Moscow ML for the Apple Macintosh™

Version 2.00 of July 2000

Doug Currie (e@flavors.com)
Flavors Technology, Inc.
Manchester, New Hampshire, USA

This document describes Moscow ML 2.00 for the Apple Macintosh. It was done by e (Doug Currie) in July 2000 based on previous ports.

The biggest changes since version 1.44 are: increased default and minimum memory requirements, Appearance and Navigation Services support, changes to the make function, IntInf and libgmp.so added, Cookies, Bin234map, etc. See `ReleaseNotes_Mac.txt` for other recent changes and improvements.

Three other documents, the *Moscow ML Owner's Manual* [3], the *Moscow ML Language Overview* [1], and the *Moscow ML Library Documentation* [2], describe general aspects of the Moscow ML system.

Moscow ML implements Standard ML (SML), as defined in the 1997 *Definition of Standard ML*, with some extensions to the SML Modules language, and supports most required parts of the new SML Basis Library. Moscow ML supports separate compilation and the generation of stand-alone executables.

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

Contents

1 Read this if nothing else

Use the installer to install Mac Moscow ML. It will place some files in a "mosml" folder in your System Folder's Extensions folder. This configuration supports launching the mosml applications as supplied. For other possibilities, see Section 13.

If you want a source code installation (i.e., the source code for Moscow ML itself, which is not required to use Moscow ML) be sure to read the information displayed by the installer when it's launched: you must install the Linux Moscow ML sources before the Mac Moscow ML source code supplement.

Typing an expression followed by the Enter key (or Option-Return), causes the expression to be sent to mosml for evaluation. Try typing:

```
help "lib";
```

followed by Enter.

There are two application sets, nearly identical in function, but customized for the PPC and m68k SANE. If you are using Moscow ML on a PowerMac, you should install only the PowerPC set; if you are using Moscow ML on a m68k Mac, you should install only the m68k set. The installer offers both of these options under the Custom Install mode. You must have a PowerMac, Quadra, or a M68030 or M68020 based Mac to use Mac Moscow ML, and 20 Mbytes of free disk space (30 Mbytes for a source code installation).

The m68k applications do not support shared libraries (dynlibs).

See `ReleaseNotes_Mac.txt` for a history of improvements and fixes.

In this document, the character sequence XXX in file names represents the current version number of Moscow ML. For example, in release 2.00, references to the file mosmlXXX.app are actually references to mosml200.app.

2 Credits/support

Moscow ML (mosml) is freeware. The entire source code for mosml is available. A number of authors have contributed to its development; see the file copyrght/copyrght.cl for restrictions on charging money for distribution of this software.

I have tested mosml on two PowerMacs (8600 and StarMax 4200); note that the recent m68k versions have been tested only in emulation. I have not extensively stress tested memory requirements. I usually run with 16M allocated to mosml, but it will probably run with much less. It will launch and open files in 3 MB. Running the full library test requires about 16 MB.

Comments/questions about the port should be directed to me, <e@flavors.com>.

Comments/questions about Moscow ML should probably be directed elsewhere, as I am not an expert in ML or its implementation! Please read this document, and any other README or documentation that accompanies this release before sending questions to me or any of the other authors, mailing lists, or newsgroups. See the SML FAQ, and newsgroup comp.lang.ml which I watch via the SML-LIST mailing list.

3 Getting started

The installation steps in brief are:

1. launch the installer, MacMoscowMLXXXinstaller (where XXX is the version, e.g. 200);
2. select Custom Install, and choose the PPC or m68k installation;
3. click the Install button.

Note that if you want a source code installation (i.e., the source code for Moscow ML itself, which is not required to use Moscow ML) be sure to read the information displayed by the installer when it's launched: you must install the Linux Moscow ML sources before the Mac Moscow ML source code supplement.

Details about the installation...

1. The installer will create a "mosmlXXX" folder on the volume of your choice, and will place several files in a "mosml" folder in your System Folder's Extensions folder. In the mosmlXXX folder are the mosml application, documentation, and example files. In the mosml folder in the Extensions folder are the shared libraries (PPC installations only) and the mosml image and library files.
2. The PPC version of mosml is supplied as a pair of files: a shared library containing the bulk of mosml with the user interface, and a small (16k byte) application. Note: The two files must be in the same folder, or else the shared library must be in a subdirectory of the Extensions folder in your System Folder, where they were placed by the installer.

   There is a second PPC shared library for use by mosml based CGIs, and a pair of application templates for the creation of stand-alone mosml based CGIs and applications. So in total there are five PPC specific files; these files are located in the mosmlXXX folder and its `e_templates` subfolder:

File	Contents
mosmlXXXapp	the mosml application
mosmlXXXapp.template	the mosml application template
mosmlXXXapp.lib	their shared library
mosmlXXXcgi.template	the cgi application template
mosmlXXXcgi.lib	its shared library

   For the m68k install, there is one m68k based mosml application, and two m68k templates:

File	Contents
mosmlXXXm68	the mosml application
mosmlXXXm68.template	the mosml application template
mosmlXXXm68cgi.template	the cgi application template

Each of the above sets, PPC and m68k, provide the same capabilities except that the PPC set only runs on PowerMacs, and runs faster on PowerMacs than the m68k set. Also, the m68k applications do not support shared libraries (dynlibs). It's best to only install one of the m68k or PPC versions, otherwise the Finder will not know which version to launch when a mosml file is double-clicked. From now on I'll call the version you kept "mosml."

