Moscow ML Owner's Manual

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

Moscow ML implements Standard ML (SML), as defined in the 1997 *Definition of Standard ML*, including the SML Modules language and some extensions. Moreover, Moscow ML supports most required parts of the SML Basis Library. 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 the most common built-in functions [9]. Another companion document, the *Moscow ML Library Documentation*, describes in detail all Moscow ML library modules [10].

Contents

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

1 Getting started

1.1 Installation

Get a copy of the Moscow ML system executables (see Section 20 for instructions) and unpack them in your home directory (under Unix) or in directory `C:\` (under MS Windows and DOS). This creates a directory `mosml`. Read the file `mosml/install.txt`. This manual, as well as the *Moscow ML Language Overview* and the *Moscow ML Library Documentation*, 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 2.00 (June 2000)
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 MS Windows and 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 10 for further information on `mosmlc`.

1.4 The Moscow ML Modules language

The Moscow ML Modules language is a conservative extension of the Standard ML Modules language, so that any Standard ML program can be compiled with Moscow ML 2.00, using toplevel-mode compilation; see Section 7.

Moreover, the Moscow ML Modules language is backwards compatible with Moscow ML versions 1.44 and before, so that any existing Moscow ML program can be compiled with Moscow ML 2.00, using structure-mode compilation; see Section 6.

1.5 What is new in release 2.00

- The Moscow ML Modules language (designed and implemented by Claudio Russo) includes the full Standard ML Modules language (structures, signatures, and functors):
- The Moscow ML Modules language extends the Standard ML Modules language in three ways:
  - higher-order functors: a functor may be defined within a structure, passed as an argument to another functor, or returned as the result of a functor; see Sections 11.1, 11.2 and 11.3
  - first-class modules: structures and functors may be packed and then handled as Core language values, which may then be unpacked as structures or functors again; see Section 11.4
  - recursive modules: structures and signatures may be recursively defined; see Section 11.5
- Value polymorphism has become friendlier: non-generalizable free type variables are left free, and become instantiated (once only) when the bound variable is used; see Section 12.
- Added facilities for creating and communicating with subprocesses (structure `Unix` and `Signal` from SML Basis Library).
- Added facilities for efficient functional generation of HTML code (structure `Msp`); also supports the writing of ML Server Page scripts.
- Added facilities setting and accessing 'cookies' in CGI scripts `Mosmlcookie`, thanks to Hans Molin.
- The `Gdimage` structure now produces PNG images (using Thomas Boutell's gd library).

- Internal changes: restrictions on datatype constructor ordering have been removed; the runtime system and bytecode format now accommodate much larger programs; the representation of exceptions has been simplified; literals are now shared inside topdecs; etc.

2 Additional documentation

Moscow ML implements Standard ML (SML) as revised in 1997, also known as SML'97 [7, 8], and some extensions to the SML Modules language. Moscow ML implements much of the Standard ML Basis Library [3], the most important omission being the functional stream input-output operations. 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* [9] describes the Moscow ML source syntax, which is SML'97 with some extensions, and the most common built-in functions.

The *Moscow ML Library Documentation* [10] describes in detail all Moscow ML library modules and has a comprehensive index to exceptions, functions, structures, types, and values. The same information is available also from `mosml`'s built-in help function (Section 3.1) and as hypertext from Moscow ML's homepage and in the file `mosml/doc/mosmllib/index.html`.

The *Moscow ML for the Apple Macintosh* [2] is a detailed user guide to the Apple Macintosh version of Moscow ML.

[1] The Basis Library authors are Andrew Appel (Princeton, USA); Emden Gansner (AT&T Research, USA); John Reppy, Lal George, Lorenz Huelsbergen, Dave MacQueen (Lucent 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).

3 The interactive system: mosml

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

```
$ mosml
Moscow ML version 2.00 (June 2000)
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 MS Windows and 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 | 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.

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 and MS Windows 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 of mosml

`-conservative`

Sets conservative mode for compilation of subsequent units: accept all extensions to the SML Modules language, but issue a warning for each use; see Section 11. This is the default.

`-I` directory

Specifies directories to be searched for interface files, bytecode files, and source files. A call to `use`, `load` or `loadOne` 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. (This option affects the variable `Meta.loadPath`; see Section 3.4).

`-imptypes`

Specifies that the type checker should distinguish between imperative and applicative type variables, generalize all applicative type variables, and generalize imperative type variables only in non-expansive expressions. See Section 12.

`-liberal`

Sets liberal mode for compilation of subsequent units: accept without warnings all extensions to the SML Modules language; see Section 11.

`-orthodox`

Sets orthodox mode for the compilation of subsequent units: reject all uses of the extensions to the SML Modules language. That is, accept only SML Modules syntax; see Section 11.

`-P` unit-set

Determines which library units will be included and open at compile-time. Any library unit in the load path can be used by the `compile` function for type checking purposes. Thus regardless of the `-P` option, the `compile` function knows the type of library functions such as `Array.foldl`.

- `-P default` The initial environment for the SML Basis Library: modules `Array`, `Char`, `List`, `String`, and `Vector` will be loaded, and `Char`, `List`, and `String` will be partially opened.
- `-P sml90` This provides an initial environment which is upwards compatible with that of the 1990 *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 1990 Definition [7, Appendix C and D] are available at top-level. In addition the same 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 SML/NJ version 110. The math functions and input-output facilities required by the 1990 Definition [7, 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 `-imptypes`.
- `-P full` This loads all the libraries marked `F` in the library list (see [9]), and partially opens the `Char`, `List`, and `String` units.
- `-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.

`-quietdec`

Turns off the interactive system's prompt and responses, except for warnings and error messages. Useful for writing scripts in SML. Sets `Meta.quietdec` to `true`; see Section 3.4.

`-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 MS Windows and DOS.

`-valuepoly`

Specifies that the type checker should use 'value polymorphism'; see Section 12. Default.

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`, `compileStructure`, `compileToplevel` and `load` deal with Moscow ML compilation units; see Section 5.

`compile : string -> unit`

Evaluating `compile "U.sig"` will compile and elaborate the specifications in file U.sig in structure mode, producing a compiled signature U in file U.ui. This function is backwards compatible with Moscow ML 1.44 and earlier. Equivalent to `compileStructure [] "U.sig"`.

Evaluating `compile "U.sml"` will elaborate and compile the declarations in file U.sml in structure mode, producing a compiled structure U in bytecode file U.uo. If there is an explicit signature U.sig, then file U.ui must exist, and the structure must match the signature. If there is no U.sig, then an inferred signature will be produced in file U.ui. No evaluation takes place. This function is backwards compatible with Moscow ML 1.44 and earlier. Equivalent to `compileStructure [] "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.

`compileStructure : string list -> string -> unit`

Evaluating `compileStructure` *opnunits* `"U.sig"` will elaborate and compile the specifications in file U.sig in structure mode, producing a compiled signature U in file U.ui. The units listed in *opnunits* are added to the compilation context in which the specifications in U.sig are compiled.

Evaluating `compileStructure` *opnunits* `"U.sml"` will elaborate and compile the declarations in file U.sig in structure mode, producing a compiled structure U in bytecode file U.uo. The contents of the units listed in *opnunits* are added to the compilation context in which the declarations in U.sml are compiled. If there is an explicit signature U.sig, then file U.ui must exist, and the structure must match the signature. If there is no U.sig, then an inferred signature will be produced in file U.ui. No evaluation takes place.

`compileToplevel : string list -> string -> unit`

Evaluating `compileToplevel` *opnunits* `"U.sig"` will elaborate and compile the specifications in file U.sig in toplevel mode, producing a compiled interface file U.ui. The units listed in *opnunits* are added to the compilation context in which the specifications in U.sig are compiled.

Evaluating `compileStructure` *opnunits* `"U.sml"` will elaborate and compile the declarations in file U.sig in toplevel mode, producing a bytecode file U.uo. The contents of the units listed in *opnunits* are added to the compilation context in which the declarations in U.sml are compiled. If there is an explicit signature U.sig, then file U.ui must exist, and declarations must match the interface. If there is no U.sig, then an inferred signature will be produced in file U.ui. No evaluation takes place.

`conservative : unit -> unit`

Evaluating `conservative ()` sets conservative mode for the compilation functions: accept all extensions to the SML Modules language, but issue a warning for each use; see Section 11. The conservative mode may be set also by the `mosml` option `-conservative`. Conservative mode is the default.

`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, either built-in (such as `int` or `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. See the example in `mosml/examples/pretty`.

`liberal : unit -> unit`

Evaluating `liberal ()` sets liberal mode for the compilation functions: accept (without warnings) all extensions to the SML Modules language; see Section 11. The liberal mode may be set also by the `mosml` option `-liberal`.

`load : string -> unit`

Evaluating `load "U"` will load and evaluate the compiled unit implementation 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. The loaded unit(s) must be in the current directory or in a directory on the `loadPath` list (see below).

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 interfaces of units mentioned by U agree with the interfaces used when compiling U, and it is checked that the interface of U has not been modified since U was compiled; these checks are necessary for type safety. The exception `Fail` is raised if the interface checks fail, or if the file containing U or a unit mentioned by U is not found.

`loaded : unit -> string list`

Evaluating `loaded ()` will return a list of the names of loaded units in some order (not including the preloaded units `Meta` and `General`).

`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 interface checks as `load`.

`loadPath : string list ref`

This variable determines the load path: which directories will be searched for interface files (.ui files), bytecode files (.uo files), and source files (.sml files). This variable affects the `load`, `loadOne`, and `use` functions. The current directory is always searched first, followed by the directories in `loadPath`, in order. By default, only the standard library directory is in the list, but if additional directories are specified using option `-I`, then these directories are prepended to `Meta.loadPath`.

`orthodox : unit -> unit`

Evaluating `orthodox ()` sets orthodox mode for the compilation functions: reject all uses of the extensions to the SML Modules language; see Section 11. That is, accept only SML Modules syntax. The orthodox mode may be set also by the `mosml` option `-orthodox`.

`printVal : 'a -> 'a`

This is a polymorphic function provided as a quick debugging aid. It 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 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.

`printLength : int ref`

This variable determines the depth to which list values are printed by the top-level value printer and the function `printVal`. Only the first `printLength` elements of a list are printed, and the remaining elements are printed as '`...`'. The initial value of `printLength` is 200.

`quietdec : bool ref`

This variable, when `true`, turns off the interactive system's prompt and responses, except warnings and error messages. Useful for writing scripts in SML. The default value is `false`; it can be set to `true` with the `-quietdec` command line option; see Section 3.3.

`quit : unit -> unit`

Evaluating `quit()` quits Moscow ML immediately.

`quotation : bool ref`

Determines whether quotations and antiquotations are permitted in declarations entered at top-level and in files compiled with `compile`; see Section 16. 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`.

`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.

`use : string -> unit`

Evaluating `use "`*f*`"` causes ML declarations to be read from file *f* as if they were entered from the console. The file must be in the current directory or in a directory on the `loadPath` list. 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.

`valuepoly : bool ref`

Determines whether the type checker should use 'value polymorphism'; see Section 12. Command-line option `-valuepoly` sets `valuepoly` to `true` (the default), whereas option `-imptypes` sets `valuepoly` to `false`; see Sections 3.3 and 10.2.

`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`.

4 Understanding inferred types and signatures

When you enter a declaration in the interactive toplevel, `mosml` responds with a type or signature. The response provides much information about the declaration. It is worth learning how to interpret this information.

In the type reported for a value binding, the generalized type variables are listed between the `val` keyword and the identifier bound by the declaration:

