Getting Started with Typed Functional Programming Using Standard ML

Alley Stoughton

Summer 2019
Standard ML:


Standard ML

Standard ML:

- is strongly typed, featuring type inference
Standard ML

Standard ML:

- is strongly typed, featuring type inference
- is statically scoped
Standard ML

Standard ML:

• is strongly typed, featuring type inference
• is statically scoped
• uses eager evaluation (but lazy evaluation can be simulated)
Standard ML:

- is strongly typed, featuring type inference
- is statically scoped
- uses eager evaluation (but lazy evaluation can be simulated)
- is mostly functional
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
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
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
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:
Compilers

There are two main compilers available:

- Standard ML of New Jersey (SML/NJ):
Compilers

There are two main compilers available:

- Standard ML of New Jersey (SML/NJ):
  - interactive front end
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)
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
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:
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
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
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 examples I present—plus some more resources on Standard ML—are available on the web at:

https://alleystoughton.us/getting-started-typed-fp
Using SML as a Calculator
Using SML as a Calculator

- 5 + 4;
Using SML as a Calculator

- 5 + 4;
val it = 9 : int
-
Using SML as a Calculator

- 5 + 4;
  val it = 9 : int
- if 3 + it < 12
Using SML as a Calculator

- 5 + 4;

val it = 9 : int

- if 3 + it < 12

=
Using SML as a Calculator

- 5 + 4;
val it = 9 : int
- if 3 + it < 12
= then it mod 3
Using SML as a Calculator

- 5 + 4;
val it = 9 : int
- if 3 + it < 12
  = then it mod 3
  =
Using SML as a Calculator

- 5 + 4;
val it = 9 : int
- if 3 + it < 12
  = then it mod 3
  = else it div 3;
Using SML as a Calculator

\[-5 + 4;\]
\[\text{val it = 9 : int}\]
\[-\text{if 3 + it < 12}\]
\[= \text{then it mod 3}\]
\[= \text{else it div 3};\]
\[\text{val it = 3 : int}\]
\[-\]
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];
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
-
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;
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
-
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;
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
-
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;
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`
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;
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
Using SML as a Calculator

- (4 * 9, 5 < 7);
Using SML as a Calculator

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

- val x = 4 + 8;
Declarations and Local Declarations

- val x = 4 + 8;
val x = 12 : int
-
Declarations and Local Declarations

- val x = 4 + 8;
val x = 12 : int
- val y = x * x;
Declarations and Local Declarations

- val x = 4 + 8;
  val x = 12 : int
- val y = x * x;
  val y = 144 : int
 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
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
  =
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;
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
-
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 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
Function Definitions

- fun fact n =
Function Definitions

- fun fact n =
  =
  =
Function Definitions

- fun fact n =
  = if n = 0
Function Definitions

- fun fact n =
  = if n = 0
  =
Function Definitions

- fun fact n =
=     if n = 0
=     then 1
Function Definitions

- fun fact n =
  = if n = 0
  = then 1
  =
Function Definitions

- fun fact n =
  = if n = 0
  = then 1
  = else n * fact(n - 1);
Function Definitions

- fun fact n =
  = if n = 0
  =  then 1
  = else n * fact(n - 1);
val fact = fn : int -> int
-
- fun fact n = 
  = if n = 0
  = then 1
  = else n * fact(n - 1);
val fact = fn : int -> int
- fact 6;
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
Function Definitions and Pattern Matching

- fun fact n =
Function Definitions and Pattern Matching

- fun fact n =
  

Function Definitions and Pattern Matching

- fun fact n =
  = case n of


Function Definitions and Pattern Matching

- fun fact n =
  = case n of
Function Definitions and Pattern Matching

- fun fact n =
  = case n of
  = 0 => 1
Function Definitions and Pattern Matching

- fun fact n =
  = case n of
  = 0 => 1
  =
Function Definitions and Pattern Matching

- fun fact n =
  case n of
  | 0 => 1
  | n => n * fact(n - 1);
Function Definitions and Pattern Matching

- fun fact n =
  = case n of
  = 0 => 1
  = | n => n * fact(n - 1);  
val fact = fn : int -> 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;
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
-
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
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  
  =
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);
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
-
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;
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

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

```
- fun fact n =
```

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

- fun fact n =

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

- fun fact n =
  = let fun fct(0, m) = m

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

- fun fact n =
  =   let fun fct(0, m) = m
  =

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

- fun fact n =
  = let fun fct(0, m) = m
  = | fct(n, m) = fct(n - 1, n * m)

