Name: Kang Zheng Rong
Student ID: 32797990

Design of the code

The code for my assignment follows the structure of the given example calculator and weekly tutorials. My main goal is to create a general parser that can parse the general expressions of each part. However, to achieve this I would need a few smaller parsers that are able to parse the keyword such as "λ", ".", "head", logic operator, arithmetic operator, etc. Therefore, For each part, I will code the builder for the corresponding keyword first then the parser for it. Finally, I will make a general expression for that part that chains and combines all of the required parsers into a parser for the general expression of that part. This approach is usable for most parts of the assignment.

In terms of high level structure of the code, the parser combinators used are

- Chain function introduced in the parser combinators course note which will handle the repeated chains of operators.
- Functions that use (|||) to apply alternate parser
- Functions that use chain to chain multiple parsers together for the input string.

In terms of code architecture choices, I am using the parser instance of Monad's bind function and do notation of Monad.

Name: Kang Zheng Rong
Student ID: 32797990

BNF Grammar for lambda expression

<lambdaExpression> ::= <longLamExp>|<shortLamExp>

<longLamExp> ::= <longLamBody>
<longLamStart> ::= “λ” <var> “.” <longLamBody>
<longLamBody> ::= <longLamStart>|< longLamParen>|<longLambdaTerm>
<longLamParen> ::= “(“ <longLamBody> “)”
<longLamTerm> ::= <var><longLamBody>

<shortLamExp> ::= <shortLamStart>|<shortLamParen>
<shortLamStart> ::= “λ” <var> <shortLamHead>
<shortLamHead> ::= <shortLamParam>|<shortLambdaParen>|<dot>
<shortLamParam> ::= <var><shortLamHead>
<dot> ::= “.” <shortLamBody>
<shortLamBody> ::=
<shortLamStart>|<shortLambdaParen>|<shortLamdaTerm>
<shortLamdaTerm> ::= <var><shortLambdaBody>
<shortLamParen> ::= “(“ <shortLamBody> “)”
<var> ::= [a-z] | [A-Z]

Parser combinators are widely used in my code. Almost all of the functions are parser combinators that chain multiple parsers together giving a parser for the expression eventually. It would be troublesome if the data type of these parser combinators are Parser Lambda as I need to use the build function every time to convert them into Parser Lambda. Therefore, Parser Builder data type is preferred. I only need to use the build function at the end. The general parser for the expression is using the fmap from functor which we apply build function to the builder expression. In fact, the bind function and do notation from Monad are widely used in our code.

Name: Kang Zheng Rong
Student ID: 32797990

Functional Programming

Small modular functions are used in my code instead of a long function that does everything because it does better in terms of cleanliness and readability. Besides, it will be way easier to debug if we are composing all those small modular functions together. We can run the code and check for the error then change the related small modular functions. On the other hand, I have used declarative style programming in this assignment that describes what the code does instead of specifying the control flow. With this, I am able to minimize the data mutability, increase code readability. Moreover, all haskell functions are pure so we do not have to worry about side effects.

Haskell Language Features Used

As mentioned above, we have used haskell type classes like Functor, Applicative and Monad all over the code. Beside from the do notation from Monad, functions from these type classes such as fmap, bind and apply helps in wrapping and applying parsers, function to our input. As we need to create an expression parser for each part, we need to make use of function composition to be able to chain functions and parsers together. Some of the time, the built-in function is not enough for our use. Therefore, we need to do some leverage to it in order for them to be useful in our code. With leveraging, we only need to modify the built-in function instead of creating a whole new code.