Motivating example: geometric figures

Suppose we want to represent geometric figures like circles, rectangles, and triangles so that we can do things like calculate their perimeters, scale them, etc. (Don’t worry about drawing them!)

These are so-called sum of products data:
- Circle, Rec, and Tri are tags that distinguish which one in a sum
- The numeric children of each tag are the product associated with that tag.

How would you do this in Java? In Python?

SML’s datatype for Sum-of-Product types

```sml
datatype figure =
| Circle of real (* radius *)
| Rect of real * real (* width, height *)
| Tri of real * real * real (* side1, side2, side3 *)

val figs = [Circle 1.0, Rect (2.0, 3.0), Tri(4.0,5.0,6.0)] (* List of sample figures *)
val circs = map Circ [7.0, 8.0, 9.0] (* List of three circles *)
```

Functions on datatype via pattern matching

```sml
(* Return perimeter of figure *)
fun perim (Circle r) = 2.0 * Math.pi * r
| perim (Rect (w, h)) = 2.0 * (w + h)
| perim (Tri(s1, s2, s3)) = s1 + s2 + s3

(* Scale figure by factor n *)
fun scale n (Circle r) = Circle (n * r)
| scale n (Rect (w, h)) = Rect (n*w, n*h)
| scale n (Tri(s1, s2, s3)) = Tri (n*s1, n*s2, n*s3)
```

- val perims = map perim figs
- val perims = [6.28318530718, 10.0, 15.0] : real list
- val scaledFigs = map (scale 3.0) figs
- val scaledFigs = [Circle 3.0, Rect (6.0, 9.0), Tri (12.0, 15.0, 18.0)] : figure list
Options

SML has a built-in option datatype defined as follows:

```sml
datatype 'a option = NONE | SOME of 'a
```

- NONE
  - `val it = NONE : 'a option`
- SOME 3;
  - `val it = SOME 3 : int option`
- SOME true;
  - `val it = SOME true : bool option`

Sample Use of Options

- `fun into_100 n = if (n = 0) then NONE else SOME (100 div n);`
  - `val into_100 = fn : int -> int option`
- `List.map into_100 [5, 3, 0, 10];`
  - `val it = [SOME 20, SOME 33, NONE, SOME 10] : int option list`
- `fun addOptions (SOME x) (SOME y) = SOME (x + y)`
  - `| addOptions (SOME x) NONE = NONE`
  - `| addOptions NONE (SOME y) = NONE`
  - `| addOptions NONE NONE = NONE;`
  - `val addOptions = fn : int option -> int option -> int option`
- `addOptions (into_100 5) (into_100 10);`
  - `val it = SOME 30 : int option`
- `addOptions (into_100 5) (into_100 0);`
  - `val it = NONE: int option`

Options and List.find

(* List.find : ('a -> bool) -> 'a list -> 'a option *)
- `List.find (fn y => (y mod 2) = 0) [5,8,4,1];`
  - `val it = SOME 8 : int option`
- `List.find (fn z => z < 0) [5,8,4,1];`
  - `val it = NONE : int option`

Thinking about options

What problem does option solve?

How is the problem solved in other languages?
Creating our own list datatype

```sml
datatype 'a mylist = Nil | Cons of 'a * 'a mylist

val ints = Cons(1, Cons(2, Cons(3, Nil))) (* : int mylist *)
val strings = Cons("foo", Cons("bar", Cons("baz", Nil)))
(* : strings mylist *)

fun myMap f Nil = Nil
  | myMap f (Cons(x,xs)) = Cons(f x, myMap f xs)
(* : ('a -> 'b) -> 'a mylist -> 'b mylist *)

val incNums = myMap (fn x => x + 1) ints
(* val incNums = Cons(2,Cons(3,Cons(4,Nil))) : int mylist *)
val myStrings = myMap (fn s => "my " ^ s) strings
(* val myStrings = Cons("my foo", Cons("my bar", Cons("my baz",Nil))) : string mylist *)
```

Binary Trees

```sml
datatype 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree

val int_tree = Node(Node(Leaf,2,Leaf),
  4,
  Node(Node(Leaf,1, Node(Leaf,5, Leaf)),
       6,
       Node(Leaf,3, Leaf)))

val string_tree = Node(Node(Leaf,"like",Leaf),
  "green",
  Node(Node(Leaf,"and",Leaf),
       "eggs",
       Node(Leaf,"ham",Leaf)))
```

