Standard ML of New Jersey

—

User's Guide

(Version 0.93)

February 15, 1993

Contents

Standard ML of New Jersey
License and Disclaimer

1 Introduction

This document provides some general information about the Standard ML of Jersey (SML/NJ) system and how to use it. This is not a tutorial on the Standard ML language, however. To learn how to program in Standard ML you should read one of the available books or tutorials on the subject, such as Paulson's *ML for the Working Programmer* (Cambridge, 1991).

1.1 Reporting Bugs and Errors

We are eager to receive reports of bugs in the compiler and related tools and libraries or errors in the documentation. Any error message beginning with "`Compiler bug`" definitely indicates a bug in the compiler and should be reported. But don't be shy about reporting any problems that may conceivably be bugs. Please use the bug report form in the file `doc/bugs/bug.form` and send comments and bug reports to `sml-bugs@research.att.com`.

Comments and suggestions regarding the new SML/NJ Library are also welcome and should be sent to John Reppy at `jhr@research.att.com`.

1.2 Keeping in Touch

There is a mailing list for news and discussion of Standard ML (and other dialects of ML). The address of the mailing list is `sml-list@cs.cmu.edu`, but administrative requests, such as requests to join the list, should be sent to `sml-request@cs.cmu.edu`. The mailing list is currently moderated by Greg Morrisett.

If things go as planned, the sml mailing list will soon be supplemented by a new netnews group, `comp.lang.sml`.

Announcements of new versions of Standard ML of New Jersey are distributed via the sml mailing list and several existing netnews groups, such as `comp.lang.misc` and `comp.lang.functional`.

1.3 Documentation

The Standard ML of New Jersey system is described by the following documents, which are found in the `doc` directory of the distribution.

- **Release Notes**. This describes what is in the Standard ML of New Jersey distribution, what hardware and software is required to use the compiler, and how to install the software.
- **Standard ML of New Jersey Reference Manual**. This manual contains several sections which are found in separate files.

  GUIDE is the document you are now reading. It contains basic information on using the system, discusses language issues pertinent to Standard ML of New Jersey, and describes some special features of the compiler. (`doc/manual/GUIDE.ps`)

  BASE describes the contents of the basic environment (the modules, types, values, *etc.* that are bound in the default environment). (`doc/manual/BASE.ps`)

  SYS documents the components of the System structure, which provides facilities for interacting with the compiler internals and with the host operating system. (`doc/manual/SYS.ps`)

  LIB is the Standard ML of New Jersey Library Manual. This describes a set of modules providing commonly useful abstractions and facilities such as sorting, hashing, regular expressions, and formatted I/O. With any public release of SML, `doc/manual/LIB.ps` will be essentially identical to `smlnj-lib-0.1/lib-manual.ps`. However, we expect releases of the library to be more frequent than the compiler, and the most recent version of the library manual should be obtained from `lib-manual.ps`, which is also stand-alone.

TOOL ML software tools, such as SourceGroup, mlyacc, etc. (`doc/manual/TOOL.ps`)

In addition, each of the tools SourceGroup, mlyacc, lexgen, and mltwig come with their own manuals or documentation files. These documentation files are located in the appropriate subdirectories of the tools directory. If you are installing SML/NJ on your system you may want to provide a local guide that tells where to find the commands, documentation, tools, and libraries on your system.

2 How to Use the System

This section explains some of the basic elements of using the Standard ML of New Jersey compiler as an interactive system into which you enter declarations and expressions or load code from source files. This is the most accessible mode of using the compiler, but in the near future there will be other modes available that are more appropriate for building "stand-alone" ML applications with a minimum of extraneous baggage. We assume below that you are using the compiler under Unix. The behavior will be somewhat different under other operating systems such as the Macintosh OS.

Running Standard ML. Type "`sml`" from the Unix shell. 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 typing your interrupt character (*e.g.* control-C) will cancel your input and return to the ML 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.

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`."

