FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Summary

Throughout the assignment, I took extra care in not writing large do blocks or writing multiple do blocks in one parser. I did this by using bind (>>=) or special bind (>>) where possible to reduce the amount of do blocks which made the code look cleaner and more readable. Other than that, I made sure to reuse functions from previous exercises wherever it seemed possible, either by directly reusing it or modifying the existing function in such a way that it can be used in any parser.

Part 1

Most of the parsers created in Exercise 2 were reused in Exercise 3. I optimised the code as much as possible to minimise the number of parsers by utilising the (|||) function as much as possible. All the reusable parsers for Part 1 have been placed inside a section before the beginning of the code for Part 1.

Exercise 1

BNF

```
<both> ::= <long> | <short>
<long> ::= "(" <var> <term> ")" | <long> <term>
<short> ::= <var> <term>
<body> ::= [a-z] | [a-z] <body> | <spaces>
<spaces> ::= " "
<term> ::= <term> <term> | <body> | <both> | "(" <term> ")"
<dot> ::= "."
<var> ::= <lamb> <body> <dot>
<lamb> ::= "\\" | "/"
```

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Exercise 2

My approach here was to first parse the beginning of the lambda expression consisting of the lambda symbol, the single alphabet and the dot symbol, then parsing the rest of the expression while also taking into account the different patterns that can occur, and finally combine both parts at the end.

The first difficulty was getting the parser to continuously parse the lambda expression an unknown amount of times, this was solved using the function ‘list1’.

The second difficulty was parsing terms inside of brackets, this was solved using the function ‘between’.

Additionally, I created a parser which used the function (|||) to try two different parsers, ‘between’ and a parser that parses terms. This parser was then put into the function ‘list1’ to run continuously which allowed for parsing of any pattern of terms.

To concatenate all the terms, I used the function ‘foldl1’ which specifically doesn’t need a base case to apply the function ‘ap’ onto all the terms.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Exercise 3

For this parser, I reused most of the parsers from Exercise 2.

The main difficulty was applying the function ‘lam’ to all the characters before the dot symbol in the lambda expression.

I solved this by first, parsing all the characters of the lambda expression until the dot, then reusing the parsers from the previous exercise to parse the terms. This is important because the combined terms will be the base case used in the function ‘foldr’ which allows us to concatenate the characters we parsed previously with the combined terms while also applying the function ‘lam’ to the characters.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Part 2

There are two functions that I extensively used throughout the whole of Part 2 which are ‘chain’ and ‘opr’. These two functions have been placed inside a section before the beginning of the code for Part 2. As there are a lot of church encodings in this part, I took the liberty of containing the church encodings inside functions that return them to maintain the cleanliness and readability of the code.

Exercise 1

My approach to this exercise was to first translate all the church encodings of the logical literals and operators into code then chain the logical literals and operators.

The main difficulty in this exercise was chaining the logical literals and operators while also parsing expressions with correct order of operations. I solved this by using the ‘chain’ function mentioned previously. I successfully parsed expressions with correct order of operations by using the (|||) function to use parsers with greater precedence first.

Another difficulty was getting the church encoding of the logical literals. This was solved by using the provided ‘boolToLam’ function.

The other difficulty was getting the church encoding of operators to the front of the church encoding such as getting the church encoding of “and” to the front of “True” and “False” in the expression “True and False”. This was solved by creating an unapplied binary function that takes two inputs and applying the inputs to the end of the church encoding of the operator using the ‘ap’ function.

The final difficulty was trying to parse expressions that came in brackets. This was solved using the ‘between’ function which allowed us to parse whatever expression was in the brackets and return it still wrapped in the brackets.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Exercise 2

My approach to this exercise is similar to that of Exercise 1.

The main difficulty in this exercise that wasn’t encountered in Exercise 1 was getting the correct order of the operations. This was solved by creating multiple parsers of which each one parsed another one of the created parsers while also parsing a different operator.

