Standard ML of New Jersey

Release Notes
(Version 0.75)

November 11, 1991

Copyright © 1989,1990,1991 by AT&T Bell Laboratories

Standard ML of New Jersey
License and Disclaimer

Copyright © 1989,1990,1991 by AT&T Bell Laboratories
All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of AT&T Bell Laboratories or any AT&T entity not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

AT&T disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall AT&T be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

UNIX is a trademark of AT&T Corporation

VAX, DECstation and ULTRIX are trademarks of Digital Equipment Corporation.

SPARC and SunOS are trademarks of Sun MicroSystems

IBM and AIX are trademarks of International Business Machines Corporation.

HP, HP9000 and HP9000/300 are trademarks of Hewlett-Packard Company. HP-UX is Hewlett-Packard's implementation of the UNIX operating system.

PostScript is a trademark of Adobe Systems Incorporated.

Contents

1 Standard ML of New Jersey **1**

1.1 Getting this release 1

1.2 Structure of the release 3

1.3 Installing the system 3

1.3.1 Makeml 4

1.3.2 Managing Memory Use 4

1.3.3 Other options 5

2 Release history **6**

2.1 Major changes since release 0.66 6

3 How to Use the System **8**

3.1 Running Standard ML 8

3.2 Interactive input 8

3.3 Changing the prompts 8

3.4 Interrupting parsing or computation 9

3.5 Exiting the interactive system 9

3.6 Loading ML source text from a file 9

3.7 Saving an image of the system 9

3.8 Executing System commands and changing directories 10

3.9 Error messages 10

3.10 Useful system flags 10

3.10.1 Printing: System.Control.Print. 11

3.10.2 Garbage collection messages 11

3.10.3 Memory usage 11

3.11 Timing 11

3.12 Basic ML environment 12

4 Profiling **13**

5 SML/NJ Langauge Notes **15**

5.1 First-class continuations 15

5.2 Weak Types 16

5.2.1 References and weak polymorphism 16

5.2.2 Weak variables and exceptions 18

5.3 Local vs Open in Signatures 18

5.4 Discrepancies between SML/NJ and the Definition 20

6 SML/NJ Compiler Notes **21**

6.1 Program optimization flags 21

6.1.1 For the speed fanatic 22

6.1.2 Increasing Optimization 22

6.2 Batch Compiler 22

6.2.1 Installation 23

6.2.2 Using the batch compiler 24

6.3 Bootstrapping and Recompiling the Compiler 25

Chapter 1

Standard ML of New Jersey

by Andrew Appel and David MacQueen

John Reppy	Lal George	Andrew Tolmach	David Tarditi
Greg Morrisett	Gene Rollins	Zhong Shao	Trevor Jim
Emden Gansner	Georges Gonthier	Damien Doligez	Conrad Slind
Bruce Duba	James Mattson	Norman Ramsey	James O'Toole
Nick Rothwell	Kevin Mitchell	Mads Tofte	Peter Weinberger

This document describes the latest release of the **Standard ML of New Jersey** (SML/NJ) system. We are eager to receive your bug reports, comments, and constructive criticism. The documentation is still in a primitive state, but your comments on the installation instructions and manual would be appreciated. Any error message beginning with "`Compiler bug`" definitely indicates a bug in the compiler and should be reported. Please use an appropriate variation on the bug reporting form in the file doc/bugs/bug.form and send comments and bug reports to:

```
David MacQueen
Room 2C-322
AT&T Bell Laboratories
Murray Hill, NJ 07974
USA
phone: (908) 582-7691
email: macqueen@research.att.com
```


1.1 Getting this release

The primary means of distributing the compiler is via anonymous ftp from hosts `princeton.edu` and `research.att.com`. For those who do not have internet access directly or indirectory, distribution by tape is possible as a last resort. The connect info and distribution directory is given by the following table:

Host	Net Address	Directory
princeton.edu	128.112.128.1	/pub/ml
research.att.com	192.20.225.2	/dist/ml

To obtain the compiler by internet ftp, connect to one of the two hosts, use the login id "anonymous" with your email address as password, and go to the distribution directory. There you will find the following files:

README	Summary of release information
75.release-notes.ps	The postscript for this document
75.release-notes.txt	This document in ascii form
75.src.tar.Z	The src directory containing source code
75.doc.tar.Z	The doc directory containing documentation
75.tools.tar.Z	The tools directory containing various tools
75.lib.tar.Z	The lib directory containing various libraries
75.mo.m68.tar.Z	The MC680x0 object files
75.mo.vax.tar.Z	The VAX object files
75.mo.sparc.tar.Z	The SPARC object files
75.mo.mipsl.tar.Z	MIPS little-endian object files (for DEC machines)
75.mo.mipsb.tar.Z	MIPS big-endian object files (for MIPS machines)

You will need to transfer the files `75.src.tar.Z` and `75.mo.`*arch*`.tar.Z` in order to build a version for the architecture *arch*. You will probably also want to retrieve the documentation, tools and library files. Here is a sample `ftp` dialog:

```
% ftp princeton.edu
Name: anonymous
Password: login@machine
ftp> binary
ftp> cd pub/ml
ftp> get README
ftp> get 75.src.tar.Z
ftp> get 75.tools.tar.Z
ftp> get 75.lib.tar.Z
ftp> get 75.doc.tar.Z
ftp> get 75.mo.m68.tar.Z
ftp> get 75.mo.vax.tar.Z
ftp> close
ftp> quit
```

NOTE: ftp must be put into binary mode before transferring the files.

