# Moscow ML Owner's Manual

Version 1.42 of July 1997


Sergei Romanenko
Russian Academy of Sciences
Moscow, Russia

Peter Sestoft
Royal Veterinary and Agricultural University
Copenhagen, Denmark


Moscow ML implements the Core language of Standard ML (SML), as defined in the 1990 *Definition of Standard ML* and revised in 1996, and supports most required parts of the new SML Basis Library. Moscow ML also provides a simple subset of the Standard ML Modules language, restricted to signatures and non-nested structures. It supports separate compilation and the generation of stand-alone executables.

This document explains how to use the Moscow ML system. A companion document, the *Moscow ML Language Overview*, summarizes Moscow ML syntax and some built-in functions [8]. For a list of textbooks and other materials on Standard ML programming, see Section 13 below.


**Acknowledgements:** The Caml Light system was instrumental in creating Moscow ML, which uses its runtime system and essentially the same bytecode generator. Many other aspects of the design were derived from Caml Light, developed by Xavier Leroy and Damien Doligez at INRIA, France [3, 4]. The ML Kit helped solving problems of parsing, infix resolution, and type inference [1].


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

## Contents

# 1 Getting started

## 1.1 Installation

Get a copy of the Moscow ML system executables (see Section 12 for instructions) and unpack them in your home directory (under Unix) or in directory `C:\` (under DOS). This creates a directory `mosml`. Read the file `mosml/install.txt`. This manual, and the *Moscow ML Language Overview*, are in directory `mosml/doc`.

## 1.2 The interactive system

The interactive system is invoked by typing `mosml` at the shell prompt. It allows you to enter declarations and evaluate expressions:

```
$ mosml
Moscow ML version 1.42 (July 1997)
Enter `quit();' to quit.
- fun fac n = if n = 0 then 1 else n * fac (n-1);
> val fac = fn : int -> int
- fac 10;
> val it = 3628800 : int
-
```

You can quit the interactive session by typing '`quit();`' or control-D (under Unix) or control-Z followed by newline (under DOS). Type `help "lib";` for an overview of built-in function libraries, and e.g. `help "Array"` for help on `Array` operations. See Section 3 for further information on `mosml`.

## 1.3 The batch compiler and linker

The batch compiler and linker is invoked by typing `mosmlc` at the shell prompt. It can compile ML source files separately (`mosmlc -c`) and link them to obtain executables (`mosmlc -o`), in a manner similar to C compilers. See Section 5 for further information on `mosmlc`.

## 1.4 A simple module system

Moscow ML provides a simple subset of the Standard ML Modules language, restricted to signatures and non-nested structures. A Moscow ML program consists of one or more units. A unit U has a signature (or interface) in file U.sig and a body (or implementation) in file U.sml. The unit signature corresponds to a Standard ML signature, and the unit body corresponds to a Standard ML structure. Moscow ML supports type-safe separate compilation and linking. Section 4 gives the syntax and an informal semantics of compilation units. Section 6 explains automatic recompilation management.

## 1.5 What is new in release 1.42

- The linker (`mosml -o A.uo B.uo ...`) now automatically includes all (and only) those bytecode files referred to by the specified files `A.uo`, `B.uo`, etc. Option `-i` makes the linker report which files were linked. Option `-noautolink` disables the autolinker and gives you full control of the order of linking.
- A new structure Mosmlcgi provides support for writing CGI scripts in Moscow ML (thanks to Jonas Barklund).
- Structures Array2, CommandLine, and Option have been added to the Basis Library. Structure Old was renamed to SML90.
- Faster compilation, thanks to changes in compiler internals.
- Faster bytecode execution, thanks to Doug Currie.
- Explicit type variable parameters (introduced in SML'96) in val and fun declarations have been implemented.
- For other minor changes and fixes, see file mosml/doc/releases.txt.

# 2 Core language and libraries

Moscow ML implements the Core language of Standard ML as revised in 1996 [7, 6], and much of the Standard ML Basis Library [2], the most important omission being the functional stream input-output operations. The new second edition of Paulson's textbook *ML for the Working Programmer* uses the revised Core language and the new SML Basis Library.

## 2.1 The Standard ML Basis Library

The Standard ML Basis Library is a joint effort of the Standard ML of New Jersey, MLWorks, and Moscow ML developers[1] to enhance the portability of Standard ML programs.

The *Moscow ML Language Overview* [8] lists the library structures implemented by Moscow ML, and contains an index to all the identifiers they define. The same information is available also from `mosml`'s on-line help; see Section 3.1.

For a comprehensive description of the libraries, see the Basis Library documentation [2], which will become available from a commercial publisher. Currently it must be obtained from the Internet; see Section 13.

The Basis Library and the revised Standard ML language are slightly incompatible with both the 1990 *Definition of Standard ML* and with SML/NJ version 0.93. Invoking Moscow ML with '`mosml -P sml90`' gives a top-level environment compatible with the 1990 Definition. Invoking Moscow ML with option '`mosml -P nj93`', gives a top-level environment compatible with the old SML/NJ version 0.93. See Section 3.3 for more information on command-line options.

An important change in SML 1996 is the adoption of value polymorphism; see Section 7.

# 3 The interactive system

The interactive system `mosml` is invoked simply by typing `mosml` at the command line:

```
$ mosml
Moscow ML version 1.42 (July 1997)
Enter 'quit();' to quit.
-
```

The interactive system can be terminated by typing `quit();` and newline, or control-D (under Unix) or control-Z and newline (under DOS). Type '`help "";`' for help on built-in functions.

Invoking the interactive system with command line arguments

```
mosml file1 ... filen
```

is equivalent to invoking `mosml` and, when Moscow ML has started, entering

```
(use "file1"; ...; use "filen");
```

## 3.1 On-line help

In a `mosml` session, you may type `help "lib";` for an overview of built-in function libraries. To get help on a particular identifier, such as `fromString`, type

```
help "fromstring";
```

This will produce a menu of all library structures which contain the identifier `fromstring` (disregarding the lowercase/uppercase distinction):

[1]The Basis Library authors are Andrew Appel (Princeton, USA); Emden Gansner, John Reppy (AT&T Research, USA); Lal George, Lorenz Huelsbergen, Dave MacQueen (Bell Laboratories, USA); Matthew Arcus, Dave Berry, Richard Brooksby, Nick Barnes, Brian Monahan, Jon Thackray (Harlequin Ltd., Cambridge, England); Carsten Müller (Berlin, Germany); and Peter Sestoft (Royal Veterinary and Agricultural University, Denmark).

```
--------------------------------
|    1 | val  Bool.fromString   |
|    2 | val  Char.fromString   |
|    3 | val  Date.fromString   |
|    4 | val  Int.fromString    |
|    5 | val  Path.fromString   |
|    6 | val  Real.fromString   |
|    7 | val  String.fromString |
|    8 | val  Time.fromString   |
|    9 | val  Word.fromString   |
|   10 | val  Word8.fromString  |
--------------------------------
```

Choosing a number from this menu will invoke the help browser on the desired structure, e.g. `Int`. The help browser is primitive but easy to use. It works best with a window size of 24 lines.

The texts accessed by `help` are found in directory `mosml/lib`. For instance, all `List` functions are described in file `mosml/lib/List.sig`.

## 3.2 Editing and running ML programs

**Unix and Emacs** You may run `mosml` as a subshell under Emacs. You should use the `mosml`-version of the SML mode for Emacs; see file `mosml/utility/emacs` for instructions. In case of errors, Emacs can interpret `mosml`'s error messages and jump to the offending piece of source code. This is very convenient.

**Window systems** In a window-oriented system, such as MacOS, MS Windows, or the X window system, you may run `mosml` in one window and edit source code in another. After (re-)editing the source file, you must issue a `use` command in the `mosml` window.

**MS DOS** You may use the simple `edit` script to invoke an editor from inside a `mosml` session; see file `mosml\utility\dosedit` for instructions. You will not need to quit the `mosml` session to edit a source file, and the script will automatically reload the newly edited file.

