

Getting Started with Typed Functional Programming Using Standard ML

Alley Stoughton

Spring 2025

Standard ML

Standard ML:

- is strongly typed, featuring type inference
- is statically scoped
- uses eager evaluation (but lazy evaluation can be simulated)
- is mostly functional
 - imperative features, but downplayed
 - data structures immutable, so sharing happens automatically
- has a powerful module language
- has moderately good libraries

Compilers

There are two main compilers available:

- Standard ML of New Jersey (SML/NJ):
 - interactive front end
 - excellent support for separate compilation using the Compilation Manager (CM)
 - generates heap images, which can be loaded into executables
- MLton:
 - whole program optimizing compiler
 - generates executables
 - development normally done using SML/NJ

Examples

These slides and the code for my examples—plus links to more resources on Standard ML—are available on the web at:

<https://alleystoughton.us/getting-started-typed-fp-sml>

Using SML as a Calculator

```
- 5 + 4;  
val it = 9 : int  
- if 3 + it < 12  
= then it mod 3  
= else it div 3;  
val it = 3 : int  
- [1, 2] @ [3, 4];  
val it = [1,2,3,4] : int list  
- 0 :: it;  
val it = [0,1,2,3,4] : int list  
- rev it;  
val it = [4,3,2,1,0] : int list  
- tl it;  
val it = [3,2,1,0] : int list  
- hd it;  
val it = 3 : int
```

Using SML as a Calculator

```
- (4 * 9, 5 < 7);  
val it = (36,true) : int * bool
```

Declarations and Local Declarations

```
- val x = 4 + 8;  
val x = 12 : int  
- val y = x * x;  
val y = 144 : int  
- let val x = x + y  
= in (x, 2 * x, 3 * x) end;  
val it = (156,312,468) : int * int * int  
- #2 it;  
val it = 312 : int
```

Function Definitions

```
- fun fact n =  
=       if n = 0  
=       then 1  
=       else n * fact(n - 1);  
val fact = fn : int -> int  
- fact 6;  
val it = 720 : int
```


Function Definitions and Pattern Matching

```
- fun fact n =  
=       case n of  
=       0 => 1  
=       | n => n * fact(n - 1);  
val fact = fn : int -> int  
- fact 7;  
val it = 5040 : int  
- fun fact 0 = 1  
=   | fact n = n * fact(n - 1);  
val fact = fn : int -> int  
- fact 8;  
val it = 40320 : int
```

Tail Recursion

```
- fun fact n =  
=       let fun fct(0, m) = m  
=       | fct(n, m) = fct(n - 1, n * m)  
=       in fct(n, 1) end;  
val fact = fn : int -> int  
- fact 6;  
val it = 720 : int
```

Compilers for functional programming languages translate *tail recursion* into loops, not allocating stack frames.

Polymorphism and List Processing Functions

```
- fun rev xs =  
=       if null xs  
=       then nil  
=       else rev(tl xs) @ [hd xs];  
val rev = fn : 'a list -> 'a list  
- rev[1, 3, 5, 7];  
val it = [7,5,3,1] : int list  
- fun rev nil      = nil  
=   | rev (x :: xs) = rev xs @ [x];  
val rev = fn : 'a list -> 'a list  
- rev[1, 3, 5, 7];  
val it = [7,5,3,1] : int list
```

List Processing Functions and Tail Recursion

```
- fun rev xs =  
=       let fun rv(nil,      ys) = ys  
=         | rv(x :: xs, ys) = rv(xs, x :: ys)  
=         in rv(xs, nil) end;  
val rev = fn : 'a list -> 'a list  
- rev[1, 3, 5, 7];  
val it = [7,5,3,1] : int list
```

Anonymous and Higher-order Functions

```
- fn x => x + 1;  
val it = fn : int -> int  
- it(3 + 4);  
val it = 8 : int  
- map;  
val it = fn : ('a -> 'b) -> 'a list -> 'b list  
- val f = map (fn x => x + 1);  
val f = fn : int list -> int list  
- f [1, 3, 5];  
val it = [2,4,6] : int list  
- map (fn x => x + 1) [2, 7];  
val it = [3,8] : int list
```

