
		sigexp where *typreal*	type realisation	
		functor (*modid* : *sigexp*) -> *sigexp*	opaque functor signature	•
		functor *modid* : *sigexp* -> *sigexp*	transparent functor signature	•
		rec (*strid* : *sigexp*) *sigexp*	recursive structure signature	•
typreal	::=	type *tyvarseq* *longtycon* = *ty* ⟨ and *typreal* ⟩	type realisation	•3

Specifications and Descriptions

spec	::=	val *tyvarseq valdesc*	value specification	•6
		type *typdesc*	abstract type	
		type *typbind*	type abbreviation	
		eqtype *typdesc*	abstract equality type	
		datatype *datdesc* ⟨ withtype *typbind* ⟩	datatype with typbind	•7
		datatype *tycon* = datatype *tyconpath*	datatype replication	
		exception *exdesc*	exception	
		structure *strdesc*	structure	
		functor *fundesc*	functor	•
		signature *sigbind*	signature	•
		include *sigid*$_1$ ⋯ *strid*$_n$	include, $n \geq 1$	
		local *lspec* in *spec* end	local specifications	•
			empty	
		spec ⟨;⟩ *spec*	sequential	
		spec sharing type	type sharing, $n \geq 2$	
		$\quad$ *longtycon*$_1$ = ⋯ = *longtycon*$_n$		
		spec sharing	structure sharing, $n \geq 2$	
		$\quad$ *longstrid*$_1$ = ⋯ = *longstrid*$_n$		
		infix ⟨*d*⟩ *id*$_1$ ⋯ *id*$_n$	infix (left) directive, $n \geq 1$	•
		infixr ⟨*d*⟩ *id*$_1$ ⋯ *id*$_n$	infix (right) directive, $n \geq 1$	•
		nonfix *id*$_1$ ⋯ *id*$_n$	nonfix directive, $n \geq 1$	•
lspec	::=	open *longstrid*$_1$ ⋯ *longstrid*$_n$	(local) open	•
		type *typbind*	type abbreviation	•
		local *lspec* in *lspec* end	local specifications	
			empty	•
		lspec ⟨;⟩ *lspec*	sequential	•
valdesc	::=	*vid* : *ty* ⟨ and *valdesc* ⟩	value description	
typdesc	::=	*tyvarseq tycon* ⟨ and *typdesc* ⟩	type constructor description	
datdesc	::=	*tyvarseq tycon* = *condesc* ⟨ and *datdesc* ⟩	datatype description	•3
condesc	::=	*vid* ⟨of *ty*⟩ ⟨ | *condesc* ⟩	constructor description	
exdesc	::=	*vid* ⟨of *ty*⟩ ⟨ and *exdesc* ⟩	exception constructor description	
strdesc	::=	*strid* : *sigexp* ⟨ and *strdesc* ⟩	structure description	
fundesc	::=	*funid* : *sigexp* ⟨ and *fundesc* ⟩	functor description	

- Although structure and functor identifiers reside in separate name-spaces, the syntax of structure and functor identifiers is identical. In the grammar, a module identifier *longmodid* may stand for either a structure identifier *longstrid* or a functor identifier *longfunid*. Thus, *a priori*, the module expression ⟨op⟩ *longmodid* may refer to a either a functor or a structure and the compiler must resolve this ambiguity (⟨op⟩ is an optional prefix of the keyword op). Fortunately, the context of the phrase often rules out one alternative, on the grounds that choosing that alternative would force type checking to fail. In particular, if ⟨op⟩ *longmodid* occurs as the right hand side of a structure (functor) binding, then *longmodid* must be interpreted as a structure (functor) identifier; if ⟨op⟩ *longmodid* occurs in the functor position of an application, then *longmodid* must be interpreted as a functor identifier; if ⟨op⟩ *longmodid* is constrained by a signature then the signature forces a unique interpretation on *longmodid* (depending on whether the signature specifies a structure or functor). Similarly, if ⟨op⟩ *longmodid* occurs as the argument of a functor application, then the functor's domain forces a unique interpretation on *longmodid*. Indeed, the only ambiguity that remains occurs when ⟨op⟩ *longmodid* is the body of a functor. In this case, the optional prefix ⟨op⟩ is used to resolve the ambiguity: the *absence* of op signals that *longmodid* refers to structure; the *presence* of op signals that op *longmodid* refers to a functor. When the interpretation of ⟨op⟩ *longmodid* is already determined by the context, the optional prefix ⟨op⟩ has *no* effect. (This method of disambiguation relies on type information and is performed during type checking.)