## 3.3 Command-line options

`-I` directory
: Specify the default directories to be searched by `use`, `compile` and `load`. A call to `use`, `compile` or `load` will first search the current directory, then all directories specified by option '`-I`' in order of appearance from left to right, and finally the standard library directory.

`-valuepoly`
: Specify that the type checker should use 'value polymorphism'; see Section 7. The default.

`-imptypes`
: Specify that the type checker should distinguish imperative and applicative type variables, generalize all applicative type variables, and generalize imperative type variables only in non-expansive expressions. See Section 7.

`-P` unit-set
: Determines which library units will be included and open at compile-time, and which library units will be linked in at link-time:

  `-P default` This provides an initial environment for the new Basis Library, compatible with future versions of SML/NJ. The units `Array`, `Char`, `List`, `String`, and `Vector` will be loaded, and units `Char`, `List`, and `String` will be partially opened. This is the default.

  `-P sml90` This provides an initial environment which is upwards compatible with that of the *Definition of Standard ML* and with pre-1.30 releases of Moscow ML. In particular, the functions `chr`, `explode`, `implode`, and `ord` work on strings, not characters. The new versions of these functions are still available as `Char.chr`, `Char.ord`, `String.explode`, and `String.implode`. The math functions and input-output facilities required by the Definition [6, Appendix C and D] are available at top-level. In addition the same (new) libraries are loaded as with `-P default`.

`-P nj93` This provides a top-level environment which is mostly compatible with that of SML/NJ 0.93. The functions `app`, `ceiling`, `chr`, `dec`, `explode`, `fold`, `hd`, `implode`, `inc`, `max`, `min`, `nth`, `nthtail`, `ord`, `ordof`, `revapp`, `revfold`, `substring`, `tl`, and `truncate` have the same type and meaning as in SML/NJ 0.93. Note that this is incompatible with future versions of SML/NJ. The math functions and input-output facilities required by the Definition [6, Appendix C and D] are available at top-level. In addition the same (new) libraries are loaded as with `-P default`. This option does *not* imply `-valuepoly` or `-imptypes`.

`-P full` This loads all the libraries marked `F` in the library list (see [8]), and partially opens the `Char`, `List`, and `String` units.

`-P parsing` Loads the `full` libraries, plus support libraries for lexers and parsers generated by `mosmllex` (Section 9) and `mosmlyac` (Section 10).

`-P none` No library units are loaded or opened initially.

Additional library units can loaded into the interactive system by using the `load` function; see Section 3.4 below.

`-stdlib` stdlib-directory

Specify the standard library directory to be stdlib-directory. The default standard library is usually `mosml/lib` under Unix and `c:\mosml\lib` under DOS.

### 3.4 Non-standard primitives in the interactive system

The following non-standard primitives are defined in unit `Meta`, loaded (and open by default) only in the interactive system. Hence these primitives cannot be used from source files which are compiled separately. The functions `compile` and `load` deal with Moscow ML compilation units; see Section 4.

`printVal : 'a -> 'a`

`printVal` is a polymorphic function provided as a quick debugging aid. This is an identity function, which as a side-effect prints its argument to standard output exactly as it would be printed at top-level. Output is flushed immediately. For printing strings, the function `print` is probably more useful than `printVal`.

`printDepth : int ref`

This variable determines the depth (in terms of nested constructors, records, tuples, lists, and vectors) to which values are printed by the top-level value printer and the function `printVal`. The components of the value whose depth is greater than `printDepth` are printed as '`#`'. The initial value of `printDepth` is 20. This value can be changed by the user's program at any moment.

`printLength : int ref`

This variable determines the way in which list values are printed by the top-level value printer and the function `printVal`. If the length of a list is greater than `printLength`, only the first `printLength` elements are printed, and the remaining elements are printed as '...'. The initial value of `printLength` is 200. This value can be changed by the user's program at any moment.

`quit : unit -> unit`

Evaluating `quit()` quits Moscow ML immediately.

`installPP : (ppstream -> 'a -> unit) -> unit`

Evaluating `installPP` *pp* installs the prettyprinter *pp* at type `ty`, provided *pp* has type `ppstream -> ty -> unit`. The type `ty` must be a nullary (parameter-less) type constructor representing a datatype, either built-in (such as bool) or user-defined. Whenever a value of type `ty` is about to be printed by the interactive system, and whenever function `printVal` is invoked on an argument of type `ty`, the prettyprinter *pp* will be invoked to print it.

The prettyprinter *pp* must not call the `flush_ppstream` function. See library unit `PP` for more information on prettyprinters.

`use : string -> unit`

Evaluating `use "`*f*`"` causes ML declarations to be read from file *f* as if they were entered from the console. A file loaded by `use` may, in turn, evaluate calls to `use`. For best results, use `use` only at top level, or at top level within a `use`'d file.

`compile : string -> unit`
: Evaluating `compile "U.sig"` will compile and elaborate the unit signature in file U.sig, producing a compiled signature file U.ui. During compilation, the compiled signatures of other units will be accessed if they are mentioned in U.sig.
: Evaluating `compile "U.sml"` will elaborate and compile the unit body in file U.sml, producing a bytecode file U.uo. If there is an explicit signature U.sig, then file U.ui must exist, and the unit body must match the signature. If there is no U.sig, then an inferred signature file U.ui will be produced also. No evaluation takes place. During compilation, the compiled signatures of other units will be accessed if they are mentioned in U.sml.
: The declared identifiers will be reported if `verbose` is `true` (see below); otherwise compilation will be silent. In any case, compilation warnings are reported, and compilation errors abort the compilation and raise the exception `Fail` with a string argument.

`load : string -> unit`
: Evaluating `load "U"` will load and evaluate the compiled unit body from file U.uo. The resulting values are not reported, but exceptions are reported, and cause evaluation and loading to stop. If U is already loaded, then `load "U"` has no effect. If any other unit is mentioned by U but not yet loaded, then it will be loaded automatically before U.
: After loading a unit, it can be opened with `open U`. Opening it at top-level will list the identifiers declared in the unit.
: When loading U, it is checked that the signatures of units mentioned by U agree with the signatures used when compiling U, and it is checked that the signature of U has not been modified since U was compiled; these checks are necessary for type safety. The exception `Fail` is raised if the signature checks fail, or if the file containing U or a unit mentioned by U does not exist.

`loadOne : string -> unit`
: Evaluating `loadOne "U"` is similar to `load "U"`, but raises exception `Fail` if U is already loaded or if some unit mentioned by U is not yet loaded. That is, it does not automatically load any units mentioned by U. It performs the same signature checks as `load`.

`exnName : exn -> string`
: Returns a name for the exception constructor in the exception. Never raises an exception itself. The name returned may be that of any exception constructor aliasing with exn. For instance, `let exception E1; exception E2 = E1 in exnName E2 end` may evaluate to `"E1"` or `"E2"`.

`exnMessage : exn -> string`
: Formats and returns a message corresponding to the exception. For the exceptions defined in the SML Basis Library, the message will include the argument carried by the exception.

`verbose : bool ref`
: Determines whether the signature inferred by a call to `compile` will be printed. The printed signature follows the syntax of Moscow ML signatures, so the output of `compile "U.sml"` can be edited to subsequently create file U.sig. The default value is `false`.

`quotation : bool ref`
: Determines whether quotations and antiquotations are permitted in declarations entered at top-level and in files compiled with `compile`; see Section 8 for a brief explanation. When `quotation` is `false` (the default), the backquote character is an ordinary symbol which can be used in ML symbolic identifiers. When `quotation` is `true`, the backquote character is illegal in symbolic identifiers, and a quotation `` `a b c` `` will be evaluated to an object of type `'a frag list`.

`valuepoly : bool ref`
: Determines whether the type checker should use 'value polymorphism'; see Section 7. Command-line option `-valuepoly` sets `valuepoly` to `true` (the default), whereas option `-imptypes` sets `valuepoly` to `false`; see Sections 3.3 and 5.2.

`system : string -> int`
: Evaluating `system "`*com*`"` causes the command *com* to be executed by the operating system. If a non-zero integer is returned, this must indicate that the operating system has failed to execute the command. Under MS DOS, the integer returned always equals 0.