Anonymous and Higher-order Functions

```
- List.exists;  
[autoloading]  
[autoloading done]  
val it = fn : ('a -> bool) -> 'a list -> bool  
- List.exists  
= (fn x => x mod 2 = 0)  
= [1, 2, 3, 4, 5, 6, 7];  
val it = true : bool  
- List.filter;  
val it = fn : ('a -> bool) -> 'a list -> 'a list  
- List.filter  
= (fn x => x mod 2 = 0)  
= [1, 2, 3, 4, 5, 6, 7];  
val it = [2,4,6] : int list
```

Exceptions

```
- exception Negative;
exception Negative
- fun fact n =
=       if n < 0
=       then raise Negative
=       else if n = 0
=       then 1
=       else n * fact(n - 1);
val fact = fn : int -> int
- fact 6;
val it = 720 : int
- fact ~2;

uncaught exception Negative
  raised at: stdIn:66.20-66.28
- fact ~2 handle Negative => 0;
val it = 0 : int
```

Option Types

```
- NONE;  
val it = NONE : 'a option  
- SOME 5;  
val it = SOME 5 : int option  
- SOME true;  
val it = SOME true : bool option  
- valOf(SOME false);  
val it = false : bool  
- valOf NONE;  
stdIn:76.1-76.11 Warning: type vars not generalized  
because of  
    value restriction are instantiated to dummy types  
(X1,X2,...)  
  
uncaught exception Option  
    raised at: smlnj/init/pre-perv.sml:25.28-25.34
```


Option Types

```
- fun firstPos(f, ys) =  
=       let fun first(_, nil)      = NONE  
=         | first(i, y :: ys) =  
=           if f y  
=             then SOME i  
=             else first(i + 1, ys)  
=       in first(0, ys) end;  
val firstPos = fn  
  : ('a -> bool) * 'a list -> int option  
- firstPos(fn x => x = 4, [1, 3, 4, 5, 4, 7]);  
val it = SOME 2 : int option  
- firstPos(fn x => x > 7, [1, 3, 2, 7]);  
val it = NONE : int option
```

Datatypes

```
- datatype tree =  
=           Leaf of int  
=           | Node of bool * tree * tree;  
datatype tree  
  = Leaf of int  
  | Node of bool * tree * tree  
- val tr =  
=       Node(false,  
=       Node(true, Leaf 0, Leaf 1),  
=       Leaf 2);  
val tr =  
  Node (false,Node (true,Leaf 0,Leaf 1),Leaf 2)  
  : tree  
- fun size(Leaf _) = 1  
=   | size(Node(_, tr1, tr2)) =  
=   1 + size tr1 + size tr2;  
val size = fn : tree -> int  
- size tr;  
val it = 5 : int
```

Extended Example: Generating Primes

Let's consider the problem of generating the first n prime numbers.

The key to generating primes semi-efficiently is this fact:

Suppose $n \in \mathbb{N}$ is at least 2. Then n is prime iff there is no $m \in \mathbb{N}$ such that

- $m < n$,
- n is divisible by m , and
- m is prime.

This holds because every natural number $n \geq 2$ can be expressed (uniquely) as a product of prime numbers (assuming n is not prime, these prime numbers will be $< n$).

Extended Example: Generating Primes

So to test whether $n \geq 2$ is prime, we can work through the prime numbers smaller than n , from *smallest* to *largest*, *rejecting* n as soon as we find a divisor of n .

Furthermore, as soon as we get to a prime m such that $n < m * m$, we can stop and *accept* n , because if n had a prime divisor p such that $m \leq p < n$, then it would also have a prime divisor less-than m , and so would already have been rejected.

To avoid possible integer overflow, we can use the equivalent test $n \text{ div } m < m$, where div is integer division.

Primes in C

```
void gen_primes(int n, int *primes) {
    int i, j, m; int next = 2; /* next candidate */
    for (i = 0; i < n; i++) {
        int found = 0;
        while (!found) {
            for (j = 0; j < i; j++) {
                m = primes[j];
                if (next % m == 0) { break; }
                else if (next / m < m) { found = 1; break; }
            }
            if (found) { }
            else if (j == i) { found = 1; }
            else if (next == INT_MAX) { integer_overflow(); }
            else { next++; }
        }
        primes[i] = next++;
    }
}
```

Primes in SML: I

```
fun next(ms, l) =  
    if List.exists (fn m => l mod m = 0) ms  
    then next(ms, l + 1)  
    else l  
  
fun prs 0 = nil  
  | prs 1 = [2]  
  | prs n =  
      let val ms = prs(n - 1)  
      in next(ms, hd ms + 1) :: ms end  
  
fun primes n = rev(prs n)
```

The preceding primes are checked from largest to smallest, delaying ruling out a candidate as prime. `prs` is not tail recursive.