3. The configuration created by the installer supports launching the mosml applications as supplied; the mosml application itself, the mosmlXXX folder, and any created stand-alone applications may be moved about freely. Don't move the files in the Extensions folder's mosml folder without considering the consequences; see Section 13.

4 Launching mosml

The binary image file "mosml.image" is loaded automatically when mosml is launched. mosml then loads some library files. It's important to avoid moving the image or library files; they should be left where installed (i.e., in the Extensions folder, or relative to mosml and the mosml folder if you follow the alternate installation in Section 13).

Now you are running mosml interactively. Typing expressions, followed by the Enter key (or Option-Return), causes the expression to be sent to mosml for evaluation. Try typing:

```
help "lib";
```

followed by Enter.

The Command menu contains mosml specific commands. Use these menu items to use, load, compile, etc. The meaning of these commands is documented in the Moscow ML documentation [3]. The File, Edit, Search, and Windows menus are for using the built in editor. Also in the Edit menu is a Styles item for setting preferences (which are recorded in a "Mosml Preferences" file in your Preferences Folder).

Later you can use the link function to build your own images. If you name your image "mosml.image" then it will be loaded automatically at launch instead of the original mosml image. You can also select an image to load at launch time using the command line, via drag and drop, or through naming conventions for applications and images. You can also build stand-alone applications that have your mosml image built-in; see below.

Section 14 describes how to use the console and editor features provided by eConsole. Here are some newer features specific to the mosml version of eConsole:

A command in the Search menu, Open Selection, can be used to open a file by selecting its name in any window and typing Command-D. This can also be used with mosml's error messages; in this case, the file that caused the error will be opened, and the error text scrolled-to and highlighted. For example, selecting the line:

```
File "errortest.sml", line 3-6, characters 4-75:
```

and typing Command-D (or selecting the Open Selection menu item) will open the file errortest.sml in an edit buffer with the four lines highlighted. This only works for files in the current directory, but this is almost always where files will be if the compiler is complaining about them.

The Console remembers the end point of the last input (plus Mosml's response, if any) and refuses to allow backspacing past that point. It beeps in response. A common new-user problem occurs when typing an expression without a trailing semicolon – it results in no response; a natural reaction is to backspace to the previous line, add the semicolon, and type Enter again (rather then simply typing the semicolon and Enter). Unfortunately, this would result in the expression being entered twice since the rule is that all text in the "run" near or behind the insert point is sent when Enter is typed. By preventing you from backing up to connect two text "runs" the problem is avoided. You can still use Cut or Clear to remove text from the Console window, if desired.

There is a Command menu item called End of Input. It is intended to emulate the Control-D in UNIX. Using the End of Input command, or its command key equivalent <Command-;> (that's the Command key and Semicolon) from toplevel will cause Mosml to quit. Using the End of Input command while awaiting input will cause any typed characters to be Entered, and then whatever characters are available to be returned, even if there are none or fewer than requested.

When mosml exits in any manner other than an appl Quit command from the user (i.e., the Quit menu item or command-Q), the console title will change to: `press ≪enter≫ to exit` and the application will not quit until you type the <enter> key. This feature lets you see the result of UNIX-like applications in the console, and even to save the console's contents. In this mode, mosml is really terminated; only the console code is still running.

5 Cow icons

My nine (now thirteen) year old daughter, Megan, drew the icons. She is better at it than me. I chose a cow for the icon for three reasons: (1) "cow" are the letters missing from mosml; (2) Caml Light uses a camel, so I though an animal face would be appropriate; (3) I think cows are nice.

Cows in New Hampshire look *something* like the icon. They are white with blotchy black spots. If there are any Moscow ML users out there who are also artists, please feel free to contribute more polished icons!

6 Library status

The Moscow ML library units, including the SML Basis Library, Mosml extensions, and Dynlibs (dynamically loaded or shared libraries) are fully supported. There are also a couple Mac-specific extensions including AppleScript, and extensions to the open function built into the runtime.

6.1 Moscow ML library units

Although much of it is UNIX derived, a surprising amount of the library now works with the Mac. In particular, walking directory structures, and getting information about files, elapsed/used time, time of day, and dates are all working. In fact, as of this release, the Mac version is ahead of the DOS and UNIX versions in capability. In particular, the Mac version:

- has the mosmldep capability built into a new make function
- has the linker integrated into the toplevel environment.

