CIS 194: Homework 5

Due Wednesday, 25 February, 2015

Preface

Setup

You will need two packages that are not part of Haskell's standard library for this assignment. They are `aeson` and `text`. You can install these with `cabal update; cabal install aeson text`.[1] If you have GHCi open, you will need to restart GHCi to use these downloaded libraries.

[1] The `cabal update` part is to make sure you download the most recent versions of these packages.

JSON files

This homework includes parsing and then querying information stored in a JSON file. JSON is a standardized data interchange format that is easy to read and easy to write. See `json.org` for the details, but you won't need to know about details for this assignment. Instead, the `aeson` library does all the work for you!

What you do have to worry about is making sure that your Haskell program can find your JSON files. Putting the files in the same directory as your `HW05.hs` file is a great start, but it's not always enough. If you're having trouble getting your code to find your file, and you're using GHCi, try running `:!pwd`. That will print out the current directory GHCi thinks it's in. (The `:!` prefix allows you to run arbitrary shell commands within GHCi.) If the JSON files aren't there, either move it there, or use `:cd` to move GHCi.[2]

[2] `:cd` is a GHCi command. The missing `!` is intentional!

String theory

Haskell's built-in `String` type is a little silly. Sure, it's programmatically convenient to think of `String`s as lists of characters, but that's a terrible, terrible way to store chunks of text in the memory of a computer. Depending on an application's need, there are several other representations of chunks of text available. This assignment will need a representation called `ByteString`.

The `ByteString` library helpfully (?) uses many of the same names for functions as the `Prelude` and `Data.List`. If you just `import Data.ByteString`, you'll get a ton of name clashes in your code. Instead, we use `import qualified ... as BS`, which means that every use of a `ByteString` function (including operators) or type must be preceded by `BS`. Thus, to get the length of a `ByteString`, you use `BS.length`.

`ByteStrings` come in several flavors, depending on whether they are lazy or strict and what encoding they use internally. For this assignment we will use lazy `ByteStrings`.

When working with non-`String` strings, it is still very handy to use the `"..."` syntax for writing literal values. So, GHC provides the `OverloadedStrings` extension. This works quite similarly to overloaded numbers, in that every use of `"blah"` becomes a call to `fromString "blah"`, where `fromString` is a method in the `IsString` type class. Values of any type that has an instance of `IsString` can then be created with the `"..."` syntax. Of course, `ByteString` is in the `IsString` class, as is `String`.

A consequence of `OverloadedStrings` is that sometimes GHC doesn't know what string-like type you want, so you may need to provide a type signature. You generally won't need to worry about `OverloadedStrings` as you write your code for this assignment, but this explanation is meant to help if you get strange error messages. If you want to use `OverloadedStrings` in GHCi just type `:set -XOverloadedStrings`.

Trouble at Haskell Bank

Haskell Bank is in trouble! Someone hacked in to their system by exploiting a careless use of `unsafePerformIO`[3]! Haskell Bank has since secured their system, however the perpetrator was able to initiate a bunch of bogus transactions between customers. You have been hired to figure out who hacked Haskell Bank and which transactions need to be reversed so that all of the customers can get their money back.

[3] Those fools should have been using safe Haskell!

Luckily, you were able to recover some files that contain clues as to who hacked Haskell Bank. These files are contained in `clues.zip`. In the following exercises, you will extract data from these files and use the clues to catch the criminal.

Exercise 1 The criminal kept a list of the transaction IDs that he initiated. Unfortunately the list is encrypted using a variant of the Vigenère cipher. The criminal encoded the encryption key in an adorable dog photo[4], `dog.jpg`. Some of the bytes in the image have been XOR'ed with a secret message. In order to extract the message, all you need to do is pairwise XOR the bytes of the encoded image with the bytes of the original image and filter out all of the bytes with value `0`. After a quick Google search, you were able to find the original image, `dog-original.jpg`. Now, implement the function:

[4] How dare he!

```
getSecret :: FilePath -> FilePath -> IO ByteString
```

That takes in the paths to the original and modified files, reads them in as `ByteString`s, and then outputs the secret that was encoded in the image. You will need to use the `xor` function in the `Data.Bits` module. Remember that `FilePath` is just a synonym for `String`!

Exercise 2 Now that you have the encryption key, you can decrypt the list of fake transaction IDs. This list is contained in `victims.json.enc`. The data is encrypted using a scheme similar to the Vigenère cipher. To decrypt it, simply pairwise XOR the ciphertext with the key. You will have to repeat the key because it is much shorter than the ciphertext. Implement the function:

```
decryptWithKey :: ByteString -> FilePath -> IO ()
```

This function should read in the encrypted file, decrypt it using the key, and then write it back to another file. This `ByteString` is the key and the `FilePath` is the path to the file that will be written (it does not have to exist). The encrypted file should have the same path, but with `".enc"` appended to the end. For example, calling `decryptWithKey key "victims.json"` should decrypt the file `victims.json.enc` and write the result to the file `victims.json`.

Exercise 3 You now have a list of all the IDs of the transactions that the criminal initiated, but this doesn't tell you anything about who the criminal is or how much money he stole. Luckily, Haskell Bank provided you with a list of all the transactions that took place during

the time that the hacker was in their system. This list is encoded in JSON format and can be found in the file `transactions.json`[5].