# 4 Modules and compilation units

## 4.1 Basic concepts

A Moscow ML program can consist of one or more *compilation units*, or *units* for short. A compilation unit consists of an optional *unit signature* and a *unit body*. The unit signature specifies the contents of the unit; it is an interface to the unit. The unit body declares the contents of the unit; it provides an implementation of the unit. The following analogies may be helpful:

| | | |
|---|---|---|
| Moscow ML | unit signature | unit body |
| Standard ML | signature | structure |
| Caml Light | module interface | module implementation |
| Modula-2 | interface module | implementation module |

The unit body is always present, whereas the signature can be omitted. When the unit signature is present, it is called the *explicit signature* to distinguish it from the signature inferred when elaborating the unit body. When present, the explicit signature must be matched by the body, and only those identifiers specified in the signature are visible outside the unit. If no signature is given, all identifiers visible at the end of the unit body are visible outside the unit.

Units are closely associated with files, as in Modula-2. The body of the unit called 'U' is defined in a file called 'U.sml', and its explicit signature (if any) in file 'U.sig'.

| Files containing program text: | |
|---|---|
| U.sig | unit signature, specifications |
| U.sml | unit body, declarations |

| Files created by the compiler: | |
|---|---|
| U.ui | compiled unit signature |
| U.uo | compiled unit body, bytecode |

## 4.2 Units without explicit signature

A unit U without an explicit signature consists of a file U.sml containing

```
structure U = struct
    ... declarations ...
end
```

This is the same as a simple SML structure declaration. There must be no corresponding explicit signature file U.sig.

## 4.3 Units with explicit signature

A unit U with an explicit signature consists of a signature file U.sig containing

```
signature U = sig
    ... specifications ...
end
```

and a file U.sml, containing

```
structure U :> U = struct
    ... declarations ...
end
```

This is the same as a SML structure declaration with an opaque signature constraint. Note that the file name, signature name, and structure name must be the same. The notation '`U :> U`' is an opaque signature constraint[2], meaning that other units have no access to the internals of U.sml, only to the signature U.sig.

To illustrate the difference between transparent and opaque signature constraints, consider the Standard ML (not Moscow ML) declarations:

[2] Corresponding to `abstraction U : U = struct ... end` in MacQueen's original proposal for SML Modules [5]. Standard ML of NJ implemented that syntax; the new syntax '`U :> U`' is required by the 1996 revision of SML.

```
signature SIG = sig
    type t
    val x: t
end;
```

```
structure S: SIG = struct
    type t = int
    val x = 17
end
```

Given these declarations, the expression `S.x+33` will typecheck. Although the signature `SIG` just says that there exists a type `t`, constraining `S` with `SIG` does not hide the fact that `S.x` is actually an integer.

On the other hand, an opaque signature constraint, as in an `abstraction` and in Moscow ML units, *does* hide the true nature of `t` and `x`:

```
structure M :> SIG = struct
    type t = int
    val x = 17
end
```

After this declaration, `M.x+33` would fail to typecheck: the type checker cannot see that `M.t` is `int` and `M.x` is an integer. Often such hiding is just what is needed for software engineering purposes.

## 4.4 Syntax of unit signatures

Moscow ML unit signatures are very similar to Standard ML signatures as defined in [6, Figure 7]; the differences are explained below. A unit signature (in file U.sig) has the form:

| | | | |
|---|---|---|---|
| *unitsig* | ::= | `signature` *unitid* `=` `sig` *uspec* `end` | named signature |
| | | *uspec* | signature (old syntax) |
| *uspec* | ::= | `val` *valdesc* | value specification |
| | | `type` *typdesc* | abstract type |
| | | `type` *typbind* | type abbreviation |
| | | `eqtype` *typdesc* | abstract equality type |
| | | `datatype` *datbind* | datatype |
| | | `datatype` *datbind* `withtype` *typbind* | datatype with typbind |
| | | `exception` *exdesc* | exception |
| | | `local` *lspec* `in` *uspec* `end` | local specifications |
| | | | empty |
| | | *uspec* ⟨;⟩ *uspec* | sequential |
| *lspec* | ::= | `open` $unitid_1 \cdots unitid_n$ | open other units |
| | | `type` *typbind* | type abbreviation |
| | | `local` *lspec* `in` *lspec* `end` | local specifications |
| | | | empty |
| | | *lspec* ⟨;⟩ *lspec* | sequential |

Note:

1. Type abbreviations `type` *typbind* are permitted in signatures.
2. There are no structure specifications and no sharing specifications.
3. No type, value, or exception may be specified twice at top-level.
4. A `local` specification can be used only to restrict the scope of `open` specifications and type abbreviations.
5. An `open` specification can appear only inside `local`.
6. The '`signature` *unitid* `= sig`' and '`end`' parts may be left out, although this is not recommended.

Restriction (2) is the most significant one. Restriction (3), and restrictions similar to (4) and (5), are imposed by the Standard ML of New Jersey implementation also.

### 4.5 Syntax of unit bodies

A unit body (in file U.sml) has the form:

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

A long identifier can refer to entities declared in other units. In Moscow ML, the syntax of long identifiers is:

| | | | |
|---|---|---|---|
| *longid* | ::= | *id* | identifier |
| | | *unitid.id* | qualified identifier |

where *unitid* and *id* are arbitrary SML identifiers (either symbolic or alphanumeric).

A qualified identifier *unitid.id* denotes an entity *id* declared in the compilation unit *unitid*. A qualified identifier can denote either a value variable, a value constructor, an exception constructor, or a type constructor. As in Standard ML, a *longid* appearing in a defining position, such as a value variable in a pattern, cannot have a qualifier: the identifier being defined will always belong to the current unit.

An `open` declaration has the form

`open` $\mathrm{U}_1 \cdots \mathrm{U}_n$

where $\mathrm{U}_1 \cdots \mathrm{U}_n$ are names of units. The units are opened from left to right, in the order $\mathrm{U}_1 \cdots \mathrm{U}_n$. The text following an `open` U declaration can reference identifiers declared in U without explicitly specifying the name of the unit, subject to the usual scope rules of Standard ML. That is, one can use *id* instead of U.*id*.

In the interactive system, a unit must be loaded before it can be opened. In the batch compilation system, the linker links in (only) the needed declarations from opened units.

A unit body U.sml must elaborate to a structure S. If there is an explicit signature U.sig corresponding to U.sml, then the resulting structure must match the explicit signature. As in Standard ML (but in contrast to Caml Light), no reference is made to the signature while elaborating the unit body.

### 4.6 An example program consisting of three units

To illustrate the module system, we present a tiny program working with arithmetic expressions. It consists of three units `Expr`, `Reduce`, and `Evaluate`. This example is in `mosml/examples/manual`.

File `Expr.sml` below contains structure `Expr`, which defines a datatype `expr` for representing expressions and a function `show` to display them. It has no signature constraint and therefore exports both the datatype and the function:

```
structure Expr =
struct
    datatype expr = Cst of int | Neg of expr | Plus of expr * expr

    fun show (Cst n)          = makestring n
      | show (Neg e)          = "(-" ^ show e ^ ")"
      | show (Plus (e1, e2)) = "(" ^ show e1 ^ "+" ^ show e2 ^ ")"
end
```

File `Reduce.sig` below contains the signature `Reduce`, which specifies a function for reducing expressions. It mentions the type `Expr.expr` from `Expr`:

```
signature Reduce =
sig
    val reduce : Expr.expr -> Expr.expr
end
```

File `Reduce.sml` below contains the structure `Reduce`, which has a signature constraint, and therefore exports only the function `reduce` specified in the signature:

```
structure Reduce :> Reduce =
struct
    local open Expr
    in
        fun negate (Neg e) = e
          | negate e       = Neg e
        fun reduce (Neg (Neg e))      = e
          | reduce (Neg e)            = negate (reduce e)
          | reduce (Plus (Cst 0, e2)) = reduce e2
          | reduce (Plus (e1, Cst 0)) = reduce e1
          | reduce (Plus (e1, e2))    = Plus (reduce e1, reduce e2)
          | reduce e                  = e
    end
end
```

