Name: Bryan Jun Kit Wong
Student ID: 32882424

FIT2102 ASSIGNMENT 1 REPORT

Part 1: Design of the code

A common theme prominent throughout the creation process of the parses was that all of them have their own set of rigid inputs and set rules to follow during parsing. Through this, I have concluded that Context-Free Grammer, particularly the Backus-Naur Form notation, can be used to guide the creation of the parsers. BNF splits each non-terminal to be on the left-hand side of their respective production rule. Thus, I modelled my parsers after my BNF, in which each non-terminal is a function with its own parsing rules. This also reinforced the concept of functional programming as I decomposed my parser into multiple parsers, leading to a more concise flow of code. The BNF is as follows:

BNF GRAMMAR

<lambdaP> ::= <longLambda>

| <shortLambda>

<longLambda> ::= <repeatedLong>

| <longLambdaBrackets>

<repeatedLong> ::= <longLambdaBrackets> <longLambda>

<longLambdaBrackets> ::= "(" <longLambdaExpr> ")"

<longLambdaExpr> ::= <lambdaSymbol> <variables> <dot> <repeatVariables>

| <lambdaSymbol> <variables> <dot> <bracket>

| <lambdaSymbol> <variables> <dot> <longLambda>

<shortLambda> ::= <shortLamdaExpr>

| <shortLambdaBrackets>

| <repeatedShortExpr>

| <repeatedShortBrac>

<repeatedShortExpr> ::= <shortLamdaExpr> <shortLambda>

Name: Bryan Jun Kit Wong
Student ID: 32882424

<repeatedShortBrac> ::= <shortLambdaBrackets> <shortLambda>

<shortLambdaBrackets> ::= "(" <shortLamdaExpr> ")"

<shortLamdaExpr> ::= <lambdaSymbol> <repeatVariables> <dot> <repeatVariables>

| <lambdaSymbol> <repeatVariables> <dot> <bracket>

<bracket> ::= <repeatedVarBracketVar>

| <repeatedBracketsVariable>

| <varBracketVar>

| <bracketsVariable>

| <repeatVariables>

<repeatedVarBracketVar> ::= <varBracketVar> <bracket>

<varBracketVar> ::= <repeatVariables> <bracketsVariable>

<repeatedBracketsVariable> ::= <bracketsVariable> <bracket>

<bracketsVariable> ::= "(" <repeatVariables> ")"

<lambdaSymbol> ::= "λ"

<variables> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"

| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"

| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<repeatVariables> ::= <variables>

| <variables> <repeatVariables>

<dot> ::= "."

One issue, however, was the inability to access the values wrapped in the Parser context, creating extremely messy code with pattern matching. To circumvent this debilitating problem, I utilised Monads, which allows me to apply the parser functions to the values within the Parser context while maintaining the context. By employing the syntactic sugar - do blocks, excessive clutter of binds (>>=) can be minimised, creating more compact parser functions, which are easier to read and debug. Anonymous functions are used too, which act as higher-order functions, compressing multiple functions into one and lessening code clutter.

To further improve the readability of my code, where clauses were used, which breaks down my complex, confusing functions into smaller and simpler functions. In addition, guards, which are syntactic sugar for if statements, also serve to reduce excessive code clutter caused by nested ternaries.

As the input string for any parser could theoretically be of infinite length, the List ADT plays a crucial role in storing Builders until the parser reaches the end of the input string. Then, Foldable types are employed to reduce the list of Builders to a single Builder. Through this, any input of arbitrary length can be parsed.

Part 2: Parsing

For the alternatives (|) in my BNF, they are represented by the parser combinator (|||). The first parser is executed at first If failure to do so occurs, attempt to run the second parser. Else if both fail, raise an error as the input string is invalid. Because of this, I could "combine" multiple parsers together, lessening the frequency of duplicated code. Furthermore, it breaks a convoluted parