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

TODO

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 1. 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 2. 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 3. 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 4. 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 5. 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 6. 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
```