Haskell Bank has also provided you with the parsing module that they use to convert data between Haskell datatypes and JSON `ByteStrings`[6]. This module uses the Aeson parsing library and can be found in the file `Parser.hs`. Two functions are exported:

```
encode :: ToJSON a => a -> ByteString
decode :: FromJSON a => ByteString -> Maybe a
```

The file also defines `FromJSON` and `ToJSON` instances for the `Transaction` datatype and instances for `[Transaction]` are provided for free by the Aeson library. This means that you can use `decode` to parse the list of transactions in `transactions.json`. The data in `victims.json` is just a list of strings. Aeson knows how to parse this without a special instance. We can therefore use one polymorphic function to parse both of these files. Define the function:

```
parseFile :: FromJSON a => FilePath -> IO (Maybe a)
```

This function should take in a path to a JSON file and attempt to parse it as a value of type `a`. *Note*: if you want to test this in GHCi, you will need to tell it what the output type should be. For example, `parseFile "victims.json"` will return `Nothing`, but `parseFile "victims.json" :: IO (Maybe [TId])` will give you what you want.

Exercise 4 You now have the ability to parse your JSON files, so you can start looking for clues! The first step is to isolate the bad `Transactions`. Implement the function:

```
getBadTs :: FilePath -> FilePath -> IO (Maybe [Transaction])
```

This function takes in the path to the victims list and the path to the transaction data (in that order) and returns only those `Transactions` that occur in the victim list.

Exercise 5 Now that you have decrypted and parsed all of the data, it's time to do some detective work. In order to figure out who the bad guy is, you have to track the flow of money resulting from the bad transactions. There is a very easy way to do this! For every name, simply keep track of how much money that person has gained (or lost) as a result of the bad transactions.

You will need some way of associating people (`Strings`) with amounts of money (`Integers`). The do this efficiently, you should use the `Data.Map.Strict` module. Using this data structure, implement the function:

[5] Thankfully, Haskell Bank gave you the data unencrypted

[6] Seriously, these guys are so helpful

```
getFlow :: [Transaction] -> Map String Integer
```

Example:

```
let ts = [ Transaction { from   = "Haskell Curry"
                       , to     = "Simon Peyton Jones"
                       , amount = 10
                       , tid    = "534a8de8-5a7e-4285-9801-8585734ed3dc"
                       } ]
in getFlow ts == fromList [ ("Haskell Curry", -10)
                          , ("Simon Peyton Jones", 10)
                          ]
```

Note: The `Data.Map.Strict` module has been imported *qualified* meaning that you need to to prefix everything in the module with `Map`. For example, the empty map is `Map.empty`.

Exercise 6 With a `Map` containing information about they flow of money, you can easily figure out who the criminal is; he is the person that got the most money. Write the function:

```
getCriminal :: Map String Integer -> String
```

This function should take in the flow `Map` and return the name of the person who got the most money.

Exercise 7 In order to give everyone their money back, Haskell Bank has requested that you use the flow information to generate a new list of `Transactions` that will undo the money transfer initiated by the hacker. In an attempt to cover his tracks, the hacker moved money through intermediate accounts, he did not just dump it all into his own account. Reversing all of these transactions will result in many more transactions than are necessary. Instead you should implement the following algorithm[7]:

- Separate the people into *payers* and *payees*; ie, people who ended up with extra money and people who ended up at a loss.
- Sort both groups in *descending* order. The payers who owe the most and the payees who are owed the most should come first. You will likely find the `sortBy` function in the `Data.List` module helpful for this stage.
- Iterate over the payers and payees in parallel. For each pair, make a new `Transaction` where the payer pays the payee the minimum between how much the payer owes and how much the payee is

[7] While this algorithm does not always yield the optimum number of transactions, it is guaranteed to only yield $O(n)$ transactions where n is the number of people. This is quite good considering that there could have been arbitrarily many transactions initially.

owed. Deduct this amount from both, remove anyone who has completely paid his/her debt or has been completely paid off, and repeat.

Implement this algorithm as the function:

```
undoTs :: Map String Integer -> [TId] -> [Transaction]
```

The first argument is the flow map and the second argument is a list of new transaction IDs that you should use when creating new `Transaction`s. Haskell Bank has kindly provided you with a list of fresh transaction IDs that you can use when creating new `Transaction`s[8]. You can load this list in for testing, but for now it will be sufficient to use `(repeat "")` as your ID list.

[8] Wow, is there anything they won't do?

Exercise 8 In order to deliver your findings back to Haskell Bank, you need to be able to write `Transaction` data back in to JSON format. Implement the function:

```
writeJSON :: ToJSON a => FilePath -> a -> IO ()
```

Exercise 9 Now it is time to put everything together! The `main` function has been defined for you so that you can compile this file into an executable that will automate the entire mystery solving process. In Haskell, `main` has type:

```
main :: IO ()
```

This is different than other languages where the `main` function takes in `argc` and `argv` parameters. In Haskell, if you want to get command line arguments, you have to use the `getArgs` function in the `System.Environment` module. The executable will take in the paths to the original dog photo, the altered dog photo, the transactions file, the decrypted victim list (even if it hasn't been created yet), the new ID list, and the desired output file in that order as command line arguments. If these arguments are not given, they are defaulted to be the files supplied in `clues.zip`. The output by default is `new-transactions.json`.

The program extracts the encryption key from the dog picture, decrypts the victim list and writes it to a new file, writes the new transactions to a JSON file, and prints the name of the hacker.

Because your homework is written in a module that is not called `Main`, you will have to use a compiler flag to tell GHC that it should generate an executable. Type `ghc HW05.hs -main-is HW05` to compiler your program. The resulting executable should be called `HW05`. Finally, run the executable to discover who hacked Haskell Bank!