One compromise was necessary in Path.sml – it assumes that the pathname string can be used to determine if a path is relative to the current working directory or absolute (relative to "root"). While it is always the case that Mac pathnames starting with ":" are relative, all other pathnames can be relative or absolute. The Mac tries its darndest to find the file somewhere. So, I made the following choice: if the name starts with a ":" then it's relative, else if it contains a ":" it's absolute, else it's relative. This seems to handle common cases (i.e., it's rare to store files at root level). Path.sml also has assumptions built in about how pathname arcs "." and ".." behave. These are just file (or directory) names on the Mac. The Mac approach of "::" meaning up versus ":" meaning down is very dissimilar to the UNIX approach; I tried to make Path.sml work as best I could. It should do the right thing in most cases, certainly all common and canonical cases.

There are also some differences in how file system links work. Mac aliases are not 100% like UNIX links. For example, readLink() returns the absolute pathname of the original file on the Mac.

The Mac implementation of readDir shares a bug with some other mosml implementations: if you modify the contents of a directory while it is open for reading, the results are unpredictable.

6.2 Filename translation

When files are opened by Moscow ML runtime, a simple UNIX to Mac pathname translation is performed if the name contains any '/' characters. This is indispensable when porting from the UNIX version of mosml to the Mac. It is less useful after the port is done, but remains in place if you need it. One drawback to this mechanism is that some legal Mac file and folder names cannot be used in mosml paths, i.e., file or folder names containing '/' characters. This has not proved to be a problem in practice.

6.3 AppleScript library module

The module AppleScript supplies facilities for compiling, running, and receiving results from AppleScripts. Examples are provided in `e_SML` of using AppleScript from mosml to communicate with Microsoft Excel and Eudora. Many Mac applications support scripting via AppleScript. AppleScript is also used in MakeMacApplication.sml.

There are also many libraries of scripting additions available for use with AppleScript, e.g., speech, encryption, etc. These libraries must be placed in your extensions folder to be used from mosml.

6.4 Process.system

The text argument to `system()` is sent to the AppleScript compiler and executed as long as AppleScriptLib is in the Extensions folder. This is only tested on PowerMacs. Using AppleScript and `system()` you can now send AppleEvents to other applications, e.g., send e-mail, launch applications with the Finder, etc. But this is not compatible with the DOS or UNIX versions of the `system()` call.

6.5 Dynlib

The dynlib Dynamic Library feature of Moscow ML has been implemented in the PPC version of mosml using Mac OS Shared Libraries.

Note that shared libraries on the Mac are referenced by library (aka code fragment) name, not file name. Also, the Mac implementation ignores the `flag` and `global` arguments to `dlopen`. Otherwise the capabilities and use of shared libraries on the Mac mirror the dynlib support on other versions of Moscow ML (e.g., Linux, Solaris, Digital Unix, HP-UX, and MS Windows). These differences are usually transparent to SML code.

David Gay's excellent implementation of a floating point reader and writer is provided as an example shared library. It can also be used to read and write floating point numbers accurately (e.g., to work around MSL problems). See `e_SML:gay.sml`.

Also provided are Mac versions of the Moscow ML IntInf, Sockets, and Regex dynlibs (libmgmp.so, libmsocket.so, and libmregex.so).

Dynlibs common to Moscow ML are provided with a ".so" pathname extension for compatibility. The Mac Moscow ML application and CGI shared libraries have the ".lib" pathname extension. The Gay floating point reader and writer library has a ".slib" pathname extension. Since the Mac accesses shared libraries by library name rather than file name, these extensions are for decoration only.

7 Mac toplevel extensions

The mosml toplevel environment on the Macintosh is extended from the UNIX/DOS version. There are four new names:

Variable	Use	Description
make	compilation manager	see Section 7.1
link	linker	see Section 7.3
moolevel	verbosity setting	see Section 7.2
getDir	current directory	same as FileSys.getDir

The last one, getDir, is used by the Command menu.

The make and link functions can be used to build complete mosml applications without any other tools, i.e., without a UNIX shell or make program. Together with mosmllex and mosmlyac, make and link can be used to bootstrap mosml. [If you start from the UNIX distribution you will also need perl, e.g., MacPerl, and you may need to preprocess some files with the C preprocessor or by hand – the Mac source distribution has these files included "pre-processed."]

7.1 The function `make`

Note: The make function only supports compilation in toplevel mode. This is adequate for rebuilding Moscow ML itself, and for making all the example code included with the distribution. The dependency tracing portion of make will need extensive work to accommodate structure mode.

The make function compiles all files in a directory needing recompilation. It has all the compile capabilities of mosmlc in toplevel mode, plus it finds dependencies among the files. It only compiles what needs to be compiled. It has the type:

```
make : string -> string -> string list -> string ->
       (string * string list) list -> unit
```

Evaluating

```
make <oset> <stdlib> [<dir1>..<dirN>] <path> [deps]
```

is equivalent to running mosmlc as follows:

```
mosmlc -c -toplevel -P <oset> -stdlib <stdlib> -I <dir1> .. -I <dirN> <files...>
```

where <files...> is a subset of the files in directory <path> determined by tracing out the dependencies among the files and their need for compilation.

The make command uses the mosmldeps parser to find dependencies for all files in a directory, computes the transitive closure of these dependencies, finds a consistent ordering for the files, tests each file in turn to see if it needs compilation (including tests upon modification times of units outside the target directory as long as they're visible to `find_in_path()`), and compiles the files if necessary. It takes as arguments -stdlib and -I paths and -P so it can be used to compile the library and compiler directories.