Interrupting compilation or execution. Typing the interrupt character (control-C, or DELETE, depending on your "stty" parameters) should interrupt the compiler and return you to top level. This is no longer accomplished by raising the `Interrupt` exception, and hence interrupts cannot be trapped by an exception handler.

On the MIPS Risc/OS operating system, you may need to type a carriage return after the control-C.

Exiting the interactive system. Typing control-D (EOF) at top level will cause an exit to the shell (or the parent process from which **sml** was run). One can also terminate by calling `System.Unsafe.CInterface.exit()`.

Loading ML source text from a file. The function `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.

For industrial-strength multi-module software systems, the SourceGroup system may be more appropriate than `use` (see `tools/sourcegroup`).

Saving an image of the system. Use the function `exportML: string -> bool` to save an image of the current **sml** system (including the environment that you have built) in a file. See Section EXPORTML(BASE) [Section EXPORTML of Chapter BASE], for details. The saved image file is an executable binary, and can be run by typing the file name as a command to the shell.

Library functions, Base environment. Many functions are available in the standard initial environment; these are described in Chapter BASE. The rest of the library is provided (at present) as source that can be loaded; see Chapter LIB.

The built-in library functions, the set of separately loadable structures and functors, and the programming interface to the Unix operating system are described in Chapter LIB.

Error messages. The compiler attempts to recover from syntactic errors so that it can also produce semantic (type-checking) errors during the same compilation. Syntactic error recovery is more accurate for source files loaded by `use` or SourceGroup than it is in the interactive system—this is because lookahead is not possible when text is entered one line at a time.

When compiling files, the error messages include line numbers and character positions within the line. For example:

```
- if true
= then 5 true
= else 6;
std_in:7.6-7.11 Error: operator is not a function
  operator: int
  in expression:
    5 true
-
```

Here the location information `std_in:7.6-7.11` indicates that the erroneous expression "`5 true`" occupies characters 6 through 11 of the 7th line of input from `std_in`. For simple syntactic errors this position information is usually accurate or perhaps off by just one line. For some classes of errors the line numbers may not be very useful, because they delineate a potentially large declaration containing the error. If the error occurs in a file being loaded by `use`, the line numbers will refer to lines in the file being loaded.

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."

Printing. The structure `System.Print` contains several useful flags and functions with which you can control or redirect compiler error, diagnostic, and result messages. See PRINT(SYS). You can also control the depth of printing of large structured values.

Useful system flags. There are a number of useful system flags and variables, which are found in the structure `System.Control` and its substructures. These are documented in Section CONTROL(SYS). They can be used, for instance, to redefine the primary and secondary prompt strings, to control the printing of garbage collection messages, or to fine tune memory management.

Interacting with the operating system. The structure `System` and several of its substructures contain functions for such tasks as executing Unix commands from within **sml** and changing the current directory. These are documented in Chapter SYS.

Emacs support. The directory contrib/emacs contains packages supporting editing ML source code and interacting with **sml** under gnu emacs.

3 SML/NJ Language Notes

This section covers some (essentially) upward compatible extensions of Standard ML provided by SML/NJ, and lists other discrepancies between SML/NJ and the Definition of Standard ML. Use of the extensions will of course cause problems when porting your ML code to another implementation of Standard ML.

3.1 Hexadecimal integer literals

Integer constants can be expressed in hexadecimal notation. Such constants start with the digit zero followed by "`x`" followed a sequence of digits and the characters "`a-f`", "`A-F`". Examples are `0x0, 0xa3, ~0x2ff`.

3.2 Vector expressions and patterns

Vectors are homogeneous, immutable arrays (see Section VECTOR(BASE)). The vector expression `#[`$exp_0, exp_1, \ldots, exp_{n-1}$`]` (where $n \geq 0$) creates a vector of length n whose elements are the values of the corresponding subexpressions. As with other aggregate expressions, the element expressions are evaluated from left to right. Vectors may be pattern-matched by vector patterns of the form `#[`$pat_0, pat_1, \ldots, pat_{n-1}$`]`. Such a pattern will only match a vector value of the same length.

Vector expressions and vector patterns are more compact and efficient than lists, and are comparable in cost to records.

