University of Arizona, Department of Computer Science

CSc 372 — Assignment 3 — Due noon, Thu Oct 6 — 5%

Christian Collberg
October 5, 2011

1 Introduction

The purpose of this assignment is to learn to program using higher-order functions.

The following rules apply for this assignment:

1. **Every function you write (except when explicitly noted) should be *non-recursive.* I.e. functions will typically be implemented using higher-order functions from the standard prelude, such as `maps`, `folds`, `zips`, etc.**
2. You may freely introduce auxiliary functions if that makes your program cleaner. In particular, you should introduce local definitions (`where`-clauses) to make your code easier to read.
3. You will be graded primarily on correctness and style, not on the execution efficiency of your code.
4. All functions (**except when explicitly noted) should use Haskell's *pattern* or *guard* syntax, *not* the `if-then-else` syntax!**
5. Functions written for one problem may be freely used in subsequent problems. In fact, this is encouraged!
6. All functions must be commented. At the very least, each function should start out with a description of what it does, what input parameters it takes, what output it produces, and an example of how it is invoked.
7. All functions (except local ones introduced by `where` must have a function signature.
8. You *cannot* change the signatures of any of the functions — they should be *exactly* as specified below!
9. Start by picking up the template Haskell script file from the course website.

The first part of the assignment (**A**) you should solve individually. The second part (**B**) you can work on in teams of two.

NOTE: By Monday Sep 26, email the TAs with the names of the members of your group.

You should hand in three files (`ass3A.hs ass3B.hs TEAM`), one for each part, and one file that lists the names and CS IDs of the students in your team. Only one team member needs to hand in `ass3B.hs`.

2 A: Individual Problems

To get used to programming with the higher-order functions from the standard prelude, work these problems by yourself!

1. Write a function `maxl xs` that generates an error `"empty list"` if `xs==[]` and otherwise returns the largest element of `xs`: [1 point]

   ```
   maxl :: (Ord a) => [a] -> a
   maxl xs = ...

   > maxl [2,3,4,5,1,2]
   5
   > maxl []

   Program error: empty list
   ```

2. Write a function `mull xs m` which returns a new list containing the elements of `xs` multiplied by `m`: [2 points]

   ```
   mull :: (Num a) => [a] -> a -> [a]
   mull xs x = ...

   > mull [1,2,3,4,5] 0.0
   [0.0,0.0,0.0,0.0,0.0]
   > mull [1,2,3,4,5] 2
   [2,4,6,8,10]
   > mull [2.0,4.0,6.0,8.0,10.0] 5
   [10.0,20.0,30.0,40.0,50.0]
   ```

3. In the definition of `sumc` below, use *function composition* to compute the sum of the cubes of all the numbers divisible by 7 in a list `xs` of integers. [2 points]

   ```
   sumc :: [Int] -> Int
   sumc =
      where cube x = ...
            by7  x = ...

   > sumc [7]
   343
   > sumc [7,14]
   3087
   > sumc [7,8,14]
   3087
   ```

NOTE: In this case, I want the definition of `sumc` to look exactly as above.

4. Implement a function `siever n xs` which returns the elements of `xs` that are not multiples of `n`: [2 points]

```
> siever 2 []
[]
> siever 2 [2..10]
[3,5,7,9]
> siever 3 [2..10]
[2,4,5,7,8,10]
```

NOTE: Use a *lambda-expression* in your implementation.

5. Re-implement the set-manipulation functions from assignment 2, but this time don't use recursion! You should implement, at least, these functions: [8 points]

```
makeSet        :: Ord a => (a -> a-> Ordering) -> [a] -> [a]
isSet          :: Ord a => (a -> a-> Ordering) -> [a] -> Bool
member         :: Ord a => (a -> a-> Ordering) -> a -  > [a] -> Bool
setIntersect   :: Ord a => (a -> a-> Ordering) -> [a] -> [a] -> [a]
setUnion       :: Ord a => (a -> a-> Ordering) -> [a] -> [a] -> [a]
setSubtract    :: Ord a => (a -> a-> Ordering) -> [a] -> [a] -> [a]
setIsSubset    :: Ord a => (a -> a-> Ordering) -> [a] -> [a] -> Bool
setSimilarity  :: Ord a => (a -> a-> Ordering) -> [a] -> [a] -> Double
```

NOTE: Each of these functions (except for `setSimilarity`) must use at least one of the higher-order functions from the standard prelude, such as `filter`, `map`, `zip`, `foldr`, etc. In assignment 2, `setSimilarity` generated an error if both the sets given to it were empty. This time, instead let `setSimilarity` return 0.0. You can use the sort functions you implemented in assignment 2, or, if you didn't get them to work, use the sort functions provided by Haskell.

