

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

1	Moscow ML's relation to Standard ML	2
2	Reserved words	2
3	Comments	2
4	Special constants	2
5	Identifiers	3
6	Infix operators	5
7	Notational conventions used in the grammar	5
8	Grammar for the Moscow ML Core language	6
9	Interactive sessions	9
10	Grammar for the Moscow ML Modules language	10
11	Grammar for the Moscow ML Unit language	13
	11.1 Syntax and semantics for units compiled in <i>structure</i> mode	13
	11.2 Syntax and semantics for units compiled in <i>oplevel</i> mode	15
12	Further restrictions imposed for Standard ML compliance	17
13	Built-in types, constructors and exceptions	18
14	Built-in variables and functions	19
15	List of all library modules	23

The Moscow ML home page is <http://mosml.org>

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:

<code>\a</code>	A single character interpreted by the system as alert (BEL, ASCII 7).
<code>\b</code>	Backspace (BS, ASCII 8).
<code>\t</code>	Horizontal tab (HT, ASCII 9).
<code>\n</code>	Linefeed, also known as newline (LF, ASCII 10).
<code>\v</code>	Vertical tab (VT, ASCII 11).
<code>\f</code>	Form feed (FF, ASCII 12).
<code>\r</code>	Carriage return (CR, ASCII 13).
<code>\^c</code>	The control character <i>c</i> , where <i>c</i> may be any character with ASCII code 64–95 (@ to _). The ASCII code of <code>\^c</code> is 64 less than that of <i>c</i> .
<code>\ddd</code>	The character with code <i>ddd</i> (3 decimal digits denoting an integer 0–255).
<code>\uxxxx</code>	The character with code <i>xxxx</i> (4 hexadecimal digits denoting an integer 0–255).
<code>\"</code>	The double-quote character (")
<code>\\</code>	The backslash character (\)
<code>\f··f\</code>	This sequence is ignored, where <i>f··f</i> 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:	<code>#"a"</code>	<code>#"\n"</code>	<code>#"\^z"</code>	<code>#"\255"</code>	<code>#"\\""</code>
Non-examples:	<code># "a"</code>	<code>#c</code>	<code>#""</code>		

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:

<i>vid</i>	(value identifiers)	long
<i>tyvar</i>	(type variables)	
<i>tycon</i>	(type constructors)	long
<i>lab</i>	(record labels)	
<i>strid</i>	(structure identifiers)	long
<i>funid</i>	(functor identifiers)	long
<i>modid</i>	(module identifiers)	long
<i>sigid</i>	(signature identifiers)	
<i>unitid</i>	(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 '.' :

<i>longx</i>	::=	<i>x</i>	identifier
		<i>longstrid.x</i>	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 Infix 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 *exp₁ id exp₂* may occur, in parentheses if necessary, wherever the application *id (exp₁, exp₂)* or *id {1=exp₁, 2=exp₂}* 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-infix. The form of the fixity directives is as follows ($n \geq 1$):

<code>infix</code>	$\langle d \rangle$	$id_1 \cdots id_n$	left associative
<code>infixr</code>	$\langle d \rangle$	$id_1 \cdots id_n$	right associative
<code>nonfix</code>		$id_1 \cdots id_n$	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`. 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:

$xseq$	$::=$	x	(singleton sequence)
			(empty sequence)
		(x_1, \dots, 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.
- 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

