CSC/MAT-220: Discrete Structures

Lab 8: ML

December 4, 2017

Datatype Declarations. In Lab 2 we introduced value and type bindings. We saw that these bindings allowed us documentation, but really had no affect on the behavior of the program. In contrast, the *datatype* declaration provides a means of introducing a new type that is distinct from all other types and does not merely stand for some other type. A *datatype* declaration introduces the following

i. One or more type constructors.

ii. One or more new 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 take zero or more arguments; a zero-argument value constructor is just a constant. The type and value constructors introduced by the declaration are new in the sense that they are distinct from all other type and value constructors. If a datatype re-defines an old type or value constructor, then the old definition is shadowed by the new one, rendering the old ones inaccessible in the scope of the new definition.

For examples, below is a datatype declaration that introduces a new type *suit* with four null value constructors: *Spades, Hearts, Diamonds, and Clubs.*

$$\textit{datatype suit} = \textit{Spades} \mid \textit{Hearts} \mid \textit{Diamonds} \mid \textit{Clubs}$$

Following this *datatype* declaration you can do a value binding, such as *val* $x =$ *Hearts*, which will create a variable x of type *suit* with value *Heart.* Note that it is conventional to capitalize the names of value constructors, but this is not required by the language.

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.

$$\begin{aligned}
&\textit{val outranks} \ : \ \textit{suit} * \textit{suit} \ \rightarrow \ \textit{bool} \ = \\
&\quad fn\ (Spades, Spades) \ => \ false \\
&\quad \mid (Spades, \_) \ => \ true \\
&\quad \mid (Hearts, Spades) \ => \ false \\
&\quad \mid (Hearts, Hearts) \ => \ false \\
&\quad \mid (Hearts, \_) \ => \ true \\
&\quad \mid (Diamonds, Clubs) \ => \ true \\
&\quad \mid (Diamonds, \_) \ => \ false \\
&\quad \mid (Clubs, \_) \ => \ false
\end{aligned}$$

Just as functions can be polymorphic, so to can datatypes. For instance, the following datatype declaration is parametrized by type $'a$.

$$\textit{datatype 'a option} = \textit{NONE} \mid \textit{SOME of 'a}$$

Here we are putting our value constructors in all caps to shadow how they appear in the built in Option structure of SML. 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 *typ option* are

i. The constant *NONE*, and

ii. Values of the form *SOME val*, where *val* is a value of type *typ*.

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

As noted above, the option type constructor is pre-defined in Standard ML. One common use of option types is to handle functions with an optional argument. For example, the function below computes the base b exponential for natural numbers and defaults to $b = 2$ when *NONE* is supplied as the first argument.

```
val rec exp : int option * int → int =
      fn (NONE, n) => exp(SOME 2, n)
      | (SOME b, 0) => 1
      | (SOME b, n) =>
            if n mod 2 = 0 then
                  exp(SOME (b * b), n div 2)
            else
                  b * exp(SOME b, n − 1)
```

Another use is to handle special cases of the evaluated expression associated with a function. For example, the function below handles division by zero.

```
val divide : int * int → int option =
      fn (x, 0) => NONE
      | (x, y) => SOME (x div y)
```

Other uses of option include in aggregate data structures where certain entries may be *NONE*. For instance, a cell phone number or email address in an application entry may be left blank.

Recursive Datatypes. The next level of generality is the recursive type definition. One interesting example of a recursive datatype is the natural numbers.

$$\textit{datatype nat} = \textit{Zero} \mid \textit{Succ of nat}$$

The values of type *nat* are

i. The constant *Zero*, and

ii. Values of the form Succ *nat*

The value constructor *Succ* creates the next *nat* value based on the previous value. For instance, the bindings *val x = Zero* and *val y = Succ x* creates the first and second natural number, respectively. Since *nat* is a recursive data structure, it is natural (pun intended) to define recursive functions over variables of type *nat*. The function below doubles the value of a given natural number.

```
val rec double : nat → nat =
    fn Zero => Zero
    | Succ n => Succ (Succ (double n))
```

Another, far more useful, example of a recursive data structure is the *binary tree*, a tree in which every node has degree at most two. Conventionally, a descendent of an internal node in a binary tree is called the left child or the right child of the respective internal node. As we saw in class, trees are very easy to define in ML. Below is a datatype declaration for a binary tree.

```
datatype 'a tree =
    Empty |
    Node of 'a tree * 'a * 'a tree
```

Note that *'a tree* is a unary type constructor with value constructors *Empty* and *Node*. How many arguments do these value constructors take on? What are the value of type *'a tree*?

As an example, consider the binary tree below.

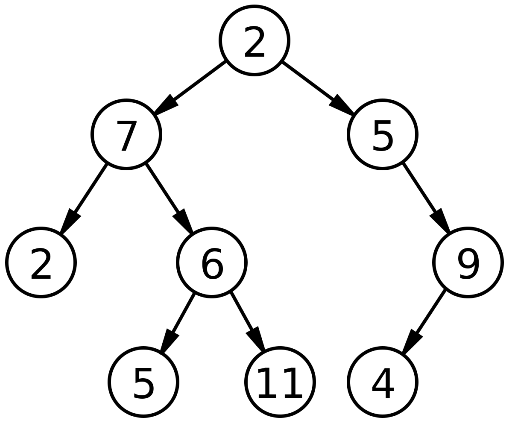


Figure 1: Binary Tree

The following is an expression for binding the right side of the binary tree in Figure 1 to the variable *rht*

$$val\ rht = Node(Empty,5,Node(Node(Empty,4,Empty),9,Empty))$$

On your own, write an expression for the left side of the binary tree in Figure 1 and bind to the variable *lft*, and then use the left and right sides to bind the entire binary tree to the variable *tree*.