- In a functor or functor signature's formal argument, (*modid* : *sigexp*) or *modid* : *sigexp* , if *sigexp* specifies a structure then *modid* binds the equivalent structure identifier *strid*; if *sigexp* specifies a functor, then *modid* binds the equivalent functor identifier *funid*.
- In a structure expression struct *dec* end, any signature declared in *dec* is local to *dec*: it does not define a component of the structure struct *dec* end, nor is it visible in the type of struct *dec* end. (Note that the syntax for signature identifiers is not long, in the sense of Section 5.)
- In a signature expression sig *spec* end, any signature declared in *spec* is local to *spec*: in particular, such a declaration does *not* specify that a structure matching sig *spec* end should also declare that signature.

Syntactic restrictions

- No binding *strbind*, *funbind*, or *sigbind* may bind the same identifier twice.
- No specification *valdesc*, *typdesc*, *typbind*, *datdesc*, *exdesc*, *strdesc* or *fundesc* 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*, *datdesc*, or *typreal*, or specification val *tyvarseq valdesc*, *tyvarseq* must not contain the same *tyvar* twice. Moscow ML requires that any *tyvar* occurring within the right side is in scope (either explictly or implicitly), but not necessarily in *tyvarseq* (cf. Section 12, restriction 3).
- No sequential specification may specify the same *tycon*, *vid*, *strid*, *funid*, *sigid* or *id* (in a fixity specification) twice.
- No *valdesc*, *datdesc*, or *exdesc* may specify true, false, nil, ::, or ref. No *datdesc* or *exdesc* may specify it.
- In a generative functor functor (*modid* : *sigexp*) => *modexp* , applicative functor functor *modid* : *sigexp* => *modexp* or recursive structure rec (*strid* : *sigexp*) *modexp*, the body of *modexp* must be *applicative* in the sense that it contains no structure or functor bindings of the form *strid* as *sigexp* = *exp* or *funid* as *sigexp* = *exp*, excluding those bindings that occur within a Core let-expression. This restriction also applies to the bodies of functors declared in a *funbind*.

11   Grammar for the Moscow ML Unit language

Moscow ML supports the separate compilation of named program fragments called *units*. A unit `unitid` consists of an optional *unit interface* in file `unitid.sig`. Each unit can be compiled in one of two *modes*: *structure* mode and *toplevel* mode. A unit's implementation and interface files must be compiled in the *same* mode.

In the batch compiler `mosmlc`, a unit's compilation mode is specified by preceding it with the command-line argument `-structure` (the default) or `-toplevel`. In the interactive system `mosml`, the compilation mode of a unit is determined by the function with which it is compiled: `compile` and `compileStructure` compile in *structure* mode; `compileToplevel` compiles in *toplevel* mode.

Note that the intended mode of a unit is not determined by file name extension or by file content: the mode must be explicitly indicated to the batch compiler and interactive system.

The syntax and semantics of a unit's interface and implementation files depends on the mode and is described in the following sections.

11.1   Syntax and semantics for units compiled in *structure* mode

In *structure* mode, the unit interface file `unitid.sig`, if present, must contain a single Moscow ML signature declaration binding the signature `unitid`; the unit implementation file `unitid.sml` must contain a single Moscow ML structure declaration, binding the structure `unitid`. The unit interface may be omitted.

With the batch compiler `mosmlc`, the files `unitid.sig` and `unitid.sml` are compiled in *structure* mode if their filenames are preceded by the command line argument `-structure`, eg:

```
mosmlc -c -structure unitid.sig unitid.sml
```

Since *structure* mode is the default compilation mode, the `-structure` option may also be omitted:

```
mosmlc -c unitid.sig unitid.sml
```

In the interactive system, a unit interface or implementation may be compiled in structure mode using the functions `compile` and `compileStructure`.