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

- fun fact n =
  = let fun fct(0, m) = m
  =       | fct(n, m) = fct(n - 1, n * m)
  =

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

fun fact n = let fun fct(0, m) = m
= | fct(n, m) = fct(n - 1, n * m)
= in fct(n, 1) end;

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

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

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

```lisp
- 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
Polymorphism and List Processing Functions

- fun rev xs =
Polymorphism and List Processing Functions

- fun rev xs =
  =
Polymorphism and List Processing Functions

- fun rev xs =
  = if null xs
Polymorphism and List Processing Functions

- fun rev xs =
  = if null xs
  =

11 / 42
Polymorphism and List Processing Functions

- fun rev xs =
  = if null xs
  = then nil
Polymorphism and List Processing Functions

- fun rev xs =
  = if null xs
  = then nil
  =
Polymorphism and List Processing Functions

- fun rev xs =
  = if null xs
  = then nil
  = else rev(tl xs) @ [hd xs];
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
-
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];
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
-
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
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
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];
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`  
-
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];
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
List Processing Functions and Tail Recursion

- fun rev xs =
List Processing Functions and Tail Recursion

- fun rev xs =
  =
List Processing Functions and Tail Recursion

- fun rev xs =
  = let fun rv(nil, ys) = ys
List Processing Functions and Tail Recursion

- fun rev xs =
  =     let fun rv(nil,  ys) = ys
  =
List Processing Functions and Tail Recursion

- fun rev xs =
  = let fun rv(nil, ys) = ys
  = | rv(x :: xs, ys) = rv(xs, x :: ys)
List Processing Functions and Tail Recursion

- fun rev xs =
  = let fun rv(nil, ys) = ys
  = | rv(x :: xs, ys) = rv(xs, x :: ys)
  =
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;
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
-
fun rev xs = 
  let fun rv(nil, ys) = ys
  in 
    | rv(x :: xs, ys) = rv(xs, x :: ys)
  end;

val rev = fn : 'a list -> 'a list 
- rev[1, 3, 5, 7];

List Processing Functions and Tail Recursion
List Processing Functions and Tail Recursion

- fun rev xs =
  = let fun rv(nil, ys) = 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
Anonymous and Higher-order Functions

- \texttt{fn x => x + 1;}

Anonymous and Higher-order Functions

- fn x => x + 1;
val it = fn : int -> int
Anonymous and Higher-order Functions

- fn x => x + 1;
val it = fn : int -> int
- it(3 + 4);
Anonymous and Higher-order Functions

- fn x => x + 1;
  val it = fn : int -> int
- it(3 + 4);
  val it = 8 : int
-
Anonymous and Higher-order Functions

- fn x => x + 1;
val it = fn : int -> int
- it(3 + 4);
val it = 8 : int
- map;
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
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
- map (fn x => x + 1) [1, 3, 5];
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
- map (fn x => x + 1) [1, 3, 5];
  val it = [2, 4, 6] : int list
Anonymous and Higher-order Functions

-
Anonymous and Higher-order Functions

- List.exists;
Anonymous and Higher-order Functions

- List.exists;
  [autoloading]
  [autoloading done]
val it = fn : ('a -> bool) -> 'a list -> bool
-
Anonymous and Higher-order Functions

- List.exists;
  [autoloading]
  [autoloading done]
val it = fn : ('a -> bool) -> 'a list -> bool
- List.exists
Anonymous and Higher-order Functions

- List.exists;
  [autoloading]
  [autoloading done]
  val it = fn : ('a -> bool) -> 'a list -> bool
- List.exists
  =
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)
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)
  =
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];
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
-
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;
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
  -
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
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
  =
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)
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)
  =
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];
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
Option Types
Option Types

- NONE;
Option Types

- NONE;
val it = NONE : 'a option
-
Option Types

- NONE;
val it = NONE : 'a option
- SOME 5;
Option Types

- NONE;
val it = NONE : 'a option
- SOME 5;
val it = SOME 5 : int option
-
Option Types

- NONE;
val it = NONE : 'a option
- SOME 5;
val it = SOME 5 : int option
- SOME true;
Option Types

- NONE;
val it = NONE : 'a option
- SOME 5;
val it = SOME 5 : int option
- SOME true;
val it = SOME true : bool option
-
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);
- 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
Option Types
Option Types

- fun firstPos(f, ys) =
Option Types

- fun firstPos(f, ys) =
Option Types

- fun firstPos(f, ys) =
  = let fun first(_, nil) = NONE
Option Types

- fun firstPos(f, ys) =
  = let fun first(_, nil) = NONE
  =
Option Types

- fun firstPos(f, ys) =
  = let fun first(_, nil) = NONE
  = | first(i, y :: ys) =
Option Types

- fun firstPos(f, ys) =
  = let fun first(_, nil) = NONE
  = | first(i, y :: ys) =
  =
Option Types

- fun firstPos(f, ys) =
  let fun first(_, nil) = NONE
  | first(i, y :: ys) = if f y
Option Types

- fun firstPos(f, ys) =
  let fun first(_, nil) = NONE
  | first(i, y :: ys) =
  |     if f y
  =
- fun firstPos(f, ys) =

  let fun first(_, nil) = NONE
  in
  first(i, y :: ys) = if f y then SOME i
Option Types

- fun firstPos(f, ys) =
  = let fun first(_, nil) = NONE
  = | first(i, y :: ys) =
  =     if f y
  =     then SOME i
  =
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)
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)
  =
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;
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
- 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]);
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
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]);
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
Datatypes

- datatype tree =
Datatypes

- datatype tree =

= 
Datatypes

- datatype tree = Leaf of int
Datatypes

- datatype tree =
  Leaf of int
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;

datatype tree
  = Leaf of int | Node of bool * tree * tree

-
Datatypes

- datatype tree =
  = Leaf of int
  = Node of bool * tree * tree;

val tr =
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;

val tr = 17 / 42
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,
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,
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),
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;

val tr =
  = Node(false,
  = Node(true, Leaf 0, Leaf 1),
  =
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);
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
-
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
```
**Datatypes**