SML bintree datatype for Binary Trees

```
val int_tree = Node(Node(Leaf,2,Leaf),
  4,
  Node(Node(Leaf,1, Node(Leaf,5, Leaf)),
       6,
       Node(Leaf,3, Leaf)))
```

bintree can have any type of element

```
val string_tree = Node(Node(Leaf,"like",Leaf),
  "green",
  Node(Node(Leaf,"and",Leaf),
       "eggs",
       Node(Leaf,"ham",Leaf)))
```
Counting nodes in a binary tree

```ml
fun num_nodes Leaf = 0
  | num_nodes (Node(l,v,r)) = 1 + (num_nodes l) + (num_nodes r)

val it = 6 : int
val it = 5 : int
```

Your turn: \texttt{height}

```ml
(* val height = fn : 'a bintree -> int *)
(* Returns the height of a binary tree. *)
(* Note: Int.max returns the max of two ints *)

fun height Leaf = 0
  | height (Node(l,v,r)) = 1 + Int.max(height l, height r)

val it = 4 : int
val it = 3 : int
```

Your turn: \texttt{sum\_nodes}

```ml
(* val sum\_nodes = fn : int bintree -> int *)
(* Returns the sum of node values in binary tree of ints *)

fun sum\_nodes Leaf = 0
  | sum\_nodes (Node(l,v,r)) = (sum\_nodes l) + v + (sum\_nodes r)

val it = 21 : int
```

Your turn: \texttt{inlist}

This returns a list of elements as they are encountered in an \textbf{in-order} traversal of a tree. We could also list them via a \textbf{pre-order} or \textbf{post-order} traversal.

```ml
(* val inlist = fn : 'a bintree -> 'a list *)
(* Returns a list of the node values in in-order *)

fun inlist Leaf = []
  | inlist (Node(l,v,r)) = (inlist l) @ [v] @ (inlist r)

val it = [2,4,1,5,6,3] : int list
val it = ["like","green","eggs","and","ham"] : string list
```
Your turn: map_tree

(`val map_tree = fn : ('a -> 'b) -> 'a bintree -> 'b bintree *)

`fun` map_tree f Leaf = Leaf
 | map_tree f (Node(l,v,r)) = Node(map_tree f l, f v, map_tree f r)

- map_tree (fn x => x*2) int_tree;
  `val` it = Node (Node (Leaf,1,Leaf),2,
  Node (Leaf,2,Node (Leaf,10,Leaf)),12,
  Node (Leaf,6,Leaf));

- map_tree (fn s => String.sub(s,0)) string_tree;
  `val` it = Node (Node (Leaf,"a",Leaf),"b",
  Node (Node (Leaf,"c",Leaf),"d",
  Node (Leaf,"e",Leaf)));

You turn: fold_tree

(`val fold_tree = fn : ('b * 'a * 'b) -> 'b -> 'b bintree -> 'b *'b *)

`fun` fold_tree comb leafval Leaf = leafval
 | fold_tree comb leafval (Node(l,v,r)) = combin(fold_tree comb leafval l, v, fold_tree comb leafval r)

- fold_tree (fn (lsum,v,rsum) => lsum + v + rsum) 0 int_tree;
  `val` it = 21 : int

- fold_tree (fn (lstr,v,rstr) => lstr ^ v ^ rstr) " " string_tree;
  `val` it = " like green eggs and ham ": string

You turn: Binary Search Tree insertion

`fun` singleton v = Node(Leaf, v, Leaf)

(`val insert: 'a bintree -> 'a -> 'a bintree *)

`fun` insert x Leaf = singleton x
 | insert x (t as (Node(l,v,r))) =
   if x = v then t
   else if x < v then Node(insert x l, v, r)
   else Node(l, v, insert x r)

`fun` listToTree xs = foldl (fn (x,t) => insert x t)
  Leaf xs

- val test_bst = listToTree [4,2,3,6,1,7,5];
  `val` test_bst = Node (Node (Node (Leaf,1,Leaf),
  2,
  Node (Leaf,3,Leaf)),
  4,
  Node (Node (Leaf,5,Leaf),
  6,
  Node (Leaf,7,Leaf)));
Your turn: Binary Search Tree membership

(val member: 'a -> 'a bintree -> bool *)
fun member x Leaf = false
| member x (Node(l,v,r)) =
  (x = v) orelse member x l orelse member x r

val test_member = map (fn i => (i, member i test_bst)) [0,1,2,3,4,5,6,7,8];
val it = [(0,false),(1,true),(2,true),(3,true),
(4,true),(5,true),(6,true),(7,true), (8,false)] : (int * bool) list

Balanced Trees (PS7)

BSTs are not guaranteed to be balanced.

But there are other tree data structures that do guarantee balance:
AVL trees, Red/Black trees, 2-3 trees, 2-3-4 trees.

In PS7 you will experiment with 2-3 trees.