Name: Loh Jing Wei

Student ID: 30856183

FIT2102 Programming Paradigms

Assignment 2

BNF for long lambda

<longLambdaP> ::= <longLambda>
<longLambda> ::= "(" <longParam> <longExpr> ")"
<longParam> ::= "λ" <lambdaVar> "."
<longExpr> ::= <longExprCont> | <longLambda>
<longExprCont> ::= <longExprList>
<longExprList> ::= <singleExpr> <longExprList> | <longExprBrac> <longExprList> | <singleExpr> | <longExprBrac>
<singleExpr> ::= "term" <lambdaVar>
<longExprBrac> ::= "(" <longExprCont> ")"
<lambdaVar> ::= "a-z"

Final BNF for short lambda

<shortLambdaP> ::= <lamList>
<lamList> ::= <shortLambda1> <lamList> | <shortLambda2> <lamList> | <shortLambda1> | <shortLambda2>
<shortLambda1> ::= "λ" <shortParam> "." <shortExpr>
<shortLambda2> ::= "(" "λ" <shortParam> "." <shortExpr> ")"
<shortParam> ::= <shortParamList>
<shortParamList> ::= <singleParam> | <singleParam> <shortParamList>
<shortExpr> ::= <shortExprList>
<shortExprList> ::= <singleExpr> <shortExprList> | <shortExprBrac> <shortExprList> | <shortLambda2> <shortExprList> | <singleExpr> | <shortExprBrac> | <shortLambda2>
<singleExpr> ::= "term" <lambdaVar>
<shortExprBrac> ::= "(" <shortExpr> ")"
<lambdaVar> ::= "a-z"

Parsing

Parser combinators

When constructing parsers for long lambda expressions and short lambda expressions, multiple parser combinators are implemented. A parser can be represented using BNF as a non-terminal, with only terminals on the right-hand side of the production rule. However, production rules usually have a mixture of terminal and non-terminal on the right-hand side, this is when parser-combinators are needed. As we know parser combinator is a higher order function which accepts parsers as input and combines them into a new parser. Hence the reason why it is especially useful when my BNF has non-terminal on the right-hand side of a production rule.

Choices made in creating parsers and parser combinators and How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses

Numerous functions from Parser.hs came in handy when implementing my parsers. Primarily the "is :: Char -> Parser Char'' function which specifies a certain character to be next on the input, together with do-blocks for multiline use of Monad instance's bind function (>>=) are used to build my BNF grammar. When creating a parser combinator, the (<-) notation in do-blocks allows me to call the other parsers. It takes the returned value resulting from the right-hand side expression (from the other parsers) out of the Monad context, in this case 'Parser', and assigns it to the variable on the left-hand side. It is important to take the returned value out of the Monad context so that I can apply functions such as "ap", "build", "term" or "lam" on the Builder. Hence, by using do-blocks, I am able to combine multiple parsers together, as well as checking multiple char using "is".

Some production rules in the BNF have alternatives on the right-hand side. The "(|||) :: Parser a -> Parser a -> Parser a" function acts like an 'or' function, it applies first Parser, and if the first parser fails alternates to the next Parser, this function helps implement the alternatives in my BNF.

Design of the code (Including Data Structures)

To parse a lambda, I use parser combinators that check if the input is a lambda with correct syntax. Then, the function builds and outputs the lambda as a Lambda data structure. For part 2, we parse strings representing boolean expressions and convert them into their respective church encodings.

For part 2, precedence of the operators are important in both the arithmetic and boolean expressions. By using chain function to handle repeated operators, it allows left recursive without the risk of running into infinite loop (blue word in BNF).

This is a BNF for the Arithmetic expressions in part 2

<arithmeticP> ::= <exprC>
<exprC> ::= <exprB> | <exprC> <add> <exprB>
<add> ::= "+" | "-"
<exprB> ::= <exprA> | <exprB> <times> <exprA>
<times> ::= "*"
<exprA> ::= <numP> | <bracP> | <numP> <expo> <numP> | <bracP> <expo> <bracP> |
<bracP> <expo> <numP> | <numP> <expo> <bracP>
<expo> ::= "**"
<bracP> ::= "(" <exprC> ")"

Hence, I am able to do left recursion (blue words) and handle the precedence for the operators at the same time.

Functional Programming

There are numerous codes that are used repeatedly, especially the lambda expressions for church encoding of an operator. Small functions such as or', and', if' which return the church encoding for the respective operators are created for repeated use.

The function to build a lambda expression ‘lam’ is of type (Char->Builder->Builder), which takes in a character and a builder as parameters and outputs a builder. When parsing a lambda expression, there are multiple choices after λ (lam). A few common possible outcomes are for instance, a single parameter followed by another lambda expression (λx.xx), or multiple parameters (λxx.xx), or a single parameter followed by another lambda expression (λx(λy.x)). Since there is always a single parameter after λ (lam), we partially applied the function with a Char to build a lambda parameter, returning a partially applied function as type Parser (Builder -> Builder). The partially applied function can then take any input with Builder type such as a term or another lambda expression to complete the lambda expression. The partially applied function is useful since we can reuse it for different types of lambda expressions.

Haskell Language Features Used

Typeclasses and Custom Types

The type Parser Lambda is used for parser combinators that parse a lambda. Whereas type Parser Builder is used for parsers which parse the components of a lambda, such as the function body. Parser (Builder -> Builder) and Parser (Builder -> Builder -> Builder) are builder function parsers, parsing functions used in a builder such as ‘lam’.

Functions

Bind functions are used in parser combinators as we need to retain the value of multiple parsers and combine them to build a lambda, the do-blocks keep the returned value in scope until end of do-block. Recursive functions and lists allow us to parse an arbitrary list of values. For lambda with unknown number of expression variables, we recursively parse a single variable and store the values into a list. We then fold the list to combine them into a single builder expression to build a lambda. The built-in function ‘foldl’ is used because we are applying the next variable to the first variable, from left to right, as per the syntax of a lambda. The chain function also serves as a recursive function that lets us parse multiple boolean expressions, numbers, and operators, while setting the precedence for each operator. For extension list operator function mapP, fmap is used because the parsed function needs to be mapped across a list of builder expressions. ‘Foldr’ is then used to apply the left expressions to the right expression, as per the syntax of church encoding of lists. The special case of bind ‘>>’ is used to sequence 2 parsers, it discards any returned value from the first parser and returns the value by the second parser. In my case, it is used when we need the condition in which the first parser must be successful for the second parser to run, and we do not need the return value of the first parser.

Extension

The **mapP** parser is a recursive list function parser which maps a function to every element in a list. In my case, the mapP function only allows the boolean operators and arithmetic operators to be mapped to a list of elements of either booleans or numbers. The output of this parser is the church encoding of the mapped list.

The **foldrP** parser parses a function, an accumulator, and a list, and folds the list from the right. Again, the function only allows boolean operators and arithmetic operators to be used as functions, while the list can only be a list of booleans or numbers. The output of this parser is the church encoding of the folded accumulator.