Lim Zheng Haur

32023952

FIT2102 Assignment 2 Report

Design of the code

The structure of my code is mainly using the do notation of Haskell to parse the string. I make use of the provided parsing functions, Lambda and Builder functions to complete most of the task. The parser is built with a character parser as a base and then built upon, using it to create Parser Builders and then using these to create Parser Lambda. I have stick to using do notation for the syntactic sugar. Inside my code, I have used functor and applicative in different areas, for example using <$> functor to build the builder in a context. The parsing of Part 1 of this assignment is done according to the BNF grammar that I have created. However, for Part 2 of this assignment, the code makes use of the chain function to chain different parsers together as a Parser Combinator. This chain function is adapted from FIT2102 Tim's note on Parser Combinator.

Parsing

```
| Part 1 Exercise 1
| BNF grammar for lambda calculus expression
<lambdaP> ::= <longLambdaP> | <shortLambdaP>
<longLambdaP> ::= <builderLong>
<shortLambdaP> ::= <builderShort>
<builderLong> ::= <open> <lambda> <var> <dot> <bodyLong> <close>
<bodyLong> ::= <builderLong> | <parentTermMid> | <parentTermFront> |
<parentTerm> | <parentTermBack> | <terms>
<builderShort> ::= <parentShortTwo> | <parentShortFront> | <bodyShort>
<bodyShort> ::= <lambda> <var>+ <dot> <shortTerms>
<shortTerms> ::= <parentTermShortBack> | <parentTermMid> |
<parentTermFront> | <parentTerm> | <parentTermBack> | <terms>
<parentShortTwo> ::= <open> <bodyShort> <close> <open> <bodyShort> <close>
<parentShortFront> ::= <open> <bodyShort> <close> <bodyShort>
<parentTermShortBack> ::= <terms> <open> <bodyShort> <close>
<parentTermMid> ::= <terms> <open> <terms> <close> <terms>
<parentTermFront> ::= <open> <terms> <close> <terms>
<parentTerm> ::= <open> <terms> <close>
<parentTermBack> ::= <terms> <open> <terms> <close>
<terms> ::= <var>+
<var> ::= [a-z]
<lambda> ::= "λ"
<dot> ::= "."
<open> ::= "("
<close> ::= ")"
```

Above is the BNF grammar that I have created for Part 1 of this Assignment. The structure of my parser in my code is built upon this BNF grammar. Starting from Parser Char of "lambda", "var",

“dot”, “open”, “close” and Parser Builder “terms”, “parentTerms” and so on. I utilized the provided parsers in the code bundle such as “is” and “string” for parsing characters and string respectively. I also used “list” to parse characters repeatedly until an unexpected character appears. These have allowed me to create my simple character parsers such as “open” which is a character parser for the open bracket character ‘(‘. The parser combinator provided in code bundle “|||” was used extensively in my code. This allows me to apply different parsers to the string until one works. I have also used the “chain” function provided in Parser Combinator Notes but adapted it to fit the Builder. This modification allows me to chain Parser Builders together. This feature is used mainly in Part 2 of this assignment. Many of the parser combinators are constructed using the do notation by using monadic binds. I have applied my functor knowledge on most of the Parser Lambda to build from the Parser Builder.

Functional Programming

I have adhered to using small modular functions in my code. This allows my code to be easier to understand and manage as the code size grew. This also allows me to reuse many of my codes such as using character parsers in Part 2 that I have already created in Part 1. There are also many codes reused from Exercise 1 of Part 2 in Exercise 2 and Exercise 3. This displays the effectiveness of adopting small modular functions. Using the small modular functions, we apply different programming syntax styles to ease readability. For example, we used infix and prefix operators to build Builders to ease readability.

Haskell Language Features Used

I have used many different typeclasses of Haskell such as Maybe and Monads. I also used custom typeclasses provided by the codebundle such as Lambda and Builder to implement my code. To modify data of these typeclasses, I have used functor and applicatives such as “fmap” and “bind”. Haskell has also allowed me to use functions such as “foldr”, “foldl”, “fmap” and many more to ease the implementation of my code. These allows my to traverse through foldables in my code such as parsing a string of “terms” in Part 1 and applying them using the “ap” function provided in the code bundle.