FREE eBook

LEARNING

sml

Free unaffiliated eBook created from
Stack Overflow contributors.

#sml

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sml

It is an unofficial and free sml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sml

Remarks

This section provides an overview of what sml is, and why a developer might want to use it.

It should also mention any large subjects within sml, and link out to the related topics. Since the Documentation for sml is new, you may need to create initial versions of those related topics.

Examples

Installation

There is a dozen implementations of Standard ML. MLton produces very optimized code, but has no REPL. SML/NJ is the most widely used, but has slightly difficult error messages for learning purposes. Moscow ML and Poly/ML are easy to get started with, but don't support the .mlb package format. That isn't essential for getting started, though.

Here are instructions for installing each of SML/NJ, Moscow ML and Poly/ML divided by operating system.

On Windows

SML/NJ:

- Go to http://www.smlnj.org/dist/working/ and find the latest release, e.g. 110.80 Distribution Files.
- Scroll down and find the MS Windows Installer, e.g. smlnj-110.80.msi. Run the installer.
- You now have a REPL in e.g. `C:\Program Files (x86)\SML NJ\bin\sml.bat`.

Moscow ML:

- Go to http://mosml.org/ and click "Download Win. Installer". Run the installer.
- You now have a REPL in e.g. `C:\Program Files (x86)\mosml\bin\mosml.exe`.

Using Homebrew On MacOS

SML/NJ:

- Run `brew install smlnj` as your own user. Test REPL with `smlnj`.

Moscow ML:

- Go to http://mosml.org/ and click "Download PKG File". Run the installer.

- *Missing... Test REPL how? Is it in `$PATH` now?*

On Ubuntu / Debian Linux

SML/NJ:

- Run `sudo apt-get install smlnj` as the super user. Test REPL with `smlnj`.

Moscow ML:

- *(Ubuntu)* Add the PPA as the super user. Test REPL with `mosml`.

```
sudo add-apt-repository ppa:kflarsen/mosml
sudo apt-get update
sudo apt-get install mosml
```

Adding readline support

In order to be able to use the arrow keys to navigate lines that were previously typed into the REPL, most of the SML compilers can benefit from the program `rlwrap`. Using Homebrew on MacOS, install this by `brew install rlwrap`, and on Ubuntu / Debian Linux, install this by `sudo apt-get install rlwrap`. Then in the terminal, try the following:

```
alias mosml='rlwrap mosml -P full'
alias sml='rlwrap sml'
alias poly='rlwrap poly'
```

These aliases can be added to e.g. your `~/.bashrc` so they work by default.

The arrows key should now work better.

Read Getting started with sml online: https://riptutorial.com/sml/topic/6953/getting-started-with-sml

Chapter 2: Comments

Syntax

- (* opens a block comment
- *) closes a block comment
- (* and *) must be balanced in number

Examples

All comments are block comments

```
(***********************************************
*  All comments in SML are block comments
*  Block Comments begin with '(*'
*  Block Comments end with '*)'
*  (* Block Comments can be nested *)
*  The additional framing asterisks at the beginning
*  and end of this block comment are common to languages
*  of SML's vintage.
*  Likewise the asterisk at the start of each line
*  But this is solely a matter of style.
***********************************************)

val _ = print "Block comment example\n" (* line ending block comment *)
```

Nested Comments

```
(* The block comment syntax allows nested comments
(* whether or not this is a good thing is probably
a matter of personal opinion (*or coding standards*)*)*)

val _ = print "Nested comment example\n" (* line ending block comment *)
```

Read Comments online: https://riptutorial.com/sml/topic/6976/comments

Chapter 3: Interactive Programming using the REPL

Syntax

- Unlike source code files, the semicolon ';' is mandatory to terminate each expression in the REPL.

Examples

Starting the SMLNJ REPL

REPL stands for 'Read Evaluate Print Loop.' The REPL can be used to write and execute code one line at a time and is an alternative to writing code to a file and then compiling or interpreting the entire file before execution.

To start the SMLNJ REPL from a command prompt:

```
smluser> sml
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- 3+4;
val it = 7 : int
- (*a comment: press contrl-d to exit *)
smluser>
```

In the Bash and similar command shells, GNU readline functionality can be added to the SML REPL using the system command `rlwrap sml`.

```
smluser> rlwrap sml
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- 3+4;
val it = 7 : int
- (* pressing the up arrow recalls the previous input *)
- 3+4;
val it = 7 : int
-
smluser>
```

Using 'it'

All SML expressions return a value. The REPL stores the return value of the last evaluated expression. `it` provides the value of the last evaluated expression within the REPL.

```
smluser> sml
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- 3+4;
val it = 7 : int
```

```
- it;
val it = 7 : int
- it + 1;
val it = 8 : int
-

[1]+  Stopped                 sml
smluser>
```

Effectively, comments are not evaluated by the REPL and do not change the value of it.

```
smluser> sml
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- 3+4;
val it = 7 : int
- (* a comment *);
- it;
val it = 7 : int

[1]+  Stopped                 sml
smluser>
```

Read Interactive Programming using the REPL online:
https://riptutorial.com/sml/topic/6975/interactive-programming-using-the-repl

Chapter 4: Module System

Examples

Lazy evaluation

Standard ML doesn't have built-in support for lazy evaluation. Some implementations, notably SML/NJ, have nonstandard lazy evaluation primitives, but programs that use those primitives won't be portable. Lazy suspensions can also be implemented in a portable manner, using Standard ML's module system.