<i>exp</i>	::=	<i>infexp</i> <i>exp</i> : <i>ty</i> <i>exp</i> ₁ andalso <i>exp</i> ₂ <i>exp</i> ₁ orelse <i>exp</i> ₂ <i>exp</i> handle <i>match</i> raise <i>exp</i> if <i>exp</i> ₁ then <i>exp</i> ₂ else <i>exp</i> ₃ while <i>exp</i> ₁ do <i>exp</i> ₂ case <i>exp</i> of <i>match</i> fn <i>match</i>	type constraint (L) sequential conjunction sequential disjunction handle exception raise exception conditional iteration case analysis function expression
<i>infexp</i>	::=	<i>appexp</i> <i>infexp</i> ₁ id <i>infexp</i> ₂	infix application
<i>appexp</i>	::=	<i>atexp</i> <i>appexp</i> <i>atexp</i>	application
<i>atexp</i>	::=	<i>scon</i> ⟨op⟩ <i>longvid</i> { { <i>exprow</i> } } # <i>lab</i> () (<i>exp</i> ₁ , … , <i>exp</i> _{<i>n</i>}) [<i>exp</i> ₁ , … , <i>exp</i> _{<i>n</i>}] #[<i>exp</i> ₁ , … , <i>exp</i> _{<i>n</i>}] (<i>exp</i> ₁ ; … ; <i>exp</i> _{<i>n</i>}) let <i>dec</i> in <i>exp</i> ₁ ; … ; <i>exp</i> _{<i>n</i>} end [structure <i>modexp</i> as <i>sigexp</i>] [functor <i>modexp</i> as <i>sigexp</i>] (<i>exp</i>)	special constant (see Section 4) value identifier record record selector 0-tuple <i>n</i> -tuple, <i>n</i> ≥ 2 list, <i>n</i> ≥ 0 vector, <i>n</i> ≥ 0 sequence, <i>n</i> ≥ 2 local declaration, <i>n</i> ≥ 1 structure package functor package
<i>exprow</i>	::=	<i>lab</i> = <i>exp</i> ⟨ , <i>exprow</i> ⟩	expression row
<i>match</i>	::=	<i>mrule</i> ⟨ <i>match</i> ⟩	
<i>mrule</i>	::=	<i>pat</i> => <i>exp</i>	

Declarations and Bindings

<pre> <i>dec</i> ::= val <i>tyvarseq valbind</i> fun <i>tyvarseq fvalbind</i> type <i>typbind</i> datatype <i>datbind</i> < withtype <i>typbind</i> > datatype <i>tycon</i> = datatype <i>tyconpath</i> abstype <i>datbind</i> < withtype <i>typbind</i> > with <i>dec</i> end exception <i>exbind</i> local <i>dec</i>₁ in <i>dec</i>₂ end open <i>longstrid</i>₁ ... <i>longstrid</i>_{<i>n</i>} structure <i>strbind</i> functor <i>funbind</i> signature <i>sigbind</i> <i>dec</i>₁ < ; > <i>dec</i>₂ infix < <i>d</i> > <i>id</i>₁ ... <i>id</i>_{<i>n</i>} infixr < <i>d</i> > <i>id</i>₁ ... <i>id</i>_{<i>n</i>} nonfix <i>id</i>₁ ... <i>id</i>_{<i>n</i>} </pre>	<pre> value declaration function declaration type declaration datatype declaration datatype replication abstype declaration exception declaration local declaration open declaration, <i>n</i> ≥ 1 structure declaration functor declaration signature declaration empty declaration sequential declaration infix (left) directive, <i>n</i> ≥ 1 infix (right) directive, <i>n</i> ≥ 1 nonfix directive, <i>n</i> ≥ 1 </pre>
<pre> <i>valbind</i> ::= <i>pat = exp</i> < and <i>valbind</i> > rec <i>valbind</i> </pre>	<pre> value binding recursive binding </pre>
<pre> <i>fvalbind</i> ::= < op > var <i>atpat</i>₁₁ ... <i>atpat</i>_{1<i>n</i>} < :ty > = <i>exp</i>₁ < op > var <i>atpat</i>₂₁ ... <i>atpat</i>_{2<i>n</i>} < :ty > = <i>exp</i>₂ ... < op > var <i>atpat</i>_{<i>m</i>1} ... <i>atpat</i>_{<i>m</i><i>n</i>} < :ty > = <i>exp</i>_{<i>m</i>} < and <i>fvalbind</i> > </pre>	<pre> <i>m, n</i> ≥ 1 </pre>
<pre> <i>typbind</i> ::= <i>tyvarseq tycon</i> = <i>ty</i> < and <i>typbind</i> > </pre>	<pre> •3 </pre>
<pre> <i>datbind</i> ::= <i>tyvarseq tycon</i> = <i>conbind</i> < and <i>datbind</i> > </pre>	<pre> •3 </pre>
<pre> <i>conbind</i> ::= < op > <i>vid</i> < of ty > < <i>conbind</i> > </pre>	
<pre> <i>exbind</i> ::= < op > <i>vid</i> < of ty > < and <i>exbind</i> > < op > <i>vid</i> = < op > <i>longvid</i> < and <i>exbind</i> > </pre>	

