FIT 2102 Assignment 2 Report

By Huiying Lin (30119413)

BNF grammar:

Long lambada BNF (playground):

<longLam> ::= "(" <longobject> ")"
<longlam> ::= <longlam> <longterm> | <longterm>
<longterm> ::= "(" <longobject> ")" | <letter>
<longobject> ::= "λ" <letter> "." <longlam> | <longlam>
<letter> ::= [a-z]

Short lambada BNF(playground):

<shortLam> ::= "(" <shortlam> ")" | "λ" <letterList> "." <shortlam>
<shortlam> ::= <shortlam> <shortterm> | <shortterm>
<shortterm> ::= "(" <shortlam> ")" | <letter> | "λ" <letterList> "." <shortlam>
<letterList> ::= <letter> | <letter> <letterList>
<letter> ::= [a-z]

Note: "λ" is replaced with "l" in the playground

In part 1, the long lambda abstraction have to have the bracts, short lambda does not, the BNF are allow to have free variable as shown as the BNF can only produce CFL, so that our lambda calculus is containing three types of forms, lambda abstraction (\x), application(\x.x) and variable. For the long paser, I used the peek function (the peek limits the first element to check if the input started with the set char, otherwise will give errors), the long lambda has to start with an open parenthesis ‘(‘.

Parser Combinator Parsing for BNF:
Recursive descent parsing is made possible by parser combinators, which makes modular piecewise building and testing easier. The lambda calculus is divided into the long lambda and short lambda in Part 1, and each of them is further divided into terms, letters, and other units. The combinator is Higher order allowing us combining them to accept our lambda calculus by using interfaces like Applicative and Monad we learnt in previous week. For example, when parsing the lambda, i have used the fmap from applicative and the ignore symbols from monads.

Design of the code:

The design of the code is dividing the code into blocks. Some parts of the code is from the initial Builder and Parser classes, also I took some functions from the previous tutorial, especially Tutorial 11. I also implemented my own parser combinator like the peek function mentioned above and the boolTok function used the bool function which returns True or False and the tok function from the tutorial 11.

Moreover, there are some additional Parser combinators for each question used for different questions, Part 2 exercise 1 parsing the logical expressions, I built the lamTrue, lamFalse, lamIF, etc. They are the Parser Combinators using Builder and use the parser combinator built at start like fun (usd to apply a list of char using fold right). All these functions had been built to make it readable and usable in the logicP.

Avoid repeating and implementing the same functions in accordance with the "Don't Repeat Yourself" principle. The additional functions for lambdas make the code practical and clean. They can be applied in a variety of settings to simplify implementation. In order to make the code logical and simple to understand, I also divided the builders and parser into independent blocks.

Data Structure:
I used NonEmpty data structure found from the Hoogle, NonEmpty lists are similar to conventional list types in complexity and API, but they always contain at least one entry. It has been used in the little “choose” function at the start of the implementation, to choose from one from a list of non- empty parses. This little choice function is used in Part 3 ListOpP and the extensionP. Also, the PaserResult in the Parser class is the data structure used to handling the errors

Parsing

The whole program used many combinators, basically every question as the example for BNF. The third part 3 used many fmaps(Functor) and the ignore(>>) from Monad and the return(Applicative). For building parser through all these types to lists.

Functional Programming
Haskell is a functional programming language, the whole task is implemented in Haskell. The parameters in the code are not modified, just returning the results. For example, small functions return pure types like the sepby functions. And functions are composing together many “~” functions in use in the whole program because it is the app from Builder it is a typical FP style. Many Point free functions in used,

basically all the functions from tutorials and the little handy functions i made like peek, bool, “~”, “~~”, they all used point free.n

Haskell Language Features Used
The program used many lazy evaluations because it avoids repeating evaluations and functions are composed together, many “fmap”s used to reduce the repetition as well and it will noy modify parameters but only the last type. Part 2 and 3 are higher order functions calling many little handy functions. I used them because it helps to reduce the workload when I implement the functions and many functions can be reused by other questions. The project itself is about the lambda expressions which is a key feature of Haskell like map functions or lamadd functions.

Extensions

I have implemented the **Recursive list function** for map, filter and foldr. The other function is the **factorial function**.

The extension function here is breaking into 3 main blocks, the main function that used for the whole main function “extensionP" and some functions for recursive list functions (“lamY”,” lamFilter”, “lamFoldr”) and the function for factorial “lamFactorial”. The idea of the extensionP is similar to the function in Part 3 exercise 1 “listOpP”. The extension P using the fmap function mapping two functions “build” from Builder and the extensionB we defined below. “~~” is liftA2 app. And the functions below are a list of functions to choose. These functions mentioned are mostly Higher order functions.

Inside the extensionP function, the “function” contains many functions to add, multiply, sub, etc. Other than just extensions function because it is all HOF inside the extension, we need some additional non-HOF functions as input to test the extension is working or not.

The function ListOpP limits its last argument to a list, in order to implement the extensions I release the limitation for list only, so the parameters can be any combinations of the applications. There is one thing that needs to be noted is that the function application is the lowest priority, which means that “succ 1 + 1 = succ(1+1)”.

The factorial function “lamFactorial” used the Y combinator I generated, however, the factorial function is not dealing with the list, we are factorial numbers here, which means recursive int function.

Something cool

1. Right associate application
   In haskell, the left-associative functions are really common like f(x) = f x; f x y z = ((f x)y)z. But the extension here is using the right associative, which means it will be f(x(y(z))). To be more specific, I will use head and rest as an example.
   head rest lst = head (rest lst) Not (head rest) lst
   In the extension code itself it will be extensionB =element ~~ extension B, this is the right association.

2. Higher Order Functions
   Functions inside extension(however not just extension used HOF) take functions as inputs or return functions as its return value. For example, the function choose section is returning all functions as its return value.

3. Recursive int Function
   The factorial function “lamFactorial” used the Y combinator “lamY” but passed the numbers for computation (int).