- `datatype tree = Leaf of int | Node of bool * tree * tree;

```
val tr = Node(false, Node(true, Leaf 0, Leaf 1), Leaf 2);
```

- `fun size(Leaf _) = 1
`

```
**Datatypes**

```
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)) =
```
Datatypes

```ocaml
- 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)) =
```
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;

val tr = Node(false, Node(true, Leaf 0, Leaf 1), Leaf 2)

fun size(Leaf _) = 1
= | size(Node(_, tr1, tr2)) = 1 + size tr1 + size tr2;
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

-
Datatypes

- datatype tree =
  = Leaf of int
  = | Node of bool * tree * tree;

val tr = Node(false, Node(true, Leaf 0, Leaf 1), Leaf 2);

fun size(Leaf _) = 1
= | size(Node(_, tr1, tr2)) =
= 1 + size tr1 + size tr2;

val size = fn : tree -> int
- size tr;
Datatypes

datatype tree =
  Leaf of int |
  Node of bool * tree * tree;

val tr = Node(false,
  Node(true, Leaf 0, Leaf 1),
  Leaf 2);

fun size(Leaf _) = 1 |
  size(Node(_, tr1, tr2)) = 1 + size tr1 + size tr2;

val size = fn : tree -> int

val it = 5 : int
Extended Example: Generating Primes

Let’s consider the problem of generating the first $n$ prime numbers.
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
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.
void gen_primes(int n, int *primes) {
    int i, j;
    int next = 2;  /* next candidate */
    for (i = 0; i < n; i++) {
        int found = 0;
        while (!found) {
            for (j = 0; j < i; j++)
                if (next % primes[j] == 0)
                    break;
            if (j == i)
                found = 1;
            else
                next++;
        }
        primes[i] = next++;
    }
}
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 as m :: _ = prs(n - 1)
    in next(ms, m + 1) :: ms end

fun primes n = rev(prs n)
fun divisible(_, nil) = false
| divisible(l, m :: ms) =
  divisible(l, ms) orelse l mod m = 0

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

fun prs 0 = nil
| prs 1 = [2]
| prs n =
  let val ms as m :: _ = prs(n - 1)
  in next(ms, m + 1) :: ms end

fun primes n = rev(prs n)
fun next(ms, l) = 
  if List.exists (fn m => l mod m = 0) ms 
  then next(ms, l + 1) 
  else l 

fun prs 0 = NONE 
  | prs 1 = SOME([2], 2) 
  | prs n = 
    let val (ms, m) = valOf(prs(n - 1)) 
      val l = next(ms, m + 1) 
    in SOME(ms @ [l], l) end 

fun primes n = 
  case prs n of 
    NONE => nil 
  | SOME(ms, _) => ms
fun next(ms, ns, l) = 
    if List.exists (fn m => l mod m = 0) ms 
    then next(ms, ns, l + 1) 
    else if List.exists (fn n => l mod n = 0) ns 
    then next(ms @ rev ns, nil, l + 1) 
    else (ms, ns, l)

fun prs 0 = NONE 
| prs 1 = SOME([2], nil, 2) 
| prs n = 
    let val (ms, ns, l) = valOf(prs(n - 1)) 
    val (ms, ns, l) = next(ms, ns, l + 1) 
    in SOME(ms, l :: ns, l) end
fun next(ms, ns, l) = 
    if List.exists (fn m => l mod m = 0) ms 
    then next(ms, ns, l + 1) 
    else if List.exists (fn n => l mod n = 0) ns 
    then next(ms @ rev ns, nil, l + 1) 
    else (ms, ns, l)

fun prs 0 = NONE 
    | prs 1 = SOME([2], nil, 2) 
    | prs n = 
        let val (ms, ns, l) = valOf(prs(n - 1)) 
        val (ms, ns, l) = next(ms, ns, l + 1) 
        in SOME(ms, l :: ns, l) end

The reconfiguration of (ms, ns) into (ms @ rev ns, nil) only happens rarely; e.g., when generating the first 100,000 primes, it only happens times.
fun next(ms, ns, l) = 
  if List.exists (fn m => l mod m = 0) ms 
    then next(ms, ns, l + 1) 
  else if List.exists (fn n => l mod n = 0) ns 
    then next(ms @ rev ns, nil, l + 1) 
  else (ms, ns, l)

fun prs 0 = NONE 