6. Using **only** list comprehension (and, if you wish `if-then-else`) write a function `row n v` which returns a list of `n` integers, all 0:s, except for the `v`:th element, which should be a 1 (you don't have to test for when `n` or `v` are out of range): [4 points]

```
row :: Int -> Int -> [Int]
> row 0 0
[]
> row 5 1
[1,0,0,0,0]
> row 5 2
[0,1,0,0,0]
> row 5 3
[0,0,1,0,0]
> row 5 4
[0,0,0,1,0]
> row 5 5
[0,0,0,0,1]
```

Using **only** set comprehension and the function `row` from above, write a function `idmatrix n` which returns a square identity matrix of size `n`. I.e., `idmatrix` should return a list of lists, where all elements are 0:s, except for the elements on the diagonal, which are all 1:

```
idmatrix :: Int -> [[Int]]

> idmatrix 0
[]
> idmatrix 1
[[1]]
> idmatrix 2
[[1,0],
 [0,1]]
> idmatrix 3
[[1,0,0],
 [0,1,0],
 [0,0,1]]
> idmatrix 4
[[1,0,0,0],
 [0,1,0,0],
 [0,0,1,0],
 [0,0,0,1]]
> idmatrix 5
[[1,0,0,0,0],
 [0,1,0,0,0],
 [0,0,1,0,0],
 [0,0,0,1,0],
 [0,0,0,0,1]]
```

NOTE: Your output will not look like this, I've just printed each row on it's own line to make it easier to read.

7. Implement functions to perform arithmetic on *Peano Numbers*. You can read more about Peano Axioms here: `http://en.wikipedia.org/wiki/Peano_axioms`. The idea is very simple: the number 0 is represented by `Zero`, and all the following integers are represented by 1+ the previous integer. "1+" is called `Succ`. Here are some examples: [6 points]

$$\begin{aligned} 0 &= \texttt{Zero} \\ 1 &= \texttt{Succ Zero} \\ 2 &= \texttt{Succ(Succ Zero)} \\ 3 &= \texttt{Succ(Succ(Succ Zero))} \\ 4 &= \texttt{Succ(Succ(Succ(Succ Zero)))} \\ 5 &= \texttt{Succ(Succ(Succ(Succ(Succ Zero))))} \end{aligned}$$

Here is the Haskell recursive data type for natural numbers represented by the Peano Axioms:

```
data Nat =
   Zero |
   Succ Nat
   deriving Show
```

Implement the following functions for conversion to and from "normal" integers, addition, multiplication, and comparisons for equality and less than:

```
toPeano :: Int -> Nat
fromPeano :: Nat -> Int
addPeano :: Nat -> Nat -> Nat
mulPeano :: Nat -> Nat -> Nat
peanoEQ :: Nat -> Nat -> Bool
peanoLT :: Nat -> Nat -> Bool
```

Here are some examples:

```
> toPeano 0
Zero
> toPeano 1
Succ Zero
> toPeano 2
Succ (Succ Zero)
> toPeano 3
Succ (Succ (Succ Zero))
> toPeano 4
Succ (Succ (Succ (Succ Zero)))
> toPeano 5
Succ (Succ (Succ (Succ (Succ Zero))))
> fromPeano Zero
0
> fromPeano (Succ (Succ (Succ (Succ Zero))))
4
> addPeano Zero Zero
Zero
> addPeano (Succ (Succ (Succ (Succ Zero)))) (Succ Zero)
Succ (Succ (Succ (Succ (Succ Zero))))
> fromPeano (addPeano (toPeano 5) (toPeano 6))
11
> mulPeano (toPeano 4) (toPeano 2)
Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
> fromPeano (mulPeano (toPeano 4) (toPeano 5))
20
> mulPeano Zero (Succ Zero)
Zero
> peanoEQ Zero Zero
True
> peanoEQ Zero (Succ Zero)
False
> peanoLT Zero (Succ Zero)
True
> peanoLT (toPeano 10) (toPeano 11)
True
```

NOTE: In these functions you are only allowed to use *structural recursion* over the recursive data type. For example, implementing `addPeano m n` as `toPeano ((fromPeano m) + (fromPeano n))` will give you zero points.

