FIT2102 Assignment 2

Chua Shin Herh (31902456)

**1 Design of the code**

The approach I used for my parsers is similar to the Calculator example provided by the teaching team. The method to parse arithmetic expressions in the Calculator example is applicable to almost all exercises in this assignment, so almost all of the exercises have the same general structure to it. The code structure will be written below. One thing to note, for the lambda expression parser lambdaP, I assigned shortLambdaP to it as shortLambdaP is able to parse both long and short lambda expressions, so there is no need to assign longLambdaP to it.

High level structure:

Most parsers to parse a string consists of the following few parser combinators:

* The parser itself, which is used to parse the string.
* A chain function, which is used to handle repeated chains of operators of unknown length.
* One or more expression functions which calls chain and chains multiple parsers together using a binary function and creates a Builder expression from the input string.
* One or more atomic functions that combines multiple individual parsers so the expression function can choose which function to use.
* One or more individual parsers that parses specific parts of the input string.

Code architecture choices:  
Do blocks are used widely in the code. For functions that require multiple lines in order, rather than using the (>>=) or (>>) functions, which is similar to the do block, the do block can make the code more readable.

Where clauses are used in the chain function to allow us to define local functions so that we do not need to create extra functions outside the function, such as the rest function in the chain function, which also makes the code cleaner.

**2 Parsing**

BNF Grammar:

<exprStart> ::= <atomStart> <exprStart> | <atomStart>

<exprLam> ::= <atomLam> <exprLam> | <atomLam>

<exprTerm> ::= <atomTerm> <exprTerm> | <atomTerm>

<atomStart> ::= "λ" <lam> | <paren>

<atomLam> ::= <lam> | <paren> | <dot>

<atomTerm> ::= "λ" <lam> | <term1> | <paren>

<lam> ::= <var> <exprLam>

<dot> ::= "." <exprTerm>

<paren> ::= "(" <exprTerm> ")"

<term1> ::= <var>

<var> ::= [a-z] | [A-Z] | "\_"

Most of the functions in the code are parser combinators, where they chain the behaviour of multiple parsers together, and returns a new parser. Most of the parser combinators are the Parser Builder type. I chose them to be Parser Builder instead of Parser Lambda because even though the functions to create lambda expressions such as “lam” and “ap” have variants that work for the Lambda type, but it would be easier to directly use the variants made for the Builder type, do all the processing needed, and use the “build” function in the end to convert it to a Parser Lambda type.

Some parsers and parser combinators are constructed using the Functor, Applicative, and Monad typeclasses.

For Functor, they are constructed using the fmap function, such as longLambdaP, where we fmap the “build” function over the expression function which is wrapped in a Parser context.

For Applicative, they are constructed by either using “pure” to wrap it in a Parser context, or by using the <\*> operator as in the rest function in the chainr1 function where fmap is performed first, which creates a function wrapped in a Parser context, then applying that function afterwards.

For Monad, they are constructed by using the >>= operator, such as the chainl1 function, where we apply the binary function to p.

**3 Functional Programming**

There are plenty of small and modular functions in the code. Each of these functions are designed to do a specific task, which makes the code cleaner, easier to read, and easier to debug.

Various small functions are also composed together to achieve functional programming style. This can be done by using the chain functions such as chainl1 and chainr1 that chain multiple functions together. Function composition is also used to achieve this. We want to compose small functions together because it allows us to write the code in less lines which also makes the code cleaner.

Declarative style programming is used extensively in the code. Functions just need to describe what the task is, rather than describing how to do the task like imperative code. Declarative style code is preferred over imperative style code because it makes the code easy to read and makes it less prone to side effects. Point free style can also be found in many functions in the code, such as composing multiple “ap” functions together, and using eta conversion. Point free style should be used as much as possible because it makes our code more concise, without needing unnecessary arguments, and allows for easy chaining of functions.

**4 Haskell Language Features**

Haskell language features for Functor, Applicative, and Monad, their functions fmap, apply, and bind are used extensively. As stated in Part 2 – Parsing, these functions are used to wrap values of types such as Builder or Lambda in a context like Parser.

Higher order functions can be found in functions like chainl1, where they take in a binary function to apply it to 2 values. Using higher order functions allows us to reuse the higher order function much easily as we can just change the function to allow for different operations.

As stated in Part 3 – Functional Programming, function composition is also used in the code. The compose operator (.) allows us to chain functions together, and allows us to get the right-most parameter on its own outside the body expression.

Some built in functions are also used to complete the tasks, such as isAlpha and isDigit which allows us to check whether the character of the input string is an alphabet or a digit.

**5 Extensions**

Extension 1 – Factorial

For my first extension feature, I decided to implement a parser for the factorial function. What I implemented was exactly what I intended to implement, which is to parse natural numbers with a “!” character after that number, which indicates factorial. For the chain function, instead of using the chainl1 function from before, which takes in a binary function, I created another chain function which only takes in a unary function, since factorial only requires one argument to it. The recursive definition of factorial is {fact(0) = 1; fact(n) = n \* fact(n-1)}, and this is appropriately implemented in the code by using if-then-else statements.

What is more complex about this extension is having to deal with recursion. As we know, recursion is not directly available when dealing with lambda calculus, since lambda functions are not able to refer to themselves, so there needs to be a way to allow recursion for factorial to work. Fortunately, there is a way to solve this, and that is by using the Y Combinator. The Y Combinator allows us to wrap the factorial function in a lambda expression and it can be passed in as a parameter, then we can refer to the factorial function by its name. We also need to set a base case for the recursion to end, and we can set it to be when n = 0, return 1.

Extension 2 – Negative Numbers

For my second extension feature, I decided to implement a parser for negative numbers. Initially, I thought there was a way to parse negative numbers using the method of parsing natural numbers, but after looking deeper into the problem, I realised there is no way to do this easily.

To parse negative numbers, my solution is to store it as a Church pair. This requires me to implement the function pair, first, and second, which are standard functions for a Church pair. First, we have to take the absolute value of the number, then convert the number to its signed form, in the form of a Church pair (a, b), where the number = a – b. Then the number is currently positive, to convert it to a negative number, we will negate it by swapping the positions of a and b, to obtain (b, a). Then the number is now negative since b – a is negative.

What is interesting about this extension is that not only can negative numbers be parsed into its signed form, natural numbers can also be parsed into its signed form with minimal modifications.