The semantics of

```
- compileStructure ["unitid₁",···,"unitidₙ"] "unitid.sig";(* if unitid.sig exists *)
- compileStructure ["unitid₁",···,"unitidₙ"] "unitid.sml";
- load "unitid";
```

is roughly equivalent to that of

```
- load "unitid₁";
...
- load "unitidₙ";
- use "unitid.sig"; (* if unitid.sig exists *)
- use "unitid.sml";
```

Note that the unit interface `unitid.sig`, if present, should be `use`'ed in the interactive system, since the interface declares a signature that is referred to in `unitid.sml`, and may be referred to in other units that depend on unit `unitid`. A structure-mode unit interface has two effects: it (a) declares a signature and (b) serves to constrain the structure defined in the unit implementation.

Structure-mode unit implementation (in file `unitid.sml`)

unitimp	::=	structure *unitid* = *modexp*	structure	
		structure *unitid* :> *unitid* = *modexp*	structure with signature	
		cdec	core declaration	deprecated

cdec	::=	val *tyvarseq valbind*	value declaration	
		fun *tyvarseq fvalbind*	function declaration	
		type *typbind*	type declaration	
		datatype *datbind* ⟨ withtype *typbind* ⟩	datatype declaration	
		datatype *tycon* = datatype *tyconpath*	datatype replication	
		abstype *datbind* ⟨ withtype *typbind* ⟩	abstype declaration	
		with *dec* end		
		exception *exbind*	exception declaration	
		local dec_1 in dec_2 end	local declaration	
		open $longstrid_1$ $\cdots$ $longstrid_n$	open declaration, $n \geq 1$	
			empty declaration	
		$cdec_1$ ⟨ ; ⟩ $cdec_2$	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, $n \geq 1$	

Structure-mode unit interface (in file `unitid.sig`)

unitint	::=	signature *unitid* = *sigexp*	signature binding	
		cspec	core specification	deprecated

cspec	::=	val *tyvarseq valdesc*	value specification	•6
		type *typdesc*	abstract type	
		type *typbind*	type abbreviation	
		eqtype *typdesc*	abstract equality type	
		datatype *datdesc* ⟨ withtype *typbind* ⟩	datatype with typbind	•7
		datatype *tycon* = datatype *tyconpath*	datatype replication	
		exception *exdesc*	exception	
		local *lspec* in *spec* end	local specifications	•
			empty	
		cspec ⟨ ; ⟩ *cspec*	sequential	
		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, $n \geq 1$	•

Syntactic restrictions

- In Moscow ML, the *unitid*, if specified in the unit interface or unit implementation, must agree with the filename (*unitid*`.sig` or *unitid*`.sml`). In the unit implementation, the name of the constraining signature, if any, must equal that of the structure.
- The unit implementation syntax *cdec* is deprecated and is provided only to support code written for earlier versions of Moscow ML (versions prior to 2.*xx*). The phrase class *cdec* is a proper subset of *dec* and is subject to the same restrictions as *dec*. The class *cdec* excludes declarations beginning with `structure`, `functor`, or `signature`.
- The unit implementation syntax *cdec* abbreviates `structure` *unitid* ⟨`:>` *unitid*⟩ = `struct` *cdec* `end` thus any fixity directives in *cdec* are local to the structure expression `struct` *cdec* `end` and are not exported in the interface.
- The unit interface syntax *cspec* is deprecated and is provided only to support code written for earlier versions of Moscow ML (versions prior to 2.*xx*). The phrase class *cspec* is a proper subset of *spec* and is subject to the same restrictions as *spec*. The class *cspec* excludes specifications beginning with `structure`, `functor`, `signature` or `include` and `sharing` specifications.
- The unit interface syntax *cspec* abbreviates `signature` *unitid* = `sig` *cspec* `end` thus any fixity directives in *cspec* are local to the signature expression `sig` *cspec* `end` and are not exported in the interface.

