Assignment 2 Report

Design of the Code

In order to implement each parser I approached the problem in an incremental way, building each individual block until we are able to build the final parser. This consisted of building individual parsers for characters such as a lambda character. Using these parsers together to build simple parsers for terms that create a Builder type. Finally combining all these parsers together and building it into the desired lambda.

This approach allowed me to strip down the problems and focus on the smaller parts first which made it much easier to code. It allows the code to be structured in a way that is easy to read and flow as it is building up to the final parser. Along with this it makes it much easier to isolate problems and solve them as you can test each component individually for errors.

Parsing

longform

```
<expression> ::= "(" <lambdaChar> <variable> "." <term>+ ")"
<term> ::= <expression> | <variable> | "(" <term>+ ")"
<lambdaChar> ::= "\\"
<variable> ::= <spaces>? [a-z] <spaces> ?
<spaces> ::= " "*
```

Shortform / general

```
<expression> ::=  <lambdaChar> <variable>+ "." <term>+ |  ("(" <expression> ")")+
<term> ::= <expression> | <variable> | "(" <term>+ ")"
<lambdaChar> ::= "\\"
<variable> ::= <spaces>? [a-z] <spaces> ?
<spaces> ::= " "*
```

The lambda calculus parsers are implemented using the grammers shown above. Note that any long form expression can be represented in the short form grammar.

In general parsers for individual characters are defined to improve readability and reusability of the code. These are mainly done through parsers such as 'is' or 'string'.

The usage of parser combinators can be seen in a lot of the terms and expressions which easily allow the BNF grammars to be translated to code. Individual parsers are combined with the (|||) operator which mirrors the '|' in the grammars. This is extremely useful as it helps us create parsers that are versatile and can handle many different cases eg. in <term> where it can parse itself with brackets. One difficulty faced with doing things such as this though is the infinite recursion that may occur if the term is left recursive. These instances were solved by using the list function and then folding over the list.

Another useful parser combinator is the chain combinator. This allowed an easy implementation of operators for arithmetic parsing and handling precedence as parsers for operators with higher precedence can be chained with parsers for operators with lower precedence eg. boolTerm for boolean operators. To use the chain the parsers for the operators needed to return a Parser (Builder -> Builder -> Builder) type. This was done by composing two *ap* functions together with the lambda equivalent for the operator. This showcases the use of haskell features such as function composition and using modular functions that can be used in many different contexts allowing us to create a simple parser of arithmetic operators.

The different typeclasses in Haskell such as the functor, applicative and monad were extremely useful as most of the values were contained in a Parser. These type classes allowed us to manipulate the values inside the Parser easily e.g. using fmap to apply '*term*', which takes a Char to '*variable*' which is a Parser Char. Monadic binds and do blocks are useful as it allows us to use parsers in a sequential manner and save or throw away results as needed. Eg. parsing spaces(throw away) then a character (save). We can manipulate these values as desired and then put them back inside a Parser context, keeping the code pure.

We manipulate these values in different ways. Eg. We use '*list1 variable'* and take it out of its Parser context. We then fmap lam to each variable in the list and use *foldl* with the (.) operator to create nested lamba functions. This sequence is a good example of functional programming as we are composing functions and using different functions in a way in which we are able to get the desired result and still maintain purity.

Extension

Parsers for fibonacci and factorials were implemented for the extension task. These were both done by first defining recursive lambda expressions for each function and then applying the z combinator which allows us to recurse those functions with reductions.

The lambda expression for fib was done by using nested if statements to define the base cases. Here's what the lambda expression translates to.
fib(n) = if (n = 0) then 0 else (if (n = 1) then 1 else (fib (n -1) + fib(n-2))
A similar thing was done for factorial.
Factorial (n) = if (n>1) then (n* Factorial(n-1)) else 1

Parsers were then defined for each of the functions. We use the parser of arithmetic expressions for the parsers so we can do things such as fib(1+3) or (5*2)! . It is also hard to calculate any of these functions with high values such as 10! As it takes ages to run. This is probably due to the fact that a number is a bunch of nested functions and a big number is a lot of functions.