Note: In the *fvalbind* form above, if *var* has infix status then either *op* must be present, or *var* must be infix. 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

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

Patterns

$atpat$	$::=$	$_$ $scon$ $\langle op \rangle longvid$ { $\langle patrow \rangle$ } () (pat_1, \dots, pat_n) [pat_1, \dots, pat_n] # [pat_1, \dots, pat_n] (pat)	wildcard special constant (see Section 4) value identifier record 0-tuple n -tuple, $n \geq 2$ list, $n \geq 0$ vector, $n \geq 0$
$patrow$	$::=$	\dots $lab = pat \langle , patrow \rangle$ $lab \langle :ty \rangle \langle as pat \rangle \langle , patrow \rangle$	wildcard pattern row label as variable
pat	$::=$	$atpat$ $\langle op \rangle longvid atpat$ $pat_1 vid pat_2$ $pat : ty$ $\langle op \rangle var \langle :ty \rangle as pat$	atomic pattern constructed value infix value construction typed 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*, *tybind*, *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 *tybind* 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 explicitly 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

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

Module bindings

<i>strbind</i>	::=	<i>strid</i> < <i>con</i> > = <i>modexp</i> < and <i>strbind</i> > <i>strid</i> as <i>sigexp</i> = <i>exp</i> < and <i>strbind</i> >	structure binding package binding	•
<i>funbind</i>	::=	<i>funid</i> <i>arg</i> ₁ ... <i>arg</i> _{<i>n</i>} < <i>con</i> > = <i>modexp</i> < and <i>funbind</i> > <i>funid</i> (<i>spec</i>) < <i>con</i> > = <i>modexp</i> < and <i>funbind</i> > <i>funid</i> as <i>sigexp</i> = <i>exp</i> < and <i>funbind</i> >	functor binding, <i>n</i> ≥ 0 abbreviated generative binding package binding	•5 •
<i>sigbind</i>	::=	<i>sigid</i> = <i>sigexp</i> < and <i>sigbind</i> >	signature binding	
<i>con</i>	::=	: <i>sigexp</i> :> <i>sigexp</i>	transparent constraint opaque constraint	
<i>arg</i>	::=	(<i>modid</i> : <i>sigexp</i>) <i>modid</i> : <i>sigexp</i>	argument of generative functor argument of applicative functor	•

Signature expressions

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

(functor) binding, then *longmodid* must be interpreted as a structure (functor) identifier; if $\langle \text{op} \rangle$ *longmodid* occurs in the functor position of an application, then *longmodid* must be interpreted as a functor identifier; if $\langle \text{op} \rangle$ *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 $\langle \text{op} \rangle$ *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 $\langle \text{op} \rangle$ *longmodid* is the body of a functor. In this case, the optional prefix $\langle \text{op} \rangle$ 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 $\langle \text{op} \rangle$ *longmodid* is already determined by the context, the optional prefix $\langle \text{op} \rangle$ 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, $(\text{modid} : \text{sigexp})$ or $\text{modid} : \text{sigexp}$, 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*, *tydesc*, *tybind*, *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 *tydesc*, *tybind*, *datdesc*, or *tyreal*, 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 explicitly 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` or applicative functor `functor modid : 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'd 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`)

<i>unitimp</i>	<pre> ::= structure <i>unitid</i> = <i>modexp</i> structure <i>unitid</i> :> <i>unitid</i> = <i>modexp</i> <i>cdec</i></pre>	<pre> structure structure with signature core declaration</pre>	<pre> deprecated</pre>
<i>cdec</i>	<pre> ::= val <i>tyvarseq valbind</i> fun <i>tyvarseq fvalbind</i> type <i>typbind</i> datatype <i>datbind</i> < withtype <i>typbind</i> > datatype <i>tycon</i> = datatype <i>tyconpath</i> abstype <i>datbind</i> < withtype <i>typbind</i> > with <i>dec</i> end exception <i>exbind</i> local <i>dec</i>₁ in <i>dec</i>₂ end open <i>longstrid</i>₁ ... <i>longstrid</i>_{<i>n</i>} <i>cdec</i>₁ <;> <i>cdec</i>₂ infix <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} infixr <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} nonfix <i>id</i>₁ ... <i>id</i>_{<i>n</i>}</pre>	<pre> value declaration function declaration type declaration datatype declaration datatype replication abstype declaration exception declaration local declaration open declaration, <i>n</i> ≥ 1 empty declaration sequential declaration infix (left) directive, <i>n</i> ≥ 1 infix (right) directive, <i>n</i> ≥ 1 nonfix directive, <i>n</i> ≥ 1</pre>	

