15–150: Functional Programming

Practice Midterm

- There are 13 pages in this examination, comprising 5 questions worth a total of 87 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- You may refer to one double-sided 8.5" x 11" page of notes, but to no other person or source, during the examination.
- Your answers for this exam must be written in blue or black ink.
- Unless otherwise indicated, you do not need to give purpose/examples/tests for functions.
- In multi-part coding questions, we will assume the helper functions are correct while grading the later tasks. So it is in your interest to attempt the later tasks, even if you don't solve the earlier ones.

Full Name: ______________________________

Andrew ID: ______________________________

Question:	Basics	Huffman Encoding	Analysis	Proof	Regexp	Total
Points:	4	25	22	28	8	87
Score:						

THIS PAGE IS FOR SCRATCH WORK ONLY

Question 1 [4]: Basics

(a) (2 points) Consider the following piece of code:

```
val x = "hi "
val f = fn y => y ^ x
val x = "there "
val z = f x
```

After this is evaluated, to what value is the variable `z` bound?

> **Solution:** `z = "there hi"`

(b) (2 points) Consider the following piece of code:

```
fun f(x : int, y : int option) =
    case y of
         SOME x => true
       | _ => false
```

What is the value of `f(2, SOME 3)` ?

> **Solution:** The answer is `true`.
> The inner `x` is a different variable, so it steps to
>
> ```
> case SOME 3 of SOME x => true | _ => false
> ```
>
> not
>
> ```
> case SOME 3 of SOME 2 => true | _ => false
> ```

Question 2 [25]: Huffman Encoding

Tree Paths

(a) (8 points) Recall trees:

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

The following definitions can be used to represent *paths* in a tree:

```
datatype direction = Left | Right
type path = direction list
```

A path represents directions to a leaf in a tree: each `direction` tells you which subtree to go into as you walk down `Node`s from the root of the tree. For example, the path

```
[Left,Right]
```

directs to `Leaf "licorice"` in the tree

```
Node (Node (Leaf "candy",
            Leaf "licorice"),
      Leaf "popcorn")
```

Question contintues on the next page

Write a function

```
findPath : ('a -> bool) -> 'a tree -> path option
```

If there is an element `x` of `t` satisfying `p`, then `findPath p t` returns `SOME path`, where `path` is a path to `Leaf x`.[1] Otherwise, it returns `NONE`.

```
fun findPath (p : 'a -> bool) (t : 'a tree) : path option =
```

Solution:

```
fun findPath (p : 'a -> bool) (t : 'a tree) : path option =
  case t of
      Empty => NONE
    | Leaf x => (case p x of
                   true => SOME []
                 | false => NONE)
    | Node (l,r) =>
          (case findPath p l of
                SOME lpath => SOME (Left :: lpath)
              | NONE => (case findPath p r of
                            NONE => NONE
                          | SOME rpath => SOME (Right :: rpath)))
```

[1] If there is more than one such element, you may return a path to any one of them.

Huffman Trees

Huffman encoding is a technique for compressing a document. The *frequency* of a word is the number of times it occurs in the document. Suppose you have a prediction about frequencies of the words in the document—the word "the" will appear many times, the word "perpendicular" fewer times, and so on. To compress the document, you can replace each word with a *code*, where frequently-used words are given short codes and less-frequently-used words are given longer codes.

We use a *frequency list* to represent the frequencies of words:

```
type 'a freqlist = (int * 'a) list
```

For example:

```
val freqs : string freqlist =
  [(156, "cow"), (60, "perpendicular"), (43, "sleepless"), (3156,"money")]
```

represents a document where "cow" appears 156 times, etc.