The *height* of a binary tree is the number of edges from the top node (root of the tree) to the deepest leaf. The *size* of a binary tree is the number of nodes in the tree. Below is a recursive function for evaluating the height of a tree.

$$\begin{aligned}&val\ rec\ height\ :\ 'a\ tree\ \rightarrow\ int\ =\\ &\quad fn\ Empty\ \ =>\ 0\\ &\quad |\ Node(Empty, \_, Empty)\ \ =>\ 0\\ &\quad |\ Node(lft, \_, rht)\ \ =>\ 1 + Int.max(height\ lft, height\ rht)\end{aligned}$$

In a similar fashion, write a function for evaluating the size of a binary tree, bind the function to the variable *size*. Then, use the function *height* and *size* to compute the height and size of the binary tree in Figure 1.

Trees are so often used in computer science, since the posses the following advantages.

i. Trees reflect structural relationships in data.

ii. Trees are used to represent hierarchies.

iii. Trees provide for efficient insertion and searching.

iv. Trees are very flexible data, allowing to move subtrees with minimum effort.

Traversals. A *traversal* is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. Traversals can be grouped into two kinds: depth-first and breadth-first. We will only consider depth-first traversals, of which there are three types:

i. PreOrder traversal : visit the parent first and then the left and right children.

ii. InOrder traversal : visit the left child, then the parent and the right child.

iii. PostOrder traversal : visit the left child, then the right child, and then the parent.

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

$$\begin{aligned}&val\ rec\ inOrder\ :\ 'a\ tree * \ 'a\ list\ \rightarrow\ 'a\ list\ =\\ &\quad fn\ (Empty, l)\ \ =>\ l\\ &\quad |\ (Node(lft, n, rht), l)\ \ =>\ inOrder(lft, n :: inOrder(rht, l))\end{aligned}$$

Applying the function *inOrder* to the tree from Figure 1 evaluates to the following list: [2, 7, 5, 6, 11, 2, 5, 4, 9]. As another example, consider the binary tree below.

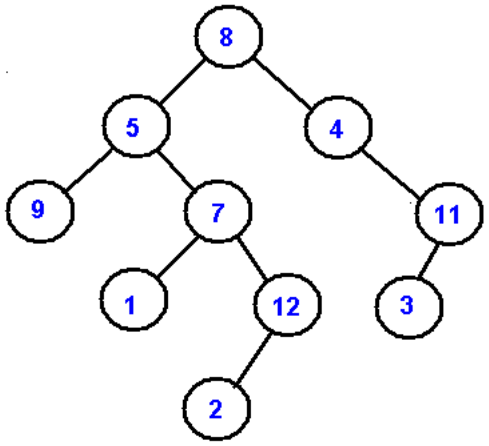


Figure 2: Binary Tree

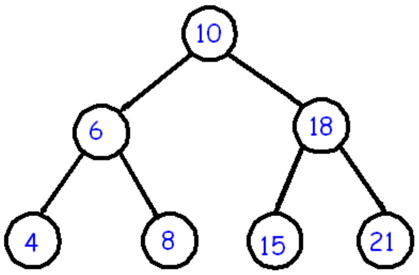
The traversals on the tree from Figure 2 are as follows

- 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].

As a challenging review, write recursive functions for *preOrder* and *postOrder* and apply all traversals to the binary tree *tree2* from Figure 2.

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

- each node contains one key (data value),
- the keys in the left subtree are less than the key in its parent node,
- the keys in the right subtree are greater than the key in its parent node,
- duplicate keys are not allowed.



It is clear from the definition of a BST, that the data stored in each node must be comparable; that is, we need an ordering. We saw in Lab 7 that it is possible to provide additional context so that we can construct polymorphic functions. While this is possible, we are content with performing the remaining expressions with integer value nodes in mind. Our first function *mkBST* makes a BST from a list of integers with no duplicates.

```
local
    val rec divide : int * int list → int list * int list =
        fn (,nil) => (nil, nil)
        | (x, h :: t) =>
            let
                val (low, high) = divide(x, t)
            in
                if x < h then (low, h :: high) else (h :: low, high)
            end
in
    val rec mkBST : int list → int tree =
        fn nil => Empty
        | (h :: t) =>
            let
                val (x, y) = divide(h, t)
            in
                Node(mkBST x, h, mkBST y)
            end
end
```

In the above function the head of the list acts as the nod in the tree, which we use as a pivot to divide the data structure into two independent parts. We place the smaller items in the left sub-tree and larger items in the right sub-tree. We can test this function by applying a traversal to the tree in Figure 2 *tree2*. This results in a list from which we can make a BST using the function *mkBST*. Then, we can check our results by performing an inOrder traversal, which should result in an ordered list.

We should point out that it is common to start with a tree that has duplicates, see Figure 1, and desire to make a BST. In this case you will need to delete duplicates, which requires consideration of what edges are removed and how to re-join the tree. This is outside of the scope for this lab, but is worthwhile for you to look into as I have seen questions of this nature on coding interviews.

Assuming that we are working with trees with no duplicates, then we are capable of making a BST and the following function will perform a search on this BST and return a bool depending on if the target is found or not.

```
$val\ rec\ search\ :\ int\ tree * int\ \rightarrow\ bool\ =$
    $fn\ (Empty, t)\ => \ false$
    $|\ (Node(lft, n, rht), t)\ =>$
        $if\ t < n\ then\ search(lft, t)$
        $else\ if\ n < t\ then\ search(rht, t)$
        $else\ true$
```

Test this function by performing a search on the BST representation of the tree in Figure 2. Perform at least one search on an entry that is not in the tree.