

Introduction To Standard ML



CS251 Programming Languages
Fall 2017
Lyn Turbak

Department of Computer Science
Wellesley College

The ML Programming Language



ML (Meta Language) was developed by Robin Milner in 1975 for specifying theorem provers. It since has evolved into a general purpose programming language.

Important features of ML:

- **static typing:** catches type errors at compile-time.
- **type reconstruction:** infers types so programmers don't have to write them explicitly
- **polymorphism:** functions and values can be parameterized over types (think Java generics, but much better).
- **function-oriented (functional):** encourages a composition-based style of programming and first-class functions
- **sum-of-products datatypes with pattern-matching:** simplifies the manipulation of tree-structured data

These features make ML an excellent language for mathematical calculation, data structure implementation, and programming language implementation.

Introduction to Standard ML. 2

ML Dialects

There are several different dialects of ML. The two we use at Wellesley are:

- **Standard ML (SML):** Version developed at AT&T Bell Labs. We'll use this in CS251. The particular implementation we'll use is Standard ML of New Jersey (SMLNJ):

<http://www.smlnj.org/>

- **Objective CAML:** Version developed at INRIA (France). We have sometimes used this in other Wellesley courses (esp. CS235).

These dialects differ in minor ways (e.g., syntactic conventions, library functions). See the following for a comparison:

<http://www.mpi-sws.mpg.de/~rossberg/sml-vs-ocaml.html>

Introduction to Standard ML. 3

Two ways to run sml

Way #1: Run sml on the wx Virtual box appliance from CS240 (see following slides)

Way #2: Run sml within a terminal window on the CS server = cs.wellesley.edu = tempest.

- Begin by connecting to your CS server account via ssh.
 - On a Mac, you can do this in your terminal window.
 - On a Windows PC, you'll need to use a terminal emulator like putty

```
[fturbak@Franklyns-MBP ~]$ ssh gdome@cs.wellesley.edu
gdome@cs.wellesley.edu's password:
Last login: Wed Oct 26 15:28:23 2016 from 149.130.142.18
```

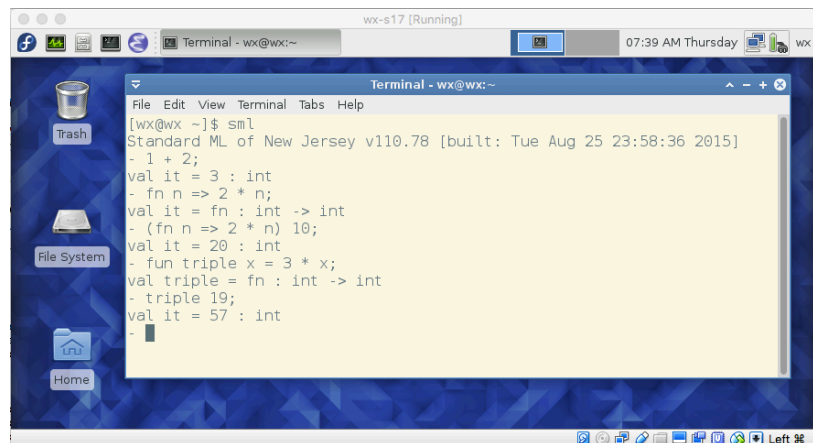
```
[gdome@tempest ~] which sml
/usr/local/smlnj/bin/sml
```

```
[gdome@tempest ~] sml
Standard ML of New Jersey v110.78 [built: Wed Jan 14 12:52:09 2015]
```

```
- 1 + 2;
val it = 3
```

Introduction to Standard ML. 4

SML and wx



We will use SML inside the wx Virtual Machine appliance. Details on how to install wx and SML within wx will be supplied.

For initial examples, it's easiest to run SML in a terminal window, as shown above. But we'll soon see (slides 18 – 19) running it in Emacs is **much** better!