After the files are transferred they should be uncompressed and unbundled (you will probably want to do this in a new directory). For example:

```
zcat 75.src.tar.Z | tar -xf -
```

unpacks the `src` directory tree, which occupies about 2.5 megabytes of disk space. The minimum required `src` tree can be extracted using the following command:

```
zcat 75.src.tar.Z | tar -xf - src/boot src/runtime src/makeml
```

1.2 Structure of the release

Each file `75.`*file*`.tar.Z` in the distribution produces a directory tree with the root directory named *file*. The `mo.`*arch* directories contain the precompiled ML object files for building the system for the architecture *arch*.

The `src` directory tree contains the source code of the compiler and run-time system, as well as the `makeml` script, which is used for building the system (see Section 1.3).

The `tools` directory contains several software tools for SML, which are also written in SML. At the moment, this consists of the following:

`tools/lexgen` contains a lexical analizer generator.

`tools/mlyacc` contains a LALR(1) parser generator.

`tools/mltwig` contains a code generator generator based on dynamic tree pattern matching.

`tools/sourcegroup` contains a system for supporting incremental recompilation of SML programs.

The `lib` directory contains various libraries. At the moment, this consists of the *Edinburgh Standard ML Library* (in `lib/edinburgh`), various `emacs` mode files (in `lib/emacs`), and a SML structure for using `emacs` tags (in `lib/gnutags`).

The `doc` directory contains various pieces of documentation in subdirectories:

`doc/bugs` contains the form for reporting bugs and a history of bug reports and fixes.

`doc/man` contains some manual pages.

`doc/release-notes` contains this document

`doc/papers` contains copies of several papers and technical reports relating to SML/NJ.

`doc/examples` contains a number of small example programs.

`doc/refman` is an incomplete language reference manual.

Documentation for the tools and libraries can be found in those directories.

1.3 Installing the system

The compiler can be configured to generate native code for the following microprocessors: MIPS (little and big endian); Motorola 68000; SPARC and VAX. The runtime system can also be configured for a variety of operating systems including SunOS, 4.3BSD, MACH, ULTRIX, RISC/os, MORE, HPUX and V9.

1.3.1 Makeml

The file `src/makeml` is a tool for building the system from source and ML-object ('.mo') files. At the very least, the arguments to `makeml` must specify the hardware architecture and operating system. For example:

```
makeml -mips riscos
```

when executed in the `src` directory, would build the interactive compiler for a MIPS processor running RISC/os. `Makeml` assumes that the `mo` files are in a directory that is at the same level as the `src` directory.

The following table gives some useful computer/opsys combinations for arguments to `makeml`.

Processor	Brand name	Options	Comments
Vax	DEC	`-vax bsd`	(`bsd` includes Ultrix)
		`-vax mach`	
68020	Sun	`-m68 sunos`	or, `-m68 mach`
68020	HP	`-m68 hpux`	or, `-m68 more`
		`-m68 hpux8`	for version 8 HPUX
68020	Apple	`-m68 aux`	not recently tested
68040	NeXT	`-next`	
32032	Encore		no longer supported
MIPS	DEC	`-decstation ultrix`	little-endian R3000
MIPS	MIPS	`-mips riscos`	or, `-mips mach`
MIPS	SGI	`-sgi irix`	
MIPS R6000			not yet supported
SPARC	Sun	`-sun4 sunos`	or, `-sun4 mach`

1.3.2 Managing Memory Use

ML provides automatic storage management through a garbage-collected heap. Since the heap is used intensively, choice of heap size can have a significant impact on performance. The compiler determines an efficient heap size automatically on startup, resizes the heap up or down as the amount of live data changes, and complains if it runs out of memory (the interactive system can be booted in approximately 8 megabytes).

There are two options to `makeml` that help to control the heap size. The options are:

-r j the desired ratio between the size of live data and the size of the heap is set to j, a positive integer greater than or equal to 3 (the default is j=5)

-m k set the soft maximum heap size (softmax) to k Kbytes (k a positive integer), with the default being k=4096 (i.e. 4MB) You should increase this if you have a machine with more than 8MB!

In general, the larger the heap size, the lower the garbage collection overhead. In principle, heap size is limited by the total virtual memory available to a process. In practice, since heavy paging will slow down a program drastically, it is desirable to keep heap size within the bounds of available physical memory, taking into account the memory requirements of other processes sharing

the physical memory. Setting a high ratio will cause the compiler to be aggressive in its use of memory, while setting a reasonable limit on heap size can prevent excessive paging.

The soft maximum heap size (softmax) specified with the -m option is a way of controlling the system's appetite for memory. The heap will grow larger than softmax only when necessary to maintain a ratio of 3. Thus when softmax is reached, the heap will not grow until the size of live data is one third of softmax, and thereafter it will grow more slowly than with the default ratio of 5 (or the user defined ratio). A reasonable strategy is to set softmax to the maximum amount of physical memory that you know will be available and set ratio to a high value (e.g. 20), so that the heap size will tend to stick at softmax, thereby taking advantage of available memory.

1.3.3 Other options

Options also exist for building a batch compiler, a version of the compiler with the debugger loaded, cross compilers, etc. The manpage for `makeml` should be consulted for further details of options that are available.

If you have trouble installing the system, please send us a request for help, including the version of the compiler (check the definition of the `version` variable in `src/boot/perv.sml` if in doubt), hardware configuration (machine type and memory size), operating system, and an input/output script showing the problem.

