B5 - Advanced Functional Programming

B-FUN-500

Bootstrap

my parsing library from the ground up

2.0

The goal of this boostrap is to help you start to write a parsing module that you will be able to use on all the projects of this year functional track.

You will be guided to write a basic recursive descent parser with combinators. Of course the version used here is just an example (and one of the simplest version), and you're invited to extend it to implement more features.

Part 1 - A simple recursive descent parsing module

Step 1.1 - My Parser Type

A parser is a **function** , which takes a stream of character (or tokens) in argument, and either **fail** or **returns a result**. The result is composed of what the parser was able to parse from the begining of the stream, and the rest of the stream.

In short, a parser for **type a** takes a **string** in argument and returns either **nothing**, or an **object of type a** and the **rest** of the string.
Here is a simple type satisfying all these features:

```
type Parser a = String -> Maybe (a, String)
```

Exercise 1.1.1

Find one different data type satisfying at least these features.

> What could you do to return more meaningful error messages in case of failure?

Step 2 - My first parsers

> For this document we will use the type we defined in **step 1** but feel free to use your own

Exercise 1.2.1

Write a function **parseChar** of the type:

```
parseChar :: Char -> Parser Char
```

This function takes a **Char** as argument and returns a **Parser Char**.

> The signature **Char -> Parser Char** can be rewritten as **Char -> String -> Maybe (Char, String)**

Usage examples:

```
> parseChar 'a' "abcd"
Just ('a', "bcd")
> parseChar 'z' "abcd"
Nothing
> parseChar 'b' "abcd"
Nothing
> parseChar 'a' "aaaa"
Just ('a', "aaa")
```

Exercise 1.2.2

Write a function **parseAnyChar** of the type:

```
parseAnyChar :: String -> Parser Char
```

This functions parse any of the characters in the string in its first argument.
Usage examples:

```
> parseAnyChar "bca" "abcd"
Just ('a', "bcd")
> parseAnyChar "xyz" "abcd"
Nothing
> parseAnyChar "bca" "cdef"
Just ('c', "def")
```

Step 3 - My higher order parsers

In the same way a higher order functions is a function taking another function in argument, a higher order parser is a parser taking another parser in argument in order to use it to do more complex parsing.

Exercise 1.3.1

Write a function **parseOr** of the type:

```
parseOr :: Parser a -> Parser a -> Parser a
```

this function takes two parsers in argument, tries to apply the first one, and if it fails, try to apply the second one.
Usage examples:

```
> parseOr (parseChar 'a') (parseChar 'b') "abcd"
Just ('a', "bcd")
> parseOr (parseChar 'a') (parseChar 'b') "bcda"
Just ('b', "cda")
> parseOr (parseChar 'a') (parseChar 'b') "xyz"
Nothing
```

Try to re-write parseAnyChar using parseOr and parseChar.

Exercise 1.3.2

Likewise, **parseAnd** takes two parsers, tries to apply the first one, and if it succeed, appies the second one and returns a tuple of what it parsed.

```
parseAnd :: Parser a -> Parser b -> Parser (a,b)
```

Usage examples:

```
> parseAnd (parseChar 'a') (parseChar 'b') "abcd"
Just (('a','b'), "cd")
> parseAnd (parseChar 'a') (parseChar 'b') "bcda"
Nothing
> parseAnd (parseChar 'a') (parseChar 'b') "acd"
Nothing
```

Exercise 1.3.3

parseAndWith is like **parseAnd**, but takes an additional function which gets the parsed elements as arguments.

```
parseAndWith :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
```

Usage examples:

```
> parseAndWith (\ x y -> [x,y]) (parseChar 'a') (parseChar 'b') "abcd"
Just ("ab", "cd")
```

Exercise 1.3.4

Write **parseMany**, which takes a parser in argument and tries to apply it zero or more times, returning a list of the parsed elements.

```
parseMany :: Parser a -> Parser [a]
```

Usage examples:

```
> parseMany (parseChar ' ') "    foobar"
Just ("    ", "foobar")
> parseMany (parseChar ' ') "foobar    "
Just ("", "foobar    ")
```

This parser can never fail.

Exercise 1.3.5

Write **parseSome**, which works like parseMany but must parse at least one element and fails otherwise.

```
parseSome :: Parser a -> Parser [a]
```

Usage examples:

```
> parseSome (parseAnyChar ['0'..'9']) "42foobar"
Just ("42", "foobar")
> parseSome (parseAnyChar ['0'..'9']) "foobar42"
Nothing
```

Try to use parseAnd and parseMany to implement parseSome

Step 4 - using my parsers for something actually usefull

Exercise 1.4.1

Write the following parsers:

```
parseUInt :: Parser Int -- parse an unsigned Int

parseInt :: Parser Int  -- parse a signed Int

parseTuple :: Parser a -> Parser (a,a) -- parse a tuple

-- Example:
> parseTuple parseInt "(123,456)foo bar"
Just ((123,456), "foo bar")
```

Part 2 - Adding Type Classes

All this is nice and good but quite cumbersome to use. Fortunately, Haskell allows us to organize our parsers in a different maner, which will have the nice benefit to remove a lot of book-keeping code.

Step 2.1 - Refactoring my parser for type classes

To do so, we want to implement a couple of **type classes** defined in the standard library of Haskell for our parser type. Only **data** and **newtypes** objects can be instances of **type classes**, therefor we first have to refactor our **Parser** type to be a data type containing a function and not just a function.

```
-- From:
type Parser a = String -> Maybe (a, String)

-- To:
data Parser a = Parser {
    runParser :: String -> Maybe (a, String)
}
```

Exercice 2.1.2 - Change your existing parsers to the new type

Example:

```
> runParser (parseChar 'a') "abc"
Just ('a', "bc")
> runParser parseInt "42"
Just (42, "")
```

Step 2.2 - Make your parser type a functor

Functor is the first type class we want to add to our type.

It may be useful to read this page about the Functor type class:
https://wiki.haskell.org/Functor
According to this page, to implement the Functor type class for our Parser type we just have to provide an implementation for the function **fmap**, which in our case will look like this:

```
instance Functor Parser where
    fmap fct parser = undefined -- your implementation goes here
```

Your fmap implementation should look a lot like a generalized version of your parseInt or parseUInt parser. Once implemented, you must be able to re-write those functions more consizely using fmap or its infix version, <$>

Step 2.3 - Make your parser an applicative functor

Another useful type class is **Applicative** (for applicative functor):
https://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html

> Once implemented, you should be able to rewrite parseSome using <$> and <*>

Step 2.4 - Make your parser an alternative functor

Also found in **Control.Applicative**, **Alternative** is a must have for your parser. It gives the operator **<|>**.

> Think about your parseOr function...

Step 2.5 - BONUS: make your parser a monad

You can optionaly make the extra step and implement the **Monad** type class for your parser type. This has the benefit to let you use the **do** notation with your parsers.