15-150 Spring 2012
Lab 4

8 February 2012

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 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) => ...
```

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 `Empty` is 0; the depth of `Node(l, x, 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

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

```
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:

$$\texttt{treeToList (revT t)} \cong \texttt{reverse (treeToList t)}$$

Code

Task 4.1 Define the function

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

according to the above spec.

Task 4.2 Explain why `revT` is total.

Solution 4.2 `revT` is recursive on the structure of trees.

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).

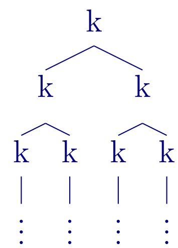
Task 4.6 Use the closed form to determine the big-O of $W_{\texttt{revT}}$.

$\boxed{\textbf{Solution 4.6}}$ We will use the definition of `revT` given in `lab04-sol.sml` file. We will determine the work, $W_{revT}(n)$, based on the number n of elements in the tree t. We assume that the tree is balanced so the work of the evaluation of each recursive call in `Node (revT t2, x , revT t1)` takes at most $W_{revT}(n/2)$ steps.

Thus the recurrence for the work of `revT` is:

$$
\begin{aligned}
W_{revT}(0) &= k_0 \\
W_{revT}(n) &= k + 2W_{revT}(n/2)
\end{aligned}
$$

Using the tree method to solve the recurrence



k

$2k$

$4k$

$2^i k$
where i is the # of levels
away from the root

The tree having $(\log_2 n) + 1$ steps gives us the following summation

$$
\begin{aligned}
W_{revT}(n) &= \sum_{i=0}^{\log_2 n} 2^i k \\
&= k \cdot \sum_{i=0}^{\log_2 n} 2^i \\
&= k(2n - 1) \\
W_{revT}(n) &\in O(n)
\end{aligned}
$$

Unlike the definition of `mergesort` which required a linear amount of work between recursive calls, this recurrence only has a constant amount of work between recursive calls. Therefore, it makes sense that we found that $W_{revT}(n)$ is in $O(n)$.

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 $S_{\texttt{revT}}$.

> $\boxed{\textbf{Solution 4.9}}$ We will determine the span, $S_{revT}(n)$, based on the number n of elements in the tree t. We assume that the tree is balanced so the span of the evaluation of each recursive call in `Node (revT t2, x , revT t1)` is at most $S_{revT}(n/2)$. As we are determining the span, we take the max of these two values. This gives the following recurrence:
>
> $$S_{revT}(0) = k_0$$
> $$S_{revT}(n) = k + W_{revT}(n/2)$$
>
> Using the tree method to solve the recurrence
>
> ```
> k            k
> |
> k            k
> |
> k            k
> |
> ⋮            k
>              at all levels
>              away from the root
> ```
>
> The tree having $(\log_2 n) + 1$ steps gives us the following summation
>
> $$\begin{aligned} W_{revT}(n) &= \sum_{i=0}^{\log_2 n} k \\ &= k(1 + \log_2 n) \\ W_{revT}(n) &\in O(\log_2 n) \end{aligned}$$
>
> This recurrence only has a constant number of steps between recursive calls and therefore it makes sense that we found that $S_{revT}(n)$ is in $O(\log n)$.

Correctness

Prove the following:

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

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

- `reverse []` $\cong$ `[]`
- For all valuable expressions l and r of type `int list`,

  `reverse (l @ (x::r))` $\cong$ `(reverse r) @ (x::(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(l,x,r)`
Two Inductive hypothes**es**:

To show:

Have the TAs check your analysis and proof before proceeding!

$\boxed{\textbf{Solution 4.9}}$ **Case for** `Empty`

To show: `treeToList (revT Empty)` ≅ `reverse(treeToList Empty)`

Proof:

```
  treeToList (revT Empty)
≅treeToList (case Empty of Empty => Empty
                          | Node(t1,x,t2)=> ...)          Step - Empty is a value
≅treeToList (Empty)                                        Step
≅case Empty of Empty => [] | Node (l,x,r) => ...           Step
≅[]                                                        Step
≅reverse []                                                By Lemma
≅reverse (case Empty of Empty => [] | Node (l,x,r) => ...) Step
≅reverse (treeToList Empty)                                Step
```

Thus `treeToList (revT Empty)` ≅ `reverse(treeToList Empty)`.

Case for `Node(l,x,r)`

Two Inductive hypothes**es**:

`treeToList (revT l)` ≅ `reverse(treeToList l)`

`treeToList (revT r)` ≅ `reverse(treeToList r)`

To show: `treeToList (revT Node(l,x,r))` ≅ `reverse(treeToList Node(l,x,r))`

Proof:

```
 treeToList (revT Node(l,x,r))
≅treeToList (case Node(l,x,r) of
                Empty => ...
              | Node(l,x,r)=>
                Node (revT r, x , revT l))              Step - Node(l,x,r) is a value
≅treeToList (Node (revT r, x , revT l))                 Step
≅case (Node (revT r, x , revT l)) of
   Empty => ...
 | Node (revT r,x,revT l) =>
   treeToList (revT r) @ (x :: (treeToList (revT l)))  Step - revT r and revT l
                                                        are valuable since revT
                                                        is total
≅treeToList (revT r) @ (x :: (treeToList (revT l)))    Step
≅treeToList (revT r) @ (x :: reverse(treeToList l))    By IH 1
≅reverse(treeToList r)) @ (x :: reverse(treeToList l)) By IH 2
≅reverse(treeToList l @ (x :: treeToList r)            By Lemma, treeTolist l
                                                        and treeToList r are
                                                        valubale since treeTolist
                                                        is total
≅reverse(case (Node (l,x,r)) of
            Empty => ...
          | Node (l,x,r) =>
            treeToList l @ (x :: treeToList r)         Step
≅reverse(treeToList Node(l,x,r))                        Step
```

Thus, `treeToList (revT Node(l,x,r))` ≅ `reverse(treeToList Node(l,x,r))`

By induction, `treeToList (revT t)` ≅ `reverse (treeToList t)`, for all values `t : tree`.

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.