Chapter 2

Release history

2.1 Major changes since release 0.66

- The `lib` directory has been split into two directories: `lib` and `tools`.
- Static environments have been reimplemented. This is a step toward "first-class" environments, which should be available in the next general release.
- The module system has been reimplemented (thanks to David Tarditi, Greg Morrisett, Georges Gonthier, and Damien Doligez). Multiple specifications in signatures are not allowed (they may have been "allowed" before, but their effect was not predictable). This may break some code, particularly if include specifications are used heavily.
- The signature of the `Array` structure has been changed to conform to the new "standard" agreed on by all implementors. The `Array` structure is no longer open by default, and the function `sub` is no longer infix.
- New structures `Vector` and `RealArray` have been added. The `Vector` structure supports immutable vectors and the `RealArray` structure provides mutable arrays of reals. Be warned that real arrays at the present are provisional and improved speed is not guaranteed. We know how to fix this and are working on it.
- The function `System.system` now has the signature

  ```
  val system : string -> int
  ```

  where it returns the Unix status code of the sub-process.
- The function `IO.execute` has a different signature and two forms:

  ```
  val execute : (string * string list) -> (instream * outstream)
  val execute_in_env : (string * string list * string list)
        -> (instream * outstream)
  ```

  The first of these allows command-line arguments to be passed to the program; the second also allows an environment to be specified. In both cases, the base name of the first argument is prepended to the argument list.

- Structure `Compile` added to pervasives to support Gene Rollins' sourcegroups separate compilation system. This is the last release in which the "`import`" facility is supported. We recommend that you use sourcegroups for separate compilation in the future. A new version of sourcegroups is included with the 0.75 release.
- Callee-save registers (for 7% better performance on MIPS and 18% on SPARC).
- In-line array update and subscript (for 25% better performance on "real" programs that use arrays; program that do nothing but array accesses are twice as fast).
- The run-time system has been revised to support shared memory multiprocessing, which includes two new primitives: `getvar` and `setvar`.
- Floating point performance is improved. The SIMPLE benchmark shows a 33% improvement in speed, while some toy programs have exhibited as much as 2 times speedup.
- The parser conforms more closely to the syntax in the Definition, so that `1+if` ... doesn't work anymore; use parentheses: i.e., `1+(if ...)`. Similarly for `case` and `fn`.
- `callcc` now has a weak polymorphic type in order to avoid a loophole in the type system. The structure `System.Unsafe.PolyCont` has the old, unsafe, version of `callcc`, plus the primitives `capture` and `escape` that are like `callcc` and `throw` but do *not* preserve the exception handler of the saved continuation.
- Hexadecimal integer literals are supported using C syntax (e.g., `0x5e48`).

Chapter 3

How to Use the System

3.1 Running Standard ML

Type "sml". This puts you into the interactive system. The top level prompt is "- ", and the secondary prompt (printed when input is incomplete) is "= ". If you get the secondary prompt when you don't expect it, typing ";return" will often complete your input, or type your interrupt character (e.g. ctrl-C) to cancel your input and return you to top level.

If "sml" doesn't work, ask where sml has been installed on your machine and use the appropriate path name or redefine your PATH environment variable.

3.2 Interactive input

Input to the top level interpreter (i.e. declarations and expressions) must be terminated by a semicolon (and carriage return) before the system will evaluate it. The system then prints out a response indicating the effect of the evaluation. Expressions are treated as implicit declarations of a standard variable `it`. For example,

```
- 3;                    user input after prompt
val it  = 3 : int       system response
```

This means that the value of the last top level expression evaluated can be referred to using the variable "`it`."

3.3 Changing the prompts

The primary and secondary prompt strings are the contents of the references

```
System.Control.primaryPrompt
System.Control.secondaryPrompt
```

```
- System.Control.secondaryPrompt := "(**)";
```

3.4 Interrupting parsing or computation

Typing your interrupt character should interrupt the compiler and return you to top level, unless some function is catching the Interrupt exception (a dangerous thing to do).

3.5 Exiting the interactive system

Typing ctrl-D (EOF) at top level will cause an exit to the shell (or the parent process from which sml was run).

3.6 Loading ML source text from a file

The function

```
val use : string -> unit
```

interprets its argument as a Unix file name relative to sml's current directory and loads the text from that file as though it had been typed in. This should normally be executed at top level, but the loaded files can also contain calls of `use` to recursively load other files. It is a bad idea to call `use` within an expression or declaration, because the effects are not well-defined.

3.7 Saving an image of the system

Use the function

```
val exportML : string -> bool
```

to dump an image of the current sml system including the environment that you have built. The argument is the path name of the image file that is created. The result `false` is returned in the original system, while in the saved image the value returned is `true`. The call of exportML can be embedded in an expression which will continue evaluation (e.g. to print a message) in both the original system and in the image when it is run, and its effect can depend on the result of the exportML call. For example:

```
if exportML("saved")
  then print "this is the saved image\n"
  else print "this is the original process\n"
```

The saved image file is an executable binary, and can be run by typing the file name as a command to the shell. (Access to command-line arguments and Unix environment variables when running the saved image may be accomplished by System.argv and System.environ.)

3.8 Executing System commands and changing directories

The function

```
val system : string -> int
```

spawns a process to execute its argument string as a subprocess command. Thus, to find out what the current directory is within sml you can evaluate the expression

```
system "pwd";
```

which will cause the current directory to be printed out (there is no way at the moment to return the current directory as a string). To change the current working directory of sml use the function

```
val cd : string -> unit
```

whose argument should be a path name denoting a directory.

3.9 Error messages

The error messages produced by the compiler are not always as helpful as they should be, and there are often too many of them.

