

List Processing in SML



**CS251 Programming Languages
Spring 2017**

**Lyn Turbak, Meera Hejmadi,
Mary Ruth Ngo, & Angela Wu**

Department of Computer Science
Wellesley College

Consing Elements into Lists

```
- val nums = 9 :: 4 :: 7 :: [];  
val nums = [9,4,7] : int list  
  
- 5 :: nums;  
val it =          : int list  
  
- nums;  
val it =          : int list (* nums is unchanged *)  
  
- (1+2) :: (3*4) :: (5-6) :: [];  
val it =          : int list  
  
- [1+2, 3*4, 5-6];  
val it = [3,12,~1] : int list  
  
- [1=2, 3 < 4, false];  
val it =          : bool list  
  
- ["I", "do", String.substring ("note",0,3), "li" ^ "ke"];  
val it =          : string list  
  
- [(#"a", 8), (#"z", 5)];  
val it = [(#"a",8), (#"z",5)] : (char * int) list  
  
- [[7,2,5], [6], 9::[3,4]];  
val it = [[7,2,5],[6],[9,3,4]] : int list list
```

List Processing in SML 2

SML lists are homogeneous

Unlike in Racket & Python, all elements of an SML list must have the same type.

```
- 1 :: [2,3,4];  
val it = [1,2,3,4] : int list  
  
- op:: (1, [2,3,4]); (* op:: is prefix version of infix :: *)  
val it = [1,2,3,4] : int list  
  
-op:: ;  
val it = fn : 'a * 'a list -> 'a list  
  
- "a" :: [1,2,3];  
stdIn:1.1-8.3 Error: operator and operand don't agree [literal]  
operator domain: string * string list  
operand:          string * int list  
in expression:  
"a" :: 1 :: 2 :: 3 :: nil  
  
-[1,2] :: [3,4,5];  
stdIn:9.1-9.17 Error: operator and operand don't agree [literal]  
operator domain: int list * int list list  
operand:          int list * int list  
in expression:  
(1 :: 2 :: nil) :: 3 :: 4 :: 5 :: nil
```

List Processing in SML 3

Tuples vs. Lists

Tuples are heterogeneous fixed-length product types:

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

Tuples are homogeneous variable-length product types:

```
- [1, 2+3, 4*5, 6-7, 8 mod 3];  
val it = [1,5,20,~1,2] : int list  
  
- [1=2, 3<4];  
val it = [false,true] : bool list  
  
- ["foo", "bar" ^ "baz", String.substring ("abcdefg", 2, 3)];  
val it = ["foo","barbaz","cde"] : string list  
  
- [#"a", String.sub("baz", 2), chr(100)];  
- val it = [#"a",#"z",#"d"] : char list
```

List Processing in SML 4

Some Simple List Operations

```
- List.length [7,3,6,1];
val it = 4 : int

- List.nth ([7,3,6,1],0);
val it = 7 : int

- List.hd [7,3,6,1];
val it = 7 : int

- List.tl [7,3,6,1];
val it = [3,6,1] : int list

- List.take ([7,3,6,1],2);
val it = [7,3] : int list

- List.take ([7,3,6,1],3);
val it = [7,3,6] : int list

- List.drop ([7,3,6,1],2);
val it = [6,1] : int list

- List.drop ([7,3,6,1],3);
val it = [1] : int list

- List.nth ([7,3,6,1],1);
val it = 3 : int

- List.nth ([7,3,6,1],2);
val it = 6 : int

- List.null [7,3,6,1];
val it = false : bool

- List.null [];
val it = true : bool

- [7,3,6,1] = [];
val it = false : bool

- List.rev [7,3,6,1];
val it = [1,6,3,7] : int list
```

