Consing Elements into Lists
- val nums = 9 :: 4 :: 7 :: [];
- 5 :: nums;
- nums;
- (1+2) :: (3*4) :: (5-6) :: [];
- [1+2, 3*4, 5-6];
- [1+2, 3 < 4, false];
- ["I", "do", String.substring("note", 0, 3), "li" ^ "ke"];
- ["I", "do", "not", "like"];
- [(#"a", 8), (#"z", 5)];
- [1,2] :: [3,4];

Unlike in Racket & Python, all elements of an SML list must have the same type.
- 1 :: [2,3,4];
- op:: (1, [2,3,4]); (* op:: is prefix version of infix :: *)
- "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
  operand: int list * int list
  in expression:
  (1 :: 2 :: nil) :: 3 :: 4 :: 5 :: nil

Tuples are heterogeneous fixed-length product types:
- (1+2, 3=4, "foo" ^ "bar", String.substring("baz", 2));
- [1,2,3,4, 5*6, 6-7, 8 mod 3];
- [1-2, 3<4];
- ["foo", "bar" ^ "baz", String.substring("abcdefg", 2, 3)];
- ["a", String.substring("baz", 2), chr(100)];
- val it = [(#"a", (#"z","d") : char 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 = [7,3,6] : int list
- `List.take ([7,3,6,1], 3);`
  val it = [7,3,6] : int list
- `List.take ([7,3,6,1], 4);`
  val it = [7,3,6] : 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.nth ([7,3,6,1], 3);`
  val it = false : bool
- `List.null [7,3,6,1];`
  val it = false : bool
- `List.null [];
  val it = true : bool
- `val it = [7,2] @ [8,1,6];`
  val it = [7,2,8,1,6] : int list
- `val it = [7,2] @ [8,1,6] @ [9] @ [];`
  val it = [7,2,8,1,6,9] : int list
- `val it = [[7,2],[8,1,6],[9]];`
  val it = [[7,2],[8,1,6],[9]] : int list
- `val it = List.rev [7,3,6,1];`
  val it = [1,6,3,7] : int list
- `val it = List.concat [[7,2],[8,1,6],[9]];`
  val it = [7,2,8,1,6,9] : int list
- `use pattern matching instead`

(* An API for all SMLNJ String operations can be found at: *)

Appendix

- `[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
- `val it = [[7,2],[8,1,6],[9]];`
  val it = [[7,2],[8,1,6],[9]] : int list
- `val it = List.concat [[7,2],[8,1,6],[9]];`
  val it = [[7,2],[8,1,6],[9]] : int list
- `val it = List.rev [7,3,6,1];`
  val it = [1,6,3,7] : int 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

- `val it = matchtest [];
  val it = [] : (int * int) list`
- `val it = matchtest [(1,2)];
  val it = [(1,2)] : (int * int) list`
- `val it = matchtest [(1,2),(3,4)];
  val it = [(1,2),(3,4)] : (int * int) list`
- `val it = matchtest [(1,2),(3,4),(5,6)];
  val it = [(1,2),(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
  (* subpatterns can be named with "as" *)

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

List Accumulation

(* Recursively sum a list of integers *)
fun sumListRec [] = 0
| sumListRec (x::xs) = x + (sumListRec 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 = sum
  | loop (y::ys) sum = loop ys (y + sum)
in loop xs 0
end
- sumListIter [5,2,4];
val it = 11 : int

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];
val it = [6,3,5] : int list
- map (fn y => y * 2) [5,2,4];
val it = [10,4,8] : int list
- map (fn z => z > 3) [5,2,4];
val it = [true,false,true] : bool list
- map (fn a => (a, (a mod 2) = 0)) [5,2,4];
val it = [(5,false),(6,true),(4,true)] : (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 *)

Instance of the Mapping Idiom

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

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

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

(* '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 [1,2,3] ["a", "b"];
val it = [(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b")]

(* SML/NJ supplies map at top-level and as List.map *)
Zipping: A Different Kind of List Product

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

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

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

Powersets (well, bags really) of Lists

(* 'a list -> 'a list list *)
fun powerBag [] = [[]]
  | powerBag (x::xs) =
      let
      val subbags = powerBag xs
      in
      subbags @ (List.map (fn bag => x::bag) subbags)
      end
    in
    powerBag [1];
    val it = [[],[1]] : int list
    | powerBag [2,1];
    val it = [[],[1],[2],[2,1]] : int list
    | powerBag [3,2,1];
    val it = [[],[1],[2],[2,1],[3],[3,1],[3,2],[3,2,1]] : int list
    | powerBag [1,2,1];
    val it = [[],[1],[2],[2,1],[1],[1,1],[1,2],[1,2,1]] : int list
  end

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)
    in
    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","like","and","ham" ] : string list
    | filter (fn xs => (sumListRec xs > 10)) [[7,2],[3],[8,4,5]];
    val it = [[8,4,5]] : int list
  end

Instance of the Filtering Idiom

fun filterPos [] = []
  | filterPos (x::xs) =
      if x > 0
      then x::(filterPos xs)
      else filterPos xs
    in
    filterPos [3, ~7, ~6, 8, 5];
    val it = [3,8,5] : int list
    | filterPos [];
    val it = [] : int 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 x => x > 0) [5,2,4,1];
  val it = true : bool
- List.exists (fn y => (y mod 2) = 0) [5,2,4,1];
  val it = false : bool

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 = 11 : int
- List.foldr op+ 0 [5,2,4];
  val it = 11 : int
- List.foldr (fn (x,y) => x * y) 1 [5,2,4];
  val it = 40 : int
- List.foldr (fn (x,y) => x andalso y) true [true,false,true];
  val it = true : 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];
  val it = true : bool
- List.foldr (fn (x,y) => (x < 0) orelse y) false [5,2,4];
  val it = false : bool

Strings of Chars

- String.explode "foobar";
  val it = ["f","o","o","b","a","r"] : char list
- String.implode ["1","0","0","1","1","0"];
  val it = "100110" : string

Define the following function:

all_1s : string -> bool
Returns true iff the given string contains only 1s.

fun all_1s s = List.all (fn c => c = "1")
  (String.explode s)