Report

Code Design

The design of the code started off by creating parsers for simple combinators to become acquainted with lambda expressions and their structure. Ground-level parsers were created which assisted throughout the assignment as every function more or less required them.

The code design seeks to break down large functions into smaller ones, such as in *longLambdaP and shortLambdaP*, which relies on smaller helper functions to aggregate so that it is simpler to solve a difficult problem, and to minimise code length by leveraging function searches from Hoogle. Making tiny maintainable functions allows for improved testing since we can guarantee the correctness of bigger functions by ensuring small functions are correctly implemented.

Using extensive nested do blocks for task 2a for parsing logical clauses might be a weakness in the code design. However, I feel it's a fine alternative to the chain functions which I didn't know how to use until task 2b.

BNF Grammar

A parser combinator is a higher-order function that takes in parsers and combines them in some way to create a new parser.

The long form syntax follow a constant starting pattern the ambiguity start after the "." Therefore, for long form I hard coded the initial part then used recursion to show the later half, as we don't know how many non terminals are there.

The short form syntax follows 2 initial patterns one with brackets another without, therefore I created 2 separate parsers for each of them and did the ambiguous part same as long form.

The pure applicative generates a Parser that always succeeds with the provided input and so serves as a foundation for composition. In the above example, it was used to return the results of a parse back into the Parser at the conclusion of a do-block.

The (<*>) allows us to map functions from one Parser to another. A frequent use case, like with other Applicative instances, would be composition with a Parser that yields a data function Object().

The Monad instance's bind function (>>=) , allows us to sequence Parsers in do-blocks to build up the BNF grammar implementation.

BNF Grammar

<lambdaExpression> ::= <longLambdaP> | <shortLambdaP>

<longLambdaP> ::= <bracketR> <lambda> <alphabet> "." (<longLambdaP>* | (<alphabet> | <bracketR> <contAlpha> <bracketL>)+) <bracketL>

<alphabet> ::= "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"

<contAlpha> ::= <alphabet>+

<bracketR> ::= "("

<bracketL> ::= ")"

<lambda> ::= "\\"

<shortLambdaP> :: = <withoutBracketShortLambdaP> | <withBracketShortLambdaP>

<withBracketShortLambdaP> ::= <bracketR> <lambda> <contAlpha> "." <contAlpha> <bracketL> <withBracketShortLambdaP>*

<withoutBracketShortLambdaP> ::= <lambda> <contAlpha> "." (<alphabet> | <bracketR> <contAlpha> <bracketL>)+ <withBracketShortLambdaP>*

Functional Programming

Due to the declarative structure of Haskell, we can't avoid utilising recursion, like in the improved repeatSequenceL function, where we utilise recursion to retrieve the proper long form syntax. Moving on, we should always specify type signatures for our functions and constants rather than relying on Haskell's type-inference scheme, which is done for all functions.

Using small modular functions is vital in ensuring code clearity and functionality. Which is clearly evident in task 2 where a Builder function was made for every arithmetic operation.

Composing small functions helped me to debug tasks as I didn't have to go through long pieces of code, identifying a bug was easier. Function *exprB* of task2b is an example of how I created different smaller functions and chained them together using *chain* functions.

Task 2a is where I think I could have improved by creating more smaller functions and chaining them together rather than creating one large nested do function.

Haskell Language features

Almost every part of code attempted to implement Haskell features and functional concepts ranging from simple pattern matching to Applicatives. We can see in the code that pattern matching is utilised to identify natural numbers into operators when using arithmetic operators on any two numbers. We learned how to conduct eta reduction, operator sectioning, and composition from lambda calculus, and it is also commonly utilised in my code, such as in the foldr and foldl for withBracketShortLambdaP to increase code readability by removing unneeded parameters from the function.
The foldable typeclass is very useful in my project because foldr and foldl1 are frequently used to combine list instances. Foldr is used with *lam* function whereas *foldl1* is used with term function respectively.

In most of the helper functions I created I used the type of return Parser Builder as all the composite function work to create 1 Builder which I converted to Lambda data type in the main functions using fmap (<$>) to convert from Parser Builder to Builder.

On various occasions in my code you may see the Prelude.map function being used to apply all 'chars' to the *term* function for getting a correct Builder type. Monads are frequently used in parsers, such as chkSOper, where do notation is employed instead.
I also made use of Bind >>= , as seen in the *chain* function to,