| prs 1 = SOME([2], nil, 2) 
| prs n = 
  let val (ms, ns, l) = valOf(prs(n - 1)) 
    val (ms, ns, l) = next(ms, ns, l + 1) 
  in SOME(ms, l :: ns, l) end

The reconfiguration of (ms, ns) into (ms @ rev ns, nil) only happens rarely; e.g., when generating the first 100,000 primes, it only happens three times.
fun primes n = 
    case prs n of 
    NONE        => nil 
    | SOME(ms, ns, _) => ms @ rev ns
signature PRIMES =
sig

val primes : int -> int list

end
structure Primes => PRIMES =
struct

  fun next(ms, ns, l) =
      if List.exists (fn m => l mod m = 0) ms
        then next(ms, ns, l + 1)
      else if List.exists (fn n => l mod n = 0) ns
        then next(ms @ rev ns, nil, l + 1)
      else (ms, ns, l)
fun prs 0 = NONE
   | prs 1 = SOME([2], nil, 2)
   | prs n =
       let val (ms, ns, l) = valOf(prs(n - 1))
           val (ms, ns, l) = next(ms, ns, l + 1)
       in SOME(ms, l :: ns, l) end

fun primes n =
    case prs n of
        NONE => nil
        | SOME(ms, ns, _) => ms @ rev ns

end
Comparison

$
Comparison

$\text{time primes-gcc 100000 > /tmp/primes-gcc}$
Comparison

$ time primes-gcc 100000 > /tmp/primes-gcc
real 0m12.413s
user 0m12.401s
sys 0m0.007s
$

Comparison

$ time primes-gcc 100000 > /tmp/primes-gcc
real 0m12.413s
user 0m12.401s
sys 0m0.007s
$ time primes-smlnj 100000 > /tmp/primes-smlnj
Comparison

$ time primes-gcc 100000 > /tmp/primes-gcc
real 0m12.413s
user 0m12.401s
sys 0m0.007s
$ time primes-smlnj 100000 > /tmp/primes-smlnj
real 0m28.271s
user 0m27.770s
sys 0m0.485s
$
Comparison

$ time primes-gcc 100000 > /tmp/primes-gcc
real 0m12.413s
user 0m12.401s
sys 0m0.007s
$ time primes-smlnj 100000 > /tmp/primes-smlnj
real 0m28.271s
user 0m27.770s
sys 0m0.485s
$ time primes-mlton 100000 > /tmp/primes-mlton
Comparison

$ time primes-gcc 100000 > /tmp/primes-gcc  
real 0m12.413s  
user 0m12.401s  
sys 0m0.007s  
$ time primes-smlnj 100000 > /tmp/primes-smlnj  
real 0m28.271s  
user 0m27.770s  
sys 0m0.485s  
$ time primes-mlton 100000 > /tmp/primes-mlton  
real 0m17.810s  
user 0m17.329s  
sys 0m0.462s
Comparison

$ cmp /tmp/primes-gcc /tmp/primes-smlnj
$ cmp /tmp/primes-smlnj /tmp/primes-mlton
$ tail /tmp/primes-mlton

1299553
1299583
1299601
1299631
1299637
1299647
1299653
1299673
1299689
1299709
signature PRIMES =
sig

type state

val init : state

val next : state -> int * state

end
structure Primes :> PRIMES =
struct

type state = (int list * int list * int)option

val init = NONE
fun nxt(ms, ns, l) =  
    if List.exists (fn m => l mod m = 0) ms  
        then nxt(ms, ns, l + 1)  
    else if List.exists (fn n => l mod n = 0) ns  
        then nxt(ms @ rev ns, nil, l + 1)  
    else (ms, ns, l)

fun next NONE = (2, SOME([2], nil, 2))
| next (SOME(ms, ns, l)) = 
    let val (ms, ns, l) = nxt(ms, ns, l + 1)  
        in (l, SOME(ms, l :: ns, l)) end 

end
Primes in SML: VI
Primes in SML: VI

- val (n, st) = Primes.next Primes.init;
Primes in SML: VI

- val (n, st) = Primes.next Primes.init;
val n = 2 : int
val st = - : Primes.state
-
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;
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 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;
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
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;
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
-
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;
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
Can we create an infinite list (stream) of all primes like this?

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

Can we create an infinite list (stream) of all primes like this?