Primes in SML: II

```
fun next(ms, l) =  
    if List.exists (fn m => l mod m = 0) ms  
    then next(ms, l + 1)  
    else l  
  
fun prs(n, i, ms) =  
    if i = n  
    then rev ms  
    else prs(n, i + 1, next(ms, hd ms + 1) :: ms)  
  
fun primes n = if n = 0 then nil else prs(n, 1, [2])
```

`prs` is tail recursive, but the preceding primes are still checked from largest to smallest.

Primes in SML: III

```
fun divisible(_, nil)      = false
  | divisible(l, m :: ms) =
    l mod m = 0 orelse
    (m * m < l handle _ => false) andalso
    divisible(l, ms)

fun next(ms, l) =
  if divisible(l, ms)
  then next(ms, l + 1)
  else l
```


Primes in SML: III

```
fun prs(n, i, ms, m) =  
    if i = n  
    then ms  
    else let val k = next(ms, m + 1)  
          in prs(n, i + 1, ms @ [k], k) end  
  
fun primes n = if n = 0 then nil else prs(n, 1, [2], 2)
```

List concatenation has to rebuild the left argument.

Primes in SML: IV

```
fun divisible(_, nil)      = false
  | divisible(l, m :: ms) =
    l mod m = 0 orelse
    (m * m < l handle _ => false) andalso
    divisible(l, ms)
```

Primes in SML: IV

```
fun next(ms, p, ls, k) =  
    let val (ms, p, ls) =  
        if null ls orelse p = NONE orelse  
            valOf p >= k  
        then (ms, p, ls)  
        else (ms @ rev ls,  
             SOME(hd ls * hd ls)  
             handle _ => NONE,  
             nil)  
    in if divisible(k, ms)  
        then next(ms, p, ls, k + 1)  
        else (ms, p, ls, k)  
    end
```

The reorganization of `ms/ls` only happens rarely; e.g., when generating the first 5,000,000 primes, the reorganization only happens five times (when `k` is 5, 10, 50, 2210 or 4870850).

Primes in SML: IV

```
fun prs(n, i, ms, p, ls, k) =  
    if i = n  
    then ms @ rev ls  
    else let val (ms, p, ls, k) = next(ms, p, ls, k + 1)  
          in prs(n, i + 1, ms, p, k :: ls, k) end  
  
fun primes n =  
    if n = 0 then nil else prs(n, 1, [2], SOME 4, [], 2)
```

Primes in SML: V

```
signature PRIMES =  
sig  
  
val primes : int -> int list  
  
end;
```

Primes in SML: V

```
structure Primes :> PRIMES =  
struct  
  
  fun divisible(_, nil)      = false  
    | divisible(l, m :: ms) =  
      l mod m = 0 orelse  
      (m * m < l handle _ => false) andalso  
      divisible(l, ms)
```

Primes in SML: V

```
fun next(ms, p, ls, k) =  
    let val (ms, p, ls) =  
        if null ls orelse p = NONE orelse  
            valOf p >= k  
        then (ms, p, ls)  
        else (ms @ rev ls,  
              SOME(hd ls * hd ls)  
                handle _ => NONE,  
              nil)  
    in if divisible(k, ms)  
        then next(ms, p, ls, k + 1)  
        else (ms, p, ls, k)  
    end
```

Primes in SML: V

```
fun prs(n, i, ms, p, ls, k) =  
    if i = n  
    then ms @ rev ls  
    else let val (ms, p, ls, k) = next(ms, p, ls, k + 1)  
          in prs(n, i + 1, ms, p, k :: ls, k) end  
  
fun primes n =  
    if n = 0 then nil else prs(n, 1, [2], SOME 4, [], 2)  
  
end;
```

There are also supporting structures for getting and processing the command line argument (the number of primes wanted), and for printing the results.