use pattern
matching instead

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

List Processing in SML 5

Appending Lists

```
- [7,2] @ [8,1,6];
val it = [7,2,8,1,6] : int list

- [7,2] @ [8,1,6] @ [9] @ [];
val it = [7,2,8,1,6,9] : int list

(* Appending is different than consing! *)
- [7,2] :: [8,1,6] :: [9] :: [];
val it = [[7,2],[8,1,6],[9]] : int list list

- op::; (* prefix cons function *)
val it = fn : 'a * 'a list -> 'a list

- op@; (* prefix append function *)
val it = fn : 'a list * 'a list -> 'a list

(* List.concat appends all elts in a list of lists *)
- List.concat [[7,2],[8,1,6],[9]];
val it = [7,2,8,1,6,9] : int list

- List.concat;
val it = fn : 'a list list -> 'a list
```

List Processing in SML 6

Pattern Matching on Lists

```
(* matchtest : (int * int) list -> (int * int) list *)
fun matchtest xs =
  case xs of
    [] => []
  | [(a,b)] => [(b,a)]
  | (a,b) :: (c,d) :: zs => (a+c,b*d) :: (c,d) :: zs
```

```
- matchtest [];
val it = : (int * int) list

- matchtest [(1,2)];
val it = : (int * int) list

- matchtest [(1,2),(3,4)];
val it = : (int * int) list

- matchtest [(1,2),(3,4),(5,6)];
val it = : (int * int) list
```

List Processing in SML 7

Other Pattern-Matching Notations

```
fun matchtest2 xs =
  case xs of
    [] => []
  | [(a,b)] => [(b,a)]
  | (a,b) :: (ys as ((c,d) :: zs)) => (a+c,b*d) :: ys
  (* subpatterns can be named with "as" *)
```

```
fun matchtest3 [] = []
  | matchtest3 [(a,b)] = [(b,a)]
  | matchtest3 ((a,b) :: (ys as ((c,d) :: zs)))
    = (a+c,b*d) :: ys
  (* parens around pattern necessary above *)
```

List Processing in SML 8

List Accumulation

```
(* Recursively sum a list of integers *)
(* sumListRec : int list -> int *)
fun sumListRec [] =
  | sumListRec (x::xs) =
```

```
- sumListRec [];
val it = 0 : int

- sumListRec [5,2,4];
val it = 11 : int
```

```
(* Iterative (tail-recursive) summation *)
fun sumListIter xs =
  let fun loop [] sum =
        | loop (y::ys) sum =
      in loop xs 0
  end
```

```
- sumListIter [5,2,4];
val it = 11 : int
```

List Processing in SML 9

Instance of the Mapping Idiom

```
(* incList : int list -> int list *)
fun incList [] =
  | incList (x::xs) =
```

```
- incList [5,2,4];
val it = [6,3,5] : int list
```

```
- incList [];
val it = [] : int list
```

List Processing in SML 10

Abstracting Over the Mapping Idiom

```
(* myMap : ('a -> 'b) -> 'a list -> 'b list *)
fun myMap f [] = []
  | myMap f (x::xs) = (f x)::(myMap f xs)
```

```
- myMap (fn x => x + 1) [5,2,4];
val it =          : int list

- myMap (fn y => y * 2) [5,2,4];
val it =          : int list

- myMap (fn z => z > 3) [5,2,4];
val it =          : bool list

- myMap (fn a => (a, (a mod 2) = 0)) [5,2,4];
val it =          : (int * bool) list

- myMap (fn s => s ^ "side") ["in", "out", "under"];
val it =          : string list

- myMap (fn xs => 6::xs) [[7,2],[3],[8,4,5]];
val it =          : int list list
```

```
(* SML/NJ supplies map at top-level and as List.map *)
```

List Processing in SML 11

Cartesian Products of Lists

```
(* 'a list -> 'b list -> ('a * 'b) list *)
fun listProd xs ys =
  List.concat (List.map
              xs)
```

```
- listProd ["a", "b"] [1,2,3];
val it = [("a",1), ("a",2), ("a",3), ("b",1), ("b",2), ("b",3)]
```

```
- listProd [1,2,3] ["a", "b"];
val it = [(1,"a"), (1,"b"), (2,"a"), (2,"b"), (3,"a"), (3,"b")]
```