The compiler attempts to recover from syntactic and type errors so that it can detect as many errors as possible during a compilation. Unfortunately, it is not very graceful in recovery, and the process can cause numerous spurious secondary error messages.

When compiling files, the error messages include a line number. For simple syntactic errors this line number is often accurate or off by just one line. For other classes of errors, including type errors, the line number may not be very useful, since it will often just indicate the end of the declaration containing the error, and this declaration can be quite large.

There are a number of different forms of type error message, and it may require some practice before you become adept at interpreting them. The most common form indicates a mismatch between the type of a function (or operator) and its argument (or operand). A representation of the offending expression is usually included, but this is an image of the internal abstract syntax for the expression and may differ significantly from the original source code. For instance, an expression "`if` e_1 `then` e_2 `else` e_3" is represented internally as a `case` expression over a boolean value: "`case` e_1 `of true =>` e_2 `| false =>` e_3."

3.10 Useful system flags

There are a number of useful system flags and variables, which are found in the structure `System.Control` and its substructures. The primary and secondary prompt variable have already been mentioned; here are some more:

3.10.1 Printing: System.Control.Print.

`printDepth : int ref` controls depth to which complex values and syntax trees are printed (default 5)

`stringDepth : int ref` controls how much of a long string will be printed (default 70)

`signatures : bool ref` when `true`, signatures, and the signatures of structures, will be printed when these are defined at top level (default true)

3.10.2 Garbage collection messages

`System.Control.Runtime.gcmessages: int ref`

- when 0, no messages are printed
- when 1, only major collections are reported
- when 2, major collections and heap resizings are reported (the default)
- when 3, minor and major collections and heap resizings are reported

3.10.3 Memory usage

`System.Control.Runtime.ratio : int ref`

determines the desired ratio between size of live data and total heap size. Default is 5, and 3 is the smallest acceptable value. A higher ratio causes more aggressive use of memory (up to the softmax bound).

`System.Control.Runtime.softmax : int ref`

suggested ceiling on heap size, in bytes. Heap size will not grow beyond this value except to maintain the "minimum" ratio of 3. Actually, when hard limits are reached (e.g. as determined by limit datasize), the system can continue to run as long as the actual ratio is greater than 2. A good value for softmax is one that reflects the amount of physical (not virtual) memory that is expected to be available for the sml process, for instance, 5000000 (5MB) might be appropriate on an 8MB Sun-3.

3.11 Timing

The structure `System.Timer` provides basic facilities for timing ML programs. It has the following signature:

```
signature TIMER =
  sig
    datatype time = TIME of {sec : int, usec : int}
    eqtype timer
    val start_timer : unit -> timer
    val check_timer : timer -> time
    val check_timer_sys : timer -> time
    val check_timer_gc : timer -> time
    val makestring : time -> string
    val add_time : time * time -> time
    val sub_time : time * time -> time
    val earlier : time * time -> bool
 end
```

Here is how a typical timing function could be implemented:

```
fun timeit (f : unit->'a) =
    let
      open System.Timer
      val start = start_timer ()
      val result = f()
      val non_gc_time = check_timer start
      val gc_time = check_timer_gc start
      val total_time = add_time(non_gc_time,gc_time)
    in
        print (makestring(makestring total_time));
        print "\n";
        result
    end;
```

3.12 Basic ML environment

Look at the files `src/boot/perv.sig` and `src/boot/system.sig` for signatures that specify what is available in the basic environment.

Chapter 4

Profiling

You can use the built-in execution profiler to find out where the bottlenecks in your programs are. The profiler will tell how much time is spent in each function and how many calls were made to each function. In order to profile a set of functions, those functions must be compiled in "profiling mode."

Here is how to use the profiler:

1. System.Control.Profile.profiling := true;

   This tells the compiler to insert profiling hooks in the compiled code for any ML declarations that are compiled from now on, or until this flag is set back to false.

2. Compile some ML code, either by typing it in to the interactive system, or by means of the "use" function.

3. System.Control.Profile.profileOn();

   This starts the timer interrupts that are used for statistical profiling.

4. Execute your program

5. System.Control.Profile.profileOff();

   This turns off the timer interrupts.

6. System.Control.Profile.report std_out;

   This prints a report of profiling information on your screen. For each function you will be told what percentage of the time was spent in that function, how many calls were made, etc. Instead of std_out you can call "report" with any outstream as an argument if, for instance, you want the report sent to a file.

Other commands:

```
System.Control.Profile.reset: unit -> unit
```

The "reset" command sets all timing and call-count information to zero, but remembers which functions have been profiled.

```
System.Control.Profile.clear: unit -> unit
```

The "clear" command tells the system to forget about profiling any functions that have been compiled so far, but allows you to compile and profile further functions.

Of course, the long names can be avoided by first opening the structure System.Control.Profile.

A slightly simpler way to use the profiler is as follows (after having opened System.Control.Profile):

```
profiling := true;
profileOn();
compile and execute some ML code
profileOff();
profiling := false;
report std_out;
```

The batch compiler can also be made to generate object files with profiling code by toggling the flag `profiling` to true, using the command "`^profiling`". We use this method to profile the compiler itself.

For more details about the profiler, and to learn more tricks you can make it do, read the paper *Profiling in the presence of optimization and garbage collection* provided in the `doc/papers` directory. Details of the implemention have changed since that paper was written, but it is still basically accurate.

