15-150 Spring 2012
Lab 4
8 February 2012

1 Introduction

The goal for this lab is to make you more comfortable writing functions that operate on trees.

Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the git repository to get the files for this week's lab as usual by running

$ git pull

from the top level directory (probably named 15150).

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.
2 Depth

Recall the definition of trees from lecture:

\[
datatype \text{tree} = \text{Empty} \\
| \text{Node} \ (\text{tree} \ast \text{int} \ast \text{tree})
\]

As with any \text{datatype}, we can \text{case} on a \text{tree} like so:

\[
case \ t \ of \\
Empty \Rightarrow ... \\
| \ Node \ (l, \ x, \ r) \Rightarrow ...
\]

Intuitively, the depth of a tree is the length of the longest path from the root to a leaf. More precisely, we define the depth of a tree inductively: the depth of \text{Empty} is 0; the depth of \text{Node}(l,x,r) is one more than the larger of the depths of its two children \text{l} and \text{r}.

\textbf{Task 2.1} Define the function

\[
\text{depth} : \text{tree} \rightarrow \text{int}
\]

that computes the depth of a tree.

\textit{Hint:} You will probably find the function \text{max} : \text{int} \ast \text{int} \rightarrow \text{int}, which we have provided for you, useful.

3 Lists to Trees

For testing, it is useful to be able to create a tree from a list of integers. To make things interesting, we will ask you to return a \textit{balanced} tree: one where the depths of any two leaves differ by no more than 1.

\textbf{Task 3.1} Define the function

\[
\text{listToTree} : \text{int list} \rightarrow \text{tree}
\]

that transforms the input \text{list} into a balanced tree. \textit{Hint:} You may use the \text{split} function provided in the support code, whose spec is as follows:

\text{If l is non-empty, then there exist l1,x,l2 such that} \\
\text{split l == (l1,x,l2) and} \\
l == l1 @ x::l2 and \\
\text{length(l1) and length(l2) differ by no more than 1}
4 Reverse

Recall the function treeToList from lecture, which computes an in-order traversal of a tree:

```haskell
fun treeToList (t : tree) : int list =
  case t of
    Empty => []
  | Node (l,x,r) => treeToList l @ (x :: (treeToList r))
```

Observe that treeToList is total.

In this problem, you will define a function to reverse a tree, so that the in-order traversal of the reverse comes out backwards:

\[
\text{treeToList } (\text{revT } t) \approx \text{reverse } (\text{treeToList } t)
\]

Code

Task 4.1 Define the function

\[\text{revT} : \text{tree} \rightarrow \text{tree}\]

according to the above spec.

Task 4.2 Explain why revT is total.

Have the TAs check your code for reverse before proceeding!

Analysis

Task 4.3 Determine the recurrence for the work of your revT function, in terms of the size (number of elements) of the tree. You may assume the tree is balanced.

Task 4.4 Use the tree method to write a closed form for the recurrence, in terms of a sum.

Task 4.5 Solve the sum (it should be one we have discussed previously in the course).

Task 4.6 Use the closed form to determine the big-O of \(W_{\text{revT}}\).

Task 4.7 Determine the recurrence for the span of your revT function, in terms of the size of the tree. You may assume the tree is balanced.

Task 4.8 Use the tree method to give a closed form for this recurrence.

Task 4.9 Use the closed form to give a big-O for \(S_{\text{revT}}\).
Correctness

Prove the following:

**Theorem 1.** For all values $t :$ tree, $\text{treeToList (revT t)} \cong \text{reverse (treeToList t)}$.

You may use the following lemmas about $\text{reverse}$ on lists:

- $\text{reverse []} \cong []$

- For all valuable expressions $l$ and $r$ of type int list,

  $$\text{reverse (l @ (x::r))} \cong (\text{reverse r}) @ (x::(\text{reverse l}))$$

In your justifications, be careful to prove that expressions are valuable when this is necessary. Follow the template on the following page.
Case for Empty
To show:

Case for Node(1, x, r)
Two Inductive hypotheses:

To show:
5 Binary Search

At this point, it behooves us to introduce another of SML’s built-in datatypes: order. order is a very simple datatype—it has precisely three values: GREATER, EQUAL, and LESS, and is defined as follows:

```
datatype order = GREATER | EQUAL | LESS
```

As you may have guessed, order represents the relative ordering of two values. At present, we care only about the relative ordering of ints. SML provides a function Int.compare : int * int -> order which compares two ints and calculates whether the first is GREATER than, EQUAL to, or LESS than the second respectively. This allows us to implement tri-valued comparisons, as follows:

```
case Int.compare (x1, x2) of
    GREATER => (* x1 > x2 *)
    | EQUAL => (* x1 = x2 *)
    | LESS => (* x1 < x2 *)
```

**Task 5.1** Define the function

```
binarySearch : tree * int -> bool
```

that, assuming the tree is sorted, returns true if and only if the tree contains the given number. Your implementation should have work and span proportional to the depth of the tree. You should use Int.compare, rather than <, in your solution.