

Name: _____

COSC 8, Fall 2008

Midterm Exam

October 23, 2008

| | | | | | | |
|----------------|----|----|----|----|----|--------------|
| Problem | 1 | 2 | 3 | 4 | 5 | Total |
| Grade | | | | | | |
| Points | 20 | 25 | 15 | 25 | 15 | 100 |
| Grader | | | | | | |

General Instructions

There are five (5) problems on this exam. Please check now that you have a complete exam booklet with nine (9) numbered pages. Please write your name on *each page* of the exam.

Be sure to look at *all* of the problems. Some are more difficult than others. Don't waste a lot of time on a problem that is giving you a hard time—if you get stuck, move on to another problem, and then return to it later.

There is space provided to answer each question, and you may request extra paper if you so desire. If you do use extra paper to record your solutions, be sure to write your name prominently on each extra page, and clearly indicate which problem is being answered there. You do not need to turn in any extra scratch-paper you use while working out your solutions; only the pages with your answers on them need to be submitted.

To help insure that you don't accidentally miss any of the questions, each section where an answer is requested has been marked with a \Leftarrow symbol in the right margin.

Please write neatly—the course staff reserves the right to ignore illegible answers. When writing solutions that involve program code, proper indentation of your code is important!

This examination covers material through Lecture 12.

| |
|---|
| <p>You may not consult any books, notes, study-guides, or other outside material during the exam period, except for one sheet of 8.5 by 11 inch notebook paper which you are allowed to make hand-written notes on (both sides). We will provide a list of functions and definitions that appear in Chapter 23.</p> |
|---|

1. (20 points) **Principal types.** Give the principal type for each of the following functions. Justify your answer. Give a non-trivial example input and output. Each part is 4 points for the principal type, 3 points for the justification, and 2 points for the example.

(a) `f = zipWith (++) ["cat", "dog"]`



(b) `f x ys = (head x, map (x :) ys)`




2. (25 points) **Sorting.**

This problem explores two methods for sorting a list.


(a) (5 points) Insertion into a sorted list. Write a function

```
insert :: Ord a => a -> [a] -> [a]
```

that takes an item and a list sorted in increasing order and inserts the item in the correct place in the list to keep it in increasing order. 

(b) (5 points) Insertion Sort. Write a function

```
insertSort :: Ord a => [a] -> [a]
```

that takes a list and insertion sorts it into increasing order, using the `insert` function written in part a. You may assume that `insert` works as specified, whether your code for part a is correct or not. 

(c) (15 points) Sorting using Data.Set. The Set ADT includes the following operations:

```
empty :: Set a                -- Returns an empty Set
null  :: Set a -> Bool        -- Tests if a Set is empty
insert :: Ord a => a -> Set a -> Set a -- Inserts an item into a Set
findMin :: Set a -> a         -- Returns the smallest item in the Set
deleteMin :: Set a -> Set a   -- Deletes the smallest item in the Set
```

Use these operations to sort a list of items. First create a set containing all of the items and then repeatedly find and remove the smallest item. The type signature for the function that performs these operations should be:

```
setSort :: Ord a => [a] -> [a]
```

Assume that Data.Set is imported as follows:

```
import qualified Data.Set as Set
```



3. (15 points) **Higher-Order Functions.** Solve the following problem using higher-order functions. For full credit, your solution may not be recursive. You may use any of the built-in functions described in the Haskell Quick Reference.

In PS 2 you wrote a Digraph module. It included the following type definitions:

```
type AdjList v e = [(v, e)]
type Digraph v e = [(v, AdjList v e)]
type Edge v e    = (v, v, e)
```

and the following function (among others):

```
-- Create a digraph out of a list of vertices and a list of edges
makeDigraph      :: (Eq v) => [v] -> [Edge v e] -> Digraph v e
```

Write a function that takes a digraph as its parameter and returns a digraph with every edge reversed. That is, the new digraph has an edge from v_2 to v_1 with label e if and only if the original digraph has an edge from v_1 to v_2 with label e . Thus if the function is called on the digraph

```
[('a', [(('b', 0), ('c', 1))], ('b', []), ('c', [(('b', 2))])]
```

it should return the digraph

```
[('a', []), ('b', [(('a', 0), ('c', 2))], ('c', [(('a', 1))])]
```

