15-150 Fall 2013
Lab 4

18 September 2013

1 Introduction

The goal for the 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 weeks 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 Warmup: Depth

Recall the definition of trees from lecture:

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

As with any `datatype`, we can `case` on a `tree` like so:

```
case t of
    Empty => ...
  | Node (l, x, r) => ...
```

Similarly, we could pattern match for a clausal function like so:

```
fun foo (Empty : tree) : t = ...
  | foo (Node (l,x,r)) = ...
```

Intuitively, the depth (or height) 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 `Empty` is 0; the depth of `Node`$(\texttt{l}, \texttt{x}, \texttt{r})$ is one more than the larger of the depths of its two children `l` and `r`.

Task 2.1 Define the function

```
depth : tree -> int
```

that computes the depth of a tree.

Hint: You will probably find the function `max : int * int -> 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 *balanced* tree: one where the depths of any two leaves differ by no more than 1.

Task 3.1 Define the function

```
listToTree : int list -> tree
```

that transforms the input `list` into a balanced tree. *Hint:* You may use the `split` function provided in the support code, whose spec is as follows:

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

4 Reverse

Here is the `trav` function from lecture, which computes an in-order traversal of a tree:

```
fun trav (Empty : tree) : int list = []
  | trav (Node (l,x,r)) = trav l @ (x :: (trav r))
```

Observe that `trav` will always return a value given an input (it will not recurse indefinitely).

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

$$\texttt{trav (revT t)} = \texttt{reverse (trav t)}$$

Code

Task 4.1 Define the function

```
revT : tree -> tree
```

according to the above spec.

Task 4.2 Explain why `revT` always returns a value given some input.

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 closed form to determine the big-O of $W_{\texttt{revT}}$.

Task 4.5 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 4.6 Use the closed form to give a big-O for $S_{\texttt{revT}}$.

Correctness

Prove the following:

Theorem 1. *For all values* `t : tree`, `trav (revT t)` = `reverse (trav t)`.

You may use the following lemmas about `reverse` on lists:

- `reverse [] = []`
- For all expressions l and r of type `int list` such that l and r both reduce to values,

```
reverse (l @ (x::r)) = (reverse r) @ (x::(reverse l))
```

In your justifications, be careful to prove that expressions evaluate to values when this is necessary. Follow the template on the following page.

Case for `Empty`
To show:

Case for `Node(l,x,r)`
Two Inductive hypothes**es**:

To show:

Have the TAs check your analysis and proof before proceeding!

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 `int`s. SML provides a function `Int.compare : int * int -> order` which compares two `int`s 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.