Assignment 2 Report:

Name: Jin En Tan

Student ID:31336574

Code Design:

Do note that I took a lot of reference from these url: https://youtu.be/dDtZLm7HIJs and https://tgdwyer.github.io/ . The trick of coding for the tasks heavily relies on the BNF grammar, basically the BNF grammar is converted into the code for longLambdaP as well as shortLambdaP so that these parsers match with the BNF grammar. Finally the parser lambdaP is basically just make use of parser combinator (|||) to combine both parsers above in order to parse in lambda calculus expression. All the code are structured in the similar way where I used the keyword 'where' and 'do notation' in order to code all the parsers after referencing from the stated sources. This is because 'where' allow us to create multiple function definitions that are visible within the scope of parent function, but they must be left-aligned with each other and to right of the start of the line containing the 'where' keyword, as can be seen in the source code file so I will not include any screenshot here. Do notation allows binding in form of a whole 'do block' and is extremely useful in defining the parsers in a similar way to match the BNF grammar. Both of these greatly improves the readability of the code in source file. While for the rest of tasks, it requires the understanding of Boolean logic and arithmetic expression in church encoding form then I just code it out.

Parsing:

BNF Grammar for Long Lambda:

<var> ::=
"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"

<body> ::= (<var> | <parens>) | (<var> | <parens>)<body>

<parens> ::= "(" <body> |<simpleLong> ")"

<simpleLong>:: = <lambda> <var> <dot> <body>

<long> ::= <simpleLong> | “(” <simpleLong> “)”

<longLambda> ::= <long> | <long> <longLambda>

<lambda> ::= "λ"

<dot> ::= "."

BNF Grammar for Short Lambda:

<var> ::=
"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"|"_"

<body> ::= [<var> | <parens>] | [(<var> | <parens>) <body>]

<parens> ::= "(" <body> | <simpleShort> ")"

<v> ::= <var>|<v><var>

<simpleShort> ::= <lambda> <v> <dot> <body>

<short> ::= <simpleShort> | "(" <simpleShort> ")"

<shortLambda> ::= <short> | <short><shortLambda>

<lambda> ::= "λ"

<dot> ::= "."

Parser combinator is a higher order function that accepts parsers as input and combines them to form a new parser. The usage of parser combinator can be seen a lot in the source file, a simple example will be the lambdaP function define in the source file. To elaborate, basically we want a general parser that can parse long lambda calculus form as well as short lambda calculus form, so the easiest way to do this is to combine the parsers for both long and short lambda. Through composition using instances of Functor, Applicative and Monad, I am able to build up sophisticated parsers for different input grammars, this is described in Tim's note as well. Basically, a parser is constructed using fmap, do notation block, as well as <$> and it can only be done because parser is defined as instance of Functor, Applicative as well as Monad type classes.

Functional Programming and Haskell Language Features Used:

As explained above, 'where' keyword is used to define multiple functions definitions under parent function. This would improve the readability since we could know the task performed by the functions directly from implementation under 'where' and parts of the parent function is bound to 'where' which provides direct access to those results. The 'do notation' block which is a syntactic sugar for bindings operation in parser makes it a look more like declarative style and since Haskell itself is a functional programming language so everything in the source code file is following the functional programming style. All of it provides faster execution process and allow lazy evaluation to work as well as improving the readability of code. Higher order functions are used when creating a parser combinator as a parser combinator needs to accept parsers as input and combines them into a new parser. Fmap allows us to create a new Parser by composing functionality onto the parse function for a given Parser, as stated in Tim's note which is applicable in the source code file as well. As for built in functions, functions such as fmap, foldl are two of the built-in functions which I used in order to save time without having to code functions with the same functionalities which reduce redundancy.

Conclusion:

In conclusion, most of my codes have the same style for this particular assignment, which extensively uses ‘where’ and ‘do notation’ block which I think is not a good thing because it’s kind of boring to code everything in a similar way. However, through the usage of Haskell in this assignment, it can be shown that the parser codes are similar to the structure of BNF grammar itself.