Learning SML by Interactive Examples

Try out these examples. (Note: many answers are missing in these slides so you can predict them. See the solns slides for answers.)

```
[wx@wx ~] sml
Standard ML of New Jersey v110.78 [built: Wed Jan 14 12:52:09 2015]

- 1 + 2;
val it = 3 : int

- 3+4;
val it = 7 : int

- 5+6
= ;
val it = 11 : int

- 7
= +
= 8;
val it = 15 : int
```

Naming Values

```
- val a = 2 + 3;
val a = 5 : int

- a * a;
val it = 25 : int

- it + a;
val it = 30 : int
```

Negative Quirks

```
- 2 - 5;
val it = ~3 : int

- -17;
stdIn:60.1 Error: expression or pattern begins with infix
identifier "-"
stdIn:60.1-60.4 Error: operator and operand don't agree
[literal]
  operator domain: 'Z * 'Z
  operand:         int
  in expression:
    - 17

- ~17;
val it = ~17 : int

- 3 * ~1;
val it = ~3 : int
```

Division Quirks

```
- 7 / 2;
stdIn:1.1-1.6 Error: operator and operand don't agree
[literal]
  operator domain: real * real
  operand:         int * int
in expression:
  7 / 2

- 7.0 / 2.0;
val it = 3.5 : real

- 7 div 2; (* integer division *)
val it = 3 : int

(* For a description of all top-level operators, see:
http://www.standardml.org/Basis/top-level-chapter.html *)
```

Introduction to Standard ML 9

Simple Functions

```
- val inc = fn x => x + 1;
val inc = fn : int -> int (* SML figures out type! *)

- inc a;
val it = 6 : int

- fun dbl y = y * 2;
  (* Syntactic sugar for val dbl = fn y => y * 2 *)
val dbl = fn : int -> int

- dbl 5;
val it = 10 : int

- (fn x => x * 3) 10; (* Don't need to name function to use it *)
val it = 30 : int
```

Introduction to Standard ML 10

When Parentheses Matter

```
- dbl(5); (* parens are optional here *)
val it = 10 : int

- (dbl 5); (* parens are optional here *)
val it = 10 : int

- inc (dbl 5); (* parens for argument subexpressions are required! *)
val it = 11 : int

- (inc dbl) 5;
stdIn:1.2-2.2 Error: operator and operand don't agree [tycon mismatch]
operator domain: int
operand:         int -> int
in expression:
  inc dbl

- inc dbl 5; (* default left associativity for application *)
stdIn:22.1-22.10 Error: operator and operand don't agree [tycon
mismatch]
operator domain: int
operand:         int -> int
in expression:
  inc dbl
```

Introduction to Standard ML 11

Booleans

```
- 1 = 1;
val it = true : bool

- 1 > 2;
val it = false : bool

- (1 = 1) andalso (1 > 2);
val it = false : bool

- (1 = 1) orelse (1 = 2);
val it = true : bool

- (3 = 4) andalso (5 = (6 div 0)); (* short-circuit evaluation *)
val it = false : bool

- fun isEven n = (n mod 2) = 0;
val isEven = fn : int -> bool (* SML figures out type! *)

- isEven 17;
val it = false : bool

- isEven 6;
val it = true : bool
```

Introduction to Standard ML 12

Conditionals

```
- fun f n = if n > 10 then 2 * n else n * n;
val f = fn : int -> int

- f 20;
val it = 40 : int

- f 5;
val it = 25 : int
```

Introduction to Standard ML 13

Recursion

```
- fun fact n =
=   if n = 0 then
=     1
=   else
=     n * (fact (n - 1)); (* fun names have recursive scope *)
val fact = fn : int -> int
    (* simpler than Java definition b/c no explicit types! *)

- fact 5;
val it = 120 : int

- fact 12;
val it = 479001600 : int

- fact 13;
uncaught exception Overflow [overflow]
  raised at: <file stdIn>
    (* SML ints have limited size ☹ *)
```

Introduction to Standard ML 14

Easier to Put Your Code in a File

```
(* This is the contents of the file
   ~cs251/download/sml/mydefns.sml
   (* By the way, comments nest properly in SML! *)
   It defines integers a and b and functions named sq, hyp, and fact *)

val a = 2 + 3
val b = 2 * a

fun sq n = n * n (* squaring function *)

(* calculate hypotenuse of right triangle with sides a and b *)
fun hyp a b = Math.sqrt(Real.fromInt(sq a + sq b))

fun fact n = (* a recursive factorial function *)
  if n = 0 then
  1
  else
  n * (fact (n - 1))
```

- File is a sequence of value/function definitions.
- Definitions are **not** followed by semi-colons in files!
- There are **no continuation characters (equal signs)** for multiple-line definitions.