```
- val f = length;
> val 'a f = fn : 'a list -> int          — polymorphic function; 'a has been generalized
```

When value polymorphism (see Section 12) prevents the generalization of a type variable, it remains free and ungeneralized. Hence `'a` will appear in the type `'a list -> int`, but will not be listed between `val` and `f`.

For a `datatype` declaration, which is generative, the response describes the new internal type names (here `u`), the datatype's type structure (consisting of the internal type name and the constructors), and the constructors:

```
- datatype u = A;
> New type names: =u
  datatype u = (u,{con A : u})
  con A = A : u
```

The notation `=u` means that the internal type name `u` admits equality. Since the `datatype` declaration is generative, repeated declaration of datatype `u` will produce distinct new internal type names `u/1`, `u/2`, etc. If the datatype is polymorphic, the internal type name will be a type function, taking as parameter a type variable `'a`:

```
- datatype 'a u = A of 'a;
> New type names: =u
  datatype 'a u = ('a u,{con 'a A : 'a -> 'a u})
  con 'a A = fn : 'a -> 'a u
```

For a signature specifying an unknown type, the response is a type function `/\t.{...}` mapping any internal type name `t` to a record `{...}`, which is the internal representation of the signature:

```
- signature SIG = sig type t val x : t end;
> signature SIG = /\t.{type t = t, val x : t}
```

For a structure declaration, the response describes any new internal type names, and then the internal signature of the structure. Here it has three components (datatype `t`, constructor `A`, and value identifier `x`):

```
- structure S = struct datatype t = A; val x = A end;
> New type names: =t
  structure S : {datatype t = (t,{con A : t}), con A : t, val x : t}
```

For a functor, the response describes a signature function `{...}->{...}`, mapping the signature of the functor argument to the signature of the functor body:

```
- functor F(X : sig val x : int end) = struct val y = X.x end;
> functor F : {val x : int}->{val y : int}
```

When the functor argument signature specifies an unknown type, the signature function will have the form `!t.{...}->{...}` in which the internal type name `t` has been generalized:

```
- functor F(X : sig type t; val x : t end) = struct val y = X.x end;
> functor F : !t.{type t = t, val x : t}->{val y : t}
```

When the body of a (standard) generative functor `G` contains a `datatype` declaration, the response describes what new internal type names will be generated at every application of the functor. The notation `{}->...` means that the functor argument signature is empty, and the notation `?=t.{...}` says that there is some internal type name `t`, admitting equality, such that the functor body has the signature `{...}`:

```
- functor F() = struct datatype t = A end;
> functor F : {}->?=t.{datatype t = (t,{con A : t}), con A : t}
```

When the functor is applicative (see Section 11.2) then no new internal type names are generated at functor application.

5 Compilation units

Moscow ML programs can be split into several source files, so-called compilation units, or units for short. A compilation unit consists of an implementation file `unitid.sml` and an optional interface file `unitid.sig`. A unit must be compiled in one of two modes:

- **Structure mode:** File `A.sml` declares a single SML structure `A`, and file `A.sig`, if present, declares an explicit SML signature `A` that is used as an opaque constraint for structure `A`.
  Structure-mode units are backwards compatible with the simple structures used in Moscow ML version 1.20 through 1.44. Hence old Moscow ML programs can be compiled (in structure mode) with no changes.
  To compile a unit in structure mode, use the function `compileStructure` (see Section 3.4), or invoke the batch compiler `mosmlc`, preceding the unit file name with option `-structure` (see Section 10.2).
- **Toplevel mode:** File `A.sml` contains a Moscow ML declaration (which may itself be a sequence of declarations), and file `A.sig`, if present, must be a Moscow ML specification (which may itself be a sequence of specifications). Toplevel-mode units can contain arbitrary Moscow ML code, and hence arbitrary Standard ML code.
  To compile a unit in toplevel mode, use the function `compileToplevel` (see Section 3.4), or invoke the batch compiler `mosmlc`, preceding the unit file name with option `-toplevel` (see Section 10.2).

The implementation file and the interface file (if any) of a unit must be compiled in the same mode. The precise syntax of structure-mode units and toplevel-mode units is given in the *Moscow ML Language Overview* [9].

A programming project may consist of a mixture of structure-mode and toplevel-mode compilation units. See the example in Section 8.

5.1 Compiling, linking and loading units

Compiling a unit interface `A.sig` produces a compiled unit interface in file `A.ui`. Compiling a unit implementation `A.sml` produces a compiled unit implementation in file `A.uo`, containing bytecode. In addition, if there is no correspond interface `A.sig`, then a file `A.ui`, containing an inferred unit interface, is produced.

Only the compiled unit interface `A.ui` is needed when compiling other units that depend on `A.sml`.

Thus the following file name extensions have a special meaning:

Extension	Contents
`.sig`	unit interface
`.sml`	unit implementation
`.ui`	compiled (or inferred) unit interface
`.uo`	compiled unit implementation (bytecode)

Compiled unit implementations can be linked together using `mosmlc -o mosmlout A.uo ...` to produce a linked executable bytecode file `mosmlout`. The linker will automatically link in any required bytecode files into `mosmlout`. See Section 10.2 for more options.

Compiled units can be loaded into the interactive system `mosml` using the primitives `load` and `loadOne`; see Section 3.4. Loading a unit makes the entities declared by the unit accessible to subsequent declarations in the interactive system.

5.2 Compiling existing SML programs that involve .sig files

The special meaning of .sig files may cause conflicts when compiling existing SML programs (written for other compilers), in which .sig files are often used rather informally to contain signature declarations. For instance, file `A.sig` may declare the signatures of structures declared in file `A.sml` (but might contain other arbitrary SML declarations besides).

By contrast, in Moscow ML, .sig files play a formal role. More precisely, Moscow ML expects file `A.sig` to contain a unit interface. Thus it may be necessary to rename `A.sig` to `A-sig.sml`, compile it before `A.sml`, and make its contents available to `A.sml` (see also Section 7):

