HASKELL PROGRAMMING PROBLEM SET 6

LENNART JANSSON AND BRANDON AZAD

This week's subject is applicative functors, a typeclass more powerful than plain functors that requires the same sort of conceptual abstract thinking you'll need to tackle monads. Though monads have a much wider range of uses, applicative functors express a few ideas really really elegantly.

LYAH does discuss applicative functors briefly at the end of Chapter 11, but this is not a very solid coverage, so in addition to reading that, please read section 4 of Typeclassopedia by Brent Yorgey (http://www.haskell.org/haskellwiki/Typeclassopedia#Applicative), which gives additional information about applicatives from a slightly more mathematical standpoint. If you are interested, explore the additional links he provides, and if you have spare time or need practice, do the exercises in that section, including implementing the applicative instance for Maybe.

This problem set is two large problems. This first is a module for dealing with encryption and decryption using the de Vigenère cipher, which will use the Applicative instance of Maybe to deal with error conditions in an elegant way. The second is a parser module that you can use to parse strings into abstract data types for easier manipulation. Parsers were some of the original motivation for the applicative typeclass, and we'll see how the applicative framework allows us to build larger parsers by combining smaller parsers in a beautiful way. We will also meet the Alternative typeclass, which is for applicative functors that are also monoids.

de Vigenère Cipher

In this problem, we'll create a simple module to do encryption and decryption using the de Vigenère cipher.

The De Vigenere cipher is a simple encryption scheme based on the idea of shifting letters in the plaintext to produce the ciphertext. The process is relatively straightforward: The key (some combination of letters, frequently a word) is first translated into a list of numbers that represent shifts. Then, this list is superimposed above the plaintext, so that each letter is given a shift value (if there are fewer letters in the key than in the plaintext, the list of shifts is repeated). Each letter in the plaintext is then shifted forward in the alphabet by the corresponding number of places (wrapping around, so that shifting one past 'z' yields 'a'). The result is the ciphertext.

See Wikipedia for a full description and examples. Most online descriptions are tailored to a graphical presentation of the cipher, so the description, while functionally equivalent, may appear to be different.

You will make a module called `DeVigenere` that exports four functions:

```
plainToKey :: String -> Maybe [Int]
deVigenere :: [Int] -> String -> String
deVigenereEncrypt :: String -> String -> Maybe String
deVigenereDecrypt :: String -> String -> Maybe String
```

Problem 1. Key creation

```
plainToKey :: String -> Maybe [Int]
```

`plainToKey` should take a key in plaintext and convert it to a list of offsets. If any character in the key is not a letter, the function should fail by returning `Nothing`.

Problem 2. The algorithm

```
deVigenere :: [Int] -> String -> String
```

`deVigenere` should take a list of offsets and use it to encrypt a plaintext string. Any non-alphabetic character in the plaintext should be self-enciphered and should not cause the function to fail by returning `Nothing`.

Problem 3. Encryption

```
deVigenereEncrypt :: String -> String -> Maybe String
```

`deVigenereEncrypt` is a helper function that takes a key and the plaintext and enciphers the plaintext with the key. If the key is invalid, the function should fail by returning `Nothing`.

Problem 4. Decryption

```
deVigenereDecrypt :: String -> String -> Maybe String
```

`deVigenereDecrypt` is just like `deVigenereEncrypt`, only it performs decryption instead of encryption.

All four functions should accept both upper and lower case letters as valid. You should manage failure conditions by using Applicative rather than by conditional execution using pattern matching. Applicative is there to help abstract away the details; pattern matching puts the details right in your face for you to manage.

Hints and Suggestions: Good decomposition is critical to be able to express the ideas neatly and succinctly. You might find it helpful to borrow the function sequenceA from Learn You a Haskell. My entire solution is about sixty lines, and uses `Just` and `Nothing` exactly once each (in part thanks to the `maybe` function).

MiniParser

In this large problem, we will build an applicative parser from scratch, then use it to parse a context-free grammar. This is really really cool, please don't skip this problem because it is well worth your time.