- Introduction to Standard ML 15

Using Code From a File

```
- Posix.FileSys.getcwd(); (* current working directory *)
val it = "/students/gdome" : string

- Posix.FileSys.chdir("/students/gdome/cs251/sml");
  (* change working directory *)
val it = () : unit

- Posix.FileSys.getcwd();
val it = "/students/gdome/cs251/sml" : string

- use "mydefns.sml"; (* load defns from file as if *)
[opening mydefns.sml] (* they were typed manually *)
val a = 5 : int
val b = 10 : int
val sq = fn : int -> int
val hyp = fn : int -> int -> real
val fact = fn : int -> intval
it = () : unit

- fact a
val it = 120 : int
```

Introduction to Standard ML 16

Another File Example

```
(* This is the contents of the file test-fact.sml *)
val fact_3 = fact 3
val fact_a = fact a
```

```
- use "test-fact.sml";
[opening test-fact.sml]
val fact_3 = 6 : int
val fact_a = 120 : int
val it = () : unit
```

Introduction to Standard ML 17

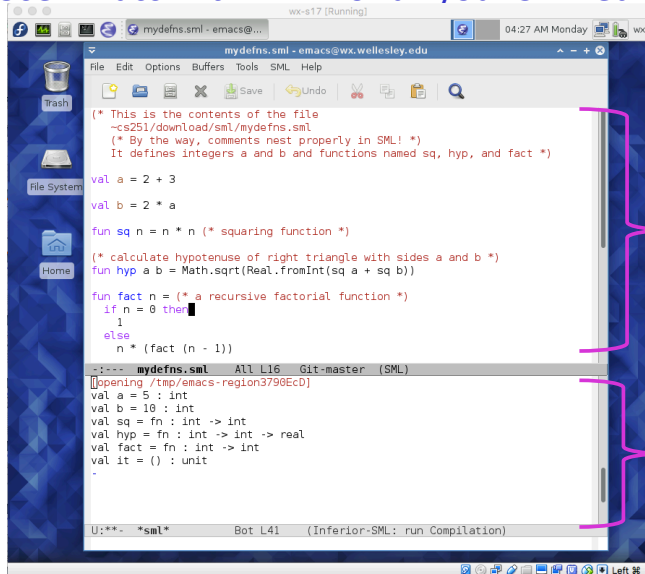
Nested File Uses

```
(* The contents of the file load-fact.sml *)
use "mydefns.sml"; (* semi-colons are required here *)
use "test-fact.sml";
```

```
- use "load-fact.sml";
[opening load-fact.sml]
[opening mydefns.sml]
val a = 5 : int
val b = 10 : int
val sq = fn : int -> int
val hyp = fn : int -> int -> real
val fact = fn : int -> intval
[opening test-fact.sml]
val fact_3 = 6 : int
val fact_a = 120 : int
val it = () : unit
val it = () : unit
```

Introduction to Standard ML 18

Use Emacs within wx for all your SML editing/testing

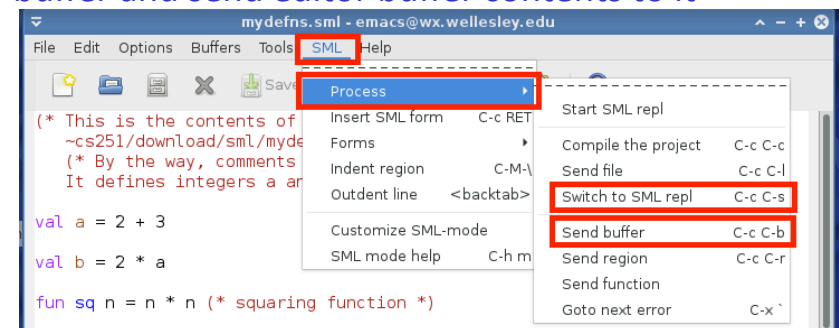


Emacs editor
buffer in
SML mode.
Edit your
SML code here.

sml interpreter
buffer.
Evaluate SML
expressions here.
Create this via
M-x sml or
C-c C-s
(see next slide).

Introduction to Standard ML 19

Use Emacs SML commands to start *sml* interpreter buffer and send editor buffer contents to it



SML>Process>Switch to SML repl (or C-c C-s) creates the *sml* interpreter buffer. This is just like the SML interpreter buffer in a terminal window, but it's in an Emacs buffer.

