LAMBDA CALCULUS PARSER ASSIGNMENT REPORT

Name : Yeoh Ming Wei
Student ID : 32205449
Subject : FIT2102

Contents

Introduction

In this assignment, we were required to create a lambda calculus parser using Haskell language which involves lambda expressions, logicals, operators, comparisons, lists and lists operators.

The first part of the assignment is to create a Backus-Naur Form (BNF) grammar which accepts long and short lambda expressions as a string and convert it into a Haskell language which need to parses a String type expression to a Lambda expression.

The second part of the assignment requires us to combine logical, arithmetic and comparison operators and create a complex calculator.

The third part is addition of list with its own list operators such as isNull, head, tail and many more.

Design of the code

The code is well made thanks to Mr. Tim's Parser Combinators example as it provides me a better understanding of how to use the parsers and create my own parser combinators. Besides that, checking the source files that contains Parser, Builder and Lambda files helps me a lot as you can check what the type of the function given such as "string" function that returns a Parser String. The examples are also given in some of the files so that we know how to use the functions.

The parser combinator is also easier to code when you use do notation. Instead of using bind (>>), we can use do notation which gives us syntatic sugar, more easier to read the code.
The process of making the parser combinator is to create multiple small parser function such as, creating parser function for logical operators like "not", "or", "and" many more. After done creating it, we combine the functions and assigned it into a parser combinator. So, the parser combinator allows us to perform multiple parsers depends on the condition.

I had created other design choices for the first part of the question which I had a parser combinator that can be used for both long and short lambda.

Parsing

This is the BNF Grammar I had created for Part 1 Exercise 1:

<expr> ::= "(" <spaces>* <expr> <spaces>* ")" | <lambda> "." <body>

<lambda> ::= <spaces>* <lambda> <spaces>* <lambda> <spaces>* | "\\" <alphabet>

<body> ::= <alphabet> | <expr> | "(" <body> ")" | <spaces>* <body> <spaces>* <body> <spaces>*

<alphabet> ::= [a-z] | <spaces>* <alphabet> <spaces>* <alphabet> <spaces>*

<spaces> ::= " "

A string can be represent as a String type or can be split into multiple characters which one character is consider as a Char type. So, we can use parsers for String and Char type. One of the most commonly used parser function for my code is "string" function as you will need to parse a full word a lot during the process, a few example of strings are "True", "False", 'and" etc.. Besides that, instead of using "char" function, we can also use "string" function that also able to parse a single character.

However, some strings input contains spaces in between characters or strings. So, we can create a parser combinator ("spacesstr" function in LambdaParser.hs) to filter the space and check if the string is valid and return string type parser. It is helps me to reduce errors and redundant codes.

As for the Part 1 which is parsing a lambda expression, I used "oneof" parser function to parse only alphabet characters (A range of alphabet from "a" to "z", no case sensitive) so that I do not need to type out all the possible values using 'is" parser function.

Example:

```
alphabet = oneof [a..z] -- Better
alphabet = is 'a' ||| is 'b' ||| … -- Redundant
```

Then all the lambda expression can be parse out using the "list" function which parse out all the alphabets of the lambda term and wrap it as a list. It is useful to create a partial applicative function just like a curried function that returns a function if you did not use all the parameter.

(|||) parser function is used when a parser fails, it will try the next parser function.

Parser combinator is a higher-order function that combines several parsers into one new parser. It is usually creating multiple smaller parser and combine it into one big parser. One of the useful function that I obtained from parser combinator notes which is the "chain" function that can handle repeated combinations of unknown length. As an example for the parser combinator for arithmetic expressions, firstly I create an expression for all the arithmetic operators such as +, -, *, **, specifically a Builder type for the expressions. Then, I create a parser for each of the operators which parse the string and return the specific operators function. If we given a "+" string, it will parse the string using "spacesstr" function and bind to the add expression which is a Parser Builder type. The bind function (>>) do not take any values and allows us to chain the actions. Lastly, we have the chain function which chain all the function for the operators and at the same time we need to careful for the precedence of operators. The function will find the number first and bind with the specific operator that takes in two combination of Builders. It will loop until the string is empty.

As most of my parser combinator using do notation and bind function, it is actually a Monad typeclass which allows is to perform another type of function application over values in a contexet. A monad typeclass is also a subclass for Applicative. Therefore, you can also use the pure function in Applicative. The bind operator (>>=) also requires a Functor and Applicative to create a Monad instances. We can also call functor or fmap in the do notation too. So, the parser combinators are related to Functor, Applicative and Monad typeclasses.

Functional Programming

After I did the coding for this assignment, I can confirm that Haskell is using Functional Reactive Programming style as the way it codes is almost similar as the coding that I did for my first assignment, but with Javascript. Firstly, Haskell codes does not use any of the global variables which means that there is no side effect. The parser function is also pure as the functions will return the same results everytime you had the same input. There is also a similar curried function way of coding which you can partially appllied values into the function without using all the parameters, and it will return a function. This can be seen at Part 1 Exercise 2 where I parse all the lambda terms into a list and partially apply the "lam" function to the elements in the list. So, there is a list of partially applied "lam" function. The process of creating many small modular function and combine it into a big complex function can also be seen at my Assignment 1. Haskell has its own declarative style by converting a function to point free style which can "hide" the parameters by refactoring. This shows that the way I code at Haskell language is almost the same as coding using RxJS in Javascript which also follows FRP style.

Haskell Language Used

I had used some of the language when coding with Haskell. The typeclasses that I used for my code is Functor, Applicative and Monad typeclasses. You can refer to the last part of "Parsing" section. I also used fmap function to map all the values in its function. You can see most of the function such as arithmeticP is require to call the build function to change from Builder to Lambda, but we got the result of Parser Builder. So, we need to use fmap to take the value out of the context and apply the function using fmap. In this case, we take the Builder value from the parser and apply build function. I also tried to use the bind function which is needed because bind function can perform an action without needing any value. For example, most of my parser code binds an action of when a string is valid, it does not return any value and proceed to another action which is produce a lambda expressions with Parser Builder as its typeclass. We have a higher order function that allow us to put any function as a parameter or return function. One of the example we can see is the chain function which accept Parser [a -> a -> a] as its second parameter. We can change the code functionality by putting any kind of function such as the add, minus and multiply operators that can take two values as input and returns an output.