Report

FIT 2102 Assignment 2

Name: Rebecca Nga
Student ID: 30720591

Introduction:

The purpose of this report is to provide a high-level overview of the assignment completed from parts 1-3, creating interpreters for parsing lambda expressions, simple arithmetic and boolean operations and extending the interpreter to handle more programmatic operations.

Part 1

Exercise 1: BNF Grammar for lambda calculus expressions

<lambdaP> ::= <longExpression> | <shortExpression>
<variable> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "x" | "y" | "z"
<variableBuilder> ::= <variable>* | <variable>*
<parameter> ::= "(" <variableBuilder> ")"
<expressionBody> ::= "(" <variableBuilder> | <parameter> ")"
<alternate> ::= <expressionBody> | <expr> | <variableBuilder> | <parameter>
<expr> ::= "(" "/" <variable>* | <alternate> ")"
<longExpression> ::= "(" <expr> | <expressionBody> ")"
<longLambdaP> ::= <longExpression>
<shortExpressionBody> ::= <variableBuilder> | <parameter>
<shortExpr> ::= "(" "/" <variable>* | <shortAlternate> ")" | "(" <variableBuilder> ")"
<shortExpression> ::= "(" <shortExpr> | <shortExpressionBody> | <shortExpr> ")"
<shortAlternate> ::= <shortExpressionBody> | <shortExpr> | <variableBuilder> | <parameter>
<shortLambdaP> ::= <shortExpression> | <longExpression>

Description for BNF Grammars :

The BNF Grammars created were to help the process of building the first part of the assignment, parser functions longLambdaP, shortLambdaP and lambdaP. The BNF Grammar helped me complete the assignment as it helps to visualize and split up necessary parts of the lambda expressions and help the coding process be more efficient in organisation and time consumption. The BNF Grammar created was the backbone of Part 1 and how the code is coded follows the BNF Grammar heavily. Though the order of the code and the BNF Grammar is not exactly the same, the reason is that when in the assignment code, the position of the code acts as a flow chart and helps with clarity in finding the errors.

Exercise 2: Parser for long-form lambda calculus expression

To complete the interpreter for long-form lambda calculus expressions, a simple BNF Grammar was created. Then BNF grammars were continuously updated and improved to suit the various breakdown functions for the main interpreter for parsing long-form lambda

calculus expressions and short-form lambda calculus expressions. The order of the BNF Grammar was also updated to match the code created, small functions with types Parser Builder, Parser [Char] and Parser Char types are created in order according to the BNF Grammar's expressions created. The primary expression function longExpression with type Parser Builder consists of the combination of all the other smaller functions for organising the code and so it's easier to trace errors and remove bugs. Also, the use of parser-combinators is used to complete the assignment and to apply the small functions created to combine parsers. As can be seen in the BNF Grammar, a variable with type Parser Char is created to keep all the alphabets from [a-z], whereas variableBuilder is created to hold either a Char or a list of Char ([Char]). And parameter is created to hold the brackets from the lambda expressions, and expressionBody is created to hold the main body for the expression. The function expr is another small sub-expression to build the required lambda expression, and longExpression is the combinator to combine the required functions to create the interpreter for the lambda expressions.

Exercise 3: Parser for short-form lambda calculus expression

Similarly to the long-form lambda calculus expression, is very similar to the first part of the exercise, but it is modified to fit the requirements of the short-form lambda calculus expression. The functions for shortLambdaP are created to reuse functions from the longLambdaP, this is to maintain the code reusability, this is also to prevent errors and when recoding parsers to help with this function, we only need to edit it minimally to suit short lambda calculus. There are very similar functions created to deal with this task, and is the same order as well, as it is the short form of lambda calculus expression, so it is very similar to the long form lambda calculus code.

Part 2

Exercise 1: Parser for logical statements

The church encodings for boolean constructs are referenced from https://tgdwyer.github.io/lambdacalculus/#church-encodings. With this reference and the help from PASS sessions, assignment specifications, Ed discussion forums and consultation sessions, for the Parsers for the logical statements, I created the necessary logical statements, and, not, or, if, and true, false statements. As some of the expressions are too long and hard to detect errors, I created builders for them, such as the IF builder as the and, not and or statements need the if builder, it was easier to just split them out. After creating these statements, I have created multiple expressions functions that handle the various logics that are required to build the interpreter such as the logicExpression, trueFalseExpression and thenElseExpression to handle various logical cases. Like Part 1 of the assignment, there is a main Parser Builder function that handles the main logic of the code and finally, it's built in the main logicP function to build. I map out the requirements for logical statements and I have built them according to the order similarly to the course notes for easy checking, editing and debugging. The code is organised and have a clear construct on how the code should flow. I've also used functions provided such as boolToLam for the evaluation of true or false.

Exercise 2: Parser for arithmetic expression

The parser for the arithmetic expression is created step by step as well, the first step was to create the integers, with code referenced from the Parser Combinator's course notes, https://tgdwyer.github.io/parsercombinators/ and then an integerBuilder is created similar to the longLambdaP code, where its suppose to take in lists of digits, whereas the op and chain functions were referenced from the Parser Combinator's course notes as well, and used in my code, there are used as we need those functions for chain function and to check for spaces. Then, the operation function holds 2 operations, addition and subtraction, where it follows the requirements of the church encodings from the assignment file provided, the helper functions are also coded according to the assignment sheet that was needed and was coded separately and then later on added into the operation block. This operation function will later be chained with the integerBuilder in another function named arithmeticExpression and then built in the final code, basicArithmeticP. The moreOperation expression holds the other operations like multiplication and exponential and brackets, the code done is similar to the basicArithmetic function, the difference is that the precedence is different.

Struggles with the assignment & Conclusion:

The main struggles with the assignments were handling the types of parsers, and type conversions with functions provided by the assignment specification. The logic of each BNF Grammar also was something challenging to figure out, as well as chain functioning and recursing the functions. When beginning the assignment, I heavily referenced the example from this Ed discussion post, https://edstem.org/au/courses/8839/discussion/1073871 , then later with more understanding I began to edit more of the code and manage to figure out what the specifications wanted, with consultations and pass sessions, I realised that I could not use 'return' in my code as it would frequently give me mistakes, especially during typing conversions from Parser Builder to Parser Lambda type, so I used 'pure' instead to ease the coding process. To conclude this report, I have learnt a lot from this assignment, even though I did not manage to complete the full assignment, but I managed to do as much as possible.