Given the frequencies of words in a document, we represent the coding scheme by a *Huffman tree*. For example, the Huffman tree for `freqs` is:

```
val ht : string tree =
 Node(Node (Node (Leaf "sleepless",
                  Leaf "perpendicular"),
            Leaf "cow"),
      Leaf "money")
```

A Huffman tree has each word in the document at a leaf. The more frequently used a word is, the closer it is to the root. Given a Huffman tree, the code of a word is the path to its leaf. For example:

Word	**Code**
money	`[Right]`
cow	`[Left,Right]`
sleepless	`[Left,Left,Left]`
perpendicular	`[Left,Left,Right]`

(b) (3 points) Define a function `encode` that, given a Huffman tree and a word, returns `SOME <the word's code>` if the word is in the tree, or `NONE` otherwise. You may not define this recursively: use `findPath`.

```
fun encode (t : string tree) (s : string) : path option =
```

Solution:

```
fun encode (t : string tree) (s : string) : path option =
    findPath (fn s' => s = s') t
```

There is a simple algorithm for building a Huffman tree, starting from the frequencies of each word in the document. First, we write a helper function `build`:

```
build : (string tree) freqlist -> string tree
```

This function's argument is a frequency list whose items are Huffman trees, where the frequency of a tree is the sum of the frequencies of all of the words in that tree.

To build a Huffman tree, we merge the trees in the frequency list as follows: At each step, extract the two trees with the lowest frequency, join the two trees into one tree, and insert this new tree back into the frequency list with the appropriate frequency. When there is only one tree left in the queue, return it.

For example,

```
build [(156,Leaf "cow"), (60,Leaf "perpendicular"),
       (43,Leaf "sleepless"), (3156,Leaf "money")]
```

evaluates to

```
Node(Node (Node (Leaf "sleepless",
                 Leaf "perpendicular"),
           Leaf "cow"),
     Leaf "money")
```

You may assume a function

```
extractMin : 'a freqlist -> (int * 'a) * 'a freqlist
```

that extracts the item with the smallest frequency, or raises an exception if the list is empty. If the input is not empty, then `extractMin fs` returns the minimum frequency in `fs`, the[2] item associated with it, and the frequency list without this (frequency, item) pair. For example:

```
extractMin freqs
  ==>
((43,"sleepless"), [(156,"cow"),(60,"perpendicular"),(3156,"money")])
```

QUESTION CONTINUES ON THE FOLLOWING PAGE

[2]If there is more than one item with the same frequency, it returns the first one.

(c) (10 points) Write the function `build`:

```
fun build (forest : (string tree) freqlist) : string tree =
```

Solution:

```
fun build (forest : (string tree) freqlist) : string tree =
    case forest of
        [] => Empty
      | [(p,t)] => t
      | _ => (* has at least two elements *)
            let val ((min1,t1), forest1) = extractMin forest
                val ((min2,t2), forest2) = extractMin forest1
            in
                build ((min1 + min2, Node(t1,t2)) :: forest2)
            end
```

(d) (4 points) Next, we write

```
huffman : string freqlist -> string tree
```

`huffman fs` should call `build`, transforming `fs` into an appropriate format by creating a single-element tree for each word, and pairing this tree with the appropriate frequency. You may not define this recursively; use a higher-order function on lists!

```
val huffman : string freqlist -> string tree =
```

Solution:

```
val huffman : string freqlist -> string tree =
  build o (List.map (fn (p,v) => (p,Leaf v)))
```

Question 3 [22]: Analysis

The following function adds one to each element of tree:

```
fun add1 (t : int tree) : int list =
    case t of
        Empty => Empty
      | Leaf x => Leaf (x+1)
      | Node (l,r) => Node(add1 l, add1 r)
```

In this problem, you will analyze `add1`. Let n be the size of t; you may assume that n is a power of 2 and that the tree is balanced (this is implies there are no `Empty`'s in the tree).