11.2 Syntax and semantics for units compiled in *toplevel* mode

In *toplevel* mode, the unit interface in file `unitid.sig`, if present, must be a Moscow ML specification (which may itself be a sequence of specifications); the unit implementation in file `unitid.sml` must be a Moscow ML declaration (which may itself be a sequence of declarations). The unit interface may be omitted.

With the batch compiler `mosmlc`, the files `unitid.sig` and `unitid.sml` are compiled in *toplevel* mode only if their filenames are preceded by the command line argument `-toplevel`.

```
mosmlc -c -toplevel unitid.sig unitid.sml
```

In the interactive system, a unit interface or implementation may be compiled in *toplevel* mode using the function `compileToplevel`.

The semantics of

```
- compileToplevel ["unitid₁",···,"unitidₙ"] "unitid.sig"; (* if unitid.sig exists *)
- compileToplevel ["unitid₁",···,"unitidₙ"] "unitid.sml";
- load "unitid";
```

Provided the compilation of `unit.sml` issues no warnings (see below), this is equivalent to

```
- load "unitid₁";
···
- load "unitidₙ";
- use "unitid.sml";
```

Note that the unit interface `unitid.sig`, if present, should not be `use`'ed in the interactive system. Unlike the interface of structure-mode unit, which declares a signature, `unitid.sig` does not contain a declaration, but merely the specification of the declarations in `unitid.sml`. The only purpose of the interface file is to support the separate compilation of units that depend on the unit `unitid` (for instance, in the absence of file `unit.sml`). Since `use`ing the implementation, as opposed to `load`ing the compiled unit, can potentially (a) declare identifiers that are not specified in the interface, or (b) declare constructors and exceptions, that are only specified as ordinary values in the interface, and both (a) and (b) may affect the meaning of subsequent code, when compiling a toplevel-mode implementation against its interface, Moscow ML will issue warning whenever (a) or (b) occurs.

Toplevel-mode unit implementation (in file `unitid.sml`)

unitimp	::=	*dec*	declaration

Toplevel-mode unit interface (in file `unitid.sig`)

unitint	::=	*spec*	specification

12   Further restrictions imposed for Standard ML compliance

In addition to the syntactic restrictions imposed by Moscow ML, compiling programs in `orthodox` or `conservative` mode (see Section 1), imposes the following additional restrictions. These are required to ensure compliance with Standard ML.

- Any instance of a Moscow ML phrase that is marked with a plain • in the grammar is illegal Standard ML.
- Any instance of a Moscow ML phrase that is marked with a •*N* in the grammar is illegal Standard ML unless it satisfies restriction *N* below:

  1. A structure declaration `structure` *strbind* may only occur at top level, within the declarations of a structure, or within a declaration local to the declarations of a structure, but not within a Core `let`-expression.
  2. A functor declaration `functor` *funbind* or signature declaration `signature` *sigbind* may only occur at the top level of a program, but not within the declarations of a structure or Core `let`-expression.
  3. In any *typbind*, *datbind*, *datdesc* or *typreal*, any *tyvar* occurring within the right side must occur in the *tyvarseq* of the left side.
  4. A functor application `appmodexp atmodexp` must be an application of a functor identifier to a single argument of the form:
     funid (*modexp*)   or
     funid (*dec*)
     The parenthesised structure expressions (*modexp*) and (*dec*), although otherwise illegal in SML, are legal when occurring as a functor argument.
  5. A functor binding must define a one-argument, generative functor, and must have the form:
     funid (*modid* : *sigexp*) ⟨ *con* ⟩ = *modexp*
        ⟨ and *funbind* ⟩
  6. In a value specification `val` *tyvarseq valdesc*, *tyvarseq* must be empty, so that the specification is of the form:
     `val` *valdesc*
  7. In a datatype specification `datatype` *datdesc* ⟨ `withtype` *typbind* ⟩ the option must be absent, so that the specification is of the form:
     `datatype` *datdesc*

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$_1$, …, U$_n$, compile unit (U.sig or U.sml) (in *structure* mode)	Fail
compileToplevel	string list -> string -> unit	In context U$_1$, …, 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 $[f(0),\cdots,f(n\text{-}1)]$
`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.