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

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

Tuples are heterogeneous fixed-length product types:

- (1+2, 3=4, "foo" ^ "bar", String.sub ("baz", 2));
  val it = (3,false,"fooobar",#"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","ode"] : string list
- [#"a", String.sub("baz", 2), chr(100)];
  val it = [#"a",#"z",#"d"] : char list
Some Simple List Operations

- **List.length** `val it = 4 : int`
- **List.hd** `val it = 7 : int`
- **List.tl** `val it = [3,6,1] : int list`
- **List.take** `val it = [7,3] : int list`
- **List.drop** `val it = [6,1] : int list`
- **List.nth** `val it = 7 : int`
- **List.null** `val it = false : bool`
- **List.rev** `val it = [1,6,3,7] : int list`

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

Appending Lists

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

(* Appending is different than consing! *)

Pattern Matching on Lists

```sml
-- (* 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 = : (int * int) list
```

Other Pattern-Matching Notations

```sml
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" *)
```

```sml
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
```

(* PaLearning Matching on Lists *)

List Processing in SML
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 = 
    in loop xs 0 
    end
  - sumListIter [5,2,4];
  val it = 11 : int

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 = [6,3,5] : int list
- myMap (fn y => y * 2) [5,2,4];
  val it = [10,4,8] : int list
- myMap (fn z => z > 3) [5,2,4];
  val it = [false, true, true] : bool list
- myMap (fn a => (a, (a mod 2) = 0)) [5,2,4];
  val it = [(5, true), (2, false)] : (int * bool) list
- myMap (fn s => s ^ "side") ["in", "out", "under"];
  val it = ["inside", "outsider", "underer"] : string list
- myMap (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) = 
    - 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 (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 9
List Processing in SML 10
List Processing in SML 11
List Processing in SML 12
Zipping: A Different Kind of List Product

(* 'a list * 'b list -> ('a * 'b) list *)
- ListPair.zip ("a",b",c",[1,2,3,4]);
- ListPair.unzip ("a",1,"b",2,"c",3);:

(* 'a list * 'b list -> 'a list * 'b list *)
- ListPair.zip ("a",1",b",2,"c",3);:

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

AbstracJing over the Filtering Idiom

fun myFilter [] = []
| 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];
- myFilter (fn y => (y mod 2) = 0) [5,2,4,1];

- List.partition (fn x => x > 0) [3, ~7, ~6, 8, 5];
- List.partition (fn y => (y mod 2) = 0) [5,2,4,1];

- List.all (fn x => x > 0) [5,2,4,1];
- List.all (fn y => (y mod 2) = 0) [5,2,4,1];

- List.exists (fn z => z < 0) [5,2,4,1];
- List.exists (fn x => x > 0) [5,2,4,1];

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];
- List.partition (fn y => (y mod 2) = 0) [5,2,4,1];

- List.all (fn x => x > 0) [5,2,4,1];
- List.all (fn y => (y mod 2) = 0) [5,2,4,1];

- List.exists (fn z => z < 0) [5,2,4,1];
- List.exists (fn x => x > 0) [5,2,4,1];
foldr : The Mother of All List Recursions

- 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 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

foldl : The Mother of All List Iterations

- List.foldl
  val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- List.foldl op+ 0 [5,2,4];
  val it = : int
- List.foldl op* 1 [5,2,4];
  val it = : int
- List.foldl (fn (bit, sumSoFar) => 2*sumSoFar + bit) 0 [1, 0, 1, 0];
  val it = : int
- List.foldl (fn (bit, sumSoFar) => 2*sumSoFar + bit) 0
  = [1, 1, 1, 1, 1, 0, 1, 1];
  val it = : int