CSC/MAT-220: Lab 6

Due: 11/26/2018

In Lab 2 we discussed value and type bindings. Recall, value bindings bind a value to a variable and are intended to be static for the life of a program. Type bindings were only seen as a way to rename current data types. Consider the example below.

```
(* SML Example 1 *)
val x = 2;
type count = int;
```

Datatype Declaration

Today, we introduce the *datatype* declaration which allows us to construct a new type that is distinct from all other types. A *datatype* deceleration introduces the following

- One or more type constructors.
- One or more value constructors for each of the type constructors.

The type constructors may take zero or more arguments; a zero-argument type constructor is just a type. Each value constructor may also take zero or more arguments; a zero-argument value constructor is just a constant. For example, below is a datatype declaration that introduces a new type *suit* with four null value constructors: *Spades*, *Hearts*, *Diamonds*, and *Clubs*.

```
(* SML Example 2 *)
datatype suit = Spades | Hearts | Diamonds | Clubs
```

Following this *datatype* declaration you are able to bind values such as *Spades*, *Hearts*, *Diamonds*, and *Clubs* to a variable x of type *suit*. Note that it is conventional to capitalize the names of value constructors, but this is not required.

We can also define functions on variables of type *suit*. For instance, the following function determines whether or not the first suit outranks the other in the game of bridge.

```
(* SML Example 3 *)
val outranks : suit*suit->bool =
fn (Spades,Spades) => false
| (Spades,_) => true
| (Hearts,Spades) => false
| (Hearts,Hearts) => false
| (Hearts,_) => true
| (Diamonds,Clubs) => true
| (Diamonds,_) => false
| (Clubs,_) => false;
```

Parametrized Datatypes

In Lab 5 we worked with polymorphic functions, i.e., functions that could take on a datatype that was parametrized by *'a*. Below we give an example of a datatype declaration that is parametrized by type *'a*.

```
(* SML Example 4 *)
datatype 'a option = None | Some of 'a
```

Note that *'a option* is a unary type constructor with two value constructors: *None* with no arguments and *Some* with one argument. The values of type *'a option* are

- The constant *None*,
- Values of the form *Some val*, where *val* is a value of type *'a*.

For example, some values of type *string option* are *None*, *Some "Graph"* and *Some "Tree"*.

A common use of the *'a option* datatype is to handle special cases of an evaluated expression. Consider the example below that handles division by zero.

```
(* SMLE Example 5 *)
val divide : int*int -> int option =
fn (x,0) => None
| (x,y) => Some (x div y);
```

Recursive Datatypes

As we saw in Lab 5, recursive datatypes are especially useful in a functional programming language such as SML. The example below shows how to define a recursive datatype that is analogous to the *'a list* type.

```
(* SML Example 6 *)
datatype 'a lst = Nil | :: of 'a*'a lst
```

Note that *'a lst* is a unary type constructor with two value constructors: *Nil* with no arguments and *::* with two arguments. The values of type *'a lst* are

- The constant *Nil*,
- Values of the form *'a :: 'a lst*.

Note that the above definition is identical to the *'a list* datatype.

Binary Tree

A *binary tree* is a tree in which every node has at most two children (left child and right child) and a degree of at most 3. Consider the example in Figure 1. Conventionally, a descendent of an interval node in a binary tree is called the left child or the right child of the respective internal node. For instance, in Figure 1, the node 11 has a left child of 3 and a right child of *Empty*. In this context, we would also say that 11 is the parent node of 3 and *Empty*.

It is convenient to allow for empty trees, i.e., tree with no vertices. With this in mind, we can define the datatype for binary trees as follows:

```
(* SML Example 7 *)
datatype tree = Empty | Node of tree*int*tree
```

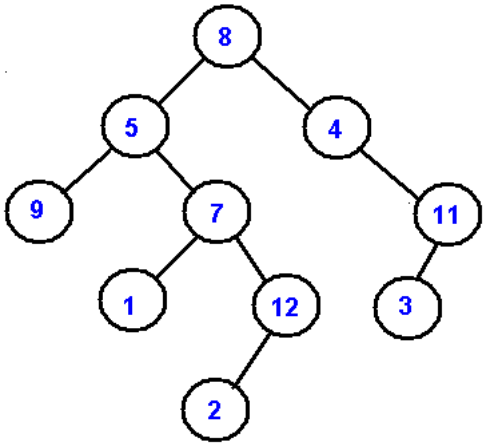

Figure 1: Binary Tree

Note that the leafs of the datatype *tree* are of the form $(Empty, int, Empty)$. As an example, we bind the right child of the node 8 in the binary tree in Figure 1 as follows:

```
(* SML Example 8 *)
val rht = Node(Empty,4,Node(Node(Empty,3,Empty),11,Empty));
```