3.3 First-class continuations

A set of 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.

The use of **callcc** is described in Section CALLCC(BASE).

3.4 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 explanation of how they work, but you should try some experiments to become familiar with the behavior of weak polymorphism.

It is well known that 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 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

```
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 viewed 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 cancelled by application 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. That is, the strength of type variables in the signature should be no greater than that in the corresponding structure component.

Compatibility. *The Definition of Standard ML* describes a less powerful system for typechecking references. The *Definition's* "underscore" type variables `'_a`, `'_b`, etc. are equivalent to variables of strength 1, that is, `'1a`, `'1b`. For compatibility, SML/NJ accepts programs using the underscore notation; and, if `System.Control.weakUnderscore` is set to true, also prints weak type variables using the underscore notation.

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. 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 in that 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, as in the following 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
```

3.5 Higher-order Modules

The module system of Standard ML has supported *first-order* parametric modules in the form of *functors*. But there are occasions when one like to parameterize over functors as well as structures, which requires a truly higher-order module system (see, for instance, the powerset functor example in `doc/examples/powerset.sml`. SML/NJ now provides an experimental higher-order extension of the module system.

Parameterization over functors can be provided in a straightforward way by allowing functors to be components of structures. Syntactically this can be accomplished merely by allowing functor declarations inside of structure bodies, and by providing syntax for functor specifications in signatures. Functor specifications were already part of the module syntax of the Definition of Standard ML (Figure 8, p. 14), so we have implemented that syntax and added it to the *spec* class (Figure 7, p. 13). In addition, it is convenient to have a way of declaring functor signatures and some syntactic sugar for curried functor definitions and partial application of curried functors, so these have also been provided. This extension is an "upward-compatible" enrichment of the language that should break no existing programs.

Functors as structure components. In the extended language, a signature can contain a functor specification:

```
signature SIG =
sig
  type t
  val a : t
  functor F(X: sig type s
                   val b: s
               end) : sig val x : t * X.s end
end
```

To match such a signature, a structure is allowed to contain a functor declaration:

```
structure S : SIG =
struct
  type t = int
  val a = 3
  functor F(X: sig type s val b: s end) =
    struct val x = (a,X.b) end
end
```

This makes it possible higher-order functors by including a functor as a component of a parameter structure or of a result structure. The case of a functor parameter is illustrated by the following example.

```
signature MONOID =
sig
  type t
  val plus: t*t -> t
  val e: t
end;
(* functor signature declaration *)
funsig PROD (structure M: MONOID
             structure N: MONOID) = MONOID
functor Square(structure X: MONOID
               functor Prod: PROD): MONOID =
    Prod(structure M = X
         structure N = X);
```

Note that this example involves the definition of a *functor signature* `PROD`. Currently functor signature declarations take one of the following forms:

`funsig` *funid* (*strid*: *sigexp*) `=` *sigexp*
`funsig` *funid* (*specs*) `=` *sigexp*

This syntax is viewed as provisional and subject to change. Possible alternative notations (for the first form) are:

`funsig` *funid* `=` (*strid*: *sigexp*) *sigexp*
`funsig` *funid* `=` (*strid*: *sigexp*) `=>` *sigexp*

A common use of functors returning functors in their result is to approximate a curried functor with multiple parameters. Here is how one might define a curried monoid product functor:

```
functor CurriedProd (M: MONOID) =
struct
  functor Prod1 (N: MONOID) : MONOID =
    struct
      type t = M.t * N.t
      val e = (M.e, N.e)
      fun plus((m1,n1),(m2,n2))=(M.plus(m1,m2),N.plus(n1,n2))
    end;
end
```

This works, but the partial application of this functor is rather awkward because it requires the explicit creation of an intermediate structure:

```
structure IntMonoid =
struct
  type t = int
  val e = 0
  val plus = (op +): int*int -> int
end;
structure Temp = CurriedProd(IntMonoid);
functor ProdInt = Temp.Prod1;
```