File `Evaluate.sig` below contains the signature `Evaluate`, which specifies a function `eval` for evaluating expressions, and a function `test`. Note the use of '`open Expr`' to make the type `expr` refer to `Expr.expr`:

```
signature Evaluate =
sig
    local open Expr
    in
        val eval : expr -> int
        val test : expr -> bool
    end
end
```

File `Evaluate.sml` below contains structure `Evaluate`, which has a signature constraint, and mentions unit `Expr` as well as `Reduce`:

```
structure Evaluate :> Evaluate =
struct
    local open Expr
    in
        fun eval (Cst n)         = n
          | eval (Neg e)         = ~ (eval e)
          | eval (Plus (e1, e2)) = eval e1 + eval e2;
        fun test e = (eval e = eval (Reduce.reduce e))
    end
end
```

## 4.7 Compiling, linking, and loading units

Units can be compiled and linked using the batch compiler `mosmlc`; see Section 5. Units compiled with option `-c` can be linked together. Use `mosml -o mosmlout A.uo` to produce a linked executable bytecode file `mosmlout` which will invoke the runtime system `camlrunm`. Use `mosml -noheader -o mosmlout A.uo` to produce a linked bytecode file which can be executed by `camlrunm mosmlout`. The linker will automatically link any required bytecode files into `mosmlout`. See Section 5.2 for more options.

Units can also be compiled from and loaded into the interactive system `mosml` using the primitives `compile` and `load`; see Section 3.4 above.

## 4.8 Organizing programs for compatibility with SML Modules

Moscow ML 1.42 and Standard ML of New Jersey (version 109 and later) implement the same core language, and many of the same libraries. Here we give advice on organizing structures and signatures so that they can be compiled by both systems.

Assume we have a software system consisting of three structures A, B, and C, where A and B each have a signature constraint, but C does not. Assume further that C depends on A and B. (There must be no functors or nested structures in A and B). We organize them in five files:

| Source file | File contents |
|---|---|
| A.sig | `signature A = sig ... end` |
| B.sig | `signature B = sig ... end` |
| A.sml | `structure A :> A = struct ... end` |
| B.sml | `structure B :> B = struct ... end` |
| C.sml | `structure C = struct ... A.foo ... B.bar ... end` |

Now we can compile these files using `mosmlc` and load them into a `mosml` session as follows (where '`$`' is the shell prompt and '`-`' is the ML prompt):

```
$ mosmlc -c A.sig B.sig A.sml B.sml C.sml
$ mosml
- load "C";
```

Or, we can load and compile them in an SML/NJ session as follows:

```
$ sml
- app use ["A.sig", "B.sig", "A.sml", "B.sml", "C.sml"];
```

Hence the same source files can be used unmodified in both systems. (However, as of version 109.14, Standard ML of New Jersey does not support the new syntax `structure A :> A = ...` for opaque signature constraints. For this reason, Moscow ML accepts the old syntax `abstraction A : A = ...` also).

Note that in Moscow ML, `mosmlc` will create bytecode files A.ui, A.uo, and so on. The `load` function does not perform any compilation and hence is very fast. If the source files do not change, there is no need to recompile them with `mosmlc`, which may save much time.

If the source files *do* change, and have to be recompiled at every use, it may be more practical to use the function `compile`:

```
$ mosml
- app compile ["A.sig", "B.sig", "A.sml", "B.sml", "C.sml"];
- load "C";
```

## 4.9 Matching a unit body against a signature

A unit body S matches a signature SIG under the conditions described in the Definition of Standard ML [6, Section 5.12]. Roughly, this means:

- a value specification `val` *v:t* must be matched by a value variable or value constructor or exception constructor *v* in S whose type generalizes *t*
- a type abbreviation `type` $t = ty$ must be matched by the same type abbreviation $t = ty$ in S
- an abstract type *t* must be matched by some type *t* in S
- an abstract equality type *t* must be matched by a type *t* in S admitting equality
- a datatype must be matched by precisely the same datatype in S
- an exception constructor *E* of type *t* must be matched by an exception constructor *E* in S whose type generalizes *t*

Moreover, to facilitate separate compilation, there are some representation constraints:

1. If the specified argument type of a value constructor (in a datatype specification) is an explicit tuple or record, then the declared argument type must be an explicit tuple or record also, and vice versa. This restriction does not apply if there is only one constructor in the datatype.
2. The order of value constructors in a datatype specification must be the same as in the matching datatype declaration.

# 5 The batch compiler

Moscow ML includes a batch compiler `mosmlc` in addition to the interactive system `mosml`. It compiles and links programs non-interactively, and can turn them into standalone executable files. The batch compiler can be invoked from a Makefile, which simplifies the (re)compilation of large programs considerably; see Section 6.

## 5.1 Overview

The `mosmlc` command has a command-line interface similar to that of most C compilers. It accepts several types of arguments: source files for unit bodies, source files for unit signatures, and compiled unit bodies.

- An argument ending in .sig is taken to be the name of a source file containing a unit signature. Given a file U.sig, the compiler produces a compiled signature in the file U.ui.
- An argument ending in .sml is taken to be the name of a source file containing a unit body. Given a file U.sml, the compiler produces compiled object code in the file U.uo. It also produces an inferred signature file U.ui if there is no explicit signature U.sig.
- An argument ending in .uo is taken to be the name of a compiled unit body. Such files are linked together, along with the compiled unit bodies obtained by compiling .sml arguments (if any), and the necessary Moscow ML library files, to produce a standalone executable program. The order in which .uo and .sml arguments are presented (from left to right) on the command line is significant: top-level declarations are evaluated and global identifiers are initialized in that order at run-time. In particular, file U.uo must come before any file which mentions identifiers U.*id* defined in U or which contains a declaration `open` U.

The output of the linking phase is a file containing compiled code that can be executed by the runtime system `camlrunm`. If `mosmlout` is the name of the file produced by the linking phase, the command

```
camlrunm mosmlout arg1 arg2 ... argn
```

executes the compiled code contained in `mosml.out`. The arguments can be accessed from the Moscow ML program via function `Mosml.argv`; see file `mosml/lib/Mosml.sig`.

**MS DOS:** If the output file produced by the linking phase has extension `.exe`, and option `-noheader` is not used, then the file is directly executable. Hence, an output file named `mosmlout.exe` can be executed with the command

```
mosmlout arg1 arg2 ... argn
```

The output file `mosmlout.exe` consists of a tiny executable file prepended to a linked bytecode file. The executable invokes the `camlrunm` runtime system to interpret the bytecode. As a consequence, this is not a standalone executable: it still requires `camlrunm.exe` to reside in one of the directories in the path.

**Unix:** The output file produced by the linking phase is directly executable (unless the `-noheader` option is used). It automatically invokes the `camlrunm` runtime system, either using a tiny executable prepended to the linked bytecode file, or using the Unix incantation `#!/usr/local/bin/camlrunm`. In the former case, `camlrunm` must be in one of the directories in the path; in the latter case it must be in `/usr/local/bin`. To create a true stand-alone executable you may simply concatenate the runtime system with the bytecode file produced by `mosmlc -noheader`, but this adds 60–150 KB to the size of the executable, depending on your version of Unix:

```
cat /usr/local/bin/camlrunm mosmlout > mosmlbin
chmod a+x mosmlbin
```

## 5.2 Command-line options

The following command-line options are recognized by `mosmlc`.

`-c`
: Compile only. Suppresses the linking phase of the compilation. Source code files are turned into compiled files (.ui and .uo), but no executable file is produced. This option is useful for compiling separate units.

`-files` response-file
: Pass the names of files listed in file response-file to the linking phase just as if these names appeared on the command line. File names in response-file are separated by blanks (spaces, tabs, newlines) and must end either in .sml or .uo. A name U.sml appearing in the response file is equivalent to U.uo. Use this option to overcome silly limitations on the length of the command line (as in DOS).

`-g`
: This option causes some information about exception names to be written at the end of the executable bytecode file.

