Lab 11

Parser combinators in Haskell

Goals

In this lab you will learn to:

1. Understand parser combinators
2. Write your own parser combinators

Resources

Table 11.1: Lab Resources

Resource	Link
Functional parsing	`https://youtu.be/dDtZLm7HIJs`
Understanding parser combinators	`https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/`
Edge case poisoning (defining data with combinators)	`https://buttondown.email/hillelwayne/archive/edge-case-poisoning/`
Understanding parser combinators: a deep dive - Scott Wlaschin	`https://www.youtube.com/watch?v=RDalzi7mhdY`

11.1 Defining basic parsers

Concept 11.1.1: Parser

A parser is a function that takes an unstructured input (a sequence of characters) and turns it into structured data (a parse tree), according to a set of rules.

We can translate this definition directly into the following type: `type Parser a = String -> a`.

The first refinement we need to make this definition usable is adding an error type to handle the case when the input doesn't have the desired form. As such we can definite the following type for the parsing result: `data ParseResult a = Success a String | Error String deriving (Show)`. In the `Success` case we return the parsed data with the remaining input and in the `Error` case we return an error message.

The updated definition now looks like: `type Parser a = String -> ParseResult a`.

Lets write a simple parser to parse the character "a":

Listing 11.1.1: Parser — Haskell code

```haskell
parserA "" = Error "End of input"
parserA (first:rest) =
  if first == 'a' then
    Success 'a' rest
  else
    Error "Expected 'a'"
```

We can use this parser as:

Haskell REPL

```
> parserA "abc"
Success 'a' "bc"
> parserA "xyz"
Error "Expected 'a'"
```

The next step to make using parsers easier is to wrap parser types instead of defining an alias: `data Parser a = Parser {runParser :: String -> ParseResult a}`.

We also need to update our simple parser as:

Listing 11.1.2 of `Simple.hs` (`parserA`) — Haskell code

```haskell
parserA :: Parser Char
parserA = Parser inner where
  inner "" = Error "End of input"
  inner (first:rest) =
    if first == 'a' then
      Success 'a' rest
    else
      Error "Expected 'a'"
```

Here we define a closure (the `inner` function) that will be wrapped by the `Parser` type. The `inner` function can directly access the input, which will be passed to the parser later.

To run the parser, we use the record field accessor function (`runParser`) to unwrap the function in the `Parser` type:

Haskell REPL

```
> runParser parserA "abc"
Success 'a' "bc"
> runParser parserA "xyz"
Error "Expected 'a'"
```

11.1.1 A more general parser

Listing 11.1.3 of `Simple.hs` (`satisfies`) — Haskell code

```
19 satisfies :: (Char -> Bool) -> Parser Char
20 satisfies predicate = Parser inner where
21   inner "" = Error "End of input"
22   inner (first:rest) =
23     if predicate first then
24       Success first rest
25     else
26       Error ("Unexpected character " ++ show first)
```

With this function we can create other parsers:

Listing 11.1.4 of `Simple.hs` (`lower, upper`) — Haskell code

```
30 lower :: Parser Char
31 lower = satisfies (\c -> elem c ['a'..'z'])
35 upper :: Parser Char
36 upper = satisfies (\c -> elem c ['A'..'Z'])
```

Exercise 11.1.1

Create a new parser using `satisfies`, `char :: Char -> Parser Char` that generates a parser that parses a given character (received as first argument).

Haskell REPL

```
> runParser (char 'a') "abc"
Success 'a' "bc"
> runParser (char 'x') "abc"
Error "Unexpected character 'a'"
```

Exercise 11.1.2

Create a new parser using `digit :: Parser Char` that parses a digit.

Haskell REPL

```
> runParser digit "123"
Success '1' "23"
> runParser digit "abc"
Error "Unexpected character 'a'"
```

Hints:

You might want to use (some of) the following functions: `satisfies`, `elem`.

11.2 Combining parsers

Parser combinators are all about creating new parsers from existing parsers *using combinator functions*.

11.2.1 Chaining (sequencing)

The first combinator is for chaining (sequencing) parsers, lets call it `andThen`:

Listing 11.2.1 of `Simple.hs` (`andThen`) — Haskell code

```haskell
andThen :: Parser a -> Parser b -> Parser (a, b)
andThen pa pb = Parser inner where
  inner "" = Error "End of input"
  inner input =
    case runParser pa input of
      Success a rest ->
        case runParser pb rest of
          Success b remaining -> Success (a,b) remaining
          Error err -> Error err
      Error err -> Error err
```

Haskell REPL

```
> runParser (andThen lower upper) "aBc"
Success ('a', 'B') "c"
> runParser (andThen lower upper) "abc"
Error "Unexpected character 'b'"
```

`andThen pa pb` tries to run parser `pa`, if it succeeds, it runs the second parser `pb`. If both parsers succeed their results are returned in a tuple.

We can use `andThen` to create a parser for a given string, expressed as a series of character parsers:

Listing 11.2.2 of `Simple.hs` (`string`) — Haskell code

```haskell
string :: String -> Parser String
string "" = Parser (\input -> Success "" input)
string (c:cs) = Parser inner where
  inner "" = Error "End of input"
  inner input =
    case runParser (andThen (char c) (string cs) ) input of
      Success (p, ps) rest -> Success (p:ps) rest
      Error err -> Error err
```

Haskell REPL

```
> runParser (string "ab") "abc"
Success "ab" "c"
> runParser (string "ab") "xy"
Error "Unexpected character 'x'"
```

11.2.2 Alternatives (choice)

The second important combinator is for choosing between multiple combinators, called `orElse`:

Listing 11.2.3 of `Simple.hs` (`orElse`) — Haskell code

```
74 orElse :: Parser a -> Parser a -> Parser a
75 orElse pa pb = Parser inner where
76   inner "" = Error "End of input"
77   inner input =
78     case runParser pa input of
79       Success a rest -> Success a rest
80       Error _ -> runParser pb input
```

`orElse pa pb` tries to run the first parser `pa`. If `pa` succeeds, it returns its result, else it runs `pb`.

