**CS380: Modern Functional Programming**
**Prof. Richard Eisenberg**
**Spring 2017**

**Homework 4: Processing JSON**
**due 9:40am on Monday, Feb. 27, 2017**

No template file is provided for this homework. Download the markets.json file from the website, and make your *Hw04.hs* Haskell file with your name, any sources you consulted, and any other relevant comments (just like in previous assignments). Then, say

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, DeriveAnyClass, GADTSyntax #-}
module Hw04 where

import Data.Aeson
import Data.Monoid
import GHC.Generics

import qualified Data.ByteString.Lazy.Char8 as B
import qualified Data.Text as T
```

and off you go. Do make sure to include this module header, as it makes grading much easier.

# 1 Preface

## 1.1 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.

## 1.2 Generics

You will see the language extension *DeriveGeneric* in the module header given above. This allows you to name the class *Generic*[2] in a **deriving** clause when declaring a new datatype. You will not use any of the methods or other features of the *Generic* class. The reason to derive *Generic* is for easy interoperability with the *aeson* JSON-parsing library. A derived *Generic* instance encodes various features of a datatype (such as its constructor names, any record-field names, etc.) that advanced Haskellers can (such as the authors of *aeson*) use to make your life easier.

## 1.3 JSON files

This homework centers around 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 `markets.json` file. Putting the file in the same directory as your *Hw04.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 `markets.json` isn't there, either move it there, or use :*cd* to move GHCi.[3]

[1] The *cabal update* part is to make sure you download the most recent versions of these packages.
[2] imported from *GHC.Generics*
[3] :*cd* is a GHCi command. The missing ' is intentional!

## 1.4 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 two other representations: *ByteString* and *Text*.

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 B*, which means that every use of a *ByteString* function (including operators) or type must be preceded by *B.* . Thus, to get the length of a *ByteString*, you use *B.length*. Even to mention the type *ByteString*, you must use *B.ByteString*.

*ByteString*s come in several flavors, depending on whether they are lazy or strict and what encoding they use internally. Laziness is a story for another day, and we *really* don't want to worry about encodings. For now, use *Data.ByteString.Lazy.Char8* and everything will work out nicely.

*Text* is quite like *ByteString*: it *also* reuses a lot of familiar names. It *also* comes in two laziness flavors. We'll be using the strict flavor, which is provided in *Data.Text*. You also may want some I/O operations, so the **import** statements above include the *Data.Text.IO* module.

When working with non-*String* strings, it is still very handy to use the `"..."` syntax for writing *Text* or *ByteString* 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* and *Text* are both 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.

## 2 Off to the market


http://www.bayoucityoutdoors.com/ClubPortal/ClubStatic.cfm?clubID=3&pubmenuoptID=11318

The `markets.json` file you have downloaded contains information about many (all?) of the Farmers' Markets that regularly take place throughout the USA, originally retrieved via `http://catalog.data.gov/dataset/farmers-markets-geographic-data`. That website produces the data in an Excel spreadsheet. I converted the spreadsheet to a comma-separated-values format (CSV) and then used `http://www.convertcsv.com/csv-to-json.htm` to get it into a JSON format. I chose JSON because the *aeson* JSON parser is more advanced yet easier to use than the CSV parser package, *cassava*.[4]

Take a look at the data by opening up the file in a text editor. You'll see that each market entry has many different fields, all with distinct names but of different types.

First, notice that the many Boolean values in the file are labeled with `"Y"` or `"N"`. As smart as *aeson* is, it doesn't know that `"Y"` corresponds to *True* and `"N"` corresponds to *False*. Your first task is to construct a *Value*[5] that has been adjusted to have proper Booleans instead of `"Y"` and `"N"`.

One *aeson* function that parses JSON is called *decode*:

$$\mathit{decode} :: \mathit{FromJSON}\ a \Rightarrow \mathit{ByteString} \rightarrow \mathit{Maybe}\ a$$

The *FromJSON* type class is also exported by the *aeson* package[6] Its method $\mathit{parseJSON} :: \mathit{Value} \rightarrow \mathit{Parser}\ a$ (which you will *not* have to write in this assignment) says how to parse from JSON to the class type *a*. Thus,

[4] I am giving you these details in case you want to look at other datasets.

[5] *Value* is a type from the *aeson* library. Hoogle does not search the *aeson* package by default, so you will have to access the package documentation on Hackage. Try this URL: `http://hackage.haskell.org/package/aeson`.

[6] In case you haven't noticed, I'm using "package" and "library" interchangeably.

anything in the *FromJSON* type class can be parsed from a JSON file. Of course, parsing can fail, so *decode* returns a *Maybe* type.

A useful member of the *FromJSON* type class is *Value*. *Value* represents JSON syntax in a Haskell type. Check out its documentation.[7] A JSON *Value* can be one of six things: an object (something in braces; a mapping from key names to other values), an array (something in brackets; a listing of JSON values), some text, a number, a Boolean value, or the special constant *null*. Look a little further down in the documentation to see the definitions for the types *Object* and *Array*.

An *Object* is a *HashMap Text Value* — that is, a way to get *Value*s indexed by some *Text*. However, the details of *HashMap* aren't important at all for you. What *is* critically important is that there is an instance *Functor* (*HashMap k*). That means that a valid type for *fmap* is $(a \to b) \to \mathit{HashMap}\ k\ a \to \mathit{HashMap}\ k\ b$.

An *Array* is a *Vector Value*. *Vector* is a type quite like normal lists but uses a different internal representation.[8] Some operations on *Vector*s are faster than for lists; some are slower. However, the details of *Vector* aren't important at all for you. What *is* critically important is that there is an instance *Functor Vector*. That means that a valid type for *fmap* is $(a \to b) \to \mathit{Vector}\ a \to \mathit{Vector}\ b$.

1. Write a function

   $$\mathit{ynToBool} :: \mathit{Value} \to \mathit{Value}$$

   that changes all occurrences of *String* `"Y"` to be *Bool True* and all occurrences of *String* `"N"` to be *Bool False*. No other part of the input *Value* should change.

2. Write a function

   $$\mathit{parseData} :: \mathit{B.ByteString} \to \mathit{Maybe}\ \mathit{Value}$$

   that takes in a *ByteString* containing JSON data and outputs either *Nothing* or a *Value* that has been processed by *ynToBool*.

   *Hint:* This can be very short, if you use *Maybe*'s *Functor* instance!

   You can easily test this by saying, for example, *filedata* ← *B.readFile* `"markets.json"` in GHCi and then calling *parseData* on *filedata*.

## 2.1 The **Market** type

If you define a datatype *Market* appropriately and include **deriving** (*Generic*, *FromJSON*), and say **instance** *FromJSON Market* with no **where** clause, you get an automatic parser for your datatype. For example, if you have[9]