`-i`
: Causes the compiler to print the inferred signature of the unit body or bodies being compiled. Also causes the linker to list all object files linked. A U.sig file corresponding to a given U.sml file can be produced semi-automatically by piping the output of the compiler to a file U.out, and subsequently editing this file to obtain a file U.sig.

`-noautolink`
: In version 1.42 and later, the linker automatically links in any additional object files required by the files explicitly specified on the command line. Option `-noautolink` reinstates the behaviour of previous versions: all object files must be explicitly specified in the appropriate order.

`-stdlib` stdlib-directory
: Specifies the standard library directory, which will be searched by the compiler and linker for the .ui and .uo files corresponding to units mentioned in the files being linked. The default standard library is set when the system is created, and is usually `${HOME}/mosml/lib` under Unix and `c:\mosml\lib` under DOS.

`-I` directory
: Add the given directory to the list of directories searched for compiled signature files (.ui) and compiled object code files (.uo). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, but before the standard library directory. When several directories are added with several `-I` options on the command line, these directories are searched from left to right.

`-valuepoly`
: Specify that the type checker should use 'value polymorphism'; see Section 7. The default.

`-imptypes`
: Specify that the type checker should distinguish imperative and applicative type variables, generalize all applicative type variables, and generalize imperative type variables only in non-expansive expressions. See Section 7.

`-o` exec-file
: Specify the name of the output file produced by the linker. In the absence of this option, a default name is used. In MS DOS, the default name is `mosmlout.exe`; in Unix it is `a.out`.

`-P` unit-set
: Determines which library units will be open at compile-time.

  - `-P default` This provides an initial environment for the new Basis Library, compatible with future versions of SML/NJ. The units `Array`, `Char`, `List`, `String`, and `Vector` will be available for linking, and units `Char`, `List`, and `String` will be partially opened. This is the default.
  - `-P sml90` This provides an initial environment which is upwards compatible with that of the *Definition of Standard ML* and with pre-1.30 releases of Moscow ML. In particular, the functions `chr`, `explode`, `implode`, and `ord` work on strings, not characters. The new versions of these functions are available as `Char.chr`, `Char.ord`, `String.explode`, and `String.implode`. The

math functions and input-output facilities required by the Definition [6, Appendix C and D] are available at top-level. In addition the same (new) libraries are available for linking as with `-P default`.

`-P nj93` This provides a top-level environment which is mostly compatible with that of SML/NJ 0.93. The functions `app`, `ceiling`, `chr`, `dec`, `explode`, `fold`, `hd`, `implode`, `inc`, `max`, `min`, `nth`, `nthtail`, `ord`, `ordof`, `revapp`, `revfold`, `substring`, `tl`, and `truncate` have the same type and meaning as in SML/NJ 0.93. The math functions and input-output facilities required by the Definition [6, Appendix C and D] are available at top-level. In addition the same (new) libraries are available for linking as with `-P default`. This option does *not* imply `-valuepoly` or `-imptypes`.

`-P full` This makes all the libraries marked `F` in the library list (see [8]) available for linking, and partially opens the `Char`, `List`, and `String` units.

`-P none` No library units are available for linking or initially opened.

Additional library units to be linked in can be specified by listing them on the command line, or by using the `-I` directory option.

`-noheader`

Causes the output file produced by the linker to contain only the bytecode, not preceded by any executable code. A file `mosmlout` thus obtained can be executed only by explicitly invoking the runtime system as follows: `camlrunm mosmlout`. This option is primarily used for recompiling the system.

`-q`

Enable the quotation/antiquotation mechanism; see Section 8.

`-v`

Print the version number of the various passes of the compiler.

# 6 Recompilation management

Recompilation management helps the programmer recompile only what is necessary after a change to a unit signature or unit body.

Type-safe linking prevents the programmer from creating unsafe or meaningless programs. The `load` function and the batch linker ensure probabilistically type-safe linking, so it is virtually impossible to cause the system to create a type-unsafe program.

## 6.1 Using 'make' to manage recompilation

Consider the example program in Section 4.6 consisting of the three units Evaluate, Expr, and Reduce. Assume their source files *.sig and *.sml reside in a particular directory. Copy a Makefile stub (see below) to that directory, and change to that directory.

1. Edit the Makefile so that the names of the bytecode files Evaluate.uo, Expr.uo, and Reduce.uo appear on the line beginning with '`all:`' (see the example makefiles below).
2. Compute the dependencies among the files by executing:

```
make depend
```

3. Recompile all those files which have not yet been compiled, or which have been modified but not yet recompiled, or which depend on modified files, by executing:

```
make
```

Step (3) must be repeated whenever you have modified a component of the program system. Step (2) need only be repeated if the inter-dependencies of some components change, or if you add or remove an explicit signature file. Step (1) need only be repeated when you add or delete an entire unit of the program system.

Old versions of the compiled *.ui and *.uo files can be removed by executing:

```
make clean
```

The inter-dependencies are computed by a small ML program `mosmldep`, which correctly handles nested comments and strings in the source files.

## 6.2 An example Makefile for Unix

To use the Makefile below, first edit it so that all the required units (.uo files) appear on the line beginning with '`all:`', then proceed as explained in Section 6.1. You do not need to edit any other part of the Makefile. In particular, the dependencies following `DO NOT DELETE THIS LINE` are generated automatically when executing `make depend` (as above). A copy of the Makefile can be found in `mosml/tools/Makefile.stub`.

You will need only the Unix utility `make`.

```
# Unix Makefile stub for separate compilation with Moscow ML.

MOSMLHOME=${HOME}/mosml
MOSMLTOOLS=camlrunm $(MOSMLHOME)/tools
MOSMLC=mosmlc -c
MOSMLL=mosmlc
MOSMLLEX=mosmllex
MOSMLYACC=mosmlyac

.SUFFIXES :
.SUFFIXES : .sig .sml .ui .uo

all: Evaluate.uo Expr.uo Reduce.uo

clean:
        rm -f *.ui
        rm -f *.uo
        rm -f Makefile.bak

.sig.ui:
        $(MOSMLC) $<

.sml.uo:
        $(MOSMLC) $<

depend:
        rm -f Makefile.bak
        mv Makefile Makefile.bak
        $(MOSMLTOOLS)/cutdeps < Makefile.bak > Makefile
        $(MOSMLTOOLS)/mosmldep >> Makefile

### DO NOT DELETE THIS LINE
Evaluate.ui: Expr.uo
Evaluate.uo: Evaluate.ui Expr.uo Reduce.ui
Reduce.uo: Reduce.ui Expr.uo
Reduce.ui: Expr.uo
```

## 6.3 An example Makefile for MS DOS

To use the Makefile below, first edit it so that all the required units (.uo files) appear on the line beginning with '`all:`', then proceed as explained in Section 6.1. You do not need to edit any other part of the Makefile. In particular, the dependencies following `DO NOT DELETE THIS LINE` are generated automatically when executing `make depend` (as above). A copy of this makefile can be found in `mosml\tools\makefile.stb`.

You will need a DOS version of `make`, such as that from Borland C++ version 2.0 or 3.0.

```
# DOS Makefile stub for separate compilation with Moscow ML.

MOSMLHOME=c:\mosml
MOSMLTOOLS=camlrunm $(MOSMLHOME)\tools
MOSMLC=mosmlc -c
MOSMLL=mosmlc
MOSMLLEX=mosmllex
MOSMLYACC=mosmlyac

all: evaluate.uo expr.uo reduce.uo

clean:
        del *.ui
        del *.uo
        del makefile.bak

.sig.ui:
        $(MOSMLC) $<

.sml.uo:
        $(MOSMLC) $<

depend:
        del makefile.bak
        ren makefile makefile.bak
        $(MOSMLTOOLS)\cutdeps < makefile.bak > makefile
        $(MOSMLTOOLS)\mosmldep >> makefile

### DO NOT DELETE THIS LINE
evaluate.uo: evaluate.ui expr.uo reduce.ui
reduce.ui: expr.uo
reduce.uo: reduce.ui expr.uo
evaluate.ui: expr.uo
```

## 6.4 Unit names and DOS file names

Recompilation management for DOS is essentially as for Unix, except for the usual complications that follow from the restrictions on the length of file names, and from their case-insensitivity.