```
mosmlc -c -toplevel A-sig.sml
mosmlc -c -toplevel A-sig.ui A.sml
```

5.3 Unit names and MS DOS/Windows/OS2 file names

In MS DOS, Windows, and OS/2, the file system may change the case of unit file names, and even truncate them (in MS DOS). Under MS Windows, file names are not truly case sensitive. 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 MS DOS and MS Windows are 'normalized': the first character is made upper case (if it is a letter), all other characters are made lower case, and (under MS DOS only) 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.
- 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 above for compatibility with the SML Basis Library.
- Under MS DOS (only), a unit name given as argument to e.g. `load` or `compile`, or to the batch compiler, is truncated and made lower case, so evaluating `load "VeryLongName"` will load bytecode file `verylong.uo`.

6 Structure-mode compilation units

This section describes the use of structure-mode compilation units, which are backwards compatible with Moscow ML versions 1.20 through 1.44.

6.1 Basic concepts

A structure-mode compilation unit consists of a *unit implementation*, in file `unitid.sml`, and an optional *unit interface*, in file `unitid.sig`.

The file `unitid.sml` must contain a declaration of a single Moscow ML structure `unitid`, and file `unitid.sig`, if present, must contain a declaration of a single Moscow ML signature `unitid`. When the unit interface is present, it is called the *explicit signature* to distinguish it from the signature inferred when elaborating the unit implementation. When present, the explicit signature must be matched by the implementation, and only those identifiers specified in the signature are visible outside the unit.

For a structure-mode unit it is immaterial whether a compilation unit's interface and implementation are compiled and directly executed in the interactive toplevel system:

```
mosml
> use "unitid.sig";
> use "unitid.sml";
```

or whether they are first compiled in structure-mode, and then loaded into the interactive toplevel system:

```
mosmlc -c -structure unitid.sig unitid.sml
mosml
> load "unitid";
```

where `unitid.sig`, if present, must be compiled before `unitid.sml`.

More generally, when a compilation unit depends on a number of other compilation units "$\texttt{unitid}_1$",...,"$\texttt{unitid}_n$", then compilation inside the interactive system:

```
mosml
> app load ["unitid1",...,"unitidn"];
> use "unitid.sig";
> use "unitid.sml";
```

is equivalent to structure-mode compilation followed by loading into the interactive system:

```
mosmlc -c -structure unitid1.ui ... unitidn.ui unitid.sig unitid.sml
mosml
> app load ["unitid1",...,"unitidn"];
> load "unitid";
```

6.2 Structure-mode units without explicit signature

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

`structure U =` modexp

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

6.3 Structure-mode units with explicit signature

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

`signature U =` sigexp

and a file U.sml, containing

`structure U :> U =` modexp

This is similar to an 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, meaning that other units have no access to the internals of U.sml, only to the signature U.sig.

7 Toplevel-mode compilation units

A toplevel-mode compilation unit may contain arbitrary Moscow ML toplevel declarations, and consequently, arbitrary Standard ML toplevel declarations (including functor, signature and fixity declarations).

7.1 Basic concepts

A toplevel-mode compilation unit consists of a *unit implementation*, in file `unitid.sml`, and an optional *unit interface*, in a file called `unitid.sig`.

The file `unitid.sml` may contain arbitrary Moscow ML declarations, and file `unitid.sig`, if present, may contain arbitrary Moscow ML specifications. When the unit interface is present, the unit implementation must implement everything specified in the unit interface, and only those identifiers specified in the interface are visible outside the unit. It is considered a mistake for the implementation to declare more identifiers than specified in the interface or for the implementation to declare an identifier as a constructor or exception, but the interface to specify that identifier as an ordinary value: a warning is issued in these cases.

Provided the interface and implementation agree on which identifiers are declared, then it is immaterial whether a compilation unit is compiled and directly executed in the interactive toplevel system:

```
mosml
> use "unitid.sml";
```

or whether it is first compiled in toplevel-mode, and then loaded into the interactive system:

```
mosmlc -c -toplevel unitid.sig unitid.sml
mosml
> load "unitid";
```

In contrast to a structure-mode unit, the unit interface `unitid.sig` should not be loaded in the interactive system, as it declares nothing. If there is no `unitid.sig`, then leave it out in the `mosmlc` command above.

More generally, when a toplevel compilation unit depends on a number of other compilation units "unitid$_1$", ..., "unitid$_n$" (compiled in structure mode or toplevel mode), then compilation inside the interactive system:

```
mosml
> app load ["unitid1",...,"unitidn"];
> use "unitid.sml";
```

is equivalent to toplevel-mode compilation followed by loading into the interactive system:

```
mosmlc -c -toplevel unitid1.ui ... unitidn.ui unitid.sig unitid.sml
mosml
> app load ["unitid1",...,"unitidn"];
> load "unitid";
```

Again, if there is no `unitid.sig`, then leave it out in the `mosmlc` command above.

8 An example program consisting of four mixed-mode units

To illustrate units, we present a tiny program working with arithmetic expressions. It consists of two structure-mode units, `Expr` and `Reduce` and two toplevel-mode units, `Evaluate` and `Test`. This example is in `mosml/examples/units`.

File `Expr.sml` below is a structure-mode unit implementation that 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 is a structure-mode unit interface that contains the signature `Reduce`:

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

File `Reduce.sml` below is a structure-mode unit implementation that declares the structure `Reduce`, with the explicit signature shown above, and therefore exports only the function `reduce` specified in that 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 is a toplevel-mode unit interface that contains the specification of a functor `Eval`, which maps any structure `R` matching the signature `Reduce` to a structure with a function `eval` for evaluating expressions, and a function `test` for testing `R`:

```
functor Eval : functor(R:Reduce) ->
                 sig val eval: Expr.expr -> int
                     val test: Expr.expr -> bool
                 end
```

File `Evaluate.sml` below is a toplevel-mode unit implementation that contains the actual functor `Eval`, which mentions the unit `Expr` as well as well as the unit interface `Reduce`:

```
functor Eval(R:Reduce) =
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 (R.reduce e))
  end
end
```

File `Test.sml` is a toplevel-mode unit implementation which mentions the unit `Expr`, the unit `Reduce`, and the functor `Eval` defined in the unit `Evaluate`, applies the functor to `Reduce` and carries out two tests:

```
structure T = Eval(Reduce)
open Expr
val t1 = T.test (Plus (Cst 244, Cst 0))
val t2 = T.test (Neg (Plus (Neg (Cst 140), Cst 0)))
```

The simplest way to compile this example is to enter:

```
mosmlc -c -structure Expr.sml Reduce.sig Reduce.sml -toplevel Evaluate.sig Evaluate.sml
Test.sml
```

which is equivalent to issuing the following individual commands:

```
mosmlc -c -structure Expr.sml
mosmlc -c -structure Expr.ui Reduce.sig
mosmlc -c -structure Expr.ui Reduce.sml
mosmlc -c -toplevel Expr.ui Reduce.ui Evaluate.sig
mosmlc -c -toplevel Expr.ui Reduce.ui Evaluate.sml
mosmlc -c -toplevel Expr.ui Reduce.ui Evaluate.ui Test.sml
```

The above command (re)compiles all files unconditionally. However, when modifying a program consisting of several units, it suffices to recompile just those units that have changed, and those units that depend on units that have changed. Moreover, compilation uses only the *interfaces* of other units, so it suffices to recompile just those units that depend on unit interfaces that have changed.

Thus if `Expr.sml` but not `Expr.sig` changes, it suffices to recompile just `Expr.sml`, not the units `Reduce`, `Evaluate`, or `Test`. Similarly, if `Reduce.sml` but not `Reduce.sig` changes, it suffices to recompile just `Reduce.sml`, not the units `Evaluate` or `Test`.

The next section explains recompilation management using `mosmldep` and `make`.

9 Recompilation management using mosmldep and make

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

Consider the example program in Section 8 consisting of the four units `Expr`, `Reduce`, `Evaluate` and `Test`. Assume their source files `*.sig` and `*.sml` reside in a particular directory (you may copy them from `mosml/examples/units`). Copy the Makefile stub from `mosml/tools/Makefile.stub` to that directory, and change to that directory.

1. Edit the Makefile so that the names of the units `Expr`, `Reduce`, `Evaluate`, and `Test` appear on the line beginning with '`UNITS=`', with each unit preceded by the option `-structure` (the default) or `-toplevel` to indicate its mode (see the example makefile in `mosml/examples/units/Makefile`):

   ```
   UNITS= -structure Expr Reduce -toplevel Evaluate Test
   ```

2. Edit the Makefile so that the names of the compiled unit implementation files `Expr.uo`, `Evaluate.uo`, `Reduce.uo` and `Test.uo` appear on the line beginning with '`all:`':

   ```
   all: Expr.uo Reduce.uo Evaluate.uo Test.uo
   ```

3. Compute the dependencies among the files by executing:

   ```
   make depend
   ```

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

The compiled `*.ui` and `*.uo` files can be removed by executing:

```
make clean
```

The unit dependencies are computed by the ML program `mosmldep`, called with

```
mosmldep $(UNITS)
```

When you explicitly list the units (together with the indication of their compilation mode):

```
UNITS= -structure Expr Reduce -toplevel Evaluate Test
```

then `mosmldep` merely constructs a dependency file on the assumption that a unit may depend on all preceding units (to the left), taking into account the presence of explicit interfaces (`.sig` files). Unfortunately, this means that `mosmldep` will generally create an inefficient Makefile, containing more dependencies than are actually in your source files. However, it still saves some recompilation (which is cheap in Moscow ML anyway) and is more convenient than hand-crafting the Makefile. We hope to improve on this situation in the future, but computing dependencies for full Standard ML programs is known to be difficult.

If you omit step (1) and leave `$(UNITS)` undefined:

```
UNITS=
```

then `mosmldep` will accurately *infer* the dependencies between all the `.sig` and `.sml` source files in the directory by inspecting their contents and looking for occurrences of unit names. Unfortunately, this process only works for programs containing just structure-mode units, that themselves contain only flat structures, but no functors or sub-structures. This is the behaviour of `mosmldep` in releases of Moscow ML prior to 2.00, and is provided for backwards compatibility.

10 The batch compiler and linker: mosmlc

Moscow ML includes a batch compiler and linker `mosmlc` in addition to the interactive system `mosml`. It compiles units and links programs, and can turn them into standalone executable bytecode files. The batch compiler can be invoked from a Makefile (see Section 9), which simplifies the (re)compilation of large programs considerably.

10.1 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 interfaces, source files for unit implementations, compiled unit interfaces (which are added to the compilation context), and compiled unit implementations (which are added to the linked executable).

- An argument ending in .sig is taken to be the name of a source file containing a unit interface. Given a file U.sig, the compiler produces a compiled interface in the file U.ui.
- An argument ending in .sml is taken to be the name of a source file containing a unit implementation. Given a file U.sml, the compiler produces compiled object code in the file U.uo. It also produces an inferred interface file U.ui if there is no explicit interface U.sig.
- An argument ending in .ui is taken to be the name of a compilation unit interface. The contents of that compilation unit are added to the compilation context of the remaining source files.
- An argument ending in .uo is taken to be the name of a compiled unit implementation. Such files are linked together, along with the compiled unit implementations obtained by compiling .sml arguments (if any), and the necessary Moscow ML library files, to produce a standalone executable program.
  The linker automatically includes any additional bytecode files required by the files specified on the command line; option `-i` makes it report all the files that were linked. The linker issues a warning if a file `B` is required by a file `A` that precedes `B` in the command line. At run-time, the top-level declarations of the files are evaluated in the order in which the files were linked; in the absence of any warning, this is the order of the .uo and .sml files on the command line.
  The linker (and the `load` function in the interactive toplevel) ensure probabilistically type-safe linking, so it is virtually impossible to cause the system to create a type-unsafe program.

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 (with option `-o mosmlout`), the command

`mosmlout` $\texttt{arg}_1$ $\texttt{arg}_2$ `...` $\texttt{arg}_n$

executes the compiled code contained in `mosmlout`. Executing the code means executing the toplevel declarations of all the bytecode files involved, in the order in which they were linked. The list of command-line arguments $\texttt{arg}_1$ `...` $\texttt{arg}_n$ can be obtained in a program by `CommandLine.arguments ()`.

There is no distinguished function which is automatically invoked when the program is executed, but it is common to define a `main` function and invoke it using the toplevel declaration `val _ = main()`, like this:

```
fun main () =
    case CommandLine.arguments () of
        [arg] => print ("The argument is " ^ arg ^ "\n")
      | _     => print "Usage: mosmlout arg\n\n"