Chapter 5

SML/NJ Langauge Notes

5.1 First-class continuations

A set of new primitives has been added to ML to give access to continuations:

```
type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw : 'a cont -> 'a -> 'b
```

The continuation of an expression is an abstraction of what the system will do with the value of the expression. For example in the expression:

```
if a orelse b then foo() else goo()
```

the continuation of the expression "a orelse b" can be described in words as "if the value is true then compute `foo()` otherwise compute `goo()`" and then continue in the context of the `if` expression. Usually the continuation of an expression is implicit, however, the primitive `callcc` allows the programmer to capture and use these continuations.

The primitive `callcc` takes a function as an argument and applies it to the current continuation. The continuation of an expression of type α has type α cont and is a first-class object. To capture the continuation described above one would write:

```
if callcc(fn k => a orelse b) then foo() else goo
```

Here the continuation of the `callcc` application is captured by being bound to k, but it is not used. Because the continuation is not used the result is simply the result of the expression "`a orelse b`." To use the continuation a value must be supplied, and the computation continues as if that value where the result of the `callcc` application. This is called throwing the continuation a value; it is performed by applying `throw` to the continuation and the value.

```
if callcc(fn k => (throw k false) orelse b) then foo() else goo()
```

Here, when the continuation k is thrown the value false, "`orelse b`" is simply ignored, the `callcc` application returns false, and goo() is then evaluated.

The type returned by a `throw` expression is unconstrained like that of a raise-expression and for the same reason: neither of these expressions ever return.

One of the less interesting uses of `callcc` is as an alternative to exception handlers. For example:

```
exception Prod

fun prod l = let
      fun loop [] = 1
        | loop(0::r) = raise Prod
        | loop(a::r) = a * loop r
      in
        loop l handle Prod => 0
      end
```

can be written with `callcc` as follows:

```
fun prod l = callcc (fn exit => let
      fun loop [] = 1
        | loop(0::r) = throw exit 0
        | loop(a::r) = a * loop r
      in
        loop l
      end)
```

But continuations are more general than exception handlers and can be used to implement sophisticated control structures. These more complex uses often involve subtle interactions with the type system. For just a taste of the techniques involved consider the following example implementing an infinite loop:

```
datatype State = S of State cont

fun state_throw (S k) = throw k

let val s = callcc S in state_throw s s end
```

5.2 Weak Types

5.2.1 References and weak polymorphism

The type checker uses weak type variables to support secure use of references and arrays and other objects like hash tables implemented in terms of references and arrays. The following is a very brief and preliminary explanation of how they work, but you should try some experiments to become familiar with the behavior of weak polymorphism.

As has been known for some time, mutable objects like references can be the source of type failures not detected by static type checking, if the reference primitives are treated as ordinary polymorphic values. A standard example is

```
let val x = ref(fn x => x)
 in x := (fn x => x+1)
    !x true
end
```

The basic principle we use to avoid such errors is that the contents of an "actual" reference must have monomorphic type. Therefore, declarations such as:

```
val x = ref []
```

are now illegal and will cause an error message. A function, like ref, that directly or indirectly creates references can have a polymorphic type, but of a special kind. Thus the type of the ref constructor itself is now

```
val ref : '1a -> '1a ref
```

where the type variable '1a is a "weak type variable of degree 1." Basically, this type indicates that when the function ref is applied, its instantiation must be given a ground type. But the notion of ground type must be interpreted relative to the context, e.g. bound type variables can be treated as type constants within the scope of their binding. For example:

```
fun f (x : '1a) = ref [x]
```

is ok, even though the type of the embedded ref expression is '1a list ref, because '1a is a bound type variable in this context. The type of the function `f` is `'1a -> '1a list ref`.

The degree of weakness (or perhaps strength is a better term) of a type variable reflects the number of lambda abstractions that have to be applied before the reference object is actually created and that type variable must be monomorphically instantiated. Ordinary type variables can be considered to have strength infinity. Each application weakens the operand type another step, and when the strength of a type variable becomes 0 it must be eliminated by instantiation to a ground type [weak type variables of degree zero are not allowed in a top-level type]. Conversely, each surrounding lambda abstraction strengthens type variables.

For example,

```
- val g = (fn x => (fn y => (ref x, ref(x,y))));
val g = fn : '2a -> '2b -> ('2a ref * ('2a *'2b) ref)
- val h = g(nil);
val h = fn :  '1a -> ('1b list ref * ('1b list * '1a) ref)
- h true;
std_in:4.1-4.6 Error: nongeneric weak type variable
  it : '0Z list ref * ('0Z list * bool) ref
```

but

```
- (h true) : int list ref * (int list * bool) ref;
val it = (ref [],ref ([],true)) : int list ref * (int list * bool) ref
```

The type constraint is necessary to instantiate the weak type variable '1c when h is applied.

If a component of a structure has a weak polymorphic type, then the corresponding signature specification should have at least as weak a type (the weaker-than relation between types should be fairly obvious).

5.2.2 Weak variables and exceptions

If the declared argument type of an exception constructor contains a type variable, then that type variable is bound in the appropriate surrounding context according to the usual rules (its binding scope is the rhs of the innermost let-binding containing the occurence of the type variable). Furthermore, the type variable must be a weak variable of the same degree as it would have were it associated with a ref argument at that point (i.e. its weakness must agree with the abstraction degree at the point of the exception declaration.) This is because the creation of an exception constructor is conceptually similar to the creation of a reference value – both can be used to transmit values between two textually unrelated points in the program.

It is best if the type variable occurring in the exception declaration had already been introduced by appearance in a type constraint on a lambda binding.