Under MS DOS, filenames are all the same case and can be at most 8 characters long (plus a 3 character extension). Since file names are used as unit names, this may cause problems. We attempt to circumvent these problems as follows:

- Unit names used inside ML programs under DOS are 'normalized': the first character is made upper case (if it is a letter), all other characters are made lower case, and the unit name is truncated to eight characters. Hence a unit which resides in file `commands.sml` can be referred to as unit `Commands` inside an ML program, and can also be referred to as `CommandStructure`, etc., since normalization transforms the latter into the former.
- The following names are exceptions to this rule: `BasicIO`, `BinIO`, `CharArray`, `CharVector`, `CommandLine`, `FileSys`, `ListPair`, `OS`, `StringCvt`, `Substring`, `TextIO`, `Word8Array`, `Word8Vector`; they are normalized precisely as shown in this list. This is to accommodate the new Basis Library.
- In DOS makefiles, the file names appearing after `all:` must be all lower case and at most 8 characters long (otherwise 'make' will not work properly). For instance, the unit `CharArray` must be called `chararra` in a DOS makefile.
- A unit name given as argument to `load`, to `compile`, or to the batch compiler, is truncated and made lower case by DOS as usual, so evaluating `load "VeryLongName"` will load bytecode file `verylong.uo`.

# 7 Value polymorphism

The 1996 revision of Standard ML [7] adopts value polymorphism, discarding the distinction between imperative (`'_a`) and applicative (`'a`) type variables, and generalizing type variables only in non-expansive expressions. Consider a `val`-binding [

```
val x = e;
```

With *value polymorphism*, the free type variables in the type of `x` are generalized only if the right-hand side `e` is non-expansive. This is a purely syntactic criterion: an expression is *non-expansive* if it has the form *nexp*, defined by the grammar below:

| | | | |
|---|---|---|---|
| *nexp* | ::= | *scon* | special constant |
| | | *longid* | (possibly qualified) identifier |
| | | { ⟨*nexprow*⟩ } | record of non-expansive expressions |
| | | ( *nexp* ) | parenthesized non-expansive expression |
| | | *con nexp* | constructor application, where *con* is not `ref` |
| | | *excon nexp* | exception constructor application |
| | | *nexp* : *ty* | typed non-expansive expression |
| | | `fn` *match* | function abstraction |
| *nexprow* | ::= | *lab* = *nexp* ⟨, *nexprow*⟩ | |

Roughly, a non-expansive expression is just a value, that is, an expression in normal form. For example, the right-hand side `length` below is an identifier, and so is non-expansive. Hence the free type variable `'a` in the type `'a list -> int` of `x` becomes generalized:

```
- val x = length;
> val x = fn : 'a list -> int
```

On the other hand, the right-hand side `(fn f => f) length` below, although it evaluates to the same value as the previous one, is expansive: it is not derivable from the above grammar. Hence the type variable `'a` will not be generalized, and type checking will fail:

```
- val x = (fn f => f) length;
! Toplevel input:
! val x = (fn f => f) length;
!         ^^^^^^^^^^^^^^^^^^^
! Value polymorphism: Free type variable at top level: 'a
```

In Standard ML, all type variables in types reported at top-level must be universally quantified; there must be no free type variables. When type checking fails for this reason, there are two remedies: Either (1) insert a type constraint to eliminate the type variables, or (2) *eta-expand* the right-hand side to make it non-expansive:

```
- val x1 = (fn f => f) length : bool list -> int;
> val x1 = fn : bool list -> int

- val x2 = fn ys => (fn f => f) length ys;
> val x2 = fn : 'a list -> int
```

In Moscow ML versions prior to 1.40, the type checker would distinguish imperative and applicative type variables, generalize all applicative type variables, and generalize imperative type variables only in non-expansive expressions, as required by the 1990 *Definition* [6]. To reinstate this behaviour, invoke `mosml` or `mosmlc` with the option `-imptypes`. This is useful for compiling old programs.

# 8 Quotations and antiquotations

Moscow ML implements *quotations*, a non-standard language feature useful for embedding object language phrases in ML programs. Quotations are disabled by default. This feature originates in the Standard ML of New Jersey implementation. To enable quotations in the interactive system (`mosml`), execute `quotation := true`. This allows quotations to appear in declarations entered at top-level and in files compiled by the primitive `compile`. To enable quotations in files compiled with the batch compiler `mosmlc`, invoke it with option `-q` as in `mosmlc -q`.

A quotation is a particular kind of expression and consists of a non-empty sequence of (possibly empty) *fragments* surrounded by backquotes:

| | | | |
|---|---|---|---|
| *exp* | ::= | ‘*frags*‘ | quotation |
| *frags* | ::= | *charseq* | character sequence |
| | | *charseq* ^*id* *frags* | antiquotation variable |
| | | *charseq* ^(*exp*) *frags* | antiquotation expression |

The *charseq* is a possibly empty sequence of printable characters or spaces or tabs or newlines. A quotation evaluates to a value of type `ty frag list` where `ty` is the type of the antiquotation variables and antiquotation expressions, and the type `'a frag` is defined as follows:

```
datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a
```

A *charseq* fragment evaluates to `QUOTE "`*charseq*`"`. An antiquotation fragment ^*id* or ^(*exp*) evaluates to `ANTIQUOTE` *value* where *value* is the value of the variable *id* resp. the expression *exp*. All antiquotations in a quotation must have the same type `ty`.

An antiquotation fragment is always surrounded by (possibly empty) quotation fragments; and no two quotation fragments can be adjacent. The entire quotation is parsed before any antiquotation inside it is evaluated. Hence changing the value of `Meta.quotation` in an antiquotation inside a quotation has no effect on the parsing of the containing quotation.

For an example, say we have written an ML program to analyse C program phrases, and that we want to enter the C declaration `char s[6] = "abcde"`. We could simply define it as a string:

```
val phrase = "char s[6] = \"abcde\"";
```

but then we need to escape the quotes (") in the C declaration, which is tiresome. If instead we use a quotation, these escapes are not needed:

```
val phrase = `char s[6] = "abcde"`;
```

It evaluates to `[QUOTE "char s[6] = \"abcde\""] : 'a frag list`. Moreover, suppose we want to generate such declarations for other strings than just `"abcde"`, and that we have an abstract syntax for C phrases:

```
datatype cprog =
    IntCst of int
  | StrCst of string;
  | ...
```

Then we may replace the string `"abcde"` by an antiquotation `^(StrCst str)`, and the array dimension 6 by an antiquotation `^(IntCst (size str + 1))`, and make the string `str` a function parameter:

```
fun mkphrase str = `char s[^(IntCst (size str + 1))] = ^(StrCst str)`;
```

Evaluating `mkphrase "longer"` produces the following representation of a C phrase:

```
[QUOTE "char s[", ANTIQUOTE (IntCst 7), QUOTE "] = ",
 ANTIQUOTE (StrCst "longer"), QUOTE ""] : cprog frag list
```

# 9 A lexer generator

This section describes `mosmllex`, a lexer generator which is closely based on `camllex` from the Caml Light implementation by Xavier Leroy. This documentation is based on that of `camllex` also.

## 9.1 Overview

The `mosmllex` command produces a lexical analyser from a set of regular expressions with attached semantic actions, in the style of `lex`. Assume that file `lexer.lex` contains the specification of a lexical analyser. Then executing

```
mosmllex lexer.lex
```

produces a file `lexer.sml` containing Moscow ML code for the lexical analyser. This file defines one lexing function per entry point in the lexer definition. These functions have the same names as the entry points. Lexing functions take as argument a lexer buffer, and return the semantic attribute of the corresponding entry point.

Lexer buffers are an abstract data type implemented in the library unit `Lexing`. The functions `createLexerString` and `createLexer` from unit `Lexing` create lexer buffers that read from a character string, or any reading function, respectively.

When used in conjunction with a parser generated by `mosmlyac` (see Section 10), the semantic actions compute a value belonging to the datatype `token` defined by the generated parsing unit.