HINT: You can use `addPeano` in the implementation of `mulPeano`.

3 B: TurnItOn

In this assignment you will write a clone of `TurnItIn` — the system used by English professors to catch plagiarism of term papers. Later in the course you'll see how a variant of this algorithm (known as MOSS) can be used to catch cheating on programming assignments.

Comparing[1] sets of k-grams of two documents is a popular method of computing their similarity. This idea has been used for plagiarism detection of text documents and source code, for authorship analysis of code, and for birthmark detection of executable code. A k-gram is a contiguous length k substring of the original document. To illustrate the idea, consider the short document `A` consisting of the string `yabbadabbadoo`:

```
    0  1  2  3  4  5  6  7  8  9  10 11 12
A:  y  a  b  b  a  d  a  b  b  a  d  o  o
```

By sweeping a window of size 3 over `A` you get the set of 3-grams for `A`:

```
A: yab abb bba bad ada dab abb bba bad ado doo
```

We'll also call these *shingles*. Here's a second document `C` consisting of the string `doobeedoobeedoo`:

```
    0  1  2  3  4  5  6  7  7  9  10 11 12 13 14
C:  d  o  o  b  e  e  d  o  o  b  e  e  d  o  o
```

The shingles for this document are:

```
C: doo oob obe bee eed edo doo oob obe bee eed edo doo
```

What are the similarities between `A` and `C`? Well, that's easy to see; just compare the sets of shingles! In this case, the only shingle that appears both in `A` and `C` is `doo`: it occurs once in `A` and three times in `C`. This could be a fluke, of course, but it could also be that we've just uncovered a case of plagiarism where `doo` was copied three times from `A` into `C`.

How well this will work in practice depends on if you can choose a good value for k. If, in the previous example, you'd have set $k = 4$ you wouldn't have seen *any* similarity between `A` and `C`:

```
A: yabb abba bbad bada adab dabb abba bbad bado adoo

C: doob oobe obee beed eedo edoo doob oobe obee beed eedo edoo
```

How you choose k depends on the type of document you're working on. In general, you need to choose k such that common idioms of the document type have length less than k. This will filter out incidental similarities such as the word *the* in English or *if* in source code.

The most interesting property of k-grams is that they're somewhat insensitive to permutations. Say that, in an effort to thwart you from detecting his attempts at plagiarism, the adversary reorders `yabbadabbadoo` into `bbadooyabbada`. From this obfuscated document you now get this set of shingles:

[1] The following text has been shamelessly plagiarized from the (totally awesome) book *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, by Christian Collberg and Jasvir Nagra, Addison-Wesley, 2009. Thus, this is a case of *self-plagiarism* which you can read about in the (equally totally awsome) article *Self-plagiarism in computer science* by Christian Collberg and Stephen Kobourov, *Communications of the ACM*, Volume 48, Issue 4 (April 2005).

```
A': bba bad ado doo ooy oya yab abb bba bad ada
```

Notice how the `doo` 3-gram still remains intact. This means that you'll compute the same rate of similarity with document `C` as before!

In practice it will be too inefficient to compute and store all the shingles of a large document, or, in the case of plagiarism detection, a large set of documents. Instead the shingles are first hashed and then a small subset of them are kept for further analysis. For ease of exposition, in the examples in the remainder of this section we'll use the following function to compute hashes from shingles:

$$
hash(s) = \left\{ \begin{array}{lcrlcrlcr}
\texttt{obe} & \Rightarrow & 15 & \texttt{abb} & \Rightarrow & 2 & \texttt{bee} & \Rightarrow & 16 \\
\texttt{bba} & \Rightarrow & 3 & \texttt{bad} & \Rightarrow & 4 & \texttt{yab} & \Rightarrow & 8 \\
\texttt{ydo} & \Rightarrow & 14 & \texttt{eed} & \Rightarrow & 17 & \texttt{byd} & \Rightarrow & 13 \\
\texttt{doo} & \Rightarrow & 1 & \texttt{ada} & \Rightarrow & 5 & \texttt{edo} & \Rightarrow & 18 \\
\texttt{ado} & \Rightarrow & 7 & \texttt{coo} & \Rightarrow & 10 & \texttt{dab} & \Rightarrow & 6 \\
\texttt{oob} & \Rightarrow & 11 & \texttt{oby} & \Rightarrow & 12 & \texttt{sco} & \Rightarrow & 9
\end{array} \right.
$$