Your solution should call `makeDigraph` with appropriate arguments in order to build the graph.

```
reverseAllEdges :: (Eq a) => Digraph a b -> Digraph a b
```



4. (20 points) **Short Answer.**

(a) (5 points) We considered the following version of `reverse`:

```
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```

While it yields the correct answer, it is not the one used in the Standard Prelude. In what way is it inferior to the one in the Standard Prelude? Explain how this inferiority arises.



(b) (6 points) Explain “currying” and “the currying simplification”.



(c) (4 points) In the program that computes motifs, we looked at functions:

```
-- First attempt at scoring: score each letter in s against distribution
-- at matching position in m.
score1 :: Motif -> String -> Score
score1 m s = product (zipWith scoreLetter s m)

-- First attempt at seeing how well the motif matches at different
-- places in s: march down s and score each suffix
scores1 :: Motif -> String -> [Score]
scores1 m s = mapTail (score1 m) s
```

where `scoreLetter` returns the appropriate score for letter `s` in the `ScoreDistrib m`. The goal of `scores1` is to get a list of scores, where position k in the returned list is to be the score of motif `m` starting in position k of string `s`. Unfortunately, this code gives the wrong scores for some positions at the end of the list. Why? ⇐

(d) (4 points) In PS 2 we defined an Automaton type using field labels as follows:

```
data Automaton a b = FA {graph :: Digraph a b, start :: a, final :: [a]}
    deriving Show
```

If we re-write this definition without using field labels as follows:

```
data Automaton a b = FA (Digraph a b) a [a]
    deriving Show
```

we no longer have the accessor functions `graph`, `start`, and `final`. Write these three functions, assuming the second definition of `Automaton`. ⇐

(e) (6 points) Consider the following type signature:

$f :: a \rightarrow [a \rightarrow a] \rightarrow a$

Write three functions that have this type signature. The three functions must be different, in the sense that no two of them give the same results for all input. (Hint - the problem does not require the functions to be interesting or useful. They just have to have the right type signature.)



5. (15 points) **Trees.** In class (and in the book) a tree with data in the leaves is defined:

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

Write a function `filterTree pred t`, which takes as parameters a predicate `pred` and a tree `t`, and returns a tree whose leaves are the leaves of `t` that satisfy the predicate. For each leaf `Leaf x` in the input tree, if `pred x` returns `True` that leaf should occur in the tree returned.

If all of the leaves satisfy the predicate, then `filterTree pred t` should return a tree with the same tree structure as `t`. However, if a leaf fails to satisfy the predicate then that leaf must be eliminated, and the shape of the tree must change. We want to keep the shape of the tree close to the shape of the original tree, and in particular we want the leaves that remain to be in the same order that they appeared in the original tree. To be more formal, we want `fringe (filterTree pred t) == filter pred (fringe t)`. (Remember that `fringe` makes an ordered list of the data in the leaves.)

What if none of the leaves satisfy the predicate? Then `filterTree` should return an empty tree, and we don't have any way to represent an empty tree. Therefore the function will have return type `Maybe (Tree a)`. That way if the tree is empty `filterTree` will return `Nothing`.

Consider the tree:

```
t = Branch (Branch (Leaf 1) (Leaf 3)) (Branch (Leaf 4) (Leaf 2))
```

Then

```
filterTree (>0) t =>
  Just (Branch (Branch (Leaf 1) (Leaf 3)) (Branch (Leaf 4) (Leaf 2)))
filterTree (>1) t => Just (Branch (Leaf 3) (Branch (Leaf 4) (Leaf 2)))
filterTree (>2) t => Just (Branch (Leaf 3) (Leaf 4))
filterTree (>3) t => Just (Leaf 4)
filterTree (>4) t => Nothing
```

Write a recursive function to solve the problem. You should not call `fringe` or any higher-order functions.

```
filterTree :: (a -> Bool) -> Tree a -> Maybe (Tree a)
```