(a) (5 points) Give a recurrence for the work of `add1`. Your answer should be exact, except you may use constants $k_0, k_1, \ldots$ to stand for constant numbers of steps of evaluation:

$W_{\texttt{add1}}(0) =$

$W_{\texttt{add1}}(1) =$

$W_{\texttt{add1}}(n) =$

Solution:

$W_{\texttt{add1}}(0) = k_0$

$W_{\texttt{add1}}(1) = k_1$

$W_{\texttt{add1}}(n) = k_2 + 2 * W_{\texttt{add1}}(\frac{n}{2})$

(b) (4 points) Use the tree method to give a closed form for $W_{\texttt{add1}}$.

Solution: Let k be the max of k_1 and k_2. The tree looks like

```
      k
  k       k
k   k   k   k
...
```

Because we do constant work at each node, and there are as many nodes as there are `Node`/`Leaf`s in the input tree, the total work is bounded by kn.

(c) (2 points) Give a tight big-O bound for the work of `add1`.

$W_{\texttt{add1}}(n)$ is

Solution: $W_{\texttt{add1}}(n)$ is $O(n)$.

(d) (5 points) Give a recurrence for the span of `add1`. Your recurrence should be exact, except you may use constants k_0, k_1, ... to stand for constant numbers of steps of evaluation:

$S_{\texttt{add1}}(0) =$

$S_{\texttt{add1}}(1) =$

$S_{\texttt{add1}}(n) =$

Solution:

$S_{\texttt{add1}}(0) = k_0$

$S_{\texttt{add1}}(1) = k_1$

$S_{\texttt{add1}}(n) = k_2 + S_{\texttt{add1}}(\frac{n}{2})$

(e) (4 points) Use the tree method to give a closed form for $S_{\texttt{add1}}$.

Solution: Let k be the max of k_1 and k_2. The tree looks like

```
k
k
...
```

and there are $\log n$ levels, so the closed form is $k \log n$.

(f) (2 points) Give a tight big-O bound for the span of `add1`:

$S_{\texttt{add1}}(n)$ is

Solution: $S_{\texttt{add1}}(n)$ is $O(\log n)$.

Question 4 [28]: Proof

Note, Spring 2012: The proof question on last year's midterm was the map fusion problem we used this year on Homework 5. We suggest you try to do this proof yourself, without looking at Homework 5, and then use the Homework 5 solutions to check your work. Make sure you understand how valuability and totality are used in the justifications.

Theorem 1. *For all types `a`, `b`, `c`, all values `f : a -> b` and `g : b -> c`, if `f` and `g` are total, then*

$$\texttt{(map g) o (map f)} \cong \texttt{map (g o f)} : \quad \texttt{a list -> c list}$$

(a) (28 points) Prove Theorem 1.

Question 5 [8]: Regexp

(a) (8 points) Define

$$L(\texttt{TimesN}(r_1, r_2)) \;=\; \{cs \;\mid \exists p_1, p_2 \text{ where } p_1 @ p_2 \Longrightarrow cs \text{ and } p_1 \in L(r_1) \text{ and } p_2 \in L(r_2) \text{ and } \exists n \text{ such that } \texttt{length}(p_1) \Longrightarrow n \text{ and } \texttt{length}(p_2) \Longrightarrow n\}$$

That is, $\texttt{TimesN}(r_1, r_2)$ matches a string in the language of r_1 followed by a string in the language of r_2, where both strings have the same length.
For example, $aabb \in L(\texttt{TimesN}(a^*, b^*))$, but $aabbb$ is not.

Extend `match` with a case for `TimesN`:

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
   case r of
       ...
     | TimesN(r1,r2) =>
```

Solution:

```
| TimesN (r1,r2) =>
     match r1 cs
      (fn s1 => match r2 s1
                 (fn s2 =>
                   inteq(length cs - length s1, length s1 - length s2)
                                 andalso k s2))
```

In one sentence or doodle, explain the soundness of your solution: why does it ensure that p_1 and p_2 have the same length?

Solution: Because in any call `match r cs (fn s => ...)`, `length cs - length s` is the length of the prefix.