The *height* of a binary tree is the number of edges from the top node (root of the tree) to the deepest leaf, i.e., the length of the longest $(root, leaf)$-path in the binary tree. For instance, the height of the binary tree in Figure 1 is equal to 4 and is attained by the $(8, 5, 7, 12, 2)$-path. Below is a recursive function for evaluating the height of a *tree*.

```
(* SML Example 9 *)
val rec height : tree -> int =
fn Empty => 0
| Node(Empty,_,Empty) => 0
| Node(lft,_,rht) =>
let
        val x = height(lft)
        val y = height(rht)
in
        if(x>y) then 1+x else 1+y
end;
```

Traversals

A *traversal* is a process that visits all the nodes in a tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider the three most common traversals:

- PreOrder traversal visits the parent first, then the left child, and finally the right child.
- InOrder traversal visits the left child, then the parent, and finally the right child.
- PostOrder traversal visits the left child, then the right child, and then the parent.

For example, the traversals for the binary tree in Figure 1 are below

- preOrder : $[8, 5, 9, 7, 1, 12, 2, 4, 11, 3]$,
- inOrder : $[9, 5, 1, 7, 2, 12, 8, 4, 3, 11]$,
- postOrder : $[9, 1, 2, 12, 7, 5, 3, 11, 4, 8]$.

Below is a recursive function that generates an int list corresponding to the inOrder traversal of the given tree.

```
(* SML Example 10 *)
local
        val rec helper : tree*int list -> int list =
        fn (Empty,l) => l
        | (Node(lft,n,rht),l) => helper(lft,n::helper(rht,l))
in
        val inOrder : tree -> int list =
        fn t => helper(t,nil)
end;
```

Binary Search Trees

We are now ready to consider a particular kind of binary tree called a Binary Search Tree (BST). A BST is a binary tree where the nodes are order in the following way:

- Each node contains one key (data value), for simplicity we will assume each key is an integer.
- The keys in the left child are less than the key in its parent node.
- The keys in the right child are greater than the key in its parent node.
- Duplicate keys are not allowed.

For example, consider the BST in Figure 2.

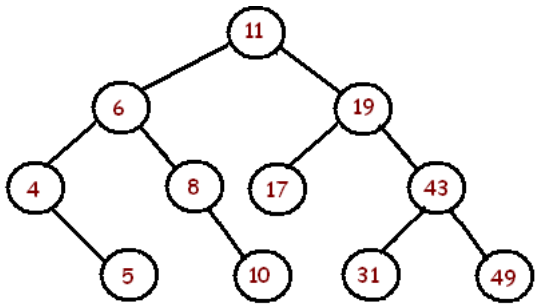

Figure 2: Binary Search Tree

Binary search trees are used often in computer science since they allow for efficient insertion and searching. Below is a recursive function that inserts an integer into a tree, assuming it is a binary search tree.

```
(* SML Example 11 *)
val rec insert : tree*int -> tree =
fn (Empty,e) => Node(Empty,e,Empty)
| (t as Node(lft,n,rht),e) =>
if e<n then Node(insert(lft,e),n,rht)
else (if e=n then t else Node(lft,n,insert(rht,e)));
```

Assignment

At the top of your file include a commented line with your name, lab number, and date. In addition, be sure to comment generously throughout so I can follow your steps. Finally, when you are done upload your source code to Dropbox. The file you you turn in should be labeled as follows: *yourname_labnumber.sml*.

I. The datatype *tree* is a type with value constructors *Empty* and *Node*. How many arguments do these value constructors take on? What are the values of type *tree*?

II. Bind the left child of the node 8 in the binary tree in Figure 1 to the variable lft. Then, use lft and rht from SML Example 8 to bind the whole binary tree to the variable *bin_tree*.

III. The *size* of a binary tree is the number of nodes (vertices) in the tree. Write a function that evaluates the size of a given variable of type tree and bind to the variable size. Test both the height function (from the reading) and the size function by computing the height and size of the binary tree in Figure 1.

IV. Write a function for the preOrder and postOrder traversal of a binary tree. Note that you should be able to slightly modify the function in SML Example 10.

V. Write a function that performs a search on a binary search tree and returns true if the target is found, and false otherwise. Store the binary search tree from Figure 2 as the variable *bst*. Then perform three searches on *bst*, at least one search should be for a value not in the binary search tree.