Using this function you can create a hash for each of the shingles in document `A` above:

A:	yab	abb	bba	bad	ada	dab	abb	bba	bad	ado	doo
	A_0:8	A_1:2	A_2:3	A_3:4	A_4:5	A_5:6	A_6:2	A_7:3	A_8:4	A_9:7	A_{10}:1

The notation we use here is $D_P : V$, where D is the name of the document from whence the shingle came, P is the shingle's position within D, and V is its value.

There will be approximately the same number of hashes as there are tokens in the original document. It therefore becomes impractical to keep more than a small number of them. A common approach is to only keep those that are $0 \bmod p$, for some p. This has the disadvantage that there could be long gaps in a document from which no hash is selected. A better approach is to use a technique known as *winnowing*. The idea is to sweep a window of size W over the sequence of hashes and choose the smallest one from each window (in case of ties, choose the rightmost smallest). This ensures that there's no gap longer than $W + K - 1$ between two selected hashes. Here are the windows of size 4 for document `A` above, where we've marked the selected hashes in gray:

(	A_0:8	A_1:2	A_2:3	A_3:4	)
(	A_1:2	A_2:3	A_3:4	A_4:5	)
(	A_2:3	A_3:4	A_4:5	A_5:6	)
(	A_3:4	A_4:5	A_5:6	A_6:2	)
(	A_4:5	A_5:6	A_6:2	A_7:3	)
(	A_5:6	A_6:2	A_7:3	A_8:4	)
(	A_6:2	A_7:3	A_8:4	A_9:7	)
(	A_7:3	A_8:4	A_9:7	A_{10}:1	)

The final set of hashes chosen from `A` then becomes {A_1:2, A_2:3, A_6:2, A_{10}:1}.

3.1 Utility functions

We'll start by implementing some useful functions that we will need later on.

1. Write a function `windows k xs` which generates a list of sublists of length `k`, by sweeping a window across the list `xs`: [5 points]

   ```
   windows :: Int -> [a] -> [[a]]

   > windows 3 []
   []
   > windows 3 [1..10]
   [[1,2,3],[2,3,4],[3,4,5],[4,5,6],[5,6,7],[6,7,8],[7,8,9],[8,9,10]]
   > windows 2 ["a","b","c","d","e"]
   [["a","b"],["b","c"],["c","d"],["d","e"]]
   ```

   HINT: Have a look at the `commaInt` function that we discussed in class.

2. Write a function `mapAllPairs f xs` which (essentially) computes the cross product of xs with itself and then applies the function `f` to all the resulting pairs of elements. [5 points]

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

   > mapAllPairs (\ x y -> (x,y)) [1,2,3]
   [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
   > mapAllPairs (\ x y -> x+y) [1,2,3]
   [2,3,4,3,4,5,4,5,6]
   > mapAllPairs (\ x y -> x++y) ["hi","bye","!"]
   ["hihi","hibye","hi!","byehi","byebye","bye!","!hi","!bye","!!"]
   ```

   NOTE: You can use recursion and/or higher-order functions.

3. Re-implement the hash-function from assignment 2, this time without using recursion! [5 points]

   ```
   type Hash = Int
   type HashSet = [Hash]

   hash :: String -> Hash

   > hash "hello sailor!"
   49012778
   ```

3.2 Reading documents

In order to compare documents for similarity, we first need to read them in! We represent a document by its filename (a String). When store the text of the document itself as a list of the lines read, paired with the line number:

```
type DocName = FilePath
type Line = (Int,String)
type Document = [Line]
```

1. Write a function `splitLines xs` which takes string with embedded newlines and splits it into a list of `Line`:s: [5 points]