The final argument to make is a list of unit dependencies that are necessary for correct compilation, but which cannot be inferred by mosmldep. It has the format of a list of pairs: a unit name and a list of units it depends upon. See the *Moscow ML Owner's Manual* [3] for a description of why this may be necessary. In all of Moscow ML itself, there is only one directory which requires additional dependency info: mosmllib. See Section 7.6 for an example.

A file needs compilation if:

- there is no corresponding object file, "object"; or
- the mtime of the object is older than the epoch (where the epoch is a built in time variable, currently Time.zeroTime); or

- the mtime of the source is newer than mtime of the object; or
- the mtime of any dependency is newer than the mtime of the object.

Make keeps a cache of file dependencies; when run a second or subsequent time the cache is consulted and verified based on the mtime of the source file. This can save considerable time. If you ever need to flush this cache (although I can't think of why you would) see moolevel below.

Caveats: make doesn't know about cpp, yacc, lex, sed, awk, perl, etc. It does report when it finds ".grm" files without corresponding ".sig" and ".sml" files, for example, (and the same for .mlp and .lex) but these must be made manually. I have created a mac mosmlyac which I have used to compile the .grm files, and mosml can run mosmllex with the (see below) command line; I have used this technique to process the .lex files.

7.2 The variable moolevel

The variable moolevel : int ref controls the verbosity of make as follows:

moolevel	Effect
0	no messages
1	error messages
2	compile messages
3	progress messages

The moolevel defaults to 1, but I personally like level 2.

As a special hack, if moolevel is negative, then its absolute value is used, and additionally the cache of file dependencies is cleared by make. You should never need this.

7.3 The function link

The function link is used to create stand-alone mosml applications from compiled files. It has type:

```
link : string -> bool * bool -> bool * string -> string
              -> string list -> string list -> unit
```

Evaluating:

```
link <file> (<g>,<h>) (<a>,<oset>) <stdlib> [<dir1>..<dirN>] [<file1>..<fileN>]
```

is equivalent to running mosmllnk as follows

```
mosmllnk -o <file> {-g} {-noheader} {-noautolink} -P <oset> -stdlib <stdlib>
         -I <dir1> ..  -I <dirN> <file1>..<fileN>
```

where -g is included if <g> is true, -noheader is included if <h> is true, and -noautolink is included if <a> is false. If <oset> is "" then it is replaced by "default".

If <a> is true then link does the following:

1. <oset> is ignored, and may be "" or any other string;
2. trace out all load dependencies of <file1>..<fileN>;
3. include in the list of files to be linked all other files depended upon if they can be found in any of the specified paths;
4. construct a legal link order for the files;
5. then link this list as usual.

Typically, simply specifying your Main program to be linked will find all the required .uo files and put them in the right order.

7.4 The function `find_lib`

The functions make and link require the path to the mosml lib. If the lib is installed in the Extensions folder as directed above, there isn't an easy (non installation specific) way to specify that lib folder. So, FindStdlib.sml is provided in the `e_SML` folder; it defines one function called `find_lib` which does a heuristic search for the mosml libraries. It is documented in FindStdlib.sml.

7.5 The function `get_home`

It is often necessary to construct a pathname for make and link. It is good to avoid the use of literal pathnames in released software, i.e., to make these names non-specific to a particular volume. So, the file GetHome.sml is provided in the `e_SML` folder; it defines one function called `get_home` which uses ENV variables (see below) to generate the name of the directory containing the current mosml application.

7.6 Examples

To create a new mosml project, put all the source files in a new directory, "project." Don't put any incomplete or irrelevant sources in the same directory or they will be compiled, too. Assuming your project only depends on library files, use the Command menu to Use GetHome.sml and FindStdlib.sml and then evaluate:

```
let val base = get_home()
in
  make "full" (find_lib()) [] (base ^ "project") []
end;
```

to compile all your files. Evaluate this each time you have made changes in your sources to keep the object files up to date. Here is a sample make and link script to build mosml.image:

```
app load ["Path", "Process"] ;

val home =
  case Process.getEnv "PATH_TRANSLATED" of
    SOME n => Path.dir n
  | NONE => ":" ;

moolevel := 2;

(* to compile the toplevel mosml image and the lexer *)

let val base = home ^ "src:"
in
  valuepoly := false;
  make "none" (base ^ "mosmllib") [] (base ^ "mosmllib")
        [("OS",["FileSys","Path","Process"])];
  valuepoly := true;
  make "none" (base ^ "mosmllib") [] (base ^ "compiler") [];
  make "none" (base ^ "mosmllib") [(base ^ "compiler")] (base ^ "toolssrc") [];
  make "none" (base ^ "mosmllib") [(base ^ "compiler")] (base ^ "lex") []
end;

(* to build the toplevel mosml image into file "testtop.image" *)

let val base = home ^ "src:"
in
```

```
    chDir base; chDir "::";
    link "testtop.image"
         (true,true) (* -g -noheader *)
         (true,"")   (* -autolink -P *)
         (base ^ "mosmllib") [(base ^ "compiler"),(base ^ "toolssrc")]
         ["Maine.uo"]
  end;
```

Other examples of the use of make and link can be found in the e_SML folder.

8 Creating standalone Mac applications

Mosml can be used to create stand-alone Macintosh applications.

Well, on PPC machines the applications mosml creates depend on the mosmlXXXapp.lib just like mosmlXXXapp does. This saves a few hundred kbytes per appl, and is hardly noticeable once mosmlXXXapp.lib is installed. Just remember to ship mosmlXXXapp.lib with your applications if you distribute them to others. When mosmlXXXapp.lib is in the usual place, or in any subfolder of the Extensions folder, the application can be moved about freely and is double-clickable.

Mosml has the Mac creator code 'Moml' [if you don't know what a Mac creator code is, don't worry, there won't be a quiz]. Stand-alone applications made by mosml have the Mac creator code 'Msml' – this prevents Finder confusion, e.g., the newly generated stand-alone applications won't be launched when double-clicking on mosml documents.

Note that the Mac Finder is sometimes slow and/or stubborn about recognizing new applications; it seems to defer entering the necessary info into the desktop database until some somewhat arbitrary activity takes place. The effect of this is that new applications will launch mosml instead of self-launching. I have found that closing and reopening the mosml folder in the Finder triggers the Finder to fix the desktop database. As a last resort, a rebuild of the desktop (restart the Mac with option-command depressed) will certainly fix it, but I have never needed to resort to this extreme. If someone knows a method of consistently prodding the Finder to do the right thing, e.g., via an AppleScript I can call from MakeMacApplication, please let me know.

To create a stand-alone Macintosh application, first make a mosml image as described above, e.g., using make and link. You can test your image by loading it manually with the command line (see below for details). Use the Command menu to Use GetHome.sml if you haven't already done so in the build process. Then evaluate:

```
val home = get_home();
chDir (home ^ "e_SML:");
load "MakeMacApplication";
```

MakeMacApplication defines several useful functions, including an interactive application builder. You can also build the application programmatically, e.g.,

```
MakeMacApplication.merge_template_image
  (home ^ "e_templates:mosmlXXXapp.template")
  "hello.image"
  "hello!";
```

or on m68k machines. . .

```
MakeMacApplication.merge_template_image
  (home ^ "e_templates:mosmlXXXm68.template")
  "hello.image"
  "hello68k!";
```

See the helloWorld folder in the e_SML folder for a complete example.

8.1 The function `merge_template_image`

The function `merge_template_image : string -> string -> string -> bool` takes three arguments: a path to an application template, a path to a mosml image, and an application name. Using AppleScript, the template file is copied and renamed. Then the image is appended to it, and using AppleScript again the file type changed to 'APPL'.

Function `merge_template_image` will complain if the target application name already exists, and refuses to create the application. This is a feature designed to prevent accidental overwrites of mosml applications. If you run into this situation, simply delete the old version and try again.

8.2 The function `make_mac_application`

The function `make_mac_application : string -> bool` takes one argument, the name of the application to build, and prompts you for the template and image with Standard File Dialogs.

9 Creating standalone Mac CGI applications

Using the process described above, you can create stand-alone CGIs. The only differences are: the shared lib is mosmlXXXcgi.lib, and the template is mosmlXXXcgi.template. For m68k, the template is mosmlXXXm68cgi.template. See the cgi folder in the `e_SML` folder for a couple of complete examples.

With m68k this creates a complete and rather large application, with PPC is a small application which relies on the shared library mosmlXXXcgi.lib.

CGIs interface to the web server through cgi AppleEvents ('WWWΩ' 'sdoc') and standard in and out. The cgi template based applications accept cgi AppleEvents and provide the mosml code with access to the data. The mosml code has a full mosml runtime, but no console or editor. The CGI application should only be launched by a web server, which will send the application a cgi AppleEvent. All output to stdOut is collected and returned to the web server in an AppleEvent reply at exit.

The environment of these applications is extended with several new variables. They correspond to the UNIX CGI environment variables as closely as possible, and can be obtained with Process.getEnv. In some cases stdIn will supply additional data for the cgi request, e.g., POST. Use the environment variable `CONTENT_LENGTH` to determine the number of characters waiting at stdIn.

The environment variables are documented in the Environment Variables section.

CGI applications have a bare minimum user interface: Quit. Quitting should only be necessary when your X.image has a bug which prevents it from exiting. You can Quit from the File menu or with Command-Q.

The old (v 1.43) method of duplicating the mosml-cgi application and giving it and an image file appropriate names is still supported, but less nice…

1. copy mosmlXXXm68cgi [you must make this from the mosmlXXXm68cgi.template]
2. rename it "X.cgi" where X is a name of your choice but shouldn't have spaces, slashes, or other nasty characters in it
3. make and link your mosml cgi program; name it "X.image"
4. move X.cgi and X.image to a folder accessible to your web server
5. point an http link at X.cgi

See the 'image pathname heuristic' below.