List Processing in SML 12

Zippping: A Different Kind of List Product

```
(* 'a list * 'b list -> ('a * 'b) list *)
- ListPair.zip (["a","b","c"], [1,2,3,4]);
val it = (["a",1], ("b",2), ("c",3)) : (string * int) list

(* ('a * 'b) list -> 'a list * 'b list *)
- ListPair.unzip (["a",1], ("b",2), ("c",3));
val it = (["a","b","c"], [1,2,3]) : string list * int list
```

(* An API for all SMLNJ ListPair operations can be found at:
<http://www.standardml.org/Basis/list-pair.html> *)

List Processing in SML 13

Instance of the Filtering Idiom

```
fun filterPos [] =
  | filterPos (x::xs) =

- filterPos [3, ~7, ~6, 8, 5];
val it = [3,8,5] : int list

- filterPos [];
val it = [] : int list
```

List Processing in SML 14

Abstracting over the Filtering Idiom

```
(* myFilter : ('a -> bool) -> 'a list -> 'a list *)
fun myFilter pred [] = []
  | myFilter pred (x::xs) =
    if (pred x) then
      x :: (myFilter pred xs)
    else
      (myFilter pred xs)
```

```
- myFilter (fn x => x > 0) [3, ~7, ~6, 8, 5];
val it =          : int list

- myFilter (fn y => (y mod 2) = 0) [5,2,4,1];
val it =          : int list

- myFilter (fn s => (String.size s) <= 3)
= ["I","do","not","like","green","eggs","and","ham"];
val it =          : string list

- myFilter (fn xs => (sumListRec xs > 10)) [[7,2],[3],[8,4,5]];
val it =          : int list list
```

(* SML/NJ supplies this function as List.filter *)

List Processing in SML 15

Some Other Higher-Order List Ops

```
(* List.partition : ('a -> bool) -> 'a list -> 'a list * 'a list
   splits a list into two: those elements that satisfy the
   predicate, and those that don't *)
- List.partition (fn x => x > 0) [3, ~7, ~6, 8, 5];
val it = ([3,8,5],[~7,~6]) : int list * int list

- List.partition (fn y => (y mod 2) = 0) [5,2,4,1];
val it = ([2,4],[5,1]) : int list * int list

(* List.all : ('a -> bool) -> 'a list -> bool returns true iff
   the predicate is true for all elements in the list. *)
- List.all (fn x => x > 0) [5,2,4,1];
val it = true : bool

- List.all (fn y => (y mod 2) = 0) [5,2,4,1];
val it = false : bool

(* List.exists : ('a -> bool) -> 'a list -> bool returns true iff
   the predicate is true for at least one element in the list. *)
- List.exists (fn y => (y mod 2) = 0) [5,2,4,1];
val it = true : bool

- List.exists (fn z => z < 0) [5,2,4,1];
val it = false : bool
```

List Processing in SML 16

foldr : The Mother of All List Recursive Functions

```
- List.foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- List.foldr (fn (x,y) => x + y) 0 [5,2,4];
val it =      : int

- List.foldr op+ 0 [5,2,4];
val it =      : int

- List.foldr (fn (x,y) => x * y) 1 [5,2,4];
val it =      : int

- List.foldr (fn (x,y) => x andalso y) true [true,false,true];
val it =      : bool

- List.foldr (fn (x,y) => x andalso y) true [true,true,true];
val it =      : bool

- List.foldr (fn (x,y) => x orelse y) false [true,false,true];
val it =      : bool

- List.foldr (fn (x,y) => (x > 0) andalso y) true [5,2,4];
val it =      : bool

- List.foldr (fn (x,y) => (x < 0) orelse y) false [5,2,4];
val it =      : bool
```