Here is an example:

```
fun f(l: '1a list, p: '1a -> bool) = let
      exception E of '1a list
      fun search(x::r) = if p x then raise E r else search r
        | search [] = []
      in
        search r handle E l => l
      end
```

As an exercise, show how this rule prevents the usual type insecurity example associated with "polymorphic" references:

```
let val (r,h) = let
      exception E of 'a
      in
        ((fn x => raise x), (fn f => f() handle E y => y))
      end
```

5.3 Local vs Open in Signatures

.. by David B. MacQueen

I am not sure what the purpose of the local spec form was supposed to be. Specifications such as

```
local val x : t
 in val y : s
end
```

are certainly useless, and

```
local type t
 in val x : t
end
```

does not seem very interesting either. Presumably the purpose of local specs was to localize the scope of open specs, as in

```
structure A : sig type t ... end

local open A in
  val x : t
end
```

But I would argue that open specs are really just an abbreviation device and do not add specifications to a signature in any case. The bindings in a structure, such as A above, are a different kind of entity from normal signature specifications, and they can be treated differently from specifications. It makes sense to include the specifications in a signature in another signature, as is done by the include spec, but it does not make sense to include the components of a structure in a signature. For instance, consider the following example

```
structure A = struct type t = int val x = 3 end
signature SIG =
  sig
    open A
  end
```

What is the meaning of SIG? Since the unconstrained structure A matches many different signatures, it would seem that SIG is not well defined.

As we have implemented the open specification, it simply makes the identifiers inside the opened structures visible in unqualified form, so that "t" can be used as an abbreviation for "A.t" in the above example. The scope of the open specification is by default the remainder of the signature expression, and an open specification does not add any new components to a signature. Thus the above definition of SIG is equivalent to

```
signature SIG = sig end
```

This view of open specs seems to eliminate any justification for local specs, so they are suppored only in a dummy form for compatibility. The meaning of

`local` $spec_1$ `in` $spec_2$ `end`

is the same as

$spec_1$ $spec_2$

except that the specifications in spec1 are ignored during signature matching. If spec1 is an open spec, which is the only reasonable possibility, the open spec remains in force even after the end of the local spec. For instance,

```
sig
  structure A : sig type t ... end
  type t
  local open A in
    val x : t  (* t is A.t *)
  end
  val y : t  (* t is still A.t, not the top level t *)
end
```

We would be interested to hear any arguments against this treatment. If it is generally accepted, it would seem to make sense to delete local specs from the language. In any case, it should be considered a dubious feature to be used only when necessary for compatibility. I also consider the let structure expression a dubious feature, but it is at least fairly simple to implement.

5.4 Discrepancies between SML/NJ and the Definition

The pervasive environment of SML of New Jersey is a superset of the initial basis of the Definition of Standard ML (Milner/Tofte/Harper) and differs from in it various ways:

- The arithmetic overflow exceptions:
  - `Sum`, `Prod`, `Diff`, `Neg`, `Exp`, `Floor` are all equivalent to `Overflow`.
  - `Div` and `Mod` are equivalent to `Div` and distinct from Overflow.
- The `@` operator is right-associative.
- Strings carried by the Io exception are more informative.

The language implemented by SML-NJ also differs from that described in the Definition. Here is a partial list of the discrepancies:

- Different right-associative operators of the same precedence associate to the right.
- "local" and "open" specs in signatures have a different semantics (see Section 5.3).
- The symbol `=` can be re-bound (though usually with a warning message)
- The construct `val ...` and `rec ...` is not permitted; the `rec` must immediately follow the `val`.
- The Definition prohibits some, but not all, signatures that could never be matched by any structure. We are slightly more liberal about accepting such signatures.
- Multiple specifications of a name (in the same name space) are not allowed in signatures.

Chapter 6

SML/NJ Compiler Notes

6.1 Program optimization flags

Each compilation unit is compiled separately. None of the optimizations take place across compilation-unit boundaries. Example:

```
fun f(x) = (x,x);
fun g 0 = nil | g i = f i :: g(i-1);
```

This is two compilation units if typed at top level, or if loaded from a file because at the first semicolon, the function f is compiled, and then at the next semicolon, g is compiled. The function g will run significantly faster if any of the following is used instead:

```
fun f(x) = (x,x)
fun g 0 = nil | g i = f i :: g(i-1);

local
  fun f(x) = (x,x);
in
  fun g 0 = nil | g i = f i :: g(i-1)
end;

structure S =
  struct
    fun f(x) = (x,x);
    fun g 0 = nil | g i = f i :: g(i-1);
  end;
```

In either of these last two, of course, the semicolons are optional.

Moral of the story: use small compilation units while typing to the interactive system and seeing how things work. Use larger compilation units when compiling large programs. I recommend the use of the module system, or of `let` and `local` declarations, to bind things together in a well-structured way.

The use of signature constraints to minimize the number of things exported from structures will reduce memory usage, and is just clean style.

6.1.1 For the speed fanatic

The initial environment (i.e., the `List`, `Array`, `Ref`, etc. structures) is normally in a separate module from the user program. If you would like a copy of this stuff in your program so that calls to the pervasive functions will have less overhead, textually insert `src/boot/fastlib.sml` near the beginning of your own structure. This only helps, of course, if `fastlib.sml` is put into the same compilation unit as the functions calling it, using the module system as described above.

You can nest structures. To get better performance, after you have developed your program, nest the whole thing in one huge structure, e.g.,

```
structure Whole =
  struct
    your program
  end
```

You can even put signatures and functors at top level inside such a structure, although this is not "Standard" ML.