val _ = main ();
```

MS Windows and 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 `camlrt.dll` to be present in directory `C:\mosml\bin`.

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` or similar. 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, use `mosmlc` option `-standalone`.

10.2 Command-line options of mosmlc

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.

`-conservative`
: Sets conservative mode for compilation of subsequent units: accept all extensions to the SML Modules language, but issue a warning for each use; see Section 11. This is the default.

`-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 MS 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 interface(s) of the unit implementation(s) 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.

`-I` directory
: Add the given directory to the list of directories searched for compiled interface files (.ui) and compiled implementation 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.

`-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 12.

`-liberal`
: Sets liberal mode for compilation of subsequent units: accept without warnings all extensions to the SML Modules language; see Section 11.

`-msgstyle` style
: By specifying `-msgstyle msdev`, one can make the compiler generate error messages understandable by Microsoft Developer Studio. The default behaviour is to generate error messages understandable the Emacs editor in SML mode.

`-noautolink`
: The linker automatically links in any additional object files required by the files explicitly specified on the command line. With option `-noautolink` all required object files must be explicitly specified in the appropriate order.

`-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`.

`-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 Windows and DOS, the default name is `mosmlout.exe`; in Unix it is `a.out`.

`-orthodox`
: Sets orthodox mode for the compilation of subsequent units: reject all uses of the extensions to the SML Modules language. That is, accept only SML Modules syntax; see Section 11.

`-P` unit-set
: Determines which library units will be *open* at compile-time. Any library unit in the load path can be used by the compiler for type checking purposes. Thus regardless of the `-P` option, the compiler knows the type of library functions such as `Array.foldl`.

  `-P default` The units `Char`, `List`, and `String` will be partially opened. This is the default, permitting e.g. `String.concat` to be referred to just as `concat`.

`-P sml90` Provides an initial environment which is upwards compatible with that of the 1990 *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 math functions and input-output facilities required by the 1990 Definition [7, Appendix C and D] are available at top-level. In addition the same (new) libraries are opened as with `-P default`.

`-P nj93` 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 1990 Definition [7, Appendix C and D] are available at top-level. In addition the same (new) libraries are opened as with `-P default`. This option does *not* imply `-imptypes`.

`-P full` Same as `-P default`.

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

Additional directories to be searched for library units can be specified with the `-I` directory option.

`-q`

Enables the quotation/antiquotation mechanism; see Section 16.

`-standalone`

Unix: Specifies that the runtime system should be prepended to the linked bytecode, thus creating a stand-alone executable. This adds 75–100 KB to the size of the linked file. MS Windows: This option cannot be used. Under MS Windows, all mosmlc-generated executables require the dynamically linked library `camlrt.dll` to be present in the directory `C:\mosml\bin`.

`-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 MS Windows and DOS.

`-structure`

Specifies that subsequent .sml and .sig source files must be compiled in structure mode; see Section 6. Default.

`-toplevel`

Specifies that subsequent .sml and .sig source files must be compiled in toplevel mode; see Section 7.

`-v`

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

`-valuepoly`

Specify that the type checker should use ‘value polymorphism’; see Section 12. Default.

11 Extensions to the Standard ML Modules language

The Moscow ML Modules language extends the Standard ML Modules language in several ways:

- higher-order functors: a functor may be defined within a structure, passed as an argument to another functor, or returned as the result of a functor; see Section 11.1
- applicative as well as (the usual) generative functors; see Section 11.2
- transparent as well as opaque functor signatures; see Section 11.3
- first-class modules: structures and functors may be packed and then handled as Core language values, which may then be unpacked as structures or functors again; see Section 11.4
- recursive modules: structures and signatures may be recursively defined; see Section 11.5
- relaxation of miscellaneous Standard ML restrictions; see Section 11.6.

The *Moscow ML Language Overview* defines the precise syntax of the extensions. For some additional examples of their use, see the directory `mosml/examples/modules` and its `README` file.

By default, Moscow ML accepts all non-Standard ML extensions, but issues a warning at every use. This is the behaviour of Moscow ML's *conservative* compilation mode. Two other compilation modes are available: *liberal*, in which all extensions are silently accepted, and *orthodox*, in which all extensions are rejected as errors.

The compilation mode can be set by the command-line options `-conservative`, `-liberal`, and `-orthodox`; see Sections 3.3 and 10.2. In the interactive system `mosml`, the compilation mode affects ML code entered interactively, as well as the functions `compile`, `compileStructure`, `compileToplevel`, and `use`. The compilation mode can be set by the functions `conservative`, `liberal`, and `orthodox`.

The extensions described here are based in part on [11, 12]; a formal definition of the extensions, derived from [7, 8], is available on request.

11.1 Higher-order modules

In Standard ML Modules, structures and functor bodies cannot declare functors. In Moscow ML Modules, they can:

```
functor F1(S : sig type t; val x : t end) = struct
    functor G(T : sig type u; val y : u end) = struct val pair = (S.x, T.y) end
end
structure R11 = F1(struct type t=int val x=177 end)
structure R12 = R11.G(struct type u=string val y="abc" end)
val (a, b) = R12.pair
```

A functor that returns a functor can be curried, avoiding the intermediate structure:

```
functor F2(S : sig type t; val x : t end) (T : sig type u; val y : u end) =
   struct val pair = (S.x, T.y) end
structure R2 =
   F2(struct type t=int val x=177 end)(struct type u=string val y="abc" end)
val (a, b) = R2.pair
```

A functor may be declared to take another functor as an argument, whose type is specified using a functor signature:

```
(* G is a functor signature *)
signature G = functor(X:sig type t=int val x: t end)->sig type u val y : u end

(* F3 takes the functor F as an argument, and applies it *)
functor F3(F:G) = F(struct type t=int val x=177 end)

(* R3 is the result of F3 applied to an anonymous functor *)
structure R3 = F3(functor(X:sig type t=int val x:t end)=>
                  struct type u = X.t * X.t val y= (X.x,X.x) end)
```

In the definition of `R3`, the argument to `F3` happens to be an anonymous functor.
Wherever a functor of a certain type is expected, one may supply instead a functor that has more a general type, that is, one which is more polymorphic, expects a less general argument, or produces a more general result:

```
(* F3  is applied to a more general functor than it specifically requires *)
structure R4 = F3(functor(X:sig type t end)=>
                  struct type u = X.t val y = 1 val z = [] end)
```

As in Standard ML, functors and structures reside in separate name spaces, so it is perfectly legal to re-use the same name for both a structure and a functor, without one hiding the other:

```
structure M = struct end
functor M() = struct end
structure N1 = M             (* structure N1 bound to structure M *)
functor   N2 = M             (* functor N1 bound to functor M *)
structure N3 = M(M)          (* functor M applied to structure M *)
```

However, when another functor, say `P`, simply returns the identifier `M`, as in:

```
functor P () = M
```

it is not clear whether `M` refers to the structure `M` or the functor `M`. In this ambiguous case, Moscow ML always interprets `M`, on its own, as a structure, but you can write `op M` to refer to the functor `M` instead:

```
functor P () = M      (* P returns the structure M *)
functor Q () = op M   (* Q returns the functor M *)
```

The files `mosml/examples/modules/{poly.sml,bootstrap.sml}` contain examples that use higher-order functors.

11.2 Applicative functors

In the Standard ML Modules language, all functors are generative. If the body of generative functor `FG` declares a datatype or opaque type `t`, then two applications of `FG` will create two structures `SG1` and `SG2` with distinct types `SG1.t` and `SG2.t`:

```
functor FG (S : sig end) = struct datatype t = C end    (* generative  *)
structure SG1 = FG() and SG2 = FG()
val res = if true then SG1.C else SG2.C;                (* ill-typed   *)
```

Recall that a conditional expression requires both branches to have equivalent types, so the last declaration above is well-typed only if the type `SG1.t` is equivalent to `SG2.t`.

If functors had an *applicative*, not generative, semantics, the two types would be equivalent. Moscow ML Modules allows the declaration of applicative functors[2] as well as generative functors. An applicative version `FA` of the above functor is declared the same way, except that the formal functor argument `S : sig end` is not enclosed in parentheses:

```
functor FA S : sig end = struct datatype t = C end      (* applicative *)
structure SA1 = FA() and SA2 = FA()
val res = if true then SA1.C else SA2.C;                (* well-typed  *)
```

More generally, if a type in an applicative functor's body depends on a datatype or opaque type of the functor's formal argument, then the types returned by separate applications of the applicative functor will be equivalent, provided the functor is applied to equivalent type arguments:

```
functor GA S : sig type t end = struct datatype u = C of S.t end  (* applicative, *)
                                                                   (* but u depends on S.t *)
structure TA1 = GA(type t = int) and TA2 = GA(type t = bool) and TA3 = GA(type t = int)
val res = if true then TA1.C 1 else TA3.C 1;            (* well-typed *)
val res = if true then TA1.C 1 else TA2.C true;         (* ill-typed  *)
```

Moscow ML supports type projections `X.u` (the dot notation for types) with local structure bindings. This makes it possible to refer directly to a type returned by an applicative functor within another type expression, which is useful for expressing sharing constraints:

```
signature S = sig functor F: functor X:sig type t end -> sig type u end
                  type v = X.u where X = F(type t = int)          (* X is local to X.u *)
              end
```

The local binding has no run-time effect and is only elaborated at compile-time for its type information. For more examples of applicative functors, see the files `mosml/examples/modules/{collect.sml,bootstrap.sml}`.

[2] Similar to Objective Caml.

11.3 Opaque and transparent functor signatures

The types of functors are specified using functor signatures. Similar to the distinction between generative and applicative functor expressions, functor signatures may be *opaque* or *transparent*. Whether a functor signature is opaque or transparent affects the interpretation of any datatype or opaque type specifications in its body signature:

The *opaque* functor signature

```
signature GO = functor(X:sig type t val x:t end)->sig type u val y:u end   (* opaque      *)
```

specifies the type of those functors that, when applied to an actual argument matching the argument signature `sig type t val x:t end`, return a result matching the body signature `sig type u val y:u end`, for some unknown implementation of the type `u` (possibly depending on `X.t`).

A *transparent* version `GT` of the above functor signature is written the same way, except that the formal functor argument `X:sig type t val x:t end` is *not* enclosed in parentheses:

```
signature GT = functor X:sig type t val x:t end -> sig type u val y:u end  (* transparent *)
```

This functor signature specifies the *family* of functor types that, for a given implementation of the result type `u` (possibly depending on `X.t`), map structures matching the argument signature `sig type t val x:t end` to structures matching the body signature `sig type u val y:u end`.

In practice, when writing a functor `H` that takes a functor `F` as an argument, the choice between specifying that argument using an opaque or transparent signature will affect the amount of type information that is propagated whenever `H` is applied to an actual functor. For instance, consider the four functors:

```
functor F1 (X:sig type t val x:t end) = struct type u = X.t val y = X.x end
functor F2 (X:sig type t val x:t end) = struct type u = int val y = 1 end

functor HO(F:GO) = F(struct type t = string val x = "abc" end)
functor HT(F:GT) = F(struct type t = string val x = "abc" end)
```

Functor `F1` returns a renamed version of its argument, and functor `F2` just ignores its argument and returns the same structure regardless. The two higher-order functors `HO` and `HT` apply the supplied `F` to the same argument (in which `x` is a string), but assume, respectively, an opaque and a transparent signature for `F`.

Since functor `HO` uses the opaque signature `GO`, its formal argument `F` is assumed to return some new unknown type `u` whenever it is applied, so that the two applications of `HO` return new abstract types `RO1.u` and `RO2.u`:

```
structure RO1 = HO(F1) and RO2 = HO(F2)
val resO1 = if true then RO1.y else "def";          (* ill-typed *)
val resO2 = if true then RO2.y else 1;              (* ill-typed *)
```

Functor `HT`, on the other hand, uses the transparent signature `GT`. This ensures that, no matter what the actual dependency of result type `u` on argument type `t`, `HT` may be applied to any functor `F` whose argument signature and result signature match `GT`, with the actual definition of `u` reflected in the result. In particular, the two applications of `HT` return the same definitions for type `u` as would the substitution of `F1` and `F2` directly into the body of `HT`. That is, the types `RT1.u` and `RT2.u` are equivalent to `string` and `int`:

```
structure RT1 = HT(F1) and RT2 = HT(F2)
val resT1 = if true then RT1.y else "def";          (* well-typed *)
val resT2 = if true then RT2.y else 1;              (* well-typed *)
```

Another way to look at this is that `HO`'s formal argument has a generative specification, so that its application in `HO`'s body returns a new type, while `HT`'s formal argument has an applicative specification, so that its application in `HT`'s body returns the same type as `HT`'s actual argument.

11.4 First-class modules

In the Moscow ML Modules language, a structure or functor can be wrapped up as a *package*, which is a first-class Core language value just like any other value, and then subsequently unpacked to re-create the structure or functor:

```
signature NAT = sig type nat  val Z:nat  val S:nat -> nat  val plus: nat -> nat -> nat end
structure SafeNat =  (* unlimited range but slow *)
  struct datatype nat = Z | S of nat  fun plus Z m = m | plus (S n) m = S (plus n m) end
structure FastNat =  (* limited range but fast *)
  struct type nat = int  val Z = 0  fun S n = n + 1  fun plus n m = n + m end

type natpack = [ NAT ]                                (* package type *)
val safeNat = [ structure SafeNat as NAT ];           (* packing      *)
val fastNat = [ structure FastNat as NAT ];
structure Nat as NAT =                                (* unpacking    *)
      if (913 mod 7 = 5) then safeNat else fastNat
val natlist = [safeNat,fastNat] : [ NAT ] list;
```

A functor may be packed using the similar Core expression `[ functor` *modexp* `as` *sigexp* `]` and unpacked using the functor binding `functor` *funid* `as` *sigexp* `=` *exp*.

Package type equivalence is determined by structure, not name, so the following package types are equivalent:

```
[sig type t val x: t type u = t val y: u end]
[sig type u val x: u type t = u val y: t end]
```

because the signatures are equivalent (every structure that matches one also matches the other).

For type soundness reasons, a package may not be unpacked in the body of a functor (although it may be unpacked within a Core expression occurring in that body):

```
functor Fail (val nat : [ NAT ]) =
  struct structure Nat as NAT = nat end                       (* illegal *)
functor Ok (val nat : [ NAT ]) =
  struct val x = let structure Nat as NAT = nat in nat end end (* legal *)
```

The files `mosml/examples/modules/{sieve.sml,array.sml,choice.sml,matrix.sml}` contain more examples.

11.5 Recursive structures and signatures

The Standard ML Modules language does not permit recursively defined modules. For instance, two structures `Even` and `Odd` may not depend on each other. In the Moscow ML Modules language, a structure may be defined recursively:

```
structure S = rec (X:sig structure Odd : sig val test : int -> bool end end)
              struct structure Even = struct fun test 0 = true
                                               | test n = X.Odd.test (n-1)
                                      end
                     structure Odd  = struct fun test 0 = false
                                               | test n = Even.test (n-1)
                                      end
              end;
```

Here, `X` is a forward declaration of the structure's body that allows `Even.test` to refer to `X.Odd.test` before it has been defined. The body of a recursive structure must match the signature of the forward declaration; any opaque type or datatype specified in the signature must be implemented in the body by *copying* it using a forward reference:

```
structure Ok   = (* well-typed *)
                 rec (X:sig datatype t = C type u type v = int end)
                 struct datatype t = datatype X.t type u = X.u type v = int end
structure Fail = (* ill-typed *)
                 rec (X:sig datatype t = C type u type v = int end)
                 struct datatype t = C type u = int type v = int end
```

At run-time, attempting to evaluate the forward reference of a recursive structure before its body has been been fully evaluated raises the exception `Bind`:

```
structure Fail = rec (X:sig end) X;                   (* raises Bind   *)
structure Fail = rec (X:sig val x: int end)
                 struct val x = X.x end;              (* raises Bind   *)
structure Ok = rec (X:sig val f: int -> int end)
               struct fun f n = X.f n  end;           (* ok            *)
val res = Ok.f n                                      (* infinite loop *)
```

On their own, recursive structures cannot be used to declare mutually recursive datatypes that span module boundaries. For this purpose, Moscow ML also supports *recursive signatures*:

```
signature REC = rec(X: sig structure Odd: sig type t end end)
                sig structure Even: sig datatype t = Zero | Succ of X.Odd.t end
                    structure  Odd: sig datatype t = One  | Succ of Even.t end
                end;
```

Here, `X` is a forward declaration of a structure implementing the body of the signature that allows the specification of `Even.t` to refer to the type `X.Odd.t` before it has been fully specified. In a recursive signature, the body of the signature must match the forward declaration and specify an implementation for any opaque types or datatypes declared within the forward specification.

Once a recursive signature has been defined, it can be used to implement the recursive structure `T` with datatypes that span module boundaries:

```
structure T = rec(X:REC)
              struct structure Even = struct datatype t = datatype X.Even.t
                                             fun succ Zero = X.Odd.One
                                               | succ E = X.Odd.Succ E
                                      end
                     structure Odd  = struct datatype t = datatype X.Odd.t
                                             fun succ O = Even.Succ O
                                     end
              end
```

The files `mosml/examples/modules/{recursion.sml,bootstrap.sml}` contain examples of using recursive structures and signatures.

11.6 Miscellaneous relaxations of Standard ML restrictions

In Standard ML, functors and signatures may only be declared at top-level, and structures may only be declared at top-level and within structures. None of these may be declared within Core let expressions. Moscow ML removes these restrictions so that functors, signatures and structures may be declared anywhere, which is particularly useful when programming with first-class modules.

In Standard ML, every parameterised type definition, and every type scheme occurring within a signature, must be closed and must not mention any type variables that are not explicitly listed as parameters. Moscow ML does not impose this restriction, and allows free type variables, provided they are bound in an enclosing scope. Again, this is useful when programming with first-class modules.