```haskell
data Person where
  Person :: { name :: String
            , age  :: Int }   -> Person
  deriving (Show, Generic, FromJSON)

p :: Maybe Person
p = decode "{ \"name\" : \"Richard\", \"age\" : 35 }"
```

You get that *p* == *Just* (*Person* `"Richard"` 35).[10] The *aeson* library uses the field names in the *Person* record (see the lecture notes about record notation) to figure out what the JSON tags should be. The order doesn't matter – *aeson* really is using the names.

[7] Ignore the 's in the documentation. They are strictness annotations, which are a story for another day. They don't affect you at all here.

[8] Haskell lists are linked lists; *Vector*s are arrays.

[9] Note the syntax for defining a record using the newer-style syntax, with **where**.

[10] The extra backslashes in the string above are because the string must contain quotes, and Haskell's way of putting quotes in strings is to use backslashes like you see here.

3. Write a *Market* type, including the fields that interest you. At a minimum, include *marketname*, *x* (the longitude of the market), *y* (the latitude of the market), *state*, and *cheese* (which is a *Bool*). Use *T.Text* to represent text. (*String* also works, but is less efficient.)

4. Then, write a function

   ```
   parseMarkets :: B.ByteString → Maybe [Market]
   ```

   that uses *parseData* and *fromJSON* (from the *aeson* package) to parse in the list of markets in the file.

5. Write an I/O action

   ```
   loadData :: IO [Market]
   ```

   that loads the market data. Use *B.readFile* :: *String* → *IO B.ByteString*. In the event of a parsing failure, report the error using *fail* :: *String* → *IO a*. (*fail* aborts an action, reporting an error to the user. It never returns, so it can be used no matter what *IO* type is expected. That's why it returns type *IO a*, for any *a*.)

   Once this is defined, you can get your market data by saying *mkts* ← *loadData* in GHCi.

# 3 Interlude: an ordered-list monoid

Before we get to actually searching the loaded market data, it will be helpful to define a monoid for an ordered list. An ordered list, which we'll call *OrdList*, is a wrapper around lists (in the tradition of *First*, *Last*, *Sum*, and *Product*, all from the *Data.Monoid* module) that defines a *Monoid* instance which keeps the list ordered, from least to greatest. For example:

```
data OrdList a where
  OrdList :: {getOrdList :: [a]} → OrdList a
  deriving (Eq, Show)
    -- include this datatype in your code!
instance Ord a ⇒ Monoid (OrdList a) where ...
    -- you'll need to fill in the ellipses
evens :: OrdList Integer
evens = OrdList [2, 4, 6]
odds :: OrdList Integer
odds = OrdList [1, 3, 5]
combined :: OrdList Integer
combined = evens <> odds
```

Now, *combined* should have the value *OrdList* $[1, 2, 3, 4, 5, 6]$, because the (<>) operator maintains the ordering invariant.

6. Write the *OrdList* datatype and its *Monoid* instance. Make sure your implementation of (<>) is $O(m + n)$, where $m$ and $n$ are the lengths of the input lists.

# 4 Searching with **Monoid**s

Now that you have a way of loading the market data, you need a way of searching through that data. Furthermore, it would be nice if the search mechanism is flexible enough to produce data in a *Monoid* of the caller's choice. Although there are, I'm sure, other queries you might want to do, all of our queries are going to center on searching for a market's name (the *marketname* field).

Throughout this section, we will be searching among the markets returning a variety of types. To avoid code repetition, it is helpful to use a type synonym. Include the following in your code:

$$\textbf{type}\ \mathit{Searcher}\ m = \mathit{T.Text} \to [\mathit{Market}] \to m$$

Thus, a $\mathit{Searcher}\ m$ is a function that, when given the $\mathit{T.Text}$ to search for in a $[\mathit{Market}]$, will produce an $m$.

7. Write a function

   $$\mathit{search} :: \mathit{Monoid}\ m \Rightarrow (\mathit{Market} \to m) \to \mathit{Searcher}\ m$$

   that searches through the provided list of $\mathit{Market}$s for market names containing the given $\mathit{T.Text}$ ($\mathit{Data.Text.isInfixOf}$ will be useful here). With each found market record, use the function provided to convert the market record into the monoid of the caller's choice, and then combine all the individual results using $\mathit{mappend}$ and $\mathit{mconcat}$.

   Note that we can always expand type synonyms in Haskell. So, the type of $\mathit{search}$ is fully equivalent to $\mathit{Monoid}\ m \Rightarrow (\mathit{Market} \to m) \to \mathit{T.Text} \to [\mathit{Market}] \to m$. This means that the definition for $\mathit{search}$ may include up to three arguments, even though the type looks like it should take only one.

   *Hint:* This function should not be very long. If it's getting long, you're probably doing something the wrong way. You may also want to check out the $\mathit{intInts}$ example from the lecture notes.

   *Hint:* Using an *as-pattern* might be helpful. Here is an example:

   $$\mathit{marketWithName} :: \mathit{Market} \to (\mathit{T.Text}, \mathit{Market})$$
   $$\mathit{marketWithName}\ \mathit{mkt}@(\mathit{Market}\ \{\mathit{marketname} = \mathit{name}\}) = (\mathit{name}, \mathit{mkt})$$

   Note that $\mathit{mkt}$ is matched against the whole market record, while the pattern-match also binds $\mathit{name}$ to the market's name. The name is just one field in the record matched by $\mathit{mkt}$.

8. Write a function

   $$\mathit{firstFound} :: \mathit{Searcher}\ (\mathit{Maybe}\ \mathit{Market})$$

   that returns the first market found by a search, if any are found at all.

   Like in the case for $\mathit{search}$, above, your $\mathit{firstFound}$ function can be given arguments, even though the type looks like there should be no arguments. Unlike $\mathit{search}$, though, this one is definable without taking any arguments, with the right call to $\mathit{search}$.

   This function (and all future functions) will not get full credit if they do not use $\mathit{search}$, which should actually do the searching.

   *Hint:* The following function may be useful for all the searching exercises. Look at the type to figure out what it does:

   $$\mathit{compose2} :: (c \to d) \to (a \to b \to c) \to a \to b \to d$$
   $$\mathit{compose2} = (.).(.)$$

9. Write a function

   $$\mathit{lastFound} :: \mathit{Searcher}\ (\mathit{Maybe}\ \mathit{Market})$$

   that returns the last market found by a search, if any are found at all.

10. Write a function

    $allFound :: Searcher\ [Market]$

    that returns all the markets found by a search.

11. Write a function

    $numberFound :: Searcher\ Int$

    that returns the number of markets found by a search.

12. Write a function

    $orderedNtoS :: Searcher\ [Market]$

    tht returns all the markets found by a search, ordered from northernmost to southernmost. You will need an appropriate *Ord* instance.

    You may find that your function takes a little while to run. As an optional extra, make it work more efficiently by adding a definition for *mconcat* to the *Monoid* instance for *OrdList* and make sure your *search* function uses *mconcat*. The default definition for *mconcat* puts elements together one by one, but you can write a custom one that maintains the ordering in a more efficient fashion.

13. *(Optional)* Now that you've built the infrastructure to do queries on this dataset, see if you can find an interesting detail in the data. It should be fairly easy at this point to make a variety of queries. Tell us something we didn't know about farmer's markets!