COMP 212: Computer Science II
Homework 8: Higher-order functions

In this assignment, you will gain some experience working with functions as data. In the written work, you'll see that we can still prove higher-order functions correct using proof by induction; you just have to be comfortable treating function expressions just like any other expressions. In particular, some of your reasoning will involve expressions of function type that are *not* identifiers; remember, those are just expressions like any other, and there is no real difference in reasoning about a complex (non-identifier) expression of function type than there is in reasoning about a complex (non-identifier) expression of any other type. At the same time, reasoning about higher-order functions also provides a nice setting to explore some of the assumptions we have made in our proofs of correctness. For example, one such assumption is that our computation does not have side-effects. You will see how this assumption is critical by showing how the equation that you prove is correct in one problem is not correct in the presence of side-effects.

For the coding part of this assignment, you will practice using the standard list functionals by re-implementing the Bayesian analysis code you wrote in a previous assignment. When you look back at it, that assignment consisted of many list processing functions. You will rewrite all of them using the standard list functionals. You will also show how to take advantage of Currying by re-designing the `makeClassifier` function to "stage" the computation involved in training the classifier, which will allow us to train a classifier once, and use it repeatedly to classify multiple documents.

1. Written problems (10 points)

Problem 1. *For this problem, refer to the implementation of* `map` *in Program 1. Prove that*

$$\texttt{map}(g, \texttt{map}(f, xs)) = \texttt{map}(g \circ f, xs)$$

by induction on the length of xs*. The operator* $\circ$ *is the function composition operator, which is specified by:*

$$\begin{array}{rcl} \circ & : & (\beta \to \gamma) * (\alpha \to \beta) \to (\alpha \to \gamma) \\ (g \circ f)(x) & = & g(f(x)) \end{array}$$

ML defines `o` *(the letter little-oh) to be the function composition operator (which is why it is a bad idea to use* `o` *as an identifier). You may* not *use the specification of* `map` *for this problem; you must use its implementation only. You may use the specification for* $\circ$*.*

Problem 2. *We have heard in class that our proofs are valid provided there are no side-effects as a result of computation. An exception is one kind of side-effect.*

(a) *Come up with ML functions* `f` *and* `g` *and a list* `xs` *such that*

$$\texttt{map(g, map(f, xs))} \neq \texttt{map(g o f, xs)}$$

More precisely, the evaluation of the left-hand side has a different result than that of the right-hand side. Use ML evaluation rules to evaluate both sides. One way to accomplish this is for `f` *and/or* `g` *to raise exceptions (probably in a rather contrived way).*

(b) *What assumption have we been making in our proofs that does not hold when we consider ML evaluation in the presence of exceptions?*

```
fun map(f : 'a -> 'b, xs : 'a list) : 'b list =
  case xs of
       [] => []
     | y :: ys => f(y) :: map(f, ys)
```

Program 1: The `map` functional.

2. Coding problems (10 points)

For the coding project, re-implement the Bayesian analysis code from HW 4 using list functionals. You must implement (almost) the same functions as in HW 4. You must start with the posted solutions. The skeleton file provided in the code distribution also has non-curried versions of many standard list functionals which you may use if you wish. However, for maximum credit you must use the curried list functionals from the `List` structure such as `filter`, `map`, `foldr`, `foldl`, etc.

The one significant difference from HW 4 is that the `makeClassifier` function is different. Its type is now

```
makeClassifer :
    (string list) * ((category*doc) list) -> (doc -> (category*real) list)
```

`makeClassifier`(K, cds) is a function cl, where $cl(d)$ is (a list representing) the function that maps each category c that appears in cds to `computeLL`$(K, d, cps(c))$, where cps = `makeCtyProbs`(`count`(K, cds))). To be clear: cl is an actual function, whereas $cl(d)$ is a list that represents a function. For full credit, `makeClassifier`(K, cds) must compute cps and then return the function cl. The reason is that computing cps involves computing the category probability distributions from the training data, and does not depend on the document that is (eventually) analyzed. In practice, computing cps is typically much more time-intensive than using cps to analyze a given document. Since cps can be computed from just K and cds, the idea is to compute it once in order to define cl, then use cl on many different documents to (more quickly) classify each one. This technique of designing `makeClassifier` to do some computation (train the classifier), and then return a function that does more computation (classify a given document) is called *staging* computation.

To get a sense as to whether staging the computation in this setting really makes a difference, Figure 1 shows the results of running the HW 4 and HW 8 drivers on the `books` directory that was distributed with HW 4 (the two different implementations of `makeClassifier` report their results in different orders, but that is OK.) With the HW 4 driver (for which the computation is not staged), it takes no time at all to build the classifier (remember, it is just a function declaration), but the time to classify all the documents is about $5,000$ milliseconds. With the HW 8 driver (for which the computation is staged), it takes about $1,500$ milliseconds to build the classifier, but only 882 milliseconds to classify the all the documents. There is a net decrease of about 2.7 seconds in total execution time when the computation is staged, and the amount by which the total time would decrease would only get larger as more documents are analyzed.

3. Going further

You need not do any of the work described in this section.

In Problem 2 you found two functions `f` and `g` such that

$$\texttt{map(g, map(f, xs))} \neq \texttt{map(g o f, xs)}$$

Homework 4 driver:

```
$ ./driver books/keywords.txt books/training books/unknown
Building classifier...done (0 ms.).
Computing probabilities...done (5120 ms.).
alger-young-explorer.txt:
        adams: ~18965.573508
        ...

Total time: 5120 ms.
```

Homework 8 driver:

```
$ ./driver books/keywords.txt books/training books/unknown
Building classifier...done (1531 ms.).
Computing probabilities...done (882 ms.).
alger-young-explorer.txt:
        ...

Total time: 3258 ms.
```

Figure 1: Showing the difference in execution time when staging computation.

in the sense the the result of evaluating the left-hand side is different than the result of evaluating the right-hand side. That might leave you a little unsatisfied, because when you are done, you might realize why I have phrased what I mean by the inequality very carefully. However, you should be able to then come up with a function `h` such that `h(map(g, map(f, xs)))` $= 0$ but `h(map(g o f, xs))` $= 1$.

For the coding project, you should start seeing that even using list functionals, you really end up implementing the "same" functions multiple times, just with different types. So you could write some generic polymorphic functions using list functionals, and then implement the other functions in terms of those.

4. Code distribution, submission, and grading criteria

The code distribution contains testing code and a driver implementation, just as in Homework 4. You may use the `books` directory from Homework 4 for testing your code and using the driver.

Submit `hw8.pdf` (written solutions) and `hw8.sml` (code solutions).

When grading, we will look for the following:

- For written work Problem 1, a clear and well-written proof by induction; for Problem 2, a clear example and explanation of the assumption that is violated, along with a description of how your proof for Problem 1 breaks down when applied to your example.
- For the programming work, all the functions must be implemented without any explicit recursive function definitions; use list functionals such as `foldr`, `map`, etc. Keep in mind that appropriately using list functionals means that the arguments are not themselves explicitly recursive. For example, implementing a function `f` using `filter0` by writing something like

  ```
  f(xs) =
    filter0(..., xs)
  ```

  where ... calls `f` is *not* an appropriate use of `filter0`. Of course, ... might call other previously-defined functions. For full credit, use the versions in the `List` structure in the

SML Standard Basis Library and use anonymous functions (`fn` expressions) everywhere appropriate. You might also find the `valOf` function in the `Option` structure occasionally helpful. For this assignment, you may also sparingly use the selector functions such as `#1`, `#2`, etc.; remember that these are ordinary functions in ML.