Haskell REPL

```
> runParser (orElse (char 'a') (char 'x')) "xy"
Success 'x' "y"
> runParser (orElse (char 'a') (char 'x')) "by"
Error "Unexpected character 'b'"
```

We can use `orElse` to create a parser for lowercase or uppercase letters:

Listing 11.2.4 of `Simple.hs` (`letter`) — Haskell code

```
118 letter :: Parser Char
119 letter = lower `orElse` upper
```

11.2.3 Transformation

The third important combinator is for transforming the results of a parser, called `pMap`[1]:

Listing 11.2.5 of `Simple.hs` (`pMap`) — Haskell code

```
108 pMap :: (a -> b) -> Parser a -> Parser b
109 pMap f p = Parser inner where
110   inner "" = Error "End of input"
111   inner input =
112     case runParser p input of
113       Success r rest -> Success (f r) rest
114       Error err -> Error err
```

`pMap f p` transforms the result of a `Parser a` into `Parser b`. For example it can be used to transform a `Parser String` into a `Parser Int`, thus creating a number parser.

Exercise 11.2.1 *

Create a new parser for strings, `string'`, that only uses the combinators defined so far (`andThen`, `orElse` and `pMap`), without using inner functions.

[1] If only there was a typeclass that dealt with applying a mapping function to a data structure so we could implement it here ...

Exercise 11.2.2 *

Create a new parser `number :: Parser Int` that parses a number.

Haskell REPL

```
> runParser number "123"
Success 123 ""
> runParser number "123abc"
Success 123 "abc"
> runParser number "abc"
Error "Unexpected character 'a'"
```

Hints:
Use the `some`, `pMap` combinators with the `digit` parser.
Use the `read` function to obtain an integer from the string.

11.2.4 Repetition

The final combinators are utility functions for matching a parser zero or more times (`many`) or one or more times (`some`):

Listing 11.2.8 of `Simple.hs` (many) — Haskell code

```
84  many :: Parser a -> Parser [a]
85  many p = Parser inner where
86    inner "" = Success [] ""
87    inner input =
88      case runParser p input of
89        Success r rest ->
90          case runParser (many p) rest of
91            Success rs remaining -> Success (r:rs) remaining
92        Error _ -> Success [] input
```

Listing 11.2.9 of `Simple.hs` (some) — Haskell code

```
96  some :: Parser a -> Parser [a]
97  some p = Parser inner where
98    inner "" = Error "End of input"
99    inner input =
100     case runParser p input of
101       Success r rest ->
102         case runParser (many p) rest of
103           Success rs remaining -> Success (r:rs) remaining
104       Error err -> Error err
```

Haskell REPL

```
> runParser (many (char 'a')) "aaabc"
Success "aaa" "bc"
> runParser (many (char 'a')) "bc"
Success "" "bc"
> runParser (some (char 'a')) "aaabc"
Success "aaa" "bc"
> runParser (some (char 'a')) "bc"
Error "Unexpected character 'b'"
```

11.3 Practice problems

Exercise 11.3.1 *

Create a new combinator `pThen :: Parser a -> Parser b -> Parser b`. First, it runs the first parser, `pa`. If `pa` succeeds, it discards the result and runs `pb` on **the remaining input from** `pa`. If `pa` fails, the error is returned.

Haskell REPL

```
> runParser (pThen (char ' ') (char 'a')) " abc"
Success 'a' "bc"
> runParser (pThen (many (char ' ')) (some lower)) "   abc"
Success "abc" ""
> runParser (pThen (many (char ' ')) (some lower)) "abc123"
Success "abc" "123"
> runParser (pThen (many (char ' ')) (some lower)) "  BC123"
Error "Unexpected character 'B'"
```

Try to solve the problem in 2 ways:

- By creating a new parser, based on `andThen`
- By only using existing parsers (`andThen` and `pMap`)

Exercise 11.3.2 **

Create a parser `sepBy sep p` with the signature `sepBy :: Parser a -> Parser b -> Parser [b]`

Haskell REPL

```
> runParser (sepBy (char ',') (many alpha)) "a,b,c"
Success ["a", "b", "c"] ""
> runParser (sepBy (char ',') (many alpha)) "abc"
Success ["abc"] ""
> runParser (sepBy (char ',') (many alpha)) "a,,b"
Success ["a"] ",,b"
> runParser (sepBy ws (many alpha)) "a  b c     d"
Success ["a", "b", "c", "d"] ""
> runParser (sepBy (char ',') (many alpha)) ""
Success [] ""
```

Exercise 11.3.3 **

Create a new combinator `between :: Parser a -> Parser b -> Parser c -> Parser c` which takes as arguments 3 parsers: `between pHd pTl p`. It runs them in the order `pHd`, `p` and `pTl`, passing the remaining input from the one to the other. If all 3 parsers succeed, it discards the results from `pHd` and `pTl` and returns the result of `p`. If any parser fails the error is returned.

Haskell REPL

```
> runParser (between (char '[') (char ']') (string "abc")) "[abc]xyz"
Success "abc" "xyz"
```

Hints:

Think about how can you use `andThen` to chain 3 parsers. What is the type of the returned parser? How can you *transform* this type to match the required return type

(`Parser c`)?
Is there any function that already does part of this work?