Monadic Parsing

Author: Monisha Vatikuti

Class: CS 421 SU2022

Professor: Mattox Beckman

Overview

The goal of this project was to implement the functional pearl 'Monadic Parsing in Haskell'. The motivation for choosing this paticular pearl to implement was to dive deeper into monads and use them to implement a parser.

In this particular project, I followed the pearl to implemented the parser and all the different combinators mentioned in the paper as well as implemented the example on arithmetic experession provided at the end the paper. I farther extended the project to implement another example that can parse a csv format.

Note: The paper can be found at this link: https://www.cs.nott.ac.uk/~pszgmh/pearl.pdf

Implementation

First we start of by defining the parser type which is basically a representation for the parser. So this parser takes in a string and returns an output `a` of any type.

```
newtype Parser a = Parser (String -> [(a, String)])
```

Expanding on the parsers ability to return any type, the CSV parser returns a ADT that I defined to reprenset a csv.

```
data CSV
  = Number Integer
  | Text String
  deriving (Show, Eq)
```

Then the next aspect is to define the monadic parser which is the core part which will help in building out different parsers. For this we need to initiate a Monad instance for the Parser.

```
instance Monad Parser where
    return a = pure a
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = Parser (\cs -> concat [parse (f a) cs' |
                                (a, cs') <- parse p cs])
```

From here we can start to build the combinators. And this with monads, we can define differnet kinds of parsers and combine them together to be able to parse complicated formats. An example of combinator is `space` which is used heavily for csv parsing though the `token` parser. The `space` parser makes use of `sat` combinator as well as `many0` choice combinator.

```
-- parse a string of spaces
space :: Parser String
space = many0 (sat isSpace)
```

So for the CSV parser, the rules are:

- characters that fall between single quotes (i.e: `'hello'` or `'12'`) are of type `Text`
- digits that are note quoted (i.e: `12`) are of type `Number`
- row is list of items seprated by a comma
- newline character `\n` is used to seprate rows

Example CSV:

```
'Id', 'Name', 'Age'\n
1, 'Alex', 24\n
2, 'Sara', 50\n
```

To parse this csv run:

```
*Lib> parse rows "'Id', 'Name', 'Class'\n1, 'Alex', 'CS421'\n 2, 'Sara',
'CS421'\n"
[([[Text "Id",Text "Name",Text "Class"],[Number 1,Text "Alex",Text
"CS421"],[Number 2,Text "Sara",Text "CS421"]],"")]
```

The csv parser has 3 main aspects to it: This parses the input to match the ADT defined for csv:

```
csv :: Parser CSV
csv = do n <- integer; return (Number n)
      `mplus`
      do s <- csvString; return (Text s)
```

And then to actually parse the row, the follow parsers are defined:

```
-- parse a single row
row :: Parser [CSV]
row = csv `sepby` comma
```

```
-- parse many rows that end with newline
rows :: Parser [[CSV]]
rows = many1 (do r <- row; mnewline; return r)
```

This example show how powerful yet simple a monadic parse can be to implement for any type of string. This can farther expanded to use symbols other than commas to represent item separation for csv. It can be expaned to parse JSON formats or lists and many more.

The current csv parser is very limited and restrivtive to the format it can process. For example the parser will fail if there is trailing space before the newline character at the end of the row or if the row ends with a number. So if I had a bit more time I would work to make fixes to these types of issues as well as expand the funtionality to accept different kinds of csv formats.

Running Code

*Note: Added a few examples at the end of this file under the Examples Section.

To run your code, start GHCi with `stack ghci` (to load Lib.hs run `:l Lib` once GHCi started):

```
$ stack ghci
 - more output -
Ok, modules loaded: Lib.
*Lib>
```

To build the project, use `stack build`:

```
$ stack build
```

To run the tests, use `stack test`:

```
$ stack test
```

Tests

Wrote unit tests using `Test.HUnit` library to test the code. There are tests for most of the implementation. I designed tests for the monadic parser such as `fmap`, `pure`, and more as well as for the combinators such as `sat`, `space`, `digit`, and more. Additionally, there are more of functional tests for the arithmentic expressions and the csv parser.

There are a total of **46** tests written. And out of those **8** tests for arithmentic expressions and **9** tests for csv parsing.

Full details of the tests can be found in `test/Spec.hs`.

Listing

All the implementation for the parser as well as the examples are in `src/Lib.hs`.

The tests for some of the major implementation bits are in `test/Spec.hs`.

Citation

Hutton, G., & Meijer, E. (n.d.). Functional pearls monadic parsing in Haskell - cs.nott.ac.uk. Retrieved July 4, 2022, from http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf

Haskell Documentation for `Data.Char`: https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-Char.html

Examples

For testing code in `src/Lib.hs`

item

Parser `item :: Parser Char` is basis for all the following parsers. This takes string argument and consumes the 1st character. If the input is an empty string, it fails and returns an empty list.

```
*Lib> parse item ""
[]
*Lib> parse item "abc"
[('a',"bc")]
```

Functor Parser

Parser for `fmap :: (a -> b) -> Parser a -> Parser`.

```
*Lib> parse (fmap toLower item) ""
[]
*Lib> parse (fmap toLower item) "AB"
[('a',"B")]
*Lib> parse (fmap toLower item) "ab"
[('a',"b")]
```

do notation

Example `p :: Parser (Char, Char)` which takes in 3 characters and return a pair of 2 by throwing away the middle character.

```
*Lib> parse p ""
[]
*Lib> parse p "ab"
[]
*Lib> parse p "abc"
[(('a','c'),"")]
*Lib> parse p "abcd"
[(('a','c'),"d")]
```

char

`char :: Char -> Parser Char` uses `sat` to check if the consumed character is of type char. If it is a char type it will return the item as is, else will return empty `[]`.

```
*Lib> parse (char 'a') "bc"
[]
*Lib> parse (char 'a') "abc"
[('a',"bc")]
```

string

`string :: String -> Parser String` will parse a specific strings.

```
*Lib> parse (string "foo") ""
[]
*Lib> parse (string "foo") "foo"
[("foo","")]
*Lib> parse (string "foo") "foobar"
[("foo","bar")]
```

many0

`many0 :: Parser a -> Parser [a]` allows for zero or more applications of the parser. This is analogous to the regex * matcher.

```
*Lib> parse (many0 (char 'a')) ""
[("","")]
*Lib> parse (many0 (char 'a')) "baa"
[("","baa")]
*Lib> parse (many0 (char 'a')) "aaa"
[("aaa","")]
```

```
*Lib> parse (many0 (char 'a')) "aaabba"
[("aaa","bba")]
```

many1

`many1 :: Parser a -> Parser [a]` allows for one or more applications of the parser. This is analogous to the regex + matcher.

```
*Lib> parse (many1 (char 'a')) ""
[]
*Lib> parse (many1 (char 'a')) "baa"
[]
*Lib> parse (many1 (char 'a')) "aaa"
[("aaa","")]
*Lib> parse (many1 (char 'a')) "aaabba"
[("aaa","bba")]
```