To simplify the use of this sort of functor, some derived forms provide syntactic sugar for curried functor definition and partial application. Thus the above example can be written:

```
functor CurriedProd (M: MONOID) (N: MONOID) : MONOID =
struct
  type t = M.t * N.t
  val e = (M.e, N.e)
  fun plus((m1,n1),(m2,n2))=(M.plus(m1,m2),N.plus(n1,n2))
end;
functor ProdInt = CurriedProd(IntMonoid);
```

The syntax for curried forms of functor signature and functor declarations and for the corresponding partial applications can be summarized as follows:

`funsig` *funsigid* (par_1) ... (par_n) = *sigexp*
`functor` *funid* (par_1) ... (par_n) = *strexp*
`functor` *funid1* = *funid2* (arg_1) ... (arg_n)
`structure` *strid* = *funid* (arg_1) ... (arg_n)

where

par ::= *id*~:~ *sigexp*~|~ *specs*
arg ::= *strexp* | *dec*.

In the case of a partial application defining a functor, it is assumed that the *funid2* on the right hand side takes more than n arguments, while in the case of the structure declaration *funid* should take exactly n arguments. As a degenerate case where $n = 0$ we have identity functor declarations:

`functor` *funid1* = *funid2*

There is also a "let" form of functor expression:

fctexp ::= `let` *dec* `in` *fctexp* `end` which can only be used in functor definitions of the form:

functor *funid* = let *dec* in *fctexp* end.

The curried functor declaration

functor F (par_1) ... (par_n) = *strexp*

is a derived form that is translated into the following declaration

```
functor F (par_1) =
struct
  functor %fct% (par_2) ... (par_n) = strexp
end
```

and the declarations

```
structure S = F (arg_1) ... (arg_n)
functor G = F (arg_1) ... (arg_n)
```

are derived forms expanding into (respectively):

```
local
  structure %hidden% = F (arg_1)
in
  structure S = %hidden%.%fct% (arg_2) ... (arg_n)
end
```

and

```
local
  structure %hidden% = F (arg_1) ... (arg_n)
in
  functor G = %hidden%.%fct%
end
```

Currently there is no checking that a complete set of arguments is supplied when a curried functor is applied to define a structure, as illustrated by the following example:

```
functor Foo (X: sig type s end) (Y: sig type t end) =
struct type u = X.s * Y.t end
structure A = struct type s = int end
structure S = Foo (A)  (* Foo A yields a (useless) structure *)
functor G = Foo (A)    (* Foo A yields a functor *)
```

Of course, the structure `S` defined in this way is useless, since we cannot use the pseudo-identifier `%fct%` to select its functor component. Arity checking to prevent this sort of error will be added in a future release.

3.6 Discrepancies between SML/NJ and the Definition

The basic environment of SML of New Jersey is a large superset of the initial basis described in Appendices C and D of the Definition of Standard ML (Milner/Tofte/Harper) and differs from it in 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 "`@`" (list append) 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`" specifications in signatures have a more limited semantics than in the definition. A specification "`open A`" in a signature merely allows abbreviated names for components of the structure A in later specifications (*e.g.* `t` for `A.t`). It does not introduce any specifications into the signature. The `local` specification form is only partially implemented for compatibility, and a warning message is issued when it is used. Signatures using these constructs in the following style will behave as expected:

```
signature S =
sig
  specs
  local
    open A B ...
  in
    specs
  end
end
```

- The equality symbol "=" can be re-bound (though usually with a warning message). It is not a good idea to do so.
- The declaration construct "`val ... and rec ...`" is not permitted; the `rec` must immediately follow the `val`.
- Multiple specifications of a name (in the same name space) are not allowed in signatures, except that certain "compatible" respecifications of a type constructor are allowed to make `include` specifications more useful.

4 SML/NJ Compiler Notes

Program optimization 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);
```

These are 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. We 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, may improve optimization, and is just clean style.

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.

In some cases, you can improve performance by combining several compilation units into one. For example, after you have developed your program as a set of top-level structures, 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.

This technique will increase compilation time and compilation memory usage substantially, **and it is not guaranteed to improve execution time.**