10 Environment variables

Environment variables are not provided by MacOS, but Mac mosml emulates this facility for compatibilty with UNIX and CGI applications.

10.1 Basic environment variables

In all versions of Mac mosml, there are these environment variables:

Variable	Defined where
`VERSION`	from application resource fork
`CAMLRUNPARAM`	from application resource fork
`PATH_TRANSLATED`	constructed at launch time

`VERSION` is just a version string; `CAMLRUNPARAM` is accessed by the runtime system at launch time for configuration (see file mosml/src/runtime/main.c); `PATH_TRANSLATED` is the pathname of the running application (used by GetHome.sml). Examples:

```
- load "Process";
> val it = () : unit
- Process.getEnv "VERSION";
> val it = SOME "mosml 1.42e" : string option
- Process.getEnv "CAMLRUNPARAM";
> val it = SOME "v=0i=50000s=50000o=20" : string option
- Process.getEnv "PATH_TRANSLATED";
> val it = SOME "jalaMPW:ml:mosml142:mosml142app" : string option
```

10.2 CGI environment variables

In the CGI applications of Mac mosml only, at launch time (see Alternate Launching Mechanisms below) these are all constructed from the ('WWWΩ' 'sdoc') cgi AppleEvent:

Variable	Mac AE name	Description
`PATH_INFO`	'—–'	arguments following the "$" in a URL
`QUERY_STRING`	'kfor'	arguments following a "?" in a URL
`REMOTE_HOST`	'addr'	Domain name of client (or IP address if no DNS)
`REMOTE_ADDR`	'Kcip'	the TCP/IP address of the client
`REMOTE_USER`	'user'	Username if authentication was required
`REMOTE_PASS`	'pass'	Password if authentication was required
`REQUEST_METHOD`	'meth'	the HTTP method being requested (e.g., `GET`, `GET_CONDITIONAL`, `POST`, etc.)
`SCRIPT_NAME`	'scnm'	The (raw) path of the CGI being executed
`SERVER_NAME`	'svnm'	Domain name of server (or IP address if no DNS)
`SERVER_PORT`	'svpt'	TCP/IP port server is listening on
`HTTP_REFERER`	'refr'	URL of page from which this CGI was referenced
`HTTP_USER_AGENT`	'Agnt'	The WWW client software name and version
`HTTP_COOKIE`	'Kfrq'	The HTTP cookies parsed from the Full Request
`GATEWAY_INTERFACE`	'Kact'	The action being performed by the CGI, either the name of the user defined action or one of the strings: CGI, ACGI, PREPROCESSOR, POSTPROCESSOR, ERROR, INDEX, or NOACCESS.
`CONTENT_TYPE`	'ctyp'	MIME type of POST arguments if present
`CONTENT_LENGTH`	'post'	the size of the POST data; the POST data are placed on stdIn

References:

UNIX http://hoohoo.ncsa.uiuc.edu/cgi/interface.html

Mac http://www.starnine.com/support/technotes/cgiparams-table.html

11 Alternate launching mechanisms, CGI

Mac mosml has a few ad hoc features related to application launching to support automating mosml applications with AppleScript, AppleEvents, and the startup of CGI and stand-alone applications. Many of these features rely on the use of command lines; see the Command Lines section below for details on that mechanism.

11.1 AppleEvent ≪event miscdosc≫

Mosml supports an AppleEvent at launch to supply the command line to mosml. For example, using a second copy of mosmlXXXapp and the AppleScript library module, you can run specified images with command line arguments. I use this mechanism to automate the bootstrap of mosml from one version to the next. See `e_SML:bootstrap:` for examples.

Specifically, the AppleScript

```
tell application "mosmlrun" ≪event miscdosc≫ <cmdline> end tell
```

will launch the mosmlrun application (which might be acopy of mosmlXXXapp) with the command line <cmdline>.

11.2 CGI support

A version of mosml which accepts cgi AppleEvents ('WWWΩ' 'sdoc') is provided as a template used to create stand-alone applications as described above in Creating Standalone Mac CGI Applications. This works analogously to the above miscdosc mechanism, but rather than a command line, a CGI request is passed. It is the mechanism used by web servers on the Mac, so you will never need to use it in scripts, it is all handled behind the scenes.

11.3 Image pathname heuristic

[The image pathname heuristic was created before stand-alone Mac applications were supported. It remains available, but stand-alone Mac applications are nicer.]

Before using the default image name in the econfigstrs resource, mosml checks for an image name which, minus the pathname extension, matches the application name, also minus the extension, in the same folder. Specific extensions are required:

Extension	Type of application
`.app`	runtime application (e.g., a copy of mosmlXXXapp)
`.cgi`	cgi application (e.g., a copy of mosmlXXXcgi)
`.image`	image

For example, launching frob.app will try to load frob.image, but if it fails to find an image of that name, will resort to the name in the econfigstrs resource. Dragging and dropping an image onto frob.app will override this behavior: the dropped image will be used instead.