The motivation behind applicative parsers is that the way parsers can be combined corresponds to the `<*>` operation of applicative functors. This means smaller parsers can be combined together to make larger parsers that can parse more and more complex grammars.

Let's get started! Open a new file and `import Control.Applicative`. Let's familiarize ourselves with the type we will use:

```
newtype MiniParser a = MiniParser {
  unMP :: String -> Maybe (String, a)
  }
```

It's given as a newtype wrapper, which makes the code a little bit messier, but means the functor and applicative instances we will define will be specific to our parser implementation; this is considered good practice. We're using record syntax, so we can use `MiniParser` to wrap a function as a parser, and `unMP` to retrieve the function from inside the newtype wrapper.

Now the type. The parser is a function that takes a string; that should be intuitive, since we will want our parser to operate on a single input, the string it should be parsing. The return type is `Maybe (String, a)`: we have a `Maybe` since the parsing might fail, in which case the parser would return `Nothing`. If it succeeds, it will return `Just (String, a)`, where the string is the remainder of the string that wasn't consumed by that parsing pass, and we have a value of type `a` as a result.

We obviously want our parser to be able to produce some result in a type that we want, after consuming a string. That's what the type `a` is.

Problem 5. runParser

Write a function `runParser :: MiniParser a -> String -> Maybe a`. This is how we'll run parsers we create on a given input. This should be simple, all this function needs to do is unwrap the parser, apply the input string to the parser, and then collect the result.

Problem 6. runParserComplete

Write a function `runParserComplete :: MiniParser a -> String -> Maybe a`, that is a variation on `runParser` that will run the parser in the same way, but fail (return `Nothing`) if not all of the input is consumed during the parsing.

This behavior is often more useful, since you can test if the whole input string follows a certain grammar instead of just a prefix of the string.

Problem 7. Your first parser

Write a function `charP :: Char -> MiniParser Char` that parses a single given character, and only that character, returing the parsed character as a result.

If this was all a bit confusing, here are some examples of how these functions can be used. When your program can do all this you're ready to move on.

```
> runParser (charP 'a') "a"
Just 'a'
> runParser (charP 'a') "b"
Nothing
> runParser (charP 'a') "alloy"
Just 'a'
> runParser (charP 'a') "barnacle"
Nothing
> runParserComplete (charP 'a') "a"
Just 'a'
> runParserComplete (charP 'a') "alloy"
Nothing
```

That last example fails because not all of the input string "alloy" is consumed by the parser that only parses the single character 'a', since after the parser has done its work, there should still be the string "lloy" remaining.

Problem 8. Parsing words

Use the `sequenceA` function you used in the Vigenère cipher module to turn a parser for a given character into a parser for a given word. Write the function `wordP :: String -> MiniParser String`.

```
> runParserComplete (wordP "hello") "hello"
Just "hello"
> runParserComplete (wordP "hello") "ohnoes"
Nothing
```

Alright. Now the crazy typeclass fun begins!

Problem 9. Functor

Make your parser a functor:

```
instance Functor MiniParser where
  fmap <your implementation here>
```

Let's think about what `fmap` should do. It's useful to think of this functor instance as similar to the functor instance for `((->) r)`, namely, `fmap` as function composition. Since the parser is a function, what `fmap` needs to do is compose a new function around the existing parsing function.

Of course, there's a little trickery that needs to happen since the return value of the parser is not a pure value that is ready for applying to a new function, but rather a `Maybe` around a tuple. Obviously, the new function should not be applied if the parser fails, and should not affect the remainder of the string to be parsed, if any.

Make sure your functor instance abides by the functor laws!

```
> runParserComplete (fmap reverse $ wordP "hello") "hello"
Just "olleh"
> runParserComplete (fmap reverse $ wordP "hello") "ohnoes"
Nothing
```

Problem 10. Applicative

Make your parser an applicative functor:

```
instance Applicative MiniParser where
  pure <your implementation here>
  (<*>) <your implementation here>
```

This is the hardest part of the whole assignment. Good luck!

`pure` needs to produce a parser that will always succeed and return a certain value without actually consuming any input.