SML>Process>Send buffer (or C-c C-b) sends the contents of the SML editor buffer to the *sml* buffer. This is much more convenient than use for loading the contents of a file into the *sml* buffer.

Introduction to Standard ML 20

How to exit SML interpreter?

```
[wx@wx ~] sml
Standard ML of New Jersey v110.78
[built: Wed Jan 14 12:52:09 2015]
```

```
- 1 + 2;
val it = 3 : int
```

```
- ← Type Control-d at the SML prompt
```

```
[gdome@tempest ~]
```

Introduction to Standard ML 21

Your turn: fib

In an Emacs buffer, translate the following recursive Racket function into SML, and then test your SML fib function in the *sml* interpreter buffer.

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2))))))
```

```
fun fib n =
  if n < 2 then
    1
  else
    fib(n-1) + fib(n-2)
```

```
[opening /tmp/emacs-region12173z61]
val fib = fn : int -> int
val it = () : unit
- fib 8;
- val it = 34 : int
```

Introduction to Standard ML 22

Strings

```
- "foobar";
val it = "foobar" : string

- "foo" ^ "bar" ^ "baz";
val it = "foobarbaz" : string

- print ("baz" ^ "quux");
bazquuxval it = () : unit

- print ("baz" ^ "quux\n"); (* parens are essential here! *)
bazquux
val it = () : unit

- print "baz" ^ "quux\n";
stdIn:1.1-1.23 Error: operator and operand don't agree
[tycon mismatch]
operator domain: string * string
operand:          unit * string
in expression:
  print "baz" ^ "quux\n"
```

Introduction to Standard ML 23

Other String Operations

```
- String.size ("foo" ^ "bar");
val it = 6 : int

- String.substring ("abcdefg", 2, 3); (* string, start index, len *)
val it =          : string

("bar" < "foo", "bar" <= "foo", "bar" = "foo", "bar" > "foo");
val it = (true,true,false,false) : bool * bool * bool * bool

-(String.compare("bar", "foo"), String.compare("foo", "foo"),
 = String.compare("foo", "bar"));
val it = (LESS,EQUAL,GREATER) : order * order * order

- String.size;
val it = fn : string -> int

- String.substring;
val it = fn : string * int * int -> string

- String.compare;
val it = fn : string * string -> order

(* An API for all SMLNJ String operations can be found at:
http://www.standardml.org/Basis/string.html *)
```

Introduction to Standard ML 24

Characters

```
- #"a";
val it = #"a" : char

- String.sub ("foobar",0);
val it = #"f" : char

- String.sub ("foobar",5);
val it = #"r" : char

- String.sub ("foobar",6);
uncaught exception Subscript [subscript out of bounds]
  raised at: stdIn:17.1-17.11

- String.str #"a"; (* convert a char to a string *)
val it = "a" : string

- (String.str (String.sub ("ABCD",2))) ^ "S"
= ^ (Int.toString (112 + 123));
val it = "CS235" : string

- (1+2, 3=4, "foo" ^ "bar", String.sub("baz",2));
val it = (3,false, "foobar", #"z") : int * bool * string * char
```

Introduction to Standard ML 25

Tuples

```
- val tpl = (1 + 2, 3 < 4, 5 * 6, 7 = 8);
val tpl = (3,true,30,false) : int * bool * int * bool

- #1 tpl;
val it = 3: int

- #2 tpl;
val it = true : bool

(* In practice, *always* use pattern matching
  (see later slides) rather than #1, #2, etc. *)
- ((#1 tpl) + (#3 tpl), (#2 tpl) orelse (#4 tpl));
val it = (33,true) : int * bool
```

Introduction to Standard ML 26

Pattern-matching Tuple Function Arguments

```
- fun swap (x,y) = (y, x);
val swap = fn : 'a * 'b -> 'b * 'a
(* infers polymorphic type!
  'a and 'b stand for any two types. *)

- swap (1+2, 3=4);
val it = (false,3) : bool * int

- swap (swap (1+2, 3=4));
val it = (3,false) : int * bool

- swap ((1+2, 3=4), ("foo" ^ "bar", String.sub("baz",2)));
val it = (("foobar",#"z"),(3,false)) : (string * char) *
(int * bool)
```

Introduction to Standard ML 27

How to Pass Multiple Arguments