```
splitLines :: String -> Document

> splitLines ""
[]
> splitLines "aaa"
[(0,"aaa")]
> splitLines "aaa\nbbbbb\ncccccc\n"
[(0,"aaa"),(1,"bbbbb"),(2,"cccccc"),(3,"")]
```

NOTE: You can write this function recursively if you wish.

A function `loadFiles` have been provided for you. It takes a list of filenames and returns a list of tuples (file-name,file-contents), using `splitLines` to split up each file:

```
loadFiles :: [DocName] -> IO [(DocName,Document)]

> loadFiles ["fred","frank"]
[("fred",[(0,"yabbadabbadoo")]),("frank",[(0,"doobeedoobeedoo")])]
```

3.3 Computing shingles

A shingle is a substring of the original document plus the position (a (row-number,column-number) tuple) at which it occurs:

```
type Position = (Int, Int)
type Shingle = (String, Position)
```

The functions in this section compute lists of shingles from the text read from a file.

1. The function `line2shingles` extracts all the singles from a line of text by sweeping a window of size K across it. [5 points]

   ```
   line2shingles :: Int -> Line -> [Shingle]

   > line2shingles 2 (5,"yabbad")
   [("ya",(5,0)),("ab",(5,1)),("bb",(5,2)),("ba",(5,3)),("ad",(5,4))]
   > line2shingles 3 (0,"yabbadabbadoo")
   [("yab",(0,0)),("abb",(0,1)),("bba",(0,2)),("bad",(0,3)),
    ("ada",(0,4)),("dab",(0,5)),("abb",(0,6)),("bba",(0,7)),
    ("bad",(0,8)),("ado",(0,9)),("doo",(0,10))]
   ```

2. The function `shingles` uses the `line2shingles` function to extract all the shingles from a document by sweeping a window of size K over each of the lines of text. [5 points]

   ```
   shingles :: Int -> (DocName,Document) -> (DocName,[Shingle])

   > shingles 3 ("fred", [(0,"yabb"),(1,"adabba"),(3,"doo")])
   ```

```
("fred",[("yab",(0,0)),("abb",(0,1)),("ada",(1,0)),
         ("dab",(1,1)),("abb",(1,2)),("bba",(1,3)),
         ("doo",(3,0))])
```

NOTE: For readability I've sometimes re-formatted the output `ghci` produces. When you run the same function calls the output is likely to look different.

3. Given a list of shingles, the `shingles2hashSet` function converts it to a *set of hash values* using the function `hash` above. There can be multiple shingles with the same string value (and hence the same hash value) but the output of this function is a *set*, i.e. an ordered set of unique hashes. [5 points]

```
shingles2hashSet :: (DocName,[Shingle]) -> (DocName,HashSet)

> > hash "yab"
1321
> hash "abb"
1300
> shingles2hashSet ("fred",[("yab",(0,0)),("abb",(0,1)),("ada",(1,0)),
                            ("dab",(1,1)),("abb",(1,2)),("bba",(1,3)),
                            ("doo",(3,0))])
("fred",[1292,1297,1300,1321,1459])
```

3.4 Winnowing shingles

Usually, documents are large, and will result in way too many hashes being generated. *Winnowing* is any technique for reducing the number of hashes. Depending on your application you may want to use different winnowing algoriths. Every algorithm takes a list of shingles as input and returns a (usually smaller) list of shingles.

The function`winnowNone` is the identity function, returning the same list of shingles as it is given. It's been defined for you:

```
type WinnowFun = (DocName,[Shingle]) -> (DocName,[Shingle])

winnowNone :: (DocName,[Shingle]) -> (DocName,[Shingle])
winnowNone (d,xs) = (d,xs)
```

1. Write a function `winnowModP` which returns only those shingles whose hash values is 0 mod p. [5 points]

```
winnowModP :: Int -> (DocName,[Shingle]) -> (DocName,[Shingle])

> hash "abb"
1300
> hash "dab"
1300
> winnowModP 5 ("fred",[("yab",(0,0)),("abb",(0,1)),("ada",(1,0)),
                ("dab",(1,1)),("abb",(1,2)),("bba",(1,3)),("doo",(3,0))])
("fred",[("abb",(0,1)),("dab",(1,1)),("abb",(1,2))])
```

NOTE: For extra credit (5 points) you can also write a function `winnowWindow :: Int -> (DocName,[Shingle]) -> (DocName,[Shingle])` **which sweeps a window of size p across the list of shingles and, for every window, only keeps the rightmost shingle with the minimum value. (See the example in the introduction.)**

3.5 Most similar documents

Now we're almost ready to write the first main function, called `similar`, which takes n documents, compares them pairwise, and sorts them, most similar first! This function has been provided for you:

```
similar :: Int -> WinnowFun -> [DocName] -> IO [(DocName, DocName, Double)]
similar shingleSize winnowAlg fileNames = do
      docs <- loadFiles fileNames
      let w = moss shingleSize winnowAlg docs
      return w
```

`similar` starts by reading in all the files in `fileNames` and then calls the `moss` function which computes the similarity between each pair of documents. Notice that one of the arguments to `moss` is a winnowing function — this means we can choose how to winnow at runtime.

Here's an example of how to call `similar`:

```
> similar 2 winnowNone ["fred","frank"]
[("frank","frank",1.0),("fred","fred",1.0),
 ("frank","fred",0.2),("fred","frank",0.2)]
> similar 3 winnowNone ["fred","frank"]
[("frank","frank",1.0),("fred","fred",1.0),
 ("frank","fred",8.333333333333333e-2),
 ("fred","frank",8.333333333333333e-2)]
```

Notice how different window sizes give different results!

1. The function `compareAllDocs` do much of the heavy lifting. It takes a list of documents (tuples of *(docName, hash-set)*), and compares all pairs of documents for similarity. The return value is a sorted list of tuples *(DocName1,DocName2,SimilarityScore)*. *SimilarityScore* is the value returned by the `setSimilarity` function over the hash-sets of each pair of documents. The output list is sorted in descending order by *SimilarityScore*. [5 points]

   ```
   compareAllDocs :: [(DocName,HashSet)] -> [(DocName,DocName,Double)]

   > compareAllDocs [("frank",[1331,1337,1341,1353,1427,1459]),
                     ("fred",[1292,1297,1300,1316,1321,1423,1459])]
   [("frank","frank",1.0),
    ("fred","fred",1.0),
    ("frank","fred",8.333333333333333e-2),
    ("fred","frank",8.333333333333333e-2)]
   ```

HINT: Remember `mapAllPairs`?

2. So, how do we go from a document (represented as a list of strings read from a file) to a list of shingles to a set of hash-values? After all, `compareAllDocs` above works on sets of hash-values and nothing else! Well, this is the job of the `kgram` function: [5 points]

```
kgram :: Int -> WinnowFun -> (DocName,Document) -> (DocName,HashSet)
kgram windowSize winnow = ......

> kgram 3 winnowNone ("fred",[(0,"yabbadabbadoo")])
("fred",[1292,1297,1300,1316,1321,1423,1459])
> kgram 3 (winnowModP 4) ("fred",[(0,"yabbadabbadoo")])
("fred",[1292,1300,1316])
> kgram 2 winnowNone ("fred",[(0,"yabbadabbadoo")])
("fred",[398,400,401,406,421,442,453])
> kgram 2 (winnowModP 4) ("fred",[(0,"yabbadabbadoo")])
("fred",[400])
```

HINT: `kgram` simply glues together functions we've already seen, namely `shingles` (for generating the list of shingles), one of the winnowing functions (passed as argument to `kgram`), and `shingles2hashSet`.

NOTE: Use function composition to implement `kgram`! `kgram` should be declared exactly as shown above!

3. The `moss` function ties everything together by gluing together the `compareAllDocs` and `kgram` functions. We use `kgram` to compute a set of hashes for each document and `compareAllDocs` to compute, for every pair of documents, how similar they are. [5 points]

```
moss :: Int -> WinnowFun ->[(DocName,Document)] -> [(DocName,DocName,Double)]
moss windowSize winnowFun = ......