`(<*>)` needs to run the first parser on the input, which should give a partially consumed string and the result, which is a function. Then, the next parser should be run on the remainder of the string input, which should produce a value to feed to the function obtained from the first parser. Of course, if one of the parsers fail, then the whole computation should fail.

Why do we need to thread a partially consumed string from the parser in the first argument to the parser in the second argument? This is actually the key to the whole applicative parsing business in the first place! Using `(<*>)`, we can combine two parsers to create a new one that will run those two parsers one after the other. Not only that, but since the first parser can yield a function as a result, we can save the results from both parsers and combine the results however we please.

Once we have defined `pure` and `(<*>)`, we get a bunch of utility functions for free. Very useful are `(*>) :: (Applicative f) => f a -> f b -> f b` and `(<*) :: (Applicative f) => f a -> f b -> f a`, which can be used to sequence parsers along with consuming input and failing on wrong inputs, but without actually saving the result from one of the parsers. A mnemonic for keeping `(<*>)`, `(*>)`, and `(<*)` straight: an arrow points to whichever parser we want to save information from.

```
> runParserComplete ((++) <$> (wordP "hello ") <*> (wordP "world")) "hello world"
Just "hello world"
> runParserComplete ((,) <$> (wordP "hello ") <*> (wordP "world")) "hello world"
("hello ","world")
```

```
> runParserComplete ((charP 'a') *> (charP 'b')) "ab"
Just 'b'
> runParserComplete ((charP 'a') *> (charP 'b')) "b"
Nothing
> runParserComplete ((++) <$> (wordP "hello ") <*> (reverse <$> wordP "world")) "hello
world"
Just "hello dlrow"
```

Even though we can now combine two parsers to parse one thing after another, we still don't have enough power to parse even the regular languages. The problem is that we have no way of parsing something *or* something else, like `a|b` in regular expressions.

If we were to add an operator to do this, it could only combine two parsers that have the same return type, so that the overall expression for certain is a parser with that particular return type. This means our combining operator needs to have type `MiniParser a -> MiniParser a -> MiniParser a`. Hm, this seems familiar... why, that's the same type as `mappend :: Monoid a => a -> a -> a`. So, our parser needs to be made a monoid!

Instead of making `MiniParser` an instance of monoid (though of course that can be done), we're instead going to make it an instance of `Alternative`, which is a typeclass specifically for types which are both applicative functors and monoids. `Alternative` is even more useful because it gives us a couple of utility functions for free that are specific to applicative functors.

The functions we need to define are `empty :: Alternative f => f a` and `(<|>) :: Alternative f => f a -> f a -> f a`. The first of these two is analogous to `mempty` on monoids, and the second is analogous to `mappend`.

Problem 11. Alternative

Make your parser an instance of `Alternative`.

Let's think a bit about how to implement `empty` and `(<|>)`. `(<|>)` needs to set up its two input parsers as "alternatives": it should combine two parsers in a way that the first parser is used first, and if it fails, then the second parser is used, and if that fails, then the whole parser should fail. We should be able to chain lots of parsers together with `(<|>)`, and the resulting parser will try using each one in turn from left to right until one of them works.

`empty` needs to be the monoid identity, so `<|>`ing it with any other parser needs to not change that parser's behavior. This is easy: in order to not change the other parser's behavior we just need a parser that will always make the combined parser default to using the other one. This parser is... the parser that always fails.

Now we have some really cool things we can do.

```
> runParserComplete (charP 'a' <|> charP 'b') "a"
Just 'a'
> runParserComplete (charP 'a' <|> charP 'b') "b"
Just 'b'
> runParserComplete (charP 'a' <|> charP 'b') "c"
Nothing
```

With the `Alternative` instance we also get the wonderful functions `many :: Alternative f => f a -> f [a]` and `some :: Alternative f => f a -> f [a]`, which apply an alternative functor zero or more times or one or more times, respectively. If you're interested, check the Hackage page on Applicative (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Applicative.html) and click the "source" buttons to see how they're defined.

```
> runParserComplete (many $ charP 'a') "a"
Just "a"
> runParserComplete (many $ charP 'a') "aaaaa"
Just "aaaaa"
> runParserComplete ((++ "!") <$> (many $ charP 'a')) "aaaaa"
Just "aaaaa!"
> runParserComplete (many $ charP 'a') ""
Just ""
> runParserComplete (many $ charP 'a') "b"
Nothing
> runParserComplete (some $ charP 'a') "aaaaa"
Just "aaaaa"
> runParserComplete (some $ charP 'a') ""
Nothing
```

Pat yourself on the back, you've implemented an applicative parser.