Example uses of `mosmllex` can be found in directories `calc` and `lexyacc` under `mosml/examples`.

## 9.2 Hints on using `mosmllex`

A lexer definition must have a rule to recognize the special symbol `eof`, meaning end-of-file. In general, a lexer must be able to handle all characters that can appear in the input. This is usually achieved by having the wildcard case `_` at the very end of the lexer definition. If the lexer is to be used with e.g. DOS files, remember to provide a rule for the carriage-return symbol `\r`. Most often `\r` will be treated the same as `\n`, e.g. as whitespace.

## 9.3 Syntax of lexer definitions

The format of a lexer definition is as follows:

```
{ header }
rule entrypoint =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint =
  parse ...
and ...
;
```

Comments are delimited by `(*` and `*)`, as in ML.

### 9.3.1 Header

The header section is arbitrary Moscow ML text enclosed in curly braces `{` and `}`. It can be omitted. If it is present, the enclosed text is copied as is at the beginning of the output file `lexer.sml`. Typically, the header section contains the `open` directives required by the actions, and possibly some auxiliary functions used in the actions.

### 9.3.2 Entry points

The names of the entry points must be valid ML identifiers.

### 9.3.3 Regular expressions

The regular expressions regexp are in the style of `lex`, but with a more ML-like syntax.

`'char'`
: A character constant, with a syntax similar to that of Moscow ML character constants; see Section 9.3.5. Match the denoted character.

`_`
: Match any character.

`eof`
: Match the end of the lexer input.

"string"
: A string constant, with a syntax similar to that of Moscow ML string constants; see Section 9.3.6. Match the denoted string.

`[` character-set `]`
: Match any single character belonging to the given character set. Valid character sets are: single character constants `'c'`; ranges of characters `'`$c_1$`' - '`$c_2$`'` (all characters between $c_1$ and $c_2$, inclusive); and the union of two or more character sets, denoted by concatenation.

`[ ^` character-set `]`
: Match any single character not belonging to the given character set.

regexp `*`
: Match the concatenation of zero or more strings that match regexp. (Repetition).

regexp `+`
: Match the concatenation of one or more strings that match regexp. (Positive repetition).

regexp `?`
: Match either the empty string, or a string matching regexp. (Option).

regexp$_1$ `|` regexp$_2$
: Match any string that matches either regexp$_1$ or regexp$_2$. (Alternative).

regexp$_1$ regexp$_2$
: Match the concatenation of two strings, the first matching regexp$_1$, the second matching regexp$_2$. (Concatenation).

`(` regexp `)`
: Match the same strings as regexp.

The operators `*` and `+` have highest precedence, followed by `?`, then concatenation, then `|` (alternative).

### 9.3.4 Actions

An action is an arbitrary Moscow ML expression. An action is evaluated in a context where the identifier `lexbuf` is bound to the current lexer buffer. Some typical uses of `lexbuf` in conjunction with the operations on lexer buffers (provided by the `Lexing` library unit) are listed below.

`Lexing.getLexeme lexbuf`
: Return the matched string.

`Lexing.getLexemeChar lexbuf n`
: Return the `n`'th character in the matched string. The first character has number 0.

`Lexing.getLexemeStart lexbuf`
: Return the absolute position in the input text of the beginning of the matched string. The first character read from the input text has position 0.

`Lexing.getLexemeEnd lexbuf`
: Return the absolute position in the input text of the end of the matched string. The first character read from the input text has position 0.

`entrypoint lexbuf`
: Here `entrypoint` is the name of another entry point in the same lexer definition. Recursively call the lexer on the given entry point. Useful for lexing nested comments, for example.

### 9.3.5 Character constants

A character constant in the lexer definition is delimited by ` (backquote) characters. The two backquotes enclose either a space or a printable character *c*, different from ` and \, or an escape sequence:

| Sequence | Character denoted |
|---|---|
| `*c*` | the character *c* |
| `\\` | backslash (\) |
| `\`` | backquote (`) |
| `\n` | newline (LF) |
| `\r` | return (CR) |
| `\t` | horizontal tabulation (TAB) |
| `\b` | backspace (BS) |
| `\^*c*` | the ASCII character control-*c* |
| `\*ddd*` | the character with ASCII code *ddd* in decimal |

### 9.3.6 String constants

A string constant is a (possibly empty) sequence of characters delimited by " (double quote) characters.

| | | | |
|---|---|---|---|
| *string-literal* | ::= | "*strcharseq*" | non-empty string |
| | | "" | empty string |
| *strcharseq* | ::= | *strchar* ⟨*strcharseq*⟩ | character sequence |

A string character *strchar* is either a space or a printable character *c*, different from " and \, or an escape sequence:

| Sequence | Character denoted |
|---|---|
| *c* | the character *c* |
| \\ | backslash (\) |
| \" | double quote (") |
| \n | newline (LF) |
| \r | return (CR) |
| \t | horizontal tabulation (TAB) |
| \b | backspace (BS) |
| \^*c* | the ASCII character control-*c* |
| *ddd* | the character with ASCII code *ddd* in decimal |

# 10 A parser generator

This section describes `mosmlyac`, a simple parser generator which is closely based on `camlyacc` from the Caml Light implementation by Xavier Leroy; `camlyacc` in turn is based on Bob Corbett's public domain Berkeley `yacc`. This documentation is based on that in the Caml Light reference manual.

## 10.1 Overview

The `mosmlyac` command produces a parser from a context-free grammar specification with attached semantic actions, in the style of `yacc`. Assume the input file is `grammar.grm`; then executing

```
mosmlyac options grammar.grm
```

produces a file `grammar.sml` containing a Moscow ML unit with code for a parser and a file `grammar.sig` containing its interface.

The generated module defines one parsing function per entry point in the grammar. These functions have the same names as the entry points. Parsing functions take as arguments a lexical analyser (a function from lexer buffers to tokens) and a lexer buffer, and return the semantic attribute of the corresponding entry point. Lexical analyser functions are usually generated from a lexer specification by the `mosmllex` program. Lexer buffers are an abstract data type implemented in the library module `Lexing`. Tokens are values from the datatype `token`, defined in the interface file `grammar.sig` produced by `mosmlyac`.

Example uses of `mosmlyac` can be found in directories `calc` and `lexyacc` under `mosml/examples`.

## 10.2 Syntax of grammar definitions

A grammar definition has the following format:

```
%{
   header
%}
   declarations
%%
   rules
%%
   trailer
```

Comments in the declarations and rules sections must be enclosed in C comment delimiters `/*` and `*/`, whereas comments in the header and trailer sections must be enclosed in ML comment delimiters `(*` and `*)`.

### 10.2.1 Header and trailer

The header and the trailer sections consist of Moscow ML code which is copied into file `grammar.sml`. Both sections are optional. The header goes at the beginning of file `grammar.sml`, but after the `token` datatype declaration; it usually contains `open` declarations required by the semantic actions of the rules. The trailer goes at the end of file `grammar.sml`.

### 10.2.2 Declarations

Declarations are given one per line. They all start with a `%` sign.

`%token` symbol ... symbol
: Declare the given symbols as tokens (terminal symbols). These symbols become constructors (without arguments) in the `token` datatype.

`%token < type >` symbol ... symbol
: Declare the given symbols as tokens with an attached attribute of the given type. These symbols become constructors (with arguments of the given type) in the `token` datatype. The type part is an arbitrary Moscow ML type expression, but all type constructor names must be fully qualified (e.g. `Unitname.typename`) for all types except standard built-in types, even if the proper `open` declarations (e.g. `open Unitname`) were given in the header section. This is because the header is copied only to the .sml output file, not to the .sig output file, while the type part of a `%token` declaration is copied to both.

`%start` symbol
: Declare the given symbol as entry point for the grammar. For each entry point, a parsing function with the same name is defined in the output file `grammar.sml`. Non-terminals that are not declared as entry points have no such parsing function. The start symbols must be given a type with the `%type` directive below.

