Sum-of-Product (SOP) Datatypes in SML

Motivating SOP 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

<table>
<thead>
<tr>
<th>Circle</th>
<th>Rect</th>
<th>Tri</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>4.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
</tbody>
</table>

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

val figs = [Circ 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

(* Return perimeter of figure *)
```sml
fun perim (Circ 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 *)
```sml
fun scale n (Circ r) = Circ (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 = [Circ 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` 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` it = SOME 30 : int option
- `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 do options 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
(* : int mylist *)
val myStrings = myMap (fn s => "my " ^ s) strings
(* : string mylist *)
```

SML bintree datatype for Binary Trees

```sml
datatype 'a bintree = Leaf
| Node of 'a bintree * 'a * 'a bintree
(* left subtree, value, right subtree *)

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

Binary Trees

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

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

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

Sum-of-Product Datatypes in SML  13

height

(* 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 =
   | height (Node(l,v,r)) =

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

Sum-of-Product Datatypes in SML  14

sum_nodes

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

fun sum_nodes Leaf =
   | sum_nodes (Node(l,v,r)) =

val it = 21 : int

Sum-of-Product Datatypes in SML  15

inlist

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

(* 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)) =

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

Sum-of-Product Datatypes in SML  16
**map_tree**

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

(* maps function over every node in a binary tree *)

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,4,Leaf),8,
   Node (Node (Leaf,2,Node (Leaf,10,Leaf)),12,
   Node (Leaf,6,Leaf))) : int bintree

- map_tree (fn s => String.sub(s,0)) string_tree;
  val it = Node (Node (Leaf,#"l",Leaf),#"g",
   Node (Node (Leaf,#"e",Leaf),#"a",
   Node (Leaf,#"h",Leaf))) : char bintree

**fold_tree**

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

(* binary tree accumulation *)

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

**Binary Search Trees (BSTs) on integers**

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

(* val insert : int bintree -> int -> int bintree *)

fun insert x Leaf =
  | insert x (t as (Node(l,v,r))) =
    if x = v then ?
    else if x < v then (* using < here means that tree
elts *must* be ints *)
      ?
    else
      ?
    fun listToTree xs =

- val testbst = listToTree [4,2,3,6,1,7,5];
  val testbst = Node (Node (Node (Leaf,1,Leaf),
   2,
   Node (Leaf,3,Leaf)),
  4,
   Node (Node (Leaf,5,Leaf),
   6,
   Node (Leaf,7,Leaf))) : int bintree
Binary Search Tree membership

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

- 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 (PS8 Problem 2)

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.

Benefits of datatype and pattern matching

- SML’s datatype declaration allows concisely defining complex sum-of-product types, including trees with lots of different node types. E.g., here is a tree datatype you’ll see in PS8 Problem 4:

- SML’s pattern matching on datatype values greatly simplifies the processing of complex sum-of-product trees.

- These features make SML an ideal language for programming data structures a la CS230/CS231 and for metaprogramming (because program ASTs are just complex sum-of-product trees)