> moss 3 winnowNone [("fred",[(0,"yabbadabbadoo")]),
                     ("frank",[(0,"doobeedoobeedoo")])]
[("frank","frank",1.0),
 ("fred","fred",1.0),
 ("frank","fred",8.333333333333333e-2),
 ("fred","frank",8.333333333333333e-2)]
```

NOTE: Use function composition to implement `moss`! `moss` should be declared exactly as shown above!

3.6 Visualizing differences

Great! Professor Balthazar can now take all his students' English essays, run them through the `similar` function, and find that Alice and Bob's essays are 83% similar, and that Charlie's essay is 76% similar to the Wikipedia article on *Emily Brontë*[2]. But now what? He needs to somehow convince them that they have, in fact, plagiarized, that the evidence of their crime is overwhelming, and that it's in their best interests to

[2]Bad Alice! Bad Bob! Bad Charlie! Bad! Bad!

confess to their transgressions. If they don't confess then Professor Balthazar will have to go through tons of paperwork and will, perhaps, have to take the case all the way to the Dean[3]

So, what should we do? We need to visualize the similarities between Alice and Bob's files! This is done by the `visualize` function, which has been provided for you:

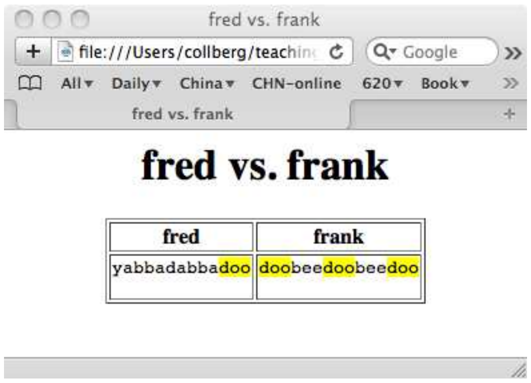
```
visualize :: Int -> DocName -> DocName -> IO()
visualize shingleSize doc1name doc2name = do
          let outfileName = doc1name ++ "-" ++ doc2name ++ ".html"
          [doc1,doc2] <- loadFiles [doc1name,doc2name]
          let shingles1 = shingles shingleSize doc1
          let shingles2 = shingles shingleSize doc2
          let common = commonStrings shingles1 shingles2
          let regions1 = shingles2regions shingles1 common
          let regions2 = shingles2regions shingles2 common
          let html = showSimilarities doc1 regions1 doc2 regions2
          writeFile outfileName html
```

`visualize` reads in two files, computes their similarities (by comparing the set of shingles), computes an html string visualizing the similarities, and writes this string to a file. We can then load this file into a web browser!

The function call

```
> visualize 3 "fred" "frank"
```

generates a file `fred-frank.html`. Here's what it looks like:



fred vs. frank

fred	frank
yabbadabbadoo	doobeedoobeedoo

To do the visualization we will be manipulating sets of *Region* tuples:

```
type Region = (Position,Int)
```

A `Region` is a tuple (pos,len) that represents a segment of a file starting in position pos (a (row,column) tuple) and extending for len characters. For simplicity, a Region never extends past the end of a line.

[3]And, it's well-known that the Dean plays golf with Charlie's Dad at the countryclub every Sunday. Not good now that Professor Balthazar's promotion case is coming up for review.

The function `inRegion` has been provided for you:

```
inRegion :: Position -> Region -> Bool
inRegion (row,col) ((r,c),len) = row==r && col>=c && col<(c+len)

> inRegion (5,2) ((5,0),1)
False
> inRegion (5,2) ((5,0),2)
False
> inRegion (5,2) ((5,0),3)
True
> inRegion (5,2) ((4,1),99)
False
```

1. Write a function `inRegions` which takes a position `pos` and a list of `Regions` `listOfRegions` as arguments and returns True if `pos` is within any of the regions in `listOfRegions`: [5 points]

   ```
   inRegions :: Position -> [Region] -> Bool

   > inRegions (5,2) [((5,0),2)]
   False
   > inRegions (5,2) [((5,0),2),((5,1),1)]
   False
   > inRegions (5,2) [((5,0),2),((5,1),2)]
   True
   ```

2. The function `commonStrings` computes the set of all shingles (well, the *strings* of the shingles, to be exact) that two documents have in common: [5 points]

   ```
   commonStrings :: (DocName,[Shingle]) -> (DocName,[Shingle]) -> [String]

