List Processing in SML

Monday, September 17, 2007

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Consing Elements into Lists

```
9 :: 4 :: 7 :: [];
val it = [9,4,7] : int list

- 9 :: 4 :: 7 :: 5 :: [];
val it = [9,4,7,5] : int list

- (1+2) :: (3*4) :: (5-6) :: [];
val it = [3,12,~1] : int list

- [1+2, 3*4, 5-6];
val it = [3,12,~1] : int list

- [1=2, 3 < 4, false];
val it = [false,true,false] : bool list

- ["I", "do", String.substring ("note",0,3), "li" ^ "ke"];
val it = ["I","do","not","like"] : 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
```

Some Simple List Operations

```
- List.length [7,3,6,1];
val it = 4 : 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],0);
val it = 7 : int

- 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
```


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

(* 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
```

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 = [(2,1)] : (int * int) list

- matchtest [(1,2),(3,4)];
val it = [(4,8),(3,4)] : (int * int) list

- matchtest [(1,2),(3,4),(5,6)];
val it = [(4,8),(3,4),(5,6)] : (int * int) list
```


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
```

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

List Accumulation

```
(* Recursively sum a list of integers *)
(* sumListRec : int list -> int *)
fun sumListRec [] = 0
  | sumListRec (x::xs) = x + (sumListRec xs)
```

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

- sumListRec [5,2,4,1];
val it = 12 : int
```

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

```
- sumListIter [5,2,4,1];
val it = 12 : int
```


Instance of the Mapping Idiom

```
(* incList : int list -> int list *)
fun incList [] = []
  | incList (x::xs) = (x+1)::(incList xs)
```

```
- incList [5,2,4,1];
val it = [6,3,5,2] : int list

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

Abstracting Over the Mapping Idiom

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

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

- map (fn y => y * 2) [5,2,4,1];
val it = [10,4,8,2] : int list

- map (fn z => z > 3) [5,2,4,1];
val it = [true,false,true,false] : bool list

- map (fn a => (a, (a mod 2) = 0)) [5,2,4,1];
val it = [(5,false),(2,true),(4,true),(1,false)] : (int * bool) list

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

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

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


Cartesian Products of Lists

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

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

- listProd ["a", "b"] [];
stdIn:6.1-6.23 Warning: type vars not generalized
because of value restriction are instantiated to
dummy types (X1,X2,...)
val it = [] : (string * ?.X1) list
```

Zipping: 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
```


Powersets (well, bags really) of Lists

```
(* 'a list -> 'a list list *)
fun powerBag [] = [[]]
  | powerBag (x::xs) =
      let val subBag = powerBag xs
      in  subBag @ (map (fn bag => x::bag) subBag)
      end
```

```
- powerBag [1];
val it = [[],[1]] : int list list

- powerBag [1,2];
val it = [[],[2],[1],[1,2]] : int list list

- powerBag [1,2,3];
val it = [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]] : int list list

- powerBag [1,2,1];
val it = [[],[1],[2],[2,1],[1],[1,1],[1,2],[1,2,1]] : int list list
```

Instance of the Filtering Idiom

```
fun filterPos [] = []
  | filterPos (x::xs) =
      if x > 0 then
      x :: (filterPos xs)
      else
      filterPos xs
```

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

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


Abstracting over the Filtering Idiom

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

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

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

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

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

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

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
```


Options and List.find

```
-List.map (fn d => if (d = 0) then NONE else SOME (100 div d))
=   [5, 3, 0, 10];
val it = [SOME 20,SOME 33,NONE,SOME 10] : int option list

- fun addOptions (SOME x) (SOME y) = x + y
=   | addOptions (SOME x) NONE = x
=   | addOptions NONE (SOME y) = y
=   | addOptions NONE NONE = 0;
val addOptions = fn : int option -> int option -> int

- List.find (fn y => (y mod 2) = 0) [5,2,4,1];
val it = SOME 2 : int option

- List.find (fn z => z < 0) [5,2,4,1];
val it = NONE : int option
```

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 [5,2,4,1];
val it = 12 : int

- List.foldr op+ 0 [5,2,4,1];
val it = 12 : int

- List.foldr (fn (x,y) => x * y) 1 [5,2,4,1];
val it = 40 : int

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

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

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

- List.foldr (fn (x,y) => (x > 0) andalso y) true [5,2,4,1];
val it = true : bool

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