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

<i>unitint</i>	<pre> ::= signature <i>unitid</i> = <i>sigexp</i> <i>cspec</i></pre>	<pre> signature binding core specification</pre>	<pre> deprecated</pre>
<i>cspec</i>	<pre> ::= val <i>tyvarseq valdesc</i> type <i>tydesc</i> type <i>typbind</i> eqtype <i>tydesc</i> datatype <i>datdesc</i> < withtype <i>typbind</i> > datatype <i>tycon</i> = datatype <i>tyconpath</i> exception <i>exdesc</i> local <i>lspec</i> in <i>spec</i> end <i>cspec</i> <;> <i>cspec</i> infix <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} infixr <<i>d</i>> <i>id</i>₁ ... <i>id</i>_{<i>n</i>} nonfix <i>id</i>₁ ... <i>id</i>_{<i>n</i>}</pre>	<pre> value specification abstract type type abbreviation abstract equality type datatype with typbind datatype replication exception local specifications empty sequential infix (left) directive, <i>n</i> ≥ 1 infix (right) directive, <i>n</i> ≥ 1 nonfix directive, <i>n</i> ≥ 1</pre>	<pre> ●6 ●7 ● ● ● ●</pre>

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 *oplevel* mode

In *oplevel* 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 *oplevel* mode only if their filenames are preceded by the command line argument `-oplevel`.

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

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

The semantics of

```
- compileOplevel ["unitid1", ..., "unitidn"] "unitid.sig"; (* if unitid.sig exists *)
- compileOplevel ["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'd 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 using the implementation, as opposed to loading 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 *oplevel*-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
<code>realint</code>	<code>int, real</code>
<code>wordint</code>	<code>int, word, word8</code>
<code>num</code>	<code>int, real, word, word8</code>
<code>numtxt</code>	<code>int, real, word, word8, char, string</code>

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

Nonfix identifiers in the initial environment

<i>id</i>	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 (<i>s, first, len</i>)	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

Infix identifiers in the initial environment

<i>id</i>	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)

<i>id</i>	type	effect	exception
compile	string -> unit	compile unit (U.sig or U.sml) (in <i>structure</i> mode)	Fail
compileStructure	string list -> string -> unit	In context U_1, \dots, U_n , compile unit (U.sig or U.sml) (in <i>structure</i> mode)	Fail
compileToplevel	string list -> string -> unit	In context U_1, \dots, U_n , compile unit (U.sig or U.sml) (in <i>toplevel</i> 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 24. 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

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

List manipulation functions (module List)

<i>id</i>	type	effect
@	'a list * 'a list -> 'a list	append
all	('a -> bool) -> 'a list -> bool	if <i>p</i> true of all elements
app	('a -> unit) -> 'a list -> unit	apply <i>f</i> to all elements
concat	'a list list -> 'a list	concatenate lists
drop	'a list * int -> 'a list	drop <i>n</i> first elements
exists	('a -> bool) -> 'a list -> bool	if <i>p</i> true of some element
filter	('a -> bool) -> 'a list -> 'a list	the elements for which <i>p</i> is true
find	('a -> bool) -> 'a list -> 'a option	first element for which <i>p</i> 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 <i>f</i> to all elements
mapPartial	('a -> 'b option) -> 'a list -> 'b list	list of the non-NONE results of <i>f</i>
nth	'a list * int -> 'a	<i>n</i> '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 <i>p</i> , false for <i>p</i>)
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 [<i>f</i> (0), ..., <i>f</i> (<i>n</i> -1)]
take	'a list * int -> 'a list	take <i>n</i> 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`)

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

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

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

- 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.

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

<i>id</i>	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	succeeding 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.