```ml
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.
Streams

Can we create an infinite list (stream) of all primes like this?

```sml
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
References

- val r = ref 10;
References

- val r = ref 10;
val r = ref 10 : int ref
-
- val r = ref 10;
val r = ref 10 : int ref
- !r;
References

- val r = ref 10;
val r = ref 10 : int ref
- !r;
val it = 10 : int
-
References

- val r = ref 10;
- !r;
- r := !r + 1;

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

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`

\[
\begin{align*}
\begin{array}{c}
\text{r} \\
\downarrow \\
\text{10 }\Rightarrow \text{ 11}
\end{array}
\end{align*}
\]
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
signature STREAM =
  sig

  type 'a stream

  val make : 'a * ('a -> 'b * 'a) -> 'b stream

  val get : 'a stream -> 'a * 'a stream

  end
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

end
Streams

Stream

\[ \text{Value}(x_0, \text{Stream}) \]

\[ \text{Value}(x_1, \text{Stream}) \]

\[ \text{Delay}(\text{fn}() = > \cdots) \]
Primes in SML: VI
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
Primes in SML: VI

val primes = Stream.make(Primes.init, Primes.next); val primes = − : int Stream.stream −
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
  =

Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
= | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
  = | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
-
val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
fun sub(stm, 0) = #1(Stream.get stm)
  | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
val sub = sub(primes, 1 - 1);
- val primes = Stream.make(Primes.init, Primes.next);  
val primes = - : int Stream.stream  
- fun sub(stm, 0) = #1(Stream.get stm)  
= | sub(stm, n) = sub(#2(Stream.get stm), n - 1);  
val sub = fn : 'a Stream.stream * int -> 'a  
- sub(primes, 1 - 1);  
val it = 2 : int
-
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
=     | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
  = | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
= | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
- sub(primes, 100000 - 1);
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);  
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)  
  | sub(stm, n) = sub(#2(Stream.get stm), n - 1);  
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);  
val it = 2 : int
- sub(primes, 2 - 1);  
val it = 3 : int
- sub(primes, 100000 - 1);  
val it = 1299709 : int
Primes in SML: VI

val primes = Stream.make(Primes.init, Primes.next);

val sub = fn : 'a Stream.stream * int -> 'a

- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
- sub(primes, 100000 - 1);
val it = 1299709 : int
- sub(primes, 99999 - 1);
Primes in SML: VI

```sml
val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
  = | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
- sub(primes, 100000 - 1);
val it = 1299709 : int
- sub(primes, 99999 - 1);
val it = 1299689 : int
- 42 / 42
```
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
=   | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
- sub(primes, 100000 - 1);
val it = 1299709 : int
- sub(primes, 99999 - 1);
val it = 1299689 : int
- sub(primes, 99998 - 1);
Primes in SML: VI

- val primes = Stream.make(Primes.init, Primes.next);
val primes = - : int Stream.stream
- fun sub(stm, 0) = #1(Stream.get stm)
  = | sub(stm, n) = sub(#2(Stream.get stm), n - 1);
val sub = fn : 'a Stream.stream * int -> 'a
- sub(primes, 1 - 1);
val it = 2 : int
- sub(primes, 2 - 1);
val it = 3 : int
- sub(primes, 100000 - 1);
val it = 1299709 : int
- sub(primes, 99999 - 1);
val it = 1299689 : int
- sub(primes, 99998 - 1);
val it = 1299673 : int