```
type t = 'a -> 'a                                (* illegal, since 'a not in scope      *)

fun f (x:'a) = let type t = 'a * 'a              (* legal Mosml, illegal SML            *)
               in (x,x):t
               end;

type 'a stackpack =                              (* legal Mosml                         *)
    [ sig
          type stack = 'a list;                  (* 'a occurs free in this type binding *)
          val push : 'a -> stack -> stack        (* 'a occurs free in this type scheme  *)
      end ]
```

12 Value polymorphism

The 1997 revision of Standard ML [8] adopts value polymorphism, discarding the distinction between imperative (`'_a`) and applicative (`'a`) type variables, and generalizing type variables only in non-expansive expressions. In Moscow ML 2.00, non-generalized type variables are left free, to be instantiated later. 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
		longvid	(possibly qualified) value 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
		`[ structure` *nmodexp* `as` *sigexp* `]`	structure package
		`[ functor` *nmodexp* `as` *sigexp* `]`	functor package
nmodexp	::=	⟨`op` ⟩ *longmodid*	module identifier
		`(` *nmodexp* `)`	
		nmodexp `:(:>)` *sigexp*	transparent (opaque) constraint
		`functor` *arg* `=>` *modexp*	functor
		`rec (` *strid* `:` *sigexp* `)` *nmodexp*	recursive structure
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, as shown by the occurrence of `'a` between `val` and `x`:

```
- val x = length;
> val 'a 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 a warning will be issued:

```
- val x = (fn f => f) length;
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier x
> val x = fn : 'a list -> int
```

Note that the type variable `'a` has not been generalized. Thus type variable `'a` is free and may become instantiated when `x` is used. If so, the type of `x` becomes more specific:

```
- x ["abc", "def"];
! Warning: the free type variable 'a has been instantiated to string
> val it = 2 : int
- x;
> val it = fn : string list -> int
```

To make sure that a type variable gets generalized, one may *eta-expand* the right-hand side expression. Eta-expansion replaces an expression *e* with `fn y => (e y)` thus making it non-expansive:

```
- val x1 = fn y => (fn f => f) length y;
> val 'b x1 = fn : 'b list -> int
```

Compatibility: All programs compiled with Moscow ML 1.44 will compile also with Moscow ML 2.00. To compile old programs that use imperative type variables as in the 1990 Definition, you may invoke `mosml` or `mosmlc` with the option `-imptypes`.

13 Weak pointers

Moscow ML supports weak pointers and arrays of weak pointers, using library structure `Weak`. A *weak pointer* is a pointer that cannot itself keep an object alive. Hence the object pointed to by a weak pointer may be deallocated by the garbage collector if the object is reachable only by weak pointers.

The interface to arrays of weak pointers is the same as that of standard arrays (structure `Array`), but the subscript function `sub` may raise exception `Fail` if the accessed object is dead. On the other hand, if `sub` returns a value, it is guaranteed not to die unexpectedly: it will be kept alive by the returned pointer. Also, the weak array iteration functions iterate only over the live elements of the arrays.

One application of weak pointers is to implement hash consing without space leaks. The idea in hash consing is to re-use pairs: whenever a new pair (a, b) is to be built, an auxiliary table is checked to see whether such a pair exists already. If so, the old pair is reused. In some applications, this may conserve much space and time. However, there is a danger of running out of memory because of a space leak: the pair (a, b) cannot be deallocated by the garbage collector because it remains forever reachable from the auxiliary table. To circumvent this problem, one creates a weak pointer from the auxiliary table to the pair, so that the auxiliary table in itself cannot keep the pair alive.

For an example, see `mosml/examples/weak`. See also the description of `Weak` in the *Moscow ML Library Documentation*, or try '`help "Weak";`'.

14 Dynamic linking of foreign functions

Moscow ML supports dynamic linking of foreign (C) functions, using library structure `Dynlib`[3]. A library of functions may be written in C and compiled into a dynamically loadable library, using appropriate compiler options. With the `Dynlib` structure one can load this library and call the C functions from Moscow ML, without recompiling the runtime system.

It is the responsibility of the C functions to access and construct SML values properly, using the macroes defined in `mosml/include/mlvalues.h`. For this reason, the foreign function interface is included only with the source distribution. As usual, type or storage mistakes in C programs may crash your programs.

The ML garbage collector may run at any time an ML memory allocation is made. This may cause ML values to be moved (from the young generation to the old one). To make sure that ML heap pointers needed by your C function are adjusted correctly by the garbage collector, register them using the `Push_roots` and `Pop_roots` macroes from `mosml/include/memory.h`.

To modify a value in the ML heap, you must use the `Modify` macro from `mosml/include/memory.h`; otherwise you may confuse the incremental garbage collector and crash your program.

When loading the compiled library one must specify the absolute path unless it has been installed as a system library. This may require putting it in a particular directory, such as `/lib` or `/usr/lib`, or editing `/etc/ld.so.conf` and running `ldconfig`.

To compile Moscow ML[4] with support for dynamic linking, edit file `mosml/src/Makefile.inc` as indicated there.

For more information, see the examples in directory `mosml/src/dynlibs`, in particular `mosml/src/dynlibs/interface`. See also the `Dynlib` section in the *Moscow ML Library Documentation* [10]; or try '`help "Dynlib";`'.

[3]Thanks to Ken Larsen.

[4]Supported under Linux, FreeBSD, NetBSD, Solaris, Digital Unix, HP-UX, MacOS, and MS Windows'95/98/NT.

15 Guide to selected dynamically loaded libraries

15.1 Using GNU gdbm persistent hash tables

Moscow ML provides an interface to GNU gdbm persistent hashtables, via structures `Gdbm` and `Polygdbm`; see the appropriate sections in [10]. GNU gdbm provides fast access even to very large hashtables stored on disk, ensuring mutual exclusion, and are handy for creating simple databases for use by CGI scripts etc.

Using `Gdbm` or `Polygdbm` requires `Dynlib` (see Section 14 above): GNU gdbm (not included with Moscow ML) must be installed, and the interface to GNU gdbm defined in `mosml/src/dynlibs/mgdbm` must be compiled and installed. For instructions, see the `README` file there.

15.2 Using POSIX regular expressions

Moscow ML provides an interface to the GNU regex implementation of POSIX 1003.2 regular expressions, with additional support for replacing matching substrings etc., via structure `Regex`; see the appropriate section in [10].

Using `Regex` requires `Dynlib` (see Section 14 above): The GNU regex library (which is included with Moscow ML) and the interface defined in `mosml/src/dynlibs/mregex` must be compiled and installed. For instructions, see the `README` file there.

15.3 Using the PostgreSQL relational database server

Moscow ML provides an interface to the PostgreSQL relational database server, via structure `Postgres`; see the appropriate section in [10].

Using `Postgres` requires `Dynlib` (see Section 14 above): the PostgreSQL database server (not included) must be installed, and the interface to PostgreSQL defined in `mosml/src/dynlibs/mpq` must be compiled and installed. For instructions, see the `README` file there.

15.4 Using the MySQL relational database server

Moscow ML provides an interface[5] to the MySQL relational database server, via structure `Mysql`; see the appropriate section in [10].