6.1.2 Increasing Optimization

You can increase the level of optimization, if you want to wait a bit longer for compiles. To make things compile more slowly but run faster, execute this before compiling your program:

```
System.Control.CG.reducemore := 0;
System.Control.CG.rounds := 10;
System.Control.CG.bodysize := 20;
```

To make things compile faster but run slower, try this:

```
System.Control.CG.reducemore := 10000;
System.Control.CG.rounds := 0;
System.Control.CG.bodysize := ~100;
System.Control.CG.reduce := false;
```

6.2 Batch Compiler

The SML/NJ batch compiler provides some (unsafe) separate compilation and cross compilation capabilities. The batch system compiles files containing signature, structure, and functor declarations, i.e. only module level declarations are allowed. Compiling a module "Foo" produces a corresponding object file "Foo.mo", which contains the names of the other .mo files required to build the module, as well as the code of the module itself. Modules are compiled separately, producing the necessary .mo files.

A program is executed by passing a module name to the run-time system with the command (assuming src is current working directory):

```
runtime/run [memory managment options] Module
```

The run-time system starts a linker which loads in the module, finds out what other modules it depends on, recursively loads them and passes them to the original module. The module then "builds" itself, by executing its top-level declarations and creating a run-time structure. By convention, the loader looks for the module object files in the directory src/mo. In this case it would look for an object file named src/mo/Module.mo.

For example, batch compiling the following module will create a file named `HelloWorld.mo`.

```
structure HelloWorld =
  struct
    val foo = print "hello world"
  end
```

When `HelloWorld.mo` is placed in the directory `src/mo` and the command

```
% runtime/run HelloWorld
```

is executed, the run-time system will build a run-time record containing foo, and as a side effect of building the record, "hello world" will be printed.

Thus a program consists of the collection of all modules on which a given top-level structure (transitively) depends. The order in which the modules will be executed/created is determined by a postorder traversal of the (acyclic) dependence graph.

Using the batch system for separate compilation is unsafe since the .mo files contain no type information, only the names of the modules on which they directly depend. However, as long as each module has the correct signature, the run-time record will be type-correct; in other words, it is safe to edit, re-compile, and re-link modules as long their signatures do not change.

6.2.1 Installation

The compiler is composed of two parts: a run-time system written in C and assembly language; and a number of ML object files which form the main part of the compiler. The run-time system provides garbage collection, signal handling, and system calls; the rest of the compiler, including the standard library, is written in ML. The .mo files for the compiler reside in the mo.vax and mo.m68 subdirectories of the ML distribution. The source for the compiler resides in the src subdirectory, and these instructions assume that the current directory is src.

To build a batch compiler:

1. Go to the src subdirectory of mldist:

   ```
   % cd src
   ```

2. Run the makeml script with the -batch option, e.g.

```
% makeml -batch
```

The other options for makeml have the same meaning as for the interactive system. See the man page doc/makeml.1.

As the command executes, you should see messages like "[Loading Foo]" and "[Executing Bar]", which indicate the modules being linked in, and then messages like "signature ASSEMBLY" and "functor CoreFunc", which indicate that the standard library is being compiled.

When the command successfully terminates, an executable image named "batch" has been created in the current directory. This is the batch ML compiler. If you wish to use a different name for this file, use the -i option of makeml.

Now you can run the batch compiler by typing "batch". You can give it commands interactively, or by redirecting its input from a file, as is done below with the command script "all".

Normally you will want to build a complete interactive or batch system, but there are occasions when you may want to build just the run-time system. The option -run of makeml will do this.

When invoked, `runtime/run` takes the name of a module (say "Foo") as a parameter. It looks for a file with path name mo/Foo.mo relative to the current directory and loads that file, and then recursively loads mo/Bar.mo for all modules Bar on which Foo depends. Finally the code for creating the module Foo is executed. For instance, to run the interactive system on a Vax, one could symbolically link mo to mldist/mo.vax (or ../mo.vax if current directory is src) and execute the command

```
runtime/run IntVax
```

This causes mo/IntVax.mo to be loaded, together with all the modules on which it depends (essentially the whole compiler). IntVax.mo (defined in src/build/glue.sml) causes the interactive top-level loop to be executed when the functor Interactive is applied. Any program can be directly loaded and executed by the run-time system in this manner, not just the compiler.

The run-time system accepts the option flags -h, -r, -m, and -g to control heap sizing and garbage collection messages.

6.2.2 Using the batch compiler

The batch compiler accepts commands on its standard input. The list of commands is:

`!file`	compile the file.
`*file`	assemble the file.
`<file`	parse the file.
`>file`	export to a file.
`%`	print the last generated lambda (intermediate code).
`#word`	comment; ignored.
`@directory`	look for files in a directory. directory should end in
`~function`	execute a function.
`^flag`	toggle a flag.
`?`	print this help message.

There should be no space between the command character and the argument string, which should not be quoted.

The execute "~" and toggle "^" commands are mainly used to control debugging facilities, which are not explained here. Typing "^" alone on a line produces a list of possible flags.

The compile command, "!", causes the named file to be compiled, generating an object file A.mo for each module A defined in the file. The assemble command, "*", generates an assembly listing A.s that can be assembled to produce the corresponding .mo file. The parse command, "<", causes the file to be parsed and type-checked, but produces no output files. These three commands all update the symbol table with the bindings defined in the file. The file must contain only signature, structure, and functor declarations.

The export command ">" exports the current state of the batch compiler to an executable file with the given name. This is a way of preserving the symbol table state of the compiler, and is useful for separate compilation; if you change a module without changing its signature, you can safely recompile it using the exported compiler (which contains the symbol table information from other modules that the modified module may require). For example, the file "all" in the src directory is a batch command script for compiling the compiler. The last command in the file exports to a file "upto.all", which can recompile any module of the compiler.