This is [was] especially useful for cgi applications (see that section). In this case, for example, a copy of mosmlXXXcgi named "echo.cgi" will run the image named "echo.image" when launched from a web server.

12 Command lines

Command lines are very non-Mac. You don't need to use them unless you need to use mosmllex, want to create an application that uses command lines, or is launch-AppleScriptable, or are bootstrapping a new version of mosml.

If mosml cannot find the image file, or if the command key is held down when mosml is launched, then mosml will display a single question mark and await a command line. [The command key is the one with an apple or a "freeway interchange" symbol on it.] Also, a command line can be supplied to mosml with the AppleEvent ≪event miscdosc≫.

The Moscow ML documentation describes command lines. The command line options are specific to the image loaded. The general format is:

```
<image-name> <command-line-options>
```

Examples:

```
:mosml.image -stdlib {Extensions}:mosml:lib:
:mosml.image -stdlib Bang:mosml:src:mosmllib: -P full
:mosmllex.image :src:toolssrc:Deplex.lex
```

Before the command line is passed to mosml, each argument is pathname expanded if it contains one of a few special prefixes. All of these prefixes begin with the character '{'. These prefixes are useful in econfigstrs (see below) where they are used extensively, but they may be used in any command line. The supported prefixes are:

{Extensions}: — replaced by an absolute path to the Extensions folder in the System Folder

{ApplicationDF} — replaced by the absolute path to the current application; this is most useful with the stand-alone application mechanism; no characters should follow this prefix

Also, for some econfigstrs arguments (see below), relative pathnames, i.e., those beginning with ':' or without any ':' characters, are expanded to absolute pathnames relative to the directory containing the application.

Note: when specifying -stdlib, or -I include paths, it is best to provide complete pathnames. A relative name for the image is fine since generally it will go unused after the image is loaded, but paths are recorded verbatim and become useless as soon as chDir() is executed. This is not a concern if any of the pathname expansion mechanisms described above are used.

When launched with the command key down, mosml will type the command line it would have executed had no keys been held down. This can be copied and pasted into the mosml console to customize the startup, e.g., add "-P full" or "-P none" after the typed command line to change the set of library files initially loaded. See the Moscow ML documentation for these sets.

Warning: The command line parser will barf at pathnames with spaces in them. The entire path from the root to the image file, and to stdlib, should only contain volumes and directories without spaces (or returns!) in their names. Alternatively, you may surround the pathname with quotes ('name with space' or "name' with space").

13 Configuration

[You need to understand Mac resources and ResEdit for the rest of the Configuration section to make sense. Don't worry, you don't need to know any of this to use mosml.]

The Mac version of mosml simulates UNIX environment variables with resources of type 'ENV '. The resource number doesn't matter, the resources are identified by name. The format of an 'ENV ' resource is simply a C string. Mosml itself only looks for one 'ENV ' variable called CAMLRUNPARAM. It is used to configure the memory manager. I don't know where it's documented (probably the Moscow ML distribution). A sensible value for this string is provided as an 'ENV ' resource in the mosml application. If the resource is not found, another sensible value is used.

You may add 'ENV ' resources to the mosml application (or to the system, although I wouldn't recommend it) and they can be read using the function Process.getEnv; here's an example...

```
- load "Process.uo";
> val it = () : unit
- Process.getEnv "VERSION";
> val it = SOME "mosml 1.42e" : string option
```

There is also a 'STR#' resource in the mosml application called econfigstrs. The five strings in this string list are used to form the default command line and console window title. They are:

Meaning	String	
image name	`{Extensions}:mosml:mosml.image`	[*]
stdlib relative path	`{Extensions}:mosml:lib:`	[*]
optional	`-P`	
optional	`default`	
console name	`Mosml Console`	

When mosml uses the default command line, it converts the stdlib relative path and the image name (marked with [*]) to absolute pathnames before passing them to the image – see the Command Line section above for a description of these translations.

When a stand-alone application is created, these are changed to:

image name	`{ApplicationDF}`	[*]
stdlib relative path	`{Extensions}:mosml:lib:`	[*]
optional	`-P`	
optional	`default`	
console name	`Mosml Console`	

This causes the image to be loaded from the Data Fork of the application file where it was placed by MakeMacApplication.sml. These values come from the PPC or m68k application template.

The image name in econfigstrs is used only if

(a) the command key was not held down when the application was launched (in which case an alternative command line is accepted from the keyboard) AND

(b) no image was dragged and dropped onto mosml to launch it (in which case the dropped image would be used) AND

(c) the application was not launched with an AppleEvent ≪event miscdosc≫ (in which case the command line is taken from the AppleEvent) AND

(d) the application name does not end in ".app" AND no image was found in the application's directory with the same name as the application, but with ".app" replaced by ".image" (in which case that image is used)

14 eConsole user notes

Mosml remembers your Styles preferences in a Preferences Folder file.

14.1 Interacting with the interpreter

Interaction with the interpreter's read-eval-print (REP) loop is done inside the "Mosml Console" window. The contents of the interaction window can be edited by the user. This window also acts as the default standard output port (and input port). Consequently, the interpreter sends its output to the interaction window. User input and system output are displayed with different fonts. By default, user input is in bold face (but this and other attributes of the window can be changed with the "Styles..." item in the Edit menu).