First we define an interface, or *signature*, for manipulating lazy suspensions:

```
signature LAZY =
sig
  type 'a lazy

  val pure : 'a -> 'a lazy
  val delay : ('a -> 'b) -> 'a -> 'b lazy
  val force : 'a lazy -> 'a

  exception Diverge

  val fix : ('a lazy -> 'a) -> 'a
end
```

This signature indicates that:

- The type constructor of lazy suspensions is *abstract* - its internal representation is hidden from (and irrelevant to) users.
- There are two ways to create a suspension: by directly wrapping its final result, and by delaying a function application.
- The only thing we can do with a suspension is force it. When a delayed suspension is forced for the first time, its result is memoized, so that the next time the result won't have to be recomputed.
- We can create self-referential values, where the self-reference goes through a suspension. This way we can create, for example, a logically infinite stream containing the same repeated element, as in the following Haskell snippet:

```
-- Haskell, not Standard ML!
xs :: [Int]
xs = 1 : xs
```

After defining the interface, we have to provide an actual implementation, also known as module or *structure*:

```
structure Lazy :> LAZY =
struct
```

```
  datatype 'a state
    = Pure of 'a
    | Except of exn
    | Delay of unit -> 'a

  type 'a lazy = 'a state ref

  fun pure x = ref (Pure x)
  fun delay f x = ref (Delay (fn _ => f x))
  fun compute f = Pure (f ()) handle e => Except e
  fun force r =
    case !r of
        Pure x => x
      | Except e => raise e
      | Delay f => (r := compute f; force r)

  exception Diverge

  fun fix f =
    let val r = ref (Except Diverge)
    in r := compute (fn _ => f r); force r end
end
```

This structure indicates that a suspension is internally represented as a mutable cell, whose internal state is one of the following:

- `Pure x`, if the suspension was already forced, and its final result is `x`.
- `Except e`, if the suspension was already forced, and an exception was thrown in the process.
- `Delay f`, if the suspension wasn't forced yet, and its final result can be obtained by evaluating `f ()`.

Furthermore, because we used *opaque ascription* (`:>`), the internal representation of the type of suspensions is hidden outside of the module.

Here's our new type of lazy suspensions in action:

```
infixr 5 :::
datatype 'a stream = NIL | ::: of 'a * 'a stream Lazy.lazy

(* An infinite stream of 1s, as in the Haskell example above *)
val xs = Lazy.fix (fn xs => 1 ::: xs)

(* Haskell's Data.List.unfoldr *)
fun unfoldr f x =
  case f x of
      NONE => NIL
    | SOME (x, y) => x ::: Lazy.delay (unfoldr f) y

(* Haskell's Prelude.iterate *)
fun iterate f x = x ::: Lazy.delay (iterate f o f) x

(* Two dummy suspensions *)
val foo = Lazy.pure 0
val bar = Lazy.pure 1

(* Illegal, foo and bar have type `int Lazy.lazy`,
```

```
 * whose internal representation as a mutable cell is hidden *)
val _ = (foo := !bar)
```

Read Module System online: https://riptutorial.com/sml/topic/7013/module-system

Chapter 5: Numeric Types

Syntax

- Real numbers must begin with one or more digits followed by a period followed by one or more digits.
- **~** is the operator to denote negative numbers
- **div** is the operator for integer division.
- **/** is the operator for real division.

Examples

Integer

Integer Basics

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
- 6;
val it = 6 : int
- ~6;
val it = ~6 : int
- 6 + ~6;
val it = 0 : int
```

Integer Division

```
- 6 div 3;
val it = 2 : int
- 6 div 4;
val it = 0 : int
- 3 div 6;
val it = 0 : int
```

Integer Value Bounds

Using Integer Basis Library Functions

```
- Int.maxInt;
val it = SOME 1073741823 : int option
- Int.minInt;
val it = SOME ~1073741824 : int option
```

Real

Real Number Basics

```
- 6.0;
val it = 6.0 : real
```

```
- ~6.0;
val it = ~6.0 : real
- 6.0 + ~6.0;
val it = 0.0 : real
- 6.0 / 3.0;
val it = 2.0 : real
- 4.0 / 6.0;
val it = 0.666666666667 : real
```

Real Value Bounds

Using Real Basis Library Functions

```
- Real.maxFinite;
val it = 1.79769313486E308 : real
- Real.minPos;
val it = 4.94065645841E~324 : real
- Real.minNormalPos;
val it = 2.22507385851E~308 : real
```

Infinity

```
- Real.posInf;
val it = inf : real
- Real.negInf;
val it = ~inf : real
```

Coercion of Real Values to Integers

Rounding

Values midway between two integers go toward the nearest even value.

```
- round(4.5);
val it = 4 : int
- round(3.5);
val it = 4 : int
```

Truncation

```
val it = 4 : int
- trunc(4.5);
val it = 4 : int
- trunc(3.5);
val it = 3 : int
```

Floor and Ceiling

```
- ceil(4.5);
val it = 5 : int
- floor(4.5);
val it = 4 : int
```

Arithmetic Operator Error with Mixed Numeric Types

Cannot add Integer and Real*

```
- 5 + 1.0;
stdIn:1.2-10.4 Error: operator and operand don't agree [overload conflict]
  operator domain: [+ ty] * [+ ty]
  operand:         [+ ty] * real
  in expression:
    5 + 1.0
```

Coersion of Integer Value to Real

```
- real(6);
val it = 6.0 : real
```

Read Numeric Types online: https://riptutorial.com/sml/topic/7010/numeric-types

Credits

S. No	Chapters	Contributors
1	Getting started with sml	4444, ben rudgers, Community, Simon Shine
2	Comments	ben rudgers
3	Interactive Programming using the REPL	ben rudgers, Nick
4	Module System	pyon
5	Numeric Types	ben rudgers, pyon