

COSC 8, Fall 2008

**Midterm Exam**  
SAMPLE SOLUTIONS

October 23, 2008



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"]` ←←

**Solution:** `f :: [[Char]] -> [[Char]]`

*Explanation:*

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c].`

`(++) :: [d] -> [d] -> [d]`

The first parameter of `zipWith` is `++`, so `a`, `b`, and `c` all map to `[d]`, giving:

`zipWith (++) :: [[d]] -> [[d]] -> [[d]].`

The second parameter of `zipWith` is of type `[[Char]]`, so `d`  $\mapsto$  `Char`.

The third parameter is not supplied, so we have curried `zipWith` to a single-parameter function from `[[Char]] -> [[Char]]`, and

`f :: [[Char]] -> [[Char]]`

*Example:*

`f ["egory", "-eared"] => ["category", "dog-eared"]`

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

**Solution:** `g :: [a] -> [[[a]]] -> (a, [[[a]]])`

*Explanation:*

`head :: [a] -> a.` Therefore the type of `x` is `[a]`.

`(:) :: b -> [b] -> [b]`

Therefore in `(x :)` we have `b`  $\mapsto$  `[a]` and thus

`(x :) :: [[a]] -> [[a]].`

`map :: (c -> d) -> [c] -> [d]`, so `c`  $\mapsto$  `[[a]]`; `d`  $\mapsto$  `[[a]]`

This means that `y` is of type `[c]`, which is `[[[a]]]`, and the return type is of type `[d]`, which is `[[[a]]]`. We conclude that

`g :: [a] -> [[[a]]] -> (a, [[[a]]])`.

*Example:*


`g [1,2] [[[3,4],[5,6]],[[7,8]]] =>  
(1, [[[1,2],[3,4],[5,6]],[[1,2],[7,8]]])`

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. 

**Solution:**

```
insert x [] = [x]
insert x (y : ys) | x < y      = x : y : ys
                  | otherwise = y : insert x ys
```

(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. 

**Solution:**

```
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)
```

or

```
insertSort lst = foldr insert [] lst
```

(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
```



**Solution:**

```
setSort xs = sortHelper initSet
  where initSet = foldr Set.insert Set.empty xs
        sortHelper s = if Set.null s then []
                       else Set.findMin s : sortHelper (Set.deleteMin s)
```

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
```



**Solution:**

```
reverseAllEdges g = makeDigraph (map fst g) (getReversedEdges g) where
  reverseEdgesForVertex (src, adjList) =
    map (\(dest, label) -> (dest, src, label)) adjList
  getReversedEdges = concat . map reverseEdgesForVertex
```

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. ⇐

**Solution:** This version takes time  $O(n^2)$  for a list of length  $n$ . The problem is that `++` takes time proportional to the length of its first argument. Because we are always appending to the end the top level of the recursion does  $n - 1$  “`:`” operations, the next level down does  $n - 2$ , and so on. The sum of the times at all levels is  $O(n^2)$ .

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

**Solution:** Currying is providing one or more arguments to a multi-parameter function to produce a new function with fewer parameters. Each argument supplied fixes a parameter, and the function is a function of the remaining parameters.

The currying simplification is dropping a parameter in a function definition when the same parameter appears in exactly two places: as the last parameter on the left side of the definition and as the last argument of the function call on the right side of the definition. For instance,

```
sum lst = foldl (+) 0 lst
```

can be simplified to

```
sum = foldl (+) 0
```

The simplification can be repeated to remove multiple parameters.

(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? ⇐

**Solution:** The problem is the `zipWith` in `score1`. When its last two arguments have different lengths it ignores the extra part of the longer string. That is what is desired when the string is longer than the motif, but when the motif is longer than the string it matches the first places and ignores the fact that the last places don't exist. The motif cannot match a shorter string.

(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`. ⇐

**Solution:**

```
graph (FA g _ _) = g
start (FA _ s _) = s
final (FA _ _ f) = f
```

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

```
f :: a -> [a -> a] -> 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.)



**Solution:**

```
f1 x _ = x
```

```
f2 x [] = x
```

```
f2 x (f : fs) = f x
```

```
f3 x lst = foldr ($) x lst
```

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)
```



**Solution:**

```
filterTree pred (Leaf x) = if pred x then Just (Leaf x) else Nothing
filterTree pred (Branch left right) =
  case ((filterTree pred left), (filterTree pred right)) of
    (Nothing, rt) -> rt
    (lt, Nothing) -> lt
    (Just l, Just r) -> Just (Branch l r)
```