`%type < type >` symbol ... symbol
: Specify the type of the semantic attributes for the given symbols. Every non-terminal symbol must have the type of its semantic attribute declared this way. This ensures that the generated parser is type-safe. The type part may be an arbitrary Moscow ML type expression, but all type constructor names must be fully qualified (e.g. `Unitname.typename`) for all types except standard built-in types, even if the proper `open` declaration (e.g. `open Unitname`) were given in the header section. This is

because the header is copied only to the .sml output file, not to the .sig output file, while the type part of a `%token` declaration is copied to both.

`%left` symbol ... symbol
`%right` symbol ... symbol
`%nonassoc` symbol ... symbol

Declare the precedence and associativity of the given symbols. All symbols on the same line are given the same precedence. They have higher precedence than symbols declared in previous `%left`, `%right` or `%nonassoc` lines. They have lower precedence than symbols declared in subsequent `%left`, `%right` or `%nonassoc` lines. The symbols are declared to associate to the left (`%left`), to the right (`%right`), or to be non-associative (`%nonassoc`). The symbols are usually tokens, but can also be dummy nonterminals, for use with the `%prec` directive inside the rules.

### 10.2.3 Rules

The format of grammar rules is as usual:

```
nonterminal :
      symbol ... symbol { semantic-action }
   | ...
   | symbol ... symbol { semantic-action }
;
```

Each right-hand side consists of a (possibly empty) sequence of symbols, followed by a semantic action.

The directive '`%prec` symbol' may occur among the symbols in a rule right-hand side, to specify that the rule has the same precedence and associativity as the given symbol.

Semantic actions are arbitrary Moscow ML expressions, which are evaluated to produce the semantic attribute attached to the defined nonterminal. The semantic actions can access the semantic attributes of the symbols in the right-hand side of the rule with the `$` notation: `$1` is the attribute of the first (leftmost) symbol, `$2` is the attribute of the second symbol, etc. An empty semantic action evaluates to `() : unit`.

Actions occurring in the middle of rules are not supported. Error recovery is not implemented.

## 10.3 Command-line options

The following command-line options are recognized by `mosmlyac`:

`-v`

Generate a description of the parsing tables and a report on conflicts resulting from ambiguities in the grammar. The description is put in file `grammar.output`.

`-b`prefix

Name the output files `prefix.sml`, `prefix.sig`, `prefix.output`, instead of using the default naming convention.

## 10.4 Reporting lexer and parser errors

Lexical errors (e.g. illegal symbols) and syntax errors can be reported in an intelligible way by using the `Location` module from the Moscow ML library. It provides functions to print out fragments of a source text, using location information provided by the lexer and parser. See `Location.sig` for more information.

## 11 Copyright and credits



**Moscow ML was written by** Sergei Romanenko (roman@keldysh.ru), Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Miusskaya Pl. 4, 125047 Moscow, Russia.

**and** Peter Sestoft (sestoft@dina.kvl.dk), Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark. Much of the work was done at the Department of Computer Science at the Technical University of Denmark, and while visiting AT&T Bell Laboratories, Murray Hill, New Jersey, USA.

**Moscow ML owes much to**

- the CAML Light implementation by Xavier Leroy and Damien Doligez (INRIA, Rocquencourt, France);
- the ML Kit by Lars Birkedal, Nick Rothwell, Mads Tofte and David Turner (Copenhagen University, Denmark, and Edinburgh University, Scotland);
- inspiration from the SML/NJ compiler developed at Princeton University and AT&T Bell Laboratories, New Jersey, USA; and
- the good work by Doug Currie, Flavors Technology, USA, on the MacOS port; and Franklin Chen, Applied Dynamics, for his MkLinux port.

## 12 How to get Moscow ML version 1.42

- The Moscow ML home page is http://www.dina.kvl.dk/~sestoft/mosml.html
- The DOS/Windows executables are in ftp://ftp.dina.kvl.dk/pub/mosml/mos14bin.zip
- The Linux executables are in ftp://ftp.dina.kvl.dk/pub/mosml/linux-mos14bin.tar.gz
- The Macintosh/MacOS (68k and PPC) executables are in ftp://ftp.dina.kvl.dk/pub/mosml/mac-mos14bin.sea.hqx
- The DOS source files are in ftp://ftp.dina.kvl.dk/pub/mosml/mos14src.zip
- The Unix source files are in ftp://ftp.dina.kvl.dk/pub/mosml/mos14src.tar.gz
- The MacOS modified source files (relative to Unix) are in ftp://ftp.dina.kvl.dk/pub/mosml/mac-mos14src.sea.hqx
- The MkLinux executables and binaries are available at http://www.ibg.uu.se/mkarchive/dev/lang

There are mirrors at `ftp://ftp.dcs.ed.ac.uk/pub/ml/versions/by_origin/Moscow` and `ftp://ftp.csd.uu.se/pub/mirror/mosml`.

## 13 Books and other materials on Standard ML

### The Definition and Commentary

- Robin Milner, Mads Tofte and Robert Harper, *The Definition of Standard ML*, MIT Press 1990, ISBN 0-262-63132-6.
- Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press 1997, ISBN 0-262-63181-4.
- Robin Milner and Mads Tofte, *Commentary on Standard ML*, MIT Press 1991, ISBN 0-262-63137-7.

### Textbooks available from publishers

- Richard Bosworth, *A Practical Course in Functional Programming Using Standard ML*, McGraw-Hill 1995, ISBN 0-07-707625-7.
- Greg Michaelson, *Elementary Standard ML*, UCL Press 1995, ISBN 1-85728-398-8.
- Colin Myers, Chris Clack, and Ellen Poon, *Programming with Standard ML*, Prentice Hall 1993, ISBN 0-13-722075-8.
- Lawrence C. Paulson, *ML for the Working Programmer*, Second edition. Cambridge University Press 1996, ISBN 0-521-56543-X.
- Chris Reade, *Elements of Functional Programming*, Addison-Wesley 1989, ISBN 0-201-12915-9.
- Ryan Stansifer, *ML Primer*, Prentice Hall 1992, ISBN 0-13-561721-9.
- Jeffrey D. Ullman, *Elements of ML Programming*, Prentice Hall 1994, ISBN 0-13-184854-2.
- Åke Wikström, *Functional Programming Using Standard ML*, Prentice Hall 1987, ISBN 0-13-331661-0.

### Texts available on the net

- Emden Gansner and John Reppy (editors): Standard ML Basis Library, hypertext version:
  `http://www.research.att.com/~jhr/sml/basis/sml-std-basis.html`
  `http://www.dina.kvl.dk/~sestoft/sml/sml-std-basis.html` (mirror site)
- Robert Harper, *Introduction to Standard ML*, LFCS Report Series ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, November 1986 (revised 1989). At ftp://ftp.cs.cmu.edu/afs/cs/project/fox/mosaic/intro-notes.ps.
- Mads Tofte, *Four Lectures on Standard ML*, LFCS Report Series ECS-LFCS-89-73, Department of Computer Science, University of Edinburgh, March 1989. At ftp://ftp.diku.dk/pub/diku/users/tofte/FourLectures/
- Mads Tofte, *Tutorial on Standard ML*, Technical Report 91/18, DIKU, University of Copenhagen, December 1991. At ftp://ftp.diku.dk/pub/diku/users/tofte/FPCA-Tutorial/

## References


[1] L. Birkedal, N. Rothwell, M. Tofte, and D.N. Turner. The ML Kit. Technical Report 93/14, DIKU, University of Copenhagen, Denmark, 1993.

[2] E. Gansner and J. Reppy. Standard ML Basis Library. Technical report, AT&T Bell Labs, 1996.

[3] X. Leroy. The Zinc experiment: An economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, France, 1990. Available as ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/economical-ML-implementation.ps.gz.

[4] X. Leroy. *The Caml Light system, release 0.6. Documentation and user's manual.* INRIA, France, September 1993. Available at ftp://ftp.inria.fr/lang/caml-light.

[5] D.B. MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conf. on LISP and Functional Programming*, pages 198–207. ACM Press, 1984.

[6] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[7] R. Milner, M. Tofte, R. Harper, and D.B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[8] S. Romanenko and P. Sestoft. *Moscow ML Language Overview, version 1.42*, July 1997.