Comparison Generating First 5,000,000 Primes

```
$ time primes-clang 5000000 > /tmp/primes-clang.txt
```

```
real    0m4.165s
```

```
user    0m3.945s
```

```
sys     0m0.047s
```

```
$ time primes-smlnj 5000000 > /tmp/primes-smlnj.txt
```

```
real    0m23.506s
```

```
user    0m21.753s
```

```
sys     0m1.713s
```

```
$ time primes-mlton 5000000 > /tmp/primes-mlton.txt
```

```
real    0m7.425s
```

```
user    0m7.030s
```

```
sys     0m0.168s
```

Comparison Generating First 5,000,000 Primes

```
$ cmp /tmp/primes-clang.txt /tmp/primes-smlnj.txt
$ cmp /tmp/primes-smlnj.txt /tmp/primes-mlton.txt
$ wc -l /tmp/primes-mlton.txt
5000000 /tmp/primes-mlton.txt
$ tail /tmp/primes-mlton.txt
86027987
86027999
86028011
86028037
86028049
86028053
86028097
86028101
86028113
86028121
```

Primes in SML: VI

```
signature PRIMES =  
sig  
  
  type state  
  
  val init : state  
  
  val next : state -> int * state  
  
end;
```

Primes in SML: VI

```
structure Primes :> PRIMES =  
struct  
  
  type state = int list * int option * int list * int  
  
  val init = ([2], SOME 4, [], 2) : state  
  
  fun divisible(_, nil)      = false  
    | divisible(l, m :: ms) =  
      l mod m = 0 orelse  
      (m * m < l handle _ => false) andalso  
      divisible(l, ms)
```

Primes in SML: VI

```
fun nxt(ms, p, ls, k) =
  let val (ms, p, ls) =
    if null ls orelse p = NONE orelse
      valOf p >= k
    then (ms, p, ls)
    else (ms @ rev ls,
          SOME(hd ls * hd ls) handle _ => NONE,
          nil)
  in if divisible(k, ms)
    then nxt(ms, p, ls, k + 1)
    else (ms, p, ls, k)
  end
fun next ((ms, p, ls, k) : state) : int * state =
  (k,
   let val (ms, p, ls, k) = nxt(ms, p, ls, k + 1)
   in (ms, p, k :: ls, k) end)

end;
```

Primes in SML: VI

```
- val (n, st) = Primes.next Primes.init;  
val n = 2 : int  
val st = - : Primes.state  
- val (n, st) = Primes.next st;  
val n = 3 : int  
val st = - : Primes.state  
- val (n, st) = Primes.next st;  
val n = 5 : int  
val st = - : Primes.state  
- val (n, st) = Primes.next st;  
val n = 7 : int  
val st = - : Primes.state  
- val (n, st) = Primes.next st;  
val n = 11 : int  
val st = - : Primes.state
```

Streams

Can we create an infinite list (stream) of all (until we have integer overflow) primes like this?

```
fun g state =  
    let (x, state) = Primes.next state  
    in x :: g state end;  
val primes = g Primes.init;
```

No, only with *lazy evaluation*, where infinite streams can be created, and the part of a stream that is visited is *memoized*.

We can simulate lazy evaluation in SML using *thunks* and *references*.

Suspensions

```
signature SUSP =  
sig  
  
  type 'a susp  
  
  val delay : (unit -> 'a) -> 'a susp  
  
  val force : 'a susp -> 'a  
  
end
```

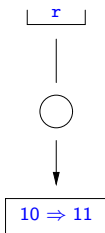
The only value of type `unit` is `()`.

References

```
type 'a ref  
  
val ref : 'a -> 'a ref  
val !    : 'a ref -> 'a  
val :=   : 'a ref * 'a -> unit
```

References

```
- val r = ref 10;  
val r = ref 10 : int ref  
- !r;  
val it = 10 : int  
- r := !r + 1;  
val it = () : unit  
- !r;  
val it = 11 : int
```



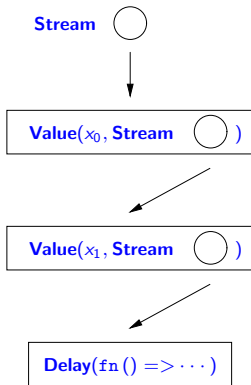
Suspensions

```
structure Susp :> SUSP =  
  struct  
  
    datatype 'a delay = Value of 'a  
                      | Delay of unit -> 'a  
  
    type 'a susp = 'a delay ref  
  
    fun delay f = ref(Delay f)  
  
    fun force(ref(Value x))      = x  
      | force(r as ref(Delay f)) =  
        let val x = f()  
        in r := Value x; x end  
  
  end;
```