Once you have compiled all of your code, you only need to move it to the mo directory and execute it in the same way as CompM68 or CompVax:

```
runtime/run Foo
```

for a module Foo. Note that the files CoreFunc.mo, Math.mo (for Vax), Initial.mo, and Loader.mo must be in the mo directory, as they are used in bootstrapping the system.

6.3 Bootstrapping and Recompiling the Compiler

`runtime/run` is the ML loader. It searches for things it needs in the mo directory. It takes one argument, which is the name of a structure. That structure is found in the mo directory and executed for side effects.

`runtime/run` loads only 4 .mo files:

```
CoreFunc.mo  Math.mo Initial.mo Loader.mo
```

It passes its argument (a structure name) to Loader.mo, which then loads many other .mo files (the entire DAG of dependencies whose root is CompFoo.mo (or whatever)).

The mo directory contains mo files. These carry names of top-level structures or functors. They are copies of the output produced by the code generator. As such they are machine-dependent. There is no software support for figuring out to which code generator (or machine) a .mo file corresponds. Thus it is entirely up to the user to make sure that the .mo files in the .mo directory make sense and are the right ones for the task at hand.

The highest-level structure of a mo file is: fn () => (["a","b","c"], fn [a,b,c] => top-level structure or functor) where the ["a", "b", "c"] is a list of names of structures on which the top-level

thing depends, and the [a,b,c] are the structures themselves. Thus a, b, and c are the only free variables in the top-level structure, and the thing in the mo file is actually a closed lambda-term. Only the run program reads .mo files.

The CompFoo structure (created with the Batch functor) is a structure that executes a small batch compiler as a side effect. IntFoo is a structure that executes the full interactive system as a side effect. The only difference between the two is in the user interface. The 'run' loader is typically used only on one of these two structures, to execute either the batch or the interactive system. Both the batch and interactive systems support an 'export ML' command that saves the currently executing system into a file in a.out format, so that it can be re-executed at will.

The batch system has three interesting commands that deal with ML source. They are:

```
!file   compile file, and write a .mo file for each top-level structure
*file   compile file, and write a .s file for each top-level structure
<file   read in and parse the file
```

All three commands enter appropriate things into the batch compiler's symbol table, so that later files can refer to them, and the appropriate top-level references can be resolved.

It is possible to set batch options with ^option The options ^printit and ^saveLvarNames will cause useful intermediate code to be written to the .s file, in addition to the assembly code.

Making a new compiler

Use the appropriate batch to generate appropriate .mo files. Then use the boot loader (run) to generate a new compiler. There it is.

Baby steps

When debugging the mips code generator, it might be worth replacing CoreFunc with a very simple structure ("hello world"), then using batchm to generate a fake CoreFunc.mo. runtime/run would execute CoreFunc before dying in Math or Initial.

NOTE: Recompiling the compiler will require a heap size of at least 6MB, so it if possible it should be done on a machine with at least 8MB of physical memory. Use the -m flag of runtime/run to set softmax as high as reasonable (e.g. -m 6000 on an 8MB Sun-3).

NOTE: Beware of using the sharable run-time system when recompiling the compiler, since it has the effect of embedding old versions of the .mo files into the system. Either use a nonsharable version of runtime/run, or rebuild a sharable runtime/run after each iteration of compiling the compiler to incorporate the latest .mo files.

If you change the compiler, you can use the batch command script "all" to recompile it with the batch system:

```
batch < all
```

or

```
runtime/run {CompM68 | CompVax | CompSparc | CompNS32} < all
```

This will generate new .mo files for all the modules in the compiler including the "boot" modules (whose source files are found in src/boot):

```
CoreFunc.mo  [boot/core.sml]
Math.mo      [boot/math.sml]
Initial.mo   [boot/perv.sml]
Loader.mo    [boot/loader.sml]
```

These are generated by the `~mBoot` command in the all script rather than using the `!` compile command. These files are treated differently because of their role in bootstraping the pervasive (i.e. built-in) environment.

The newly generated .mo files should be moved into a new mo directory, say mldist/mo.new:

```
cd ..   (to mldist)
mkdir mo.new
mv src/*.mo mo.new
```

After generating new .mo files for the compiler (including the boot modules if necessary), you should build a new batch compiler by symbolically linking mldist/mo.new to mldist/src/mo and recompiling the source code again (using the -mo option to makeml to designate the object directory).

```
% cd src
% makeml -batch -mo ../mo.new -vax bsd ...
% batch < all
```

The resulting .mo files should be identical to the previous set of .mo files in mo.new. Sometimes more than one iteration is necessary (for example if the signature of one of the boot modules has changed). The bootstrapping is successful when two successive iterations produce identical .mo files.

The last command in the all file causes an image of the batch compiler to be saved in the file upto.all. This image contains the symbol table information for all the modules in the compiler. It can be used to recompile individual files in the compiler after they have been modified (but beware of changes in signatures).

Cross compiling

If you have, say, the Vax object files in mldist/mo.vax and need to build the sparc object files, then you can build a batch compiler to generate sparc object files by using the -target option of makeml:

```
% makeml -vax bsd -target sparc
```

(the -target option implies the -batch option). Then the resulting batch compiler can be used to generate the sparc mo files by executing (in directory src):

```
smlc < all
```

Then make a directory mldist/mo.sparc and move the new object files into this directory. These object files can then be used to bootstrap the compiler on a sparc machine in the usual way.