Another difficulty was getting the church encoding of natural numbers. This was solved by using the provided ‘intToLam’ function.

Exercise 3

My approach to this exercise is similar to that of the previous exercises except that for this exercise, I had to create a lot of functions including unapplied binary functions that returned the church encodings of the respective comparison operators.

The main difficulty in this exercise that wasn’t encountered in the previous exercises was deriving the church encodings for the other comparison operators that weren’t provided. I solved this by reusing the church encodings for logical literals and operators from Exercise 1 to create the church encodings for the other comparison operators.

A decision I made in this exercise was to create two new parsers, one for “if then else” and another for “not”. The reason I did this was because the original parsers for them only work with logical operators “and” & “or”. Additionally, the reason I did not modify the original parsers instead was only because they were from different exercises.

However, one change I did make to a parser in another exercise (Exercise 2) was add the new parsers and the boolean parsers to the ‘expr’ parser. This makes it easier to add more expressions that can be parsed by the complex parser, thus making it extensible.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Part 3

In this part, as usual I stuck to my rules of keeping the code clean and readable where possible by using the same tricks I used in the previous parts. The functions I reused in this part that were implemented in previous parts are the functions ‘church’ and ‘opr’.

Exercise 1

My initial approach to this was the same as my approach to the exercises in Part 2. However, that would not be possible as it turns out I have to parse the values that come after a comma as a whole.

Hence, my new approach was to parse the beginning of the expression up until the first comma, then parse everything that comes after it altogether including the comma itself.

I solved this by creating a parser for the beginning part of the expression and another for the rest of the expression. I used the function ‘list1’ on the second parser to parse an unknown number of values in the list.

Another problem was parsing the right bracket that ends the list as the church encoding of the right bracket has to be concatenated with the second parser but the parser for the right bracket cannot be part of the second parser.

I solved this by separately calling the parser for the right bracket, then appending its result to the result of the second parser by containing it inside a square bracket and concatenating both using the ‘++’ infix operator.

Another difficulty was getting the format of the church encoding to be correct such that the list operations would work. This was solved by having the church encoding be right associative instead of left associative using the function ‘foldr1’ instead of ‘foldl1’.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Exercise 2

Fibonacci

The first parser that I have chosen to implement in this exercise is the Fibonacci sequence. My initial approach to this was to first translate the church encoding for the Y-Combinator into lambda calculus, then figure out the church encoding for the Fibonacci sequence.

The main difficulty in implementing the Fibonacci sequence was figuring out the church encoding for it. I solved this by first figuring out the base case for the recursive function which would be returning the value 1. As the Fibonacci numbers for the input numbers 1 and 2 are both 1, I set the base case as “if (n <= 2) then 1”. The next step was to figure out the church encoding for the actual recursive step. This was easier as it was just the equation for the Fibonacci sequence which is ($F_N = F_{N-1} + F_{N-2}$).

Interestingly, we already implemented all the values in the base case in the previous exercises, hence I was able to obtain the church encoding and lambda calculus for the base case by reusing functions implemented in previous exercises.

Finally, I implemented a separate parser for the Fibonacci number for the number 0 as it cannot be the base case due to the numbers 1 and 2 sharing the same Fibonacci number.

FIT2102 Assignment 2 Report

Name: Ooi Yu Zhang
Student ID: 32713339

Negative numbers

The second parser that I have chosen to implement in this exercise are negative numbers.

The main difficulty in this is understanding how to obtain negative numbers in the first place. Through independent research, I learnt the concept of church pairs and conversion of natural numbers into signed numbers using lambda calculus.

The way to obtain negative numbers is by first parsing a natural number and converting it into a signed number. Then, store the signed number inside the church pair and finally the negative number can be obtained by negation of the church pair.

The reason this works is because the church pair contains Church numerals representing a positive and a negative value. Hence, by negating the pair, we swap the positive value with the negative value and hence am able to obtain the negative value.