```
- fun avg1 (x, y) = (x + y) div 2; (* Approach 1: use pairs *)
val avg1 = fn : int * int -> int

- avg1 (10,20);
val it = 15 : int

- fun avg2 x = (fn y => (x + y) div 2); (* Approach 2: currying *)
val avg2 = fn : int -> int -> int

- avg2 10 20;
val it = 15 : int

- fun avg3 x y = (x + y) div 2; (* Syntactic sugar for currying *)
val avg3 = fn : int -> int -> int

- avg3 10 20;
val it = 15 : int

- app5 (avg3 15);
val it = 10 : int

- app5 (fn i => avg1(15,i));
val it = 10 : int
```

Introduction to Standard ML 28

Functions as Arguments

```
- fun app5 f = f 5;
val app5 = fn : (int -> 'a) -> 'a
(* infers polymorphic type!
   'a stands for "any type" *)

- app5 (fn x => x + 1);
val it = 6 : int

- fun dbl y = 2*y;
val dbl = fn : int -> int

- app5 dbl;
val it = 10 : int
```

We'll see later that functions can also be returned as results from other functions and stored in data structures, so functions are first-class in SML just as in Racket.

Your turn: translate these from Racket to SML

```
(define (sum-between lo hi)
  (if (> lo hi)
      lo
      (+ lo
         (sum-between (+ lo 1) hi))))

(sum-between 3 7)

(define (app-3-5 f) (f 3 5))

(define (make-linear a b)
  (lambda (x) (+ (* a x) b)))

((app-3-5 make-linear) 10)
```

```
(* .sml file editor buffer *)
fun sumBetween lo hi =
  (* can't use - in names! *)
  if lo > hi then
    0
  else
    lo + sumBetween (lo + 1) hi
(* parens matter! *)
fun app_3_5 f = f 3 5
fun makeLinear a b = fn x => a*x + b
```

```
(* *sml* interpreter buffer *)
sumBetween = fn : int -> int -> int
val app_3_5 = fn : (int -> int -> 'a) -> 'a
val makeLinear = fn : int -> int -> int -> int
val it = () : unit

- sumBetween 3 7;
val it = 25 : int

- ((app_3_5 makeLinear) 10);
val it = 35 : int
```

Function Composition

```
- val inc x = x + 1;
val inc = fn : int -> int

- fun dbl y = y * 2;
val dbl = fn : int -> int

- (inc o dbl) 10; (* SML builtin infix function composition *)
val it = 21 : int

- (dbl o inc) 10;
val it = 22 : int

- fun id x = x; (* we can define our own identity fcn *)
val id = fn : 'a -> 'a (* polymorphic type; compare to
   Java's public static <T> T id (T x) {return x;} *)

- (inc o id) 10;
val it = 11 : int

- (id o dbl) 10;
val it = 20 : int

- (inc o inc o inc o inc) 10;
val it = 14 : int
```

Iterating via Tail Recursion

```
(* This is the contents of the file step.sml *)

fun step (a,b) = (a+b, a*b)

fun stepUntil ((a,b), limit) = (* no looping constructs in ML; *)
  if a >= limit then (* use tail recursion instead! *)
    (a,b)
  else
    stepUntil (step(a,b), limit)

- use ("step.sml");
[opening step.sml]
val step = fn : int * int -> int * int
val stepUntil = fn : (int * int) * int -> int * int
val it = () : unit

- step (1,2);
val it = (3,2) : int * int

- step (step (1,2));
val it = (5,6) : int * int

- let val (x,y) = step (step (1,2)) in x*y end;
val it = 30 : int

- stepUntil ((1,2), 100);
val it = (371,13530) : int * int
```


Adding print statements

```
(* This is the contents of the file step-more.sml *)

fun printPair (a,b) =
  print ("(" ^ (Int.toString a) ^ ", "
        ^ (Int.toString b) ^ ")\n")

fun stepUntilPrint ((a,b), limit) =
  if a >= limit then
    (a,b)
  else
    (printPair (a,b); (* here, semicolon sequences expressions *)
     stepUntilPrint (step(a,b), limit))

- use ("step-more.sml");
[opening step-more.sml]
val printPair = fn : int * int -> unit
val stepUntilPrint = fn : (int * int) * int -> int * int
val it = () : unit