Using `Mysql` requires `Dynlib` (see Section 14 above): the MySQL database server (not included) must be installed, and the interface to Mysql defined in `mosml/src/dynlibs/mmysql` must be compiled and installed. For instructions, see the `README` file there.

15.5 Using the PNG image library

Moscow ML provides an interface to Thomas Boutell's gd graphics package for creating PNG images, via structure `Gdimage`; see the appropriate section in [10].

Using `Gdimage` requires `Dynlib` (see Section 14 above): the gd image package (not included) must be installed, and the interface defined in `mosml/src/dynlibs/mgd` must be compiled and installed. For instructions, see the `README` file there.

15.6 Using the Socket interface

Moscow ML provides an interface[6] to Internet and file sockets, via structure `Socket`, which adheres fairly closely to the SML Basis Library structure of the same name; see the appropriate section in [10].

Using structure `Socket` requires `Dynlib` (see Section 14 above): the sockets interface defined in `mosml/src/dynlibs/msocket` must be compiled and installed. For instructions, see the `README` file there.

[5] Thanks to Thomas S. Iversen.

[6] Thanks to Ken Larsen; initial development financed by the PROSPER project.

16 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
```

17 A lexer generator: mosmllex

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.

17.1 Overview

Given a set of regular expressions with attached semantic actions, `mosmllex` produces a lexical analyser in the style of `lex`. If file `lexer.lex` contains a 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 18), 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`.

17.2 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 putting the wildcard case _ at the very end of the lexer definition. If the lexer is to be used with e.g. MS Windows, MS DOS or MacOS 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.

Do not use string constants to define many keywords; this may produce large lexer programs. It is better to let the lexer scan keywords the same way as identifiers and then use an auxiliary function to distinguish between them. For an example, see the `keyword` function in `mosml/examples/lexyacc/Lexer.lex`.

17.3 Syntax of lexer definitions

The format of a lexer definition is as follows:

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

Comments are delimited by `(*` and `*)`, as in SML. An abbreviation (abbrev) for a regular expression may refer only to abbreviations that strictly precede it in the list of abbreviations; in particular, abbreviations cannot be recursive.

17.3.1 Header section and entry points

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.

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

17.3.2 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 17.3.4. 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 17.3.4. 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).

abbrev
: Match the same strings as the regexp in the most recent `let`-binding of abbrev.

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

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

17.3.3 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.

17.3.4 Character constants and string 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

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 a space, or a printable character *c* (except " 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

18 A parser generator: mosmlyac

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.

18.1 Overview

Given a context-free grammar specification with attached semantic actions, `mosmlyac` produces a parser, in the style of `yacc`. If file `grammar.grm` contains a grammar specification, then executing

```
mosmlyac 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 unit defines a parsing function `S` for each start symbol `S` declared in the grammar. Each parsing function takes as arguments a lexical analyser (a function from lexer buffers to tokens) and a lexer buffer, and returns 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 unit `Lexing`. Tokens are values from the datatype `token`, defined in the signature file `grammar.sig` produced by running `mosmlyac`.

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

18.2 The format of grammar definitions

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

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

18.2.1 Header and trailer

Any SML code in the header is copied to the beginning of file `grammar.sml`, after the `token` datatype declaration; it usually contains `open` declarations required by the semantic actions of the rules. Any SML code in the trailer is copied to the end of file `grammar.sml`. Both sections are optional.

18.2.2 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.

`%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.

`%type < type >` symbol ... symbol
: Specify the type of the semantic attributes for the given symbols. Every non-terminal symbol, including the start symbols, 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.

`%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.

18.2.3 The format of grammar rules

```
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.

18.3 Command-line options of 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.

18.4 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 from the lexer and parser. Try `help "Location"` for more information. See file `mosml/examples/lexyacc/Main.sml` for an example.

19 Copyright and credits

Main Moscow ML contributors

- Doug Currie (e@flavors.com), Flavors Technology, USA.
- Sergei Romanenko (roman@keldysh.ru), Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Russia.
- Claudio V. Russo (Claudio.Russo@cl.cam.ac.uk), University of Cambridge. Thanks to Don Sannella at LFCS, Division of Informatics, University of Edinburgh for funding under EPSRC grant GR/K63795.
- Peter Sestoft (sestoft@dina.kvl.dk), Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark and The IT University of Copenhagen, Denmark. Part of the work was done while 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). It was instrumental in creating Moscow ML, which uses its runtime system, essentially the same bytecode generator, and many other aspects of its design [4, 5].
- The Definition of Standard ML, unbeatably precise and concise.
- The ML Kit by Lars Birkedal, Martin Elsman, Niels Hallenberg, Nick Rothwell, Mads Tofte and David Turner (University of Copenhagen, Denmark, and University of Edinburgh, Scotland), which helped solving problems of parsing, infix resolution, and type inference [1].
- Inspiration from the SML/NJ compiler developed at Princeton University and AT&T/Lucent Bell Laboratories, New Jersey, USA.
- Feedback, contributions, and useful suggestions, in particular from Ken Friis Larsen, but also from Jonas Barklund, Mike Gordon, Michael Norrish, Konrad Slind, Jakob Lichtenberg, Hans Molin, and numerous other people.

20 How to get Moscow ML

- The Moscow ML home page is `http://www.dina.kvl.dk/~sestoft/mosml.html`
- The Linux executables are in `ftp://ftp.dina.kvl.dk/pub/mosml/linux-mos20bin.tar.gz`
- The MS Windows executables are in `ftp://ftp.dina.kvl.dk/pub/mosml/win32-mos20bin.zip`
- The Macintosh/MacOS (68k and PPC) executables are in `ftp://ftp.dina.kvl.dk/pub/mosml/mac-mos20bin.sea.hqx`
- The Unix and MS Windows source files are in `ftp://ftp.dina.kvl.dk/pub/mosml/mos20src.tar.gz`
- The MacOS modified source files (relative to Unix) are in `ftp://ftp.dina.kvl.dk/pub/mosml/mac-mos20src.sea.hqx`

21 Books and other materials on Standard ML

The 1997 Definition [8] is the authoritative description of Standard ML, revised from the 1990 Definition [7]. The Commentary [6] explains many finer points in the 1990 Definition.

Textbooks available from publishers

- Richard Bosworth, *A Practical Course in Functional Programming Using Standard ML*, McGraw-Hill 1995, ISBN 0-07-707625-7.
- Michael R. Hansen and Hans Rischel, *Introduction to Programming using SML*, Addison-Wesley 1999, ISBN 0-201-39820-6.
- 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

- Andrew Cumming, *A Gentle Introduction to ML*, Napier University. At `http://www.dcs.napier.ac.uk/course-notes/sml/manual.html`
- Emden Gansner and John Reppy (editors): Standard ML Basis Library, hypertext version: `http://www.cs.bell-labs.com/~jhr/sml/basis/index.html` and `http://www.dina.kvl.dk/~sestoft/sml/sml-std-basis.html` (mirror site)
- Stephen Gilmore, *Programming in Standard ML'97*, report ECS-LFCS-97-364, University of Edinburgh. At `http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/97/ECS-LFCS-97-364`
- Robert Harper, *Introduction to Standard ML*, report ECS-LFCS-86-14, University of Edinburgh, November 1986 (revised 1989). At `ftp://ftp.cs.cmu.edu/afs/cs/project/fox/mosaic/intro-notes.ps`
- Robert Harper, *Programming in Standard ML*, Carnegie Mellon University. At `http://www.cs.cmu.edu/~rwh/introsml`
- 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] Doug Currie. Moscow ML for the Apple Macintosh. Technical report, Flavors Technology, 1999.

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

[4] 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.

[5] 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.

[6] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

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

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

[9] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Language Overview, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.pdf.

[10] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Library Documentation, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmllib.pdf.

[11] C. Russo. *Types For Modules.* PhD thesis, LFCS, University of Edinburgh, 1998.

[12] C. Russo. First-class structures for standard ml. In *European Symposium on Programming (ESOP)*, 2000.