Gus Morris

Code design

The main driver for the design of my code was the Grammar that defined our Lambda Calculus, as well as taking inspiration from the previous tutorial in which implemented a calculator using a context free grammar. With what could be considered the blueprint being laid by the Grammar used, design choices unique to functional programming were used such as modular functions and combinators were used in order to improve readability and ensure correctness. At a high-level a grammar function was used to assign a parser choice depending on what is parsed through our parser, a chain function was used to link the terms together.

BNF Grammar

```
<expr> ::= <var> | λ <var> . <exp> | (<exp> <exp>)
<var>   ::= <var><var> | [a .. z]
```

The grammar presented here satisfies both the short and long lambda parser we use to parse strings within our code. The <expr> non-terminal within our grammar could be considered regenerative for a lack of a better word meaning that our expression can be built from a single <expr> into an entire lambda terms once filled entirely with non-terminals

Throughout the code much of it uses parser combinators, such that the parser functionality is chained together, examples being both the short and long grammar functions which uses combinations of the variable, body and brackets expressions. These combinations are mostly linked together using the functor and monad type classes, which is especially important as the values in which are used to build the lambda expressions are often wrapped within the parser context. Other examples of this is the long and short expression functions which chain together the terms. On a more basic level parser combinators such as those including the Tok parser are used in order to return a specified parser value while ignoring the spaces that would occur following being parsed which are redundant within the strings we are trying to parse.

Functional programming

By using Haskell our code adheres to strict functional programming. As a result the code in which is written is pure and free of side affects making the code easier to debug, and also allows an easy route to proof of correctness. Throughout the code this is mostly done by function composition as well as a point free style. By incorporating these functional features within our code we improve readability. Some times the most basic of a function is used in order to specify variables so that these are all lower cases, and once this is used within our code it becomes less ambiguous in our code given that the function is named appropriately.

The Haskell language

The Haskell functional language provides the user with many features not available with typical imperative languages. The library within haskell provides many high-order functions that can be implement. An example of this is the foldr function and is implemented within my code. By using foldr, it provides a sequential control flow in which the values in which I which to reduce into a single value can be done depending on the reducible function provided. Another example used is the fmap function which is both seen by it's name 'fmap' or used as an infix way by use of <$>. By using the fmap function we ensure only creation and no modification which also applies to the bind function which is more notably seen within the do block of the code. This provides a sequential control flow that does not rely on storing values within variables. By eliminating variables within the code it ensures that no side effects can occur within other parts of the code which provides many benefits, most notably making our code easier to debug.