Normally, the user types an expression and then "sends" it to the interpreter to be evaluated. The result of the evaluation is then output to the interaction window. The preferred way of evaluating an expression is to place the insertion point (caret) immediately after the expression to evaluate and then typing the <enter> key. An alternative method is to use the <opt-return> (i.e., press the <return> key while pressing the <option> key). When <enter> is pressed, the sequence of "user input" characters immediately preceding the insertion point are sent to the interpreter. Because it doesn't check for a properly formed expression, the <enter> key is useful for programs doing text oriented input. Arbitrary text can also be sent to the interpreter by selecting the text and typing <opt-return> or <enter>.

These interaction mechanisms also work from windows associated with files (i.e., in an edit buffer). However, the result of evaluation is always displayed in the interaction window.

The Console remembers the end point of the last input (plus Mosml's response, if any) and refuses to allow backspacing past that point. It beeps in response. A common new-user problem occurs when typing an expression without a trailing semicolon – it results in no response; a natural reaction is to backspace to the previous line, add the semicolon, and type Enter again (rather then simply typing the semicolon and Enter). Unfortunately, this would result in the expression being entered twice since the rule is that all text in the "run" near or behind the insert point is sent when Enter is typed. By preventing you from backing up to connect two text "runs" the problem is avoided. You can still use Cut or Clear to remove text from the Console window, if desired.

14.2 Basic editing

It is assumed that you are familiar with standard Macintosh editing techniques. If not, you should consult your Getting Started disks and Macintosh users' manuals.

On all keyboards the following keys are defined:

Key	Effect
delete	erase character before cursor position
tab	indent current line
return	insert a new line and move to the left margin (and then do a <tab> if autoindent is selected for the window)
opt-return	send the expression preceding the insertion point
enter	send the user input preceding the insertion point

14.3 Special keys

On extended keyboards the following special keys are supported:

Keys	Effect
F1 – F4	equivalent to Undo, Cut, Copy and Paste
F5 – F15	function keys (not used)
del>	erase character after the insertion point
home	scroll text to the top of the edit buffer
end	scroll text to the bottom of the edit buffer
page up	scroll text up a page
page down	scroll text down a page

Note that these last four do not move the insertion point, just scroll.

14.4 Arrow keys

Arrow keys (when available) move the insertion point in the corresponding direction:

Key	Effect
left	move cursor to the left one character
right	move cursor to the right one character
up	move cursor up one line
down	move cursor down one line

Arrow keys can be modified with the <shift>, <option>, and <command> keys.

The <shift> key may be used to extend the text selection per Apple standards. Each selection has an anchor end and an active end. Generally, the active end is the last end you moved, the anchor end doesn't move. When the <shift> key is held down, the selection's active end is moved by the arrow keys.

The <option> key may be used with the arrow keys to move the insertion point left or right by word, or up or down by page. <Shift> and <option> together may be used to move the active end of the selection left or right by words, or up or down by pages.

The <command> key may be used with the arrow keys to move the insertion point left or right to the beginning or end of the line, or up or down to the beginning or end of the text. <Shift> and <command> together may be used to move the active end of the selection to start or end of the line or text.

When <shift>, <option>, and <command> are held down together, the up or left arrow will make the beginning of the selection the active end, and the down or right arrow will make the end of the selection the active end.

14.5 Error finder

A command in the Search menu, Open Selection, can be used to open a file by selecting its name in any window and typing Command-D. This can also be used with mosml's error messages; in this case, the file that caused the error will be opened, and the error text scrolled-to and highlighted. For example, selecting the line:

```
File "errortest.sml", line 3-6, characters 4-75:
```

and typing Command-D (or selecting the Open Selection menu item) will open the file errortest.sml in an edit buffer with the four lines highlighted. This only works for files in the current directory, but this is almost always where files will be if the compiler is complaining about them.

14.6 Mouse

The editing action is specified by the number of mouse button clicks:

Button click	Effect
single	position insertion point
double	select word
triple	select line

Dragging after one of these clicks extends the selection by character, word, or line respectively; the anchor point is the original click position, the active end is the other end of the selection. After a click-drag, the active end can be moved with <shift> clicks or <shift> arrow keys. The anchor and active ends of the selection can be swapped with <option+shift+click>, or <option+shift+command> up/down or left/right arrow keys.

14.7 Scrolling

Holding down <command>, <shift>, <option>, and <control> keys speeds up scrolling when mousing in the arrows of the scroll bar. The more keys you hold down the faster it scrolls (exponentially). Live scrolling is also supported: grab the scrollbar's "thumb" indicator and drag it around; the text will follow.

14.8 Editor limitations

The text size is limited by memory space. There is a limit of 32,000 lines (not characters) per window. The editor has been used with files several hundred kilobytes in size.

Credits: Thanks to Marc Feeley for contributing to the eConsole User Notes.

References

[1] 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.

[2] 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.

[3] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Owner's Manual, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.pdf.