Streams

```
signature STREAM =  
sig  
  
  type 'a stream  
  
  val make : 'a * ('a -> 'b * 'a) -> 'b stream  
  
  val get : 'a stream -> 'a * 'a stream  
  
  val takeToList : 'a stream * int -> 'a list  
  
  val drop : 'a stream * int -> 'a stream  
  
  val rangeToList : 'a stream * int * int -> 'a list  
  
end;
```

Streams



Streams

```
structure Stream :> STREAM =  
struct  
  
  datatype 'a stream = Stream of ('a * 'a stream)Susp.susp  
  
  fun make(state, f) =  
    let fun g state =  
          Stream  
            (Susp.delay  
              (fn () =>  
                let val (x, state) = f state  
                in (x, g state) end))  
        in g state end  
  
  fun get(Stream x) = Susp.force x
```

Streams

```
fun takeToList(stm, n) =  
  let fun tke(stm, n, xs) =  
        if n <= 0  
        then rev xs  
        else let val (x, stm) = get stm  
              in tke(stm, n - 1, x :: xs) end  
  in tke(stm, n, nil) end  
  
fun drop(stm, n) =  
  if n <= 0  
  then stm  
  else let val (_, stm) = get stm  
        in drop(stm, n - 1) end
```

Streams

```
fun rangeToList(stm, n, m) =  
    if n <= 0 orelse m < n  
    then nil  
    else takeToList(drop(stm, n - 1), m - n + 1)  
  
end;
```


Primes in SML: VI

```
- val primes = Stream.make(Primes.init, Primes.next);  
val primes = - : int Stream.stream  
- val stm = primes;  
val stm = - : int Stream.stream  
- val (x, stm) = Stream.get stm;  
val x = 2 : int  
val stm = - : int Stream.stream  
- val (x, stm) = Stream.get stm;  
val x = 3 : int  
val stm = - : int Stream.stream  
- val (x, stm) = Stream.get stm;  
val x = 5 : int  
val stm = - : int Stream.stream  
- val (x, _) = Stream.get primes;  
val x = 2 : int
```

Primes in SML: VI

```
- Stream.rangeToList(primes, 1, 10);  
val it = [2,3,5,7,11,13,17,19,23,29] : int list  
- Stream.rangeToList(primes, 50000, 50010);  
val it =  
  [611953,611957,611969,611977,611993,611999,612011,  
   612023,612037,612041,612043] : int list  
- Stream.rangeToList(primes, 40000, 40010);  
val it =  
  [479909,479939,479951,479953,479957,479971,480013,  
   480017,480019,480023,480043] : int list
```

Primes in SML: VI

Twin primes are primes that are separated by 2, like 3/5, 5/7 and 11/13. We can write a *stream transformer* that turns the stream of all primes into the stream of all twin pairs:

```
signature TWINS =  
sig
```

```
val twins : int Stream.stream -> (int * int) Stream.stream  
  
end;
```

Primes in SML: VI

```
structure Twins :> TWINS =  
struct  
  
  fun twins stm =  
    let val init : int * int Stream.stream =  
        Stream.get stm  
  
        fun next (n, stm) =  
          let val (m, stm) = Stream.get stm  
          in if m = n + 2  
             then ((n, m), (m, stm))  
             else next (m, stm)  
          end  
        in Stream.make(init, next) end  
  
end;
```

Primes in SML: VI

```
- val twins = Twins.twins primes;  
val twins = - : (int * int) Stream.stream  
- Stream.rangeToList(twins, 1, 10);  
val it =  
  [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),  
    (59,61),(71,73),(101,103),(107,109)]  
  : (int * int) list  
- Stream.rangeToList(twins, 10000, 10010);  
val it =  
  [(1260989,1260991),(1261079,1261081),  
    (1261259,1261261),(1261487,1261489),  
    (1261697,1261699),(1261829,1261831),  
    (1261889,1261891),(1262081,1262083),  
    (1262099,1262101),(1262291,1262293),  
    (1262621,1262623)] : (int * int) list
```