- stepUntilPrint ((1,2),100);
(1,2)
(3,2)
(5,6)
(11,30)
(41,330)
val it = (371,13530) : int * int
```

Introduction to Standard ML 33

Gotcha! Scope of Top-Level Names

```
- val a = 2 + 3;
val a = 5 : int

- val b = a * 2;
val b = 10 : int

- fun adda x = x + a; (* adda adds 5 *)
val adda = fn : int -> int

- adda 7;
val it = 12 : int

- adda b;
val it = 15 : int

- val a = 42; (* this is a different a from the previous one *)
val a = 42 : int

- b; (* ML values are immutable; nothing can change b's value *)
val it = 10 : int

- adda 7;
val it = 12 : int (* still uses the a where adda was defined *)
```

Introduction to Standard ML 34

Gotcha! Mutually Recursive Function Scope

```
(* This version of stepUntil DOES NOT WORK because it can only
`see` names declared *above* it, and the step function is
defined *below* it *)

fun stepUntil ((a,b), limit) = (* no looping constructs in ML; *)
  if a >= limit then           (* use tail recursion instead! *)
    (a,b)
  else
    stepUntil (step(a,b), limit)

fun step (a,b) = (a+b, a*b)
```

```
(* This version of stepUntil DOES WORK because it uses keyword
and in place of fun to define mutually recursive functions. *)

fun stepUntil ((a,b), limit) = (* no looping constructs in ML; *)
  if a >= limit then           (* use tail recursion instead! *)
    (a,b)
  else
    stepUntil (step(a,b), limit)

and step (a,b) = (a+b, a*b) (* Note keyword fun replaced by and *)
```

Introduction to Standard ML 35

Your turn: translate fib-iter to SML

```
(define (fib-iter n)
  (fib-tail n 0 0 1))

(define (fib-tail n i fib_i fib_i_plus_1)
  (if (= i n)
      fib_i
      (fib-tail n
                (+ i 1)
                fib_i_plus_1
                (+ fib_i fib_i_plus_1))))
```

```
fun fibIter n = fibTail n 0 0 1
and fibTail n i fib_i fib_i_plus_1 = (* use and not fun! *)
  if i = n then
    fib_i
  else
    fibTail n (i+1) fib_i_plus_1 (fib_i+fib_i_plus_1)
```

Introduction to Standard ML 36

Local Naming via `let`

`let` is used to define local names. Any such names “shadow” existing definitions from the surrounding scope.

```
let val a = 27 (* 1st let binding *)
    val b = 3  (* 2nd binding *)
    fun fact x = x + 2 (* 3rd binding *)
  in fact (a div b) (* let body *)
end; (* end terminates the let *)
val it = 11 : int
```

let-bound names are only visible in the body of the `let`.

```
- fact (a div b);
(* these are global names:
 * fact is factorial function.
 * a is 42
 * b is 10 *)
val it = 24 : int
```

Introduction to Standard ML 37

Pattern Matching with Tuples

```
val tp1 = (1 + 2, 3 < 4, 5 * 6, 7 = 8)
(* val tp1 = (3,true,30,false) : int * bool * int * bool *)
```

(* It is *very bad* SML style to use #1, #2, etc. to extract the components of a tuple. *)

```
val tp12 = ((#1 tp1) + (#3 tp1), (#2 tp1) orelse (#4 tp1));
(* val tp12 = (33,true) : int * bool *)
```

(* Instead can “deconstruct” tuples via pattern matching. *Always* do this rather than using #1, #2 etc. *)

```
val tp13 =
  let val (i1, b1, i2, b2) = tp1
      in (i1 + i2, b1 orelse b2)
  end
(* val tp13 = (33,true) : int * bool *)
```

Introduction to Standard ML 38

Local Functions in SML

Functions locally defined with `let` are often used in SML to improve program organization and name hiding, especially with tail recursive functions. For example:

```
fun fibIter n =
  let fun fibTail i fib_i fib_i_plus_1 =
        if i = n then (* "sees" n from outer definition *)
            fib_i
          else
            fibTail (i+1) fib_i_plus_1 (fib_i+fib_i_plus_1)
      in fibTail 0 0 1
  end
```

Introduction to Standard ML 39