Using your Parser

This section is optional but highly encouraged as it is very satisfying. You'll use your parser to create a parser for arithmetic expressions (or a LL(1) grammar of your choice).

First some examples of writing parsers. Let's use the `MiniParser` library to write a parser for the language that consists of n `a`s in a row, then n `b`s. Some strings in this language: `"ab"`, `"aaabbb"`, `"aaaaabbbbb"`, `""`. This is a canonical example of a context-free language in CS theory. Let's make our parser return an `Int` or how many `a`s or `b`s there were in the input string.

We'll think of this language as being generated by the following context-free grammar

$$S \to \texttt{0}S\texttt{1} \mid \varepsilon$$

where ε is the empty string. Convince yourself that this works. All we need to do then is make a parser that models this grammar. First, the building blocks:

```
aP :: MiniParser Char
aP = charP 'a'

bP :: MiniParser Char
bP = charP 'b'
```

Our final parser will need to be `MiniParser Int`, but it doesn't matter what the return type of the above parsers are since we will ignore their return values.

Let's follow the grammar above:

```
sP :: MiniParser Int
sP = abPair <|> epsilon
```

It doesn't make sense for `epsilon` to actually be a parser that will fail on anything except the empty string, because by the time that epsilon is run the parser should be halfway through the string. Rather, it should be a parser that accepts everything. What should it return? We're going to use recursive depth to get the number of `a`s or `b`s, so we should start the count at zero.

```
epsilon :: MiniParser Int
epsilon = pure 0
```

And `abPair`? It needs to parse an `a`, then recurse to parse the middle of the string, then parse the `b`. It should return whatever number the middle bit returns plus one since we match an additional `a` and `b`.

```
abPair :: MiniParser Int
abPair = (+ 1) <$> (aP *> sP <* bP)
```

And that's it! Let's test.

```
> runParserComplete sP "aaaabbbb"
Just 4
> runParserComplete sP "abb"
Nothing
> runParserComplete sP ""
Just 0
> runParserComplete sP "acb"
Nothing
```

Now write your own parser for the following syntax of arithmetic expressions.

Problem 12. Arithmetic expressions

All our numbers are integers, multi-digit numbers should not start with a 0. We can combine numbers into expressions with `+` and `*`, but every operation must have parentheses around it. We can also nest expressions within expressions. Parentheses are not optional, and neither should we have too many parentheses.

So the following are valid: `1`, `(1+2)`, `(6*9)`, `((1+2)*(3+4))`. The following are invalid: `5+4`, `(1+2+3)`, `(1)`.

We want to parse strings to the following binary tree-like type:

```
data Expr = NumExpr Integer | AddExpr Expr Expr | MultExpr Expr Expr
  deriving (Show)
```

Write a parser `arithmeticParser :: MiniParser Expr` that successfully parses arithmetic expressions.

```
> runParserComplete arithmeticParser "((1+2)*3)"
Just (MultExpr (AddExpr (NumExpr 1) (NumExpr 2)) (NumExpr 3))
> runParserComplete arithmeticParser "(1+2*3)"
Nothing
```

Alternatively, you can choose to write a parser for a different language. If you are interested, the languages `MiniParser` can handle are a subset of the context-free languages called the LL(1) languages. Parsing these languages does not require a lookahead of more than 1 character reading from the left, which is the way the `MiniParser` works. Other parsing libraries for Haskell, notably `Parsec` and `Attoparsec`, are more powerful and can handle a much wider variety of languages.