   > let fred = shingles 3 ("fred",[(0,"yabbadabbadoo")])
   > fred
   ("fred",[("yab",(0,0)),("abb",(0,1)),("bba",(0,2)),("bad",(0,3)),
            ("ada",(0,4)),("dab",(0,5)),("abb",(0,6)),("bba",(0,7)),
            ("bad",(0,8)),("ado",(0,9)),("doo",(0,10))])
   > let frank = shingles 3 ("frank",[(0,"doobeedoobeedoo")])
   > frank
   ("frank",[("doo",(0,0)),("oob",(0,1)),("obe",(0,2)),("bee",(0,3)),
             ("eed",(0,4)),("edo",(0,5)),("doo",(0,6)),("oob",(0,7)),
             ("obe",(0,8)),("bee",(0,9)),("eed",(0,10)),("edo",(0,11)),
             ("doo",(0,12))])
   > commonStrings fred frank
   ["doo"]
   ```

   HINT: Uh, I'm like returning a *set* and, like, it's supposed to be the *intersection* of like two other sets, and. . . ?

3. The function `shingles2regions` takes a document (represented by its list of shingles) and a set of strings as input. It returns a list of regions where the strings in common occur in the document. [5 points]

```
shingles2regions :: (DocName,[Shingle]) -> [String] -> [Region]

> let frank = shingles 3 ("frank",[(0,"doobeedoobeedoo")])
> frank
("frank",[("doo",(0,0)),("oob",(0,1)),("obe",(0,2)),("bee",(0,3)),
          ("eed",(0,4)),("edo",(0,5)),("doo",(0,6)),("oob",(0,7)),
          ("obe",(0,8)),("bee",(0,9)),("eed",(0,10)),("edo",(0,11)),
          ("doo",(0,12))])
> shingles2regions frank ["doo"]
[((0,0),3),((0,6),3),((0,12),3)]
> shingles2regions frank ["bee","doo"]
[((0,0),3),((0,3),3),((0,6),3),((0,9),3),((0,12),3)]
```

HINT: `map` **and** `filter` **are your friends.**

4. The function `showSimilarities` have been provided for you: [5 points]

```
showSimilarities :: (DocName,Document) -> [Region] -> (DocName,Document) -> [Region] -> String
showSimilarities (doc1name,doc1) reg1 (doc2name,doc2) reg2 = html
  where
     ann1 = highlight doc1 reg1
     ann2 = highlight doc2 reg2
     html = htmlTemplate (doc1name,ann1) (doc2name,ann2)
```

It takes two documents as input, along with regions that should be highlighted, and generates an `html` string with the propper annotations.

The function `highlightChar` has also been provided for you:

```
highlightChar :: Char -> String
highlightChar x = "<FONT style=\"BACKGROUND-COLOR: yellow\">" ++ [x] ++ "</FONT>"
```

`highlightChar` wraps a character in html that gives it a yellow background color.

The function htmlTemplate has also been provided for you. It returns a string containing html that displays two documents side by side.

```
htmlTemplate :: (DocName,[String]) -> (DocName,[String]) -> String
htmlTemplate (doc1name,lines1) (doc2name,lines2)= "\
   \<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">\n\
   .....
```

Now, given these functions, you can write the function `highlight` which takes two arguments: `doc` (a text document represented as (row,line) pairs) and `regions` (a list of ((row,column),length) tuples, representing the regions of the document that should be highlighted). `highlight` should return a string where each character `c` whose position lies within any of the regions has been highlighted (i.e. replaced by the output of the `highlightChar c` function call).

```
highlight :: Document -> [Region] -> [String]

> putStr (unlines (highlight  [(0,"doobeedoobeedoo")] [((0,0),3),((0,6),3),((0,12),3)]))
<FONT style="BACKGROUND-COLOR: yellow">d</FONT>
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
```

```
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
bee
<FONT style="BACKGROUND-COLOR: yellow">d</FONT>
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
bee
<FONT style="BACKGROUND-COLOR: yellow">d</FONT>
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
<FONT style="BACKGROUND-COLOR: yellow">o</FONT>
```

NOTE: You can write `highlight` recursively if you wish.

NOTE: For larger files, the visualization function is reaaaaallly slooooow. For extra credits (5-10 points): speed it up!

4 Submission and Assessment

The deadline for this assignment is noon, Thu Oct 6. It is worth 5% of your final grade.

Create a `zip`-file named `ass3.zip` containing the files ass3A.hs ass3B.hs TEAM and upload it to `d2l.arizona.edu`.

The assignment will be evaluated using `ghci` on lectura. Make sure it works on lectura before handing it in — the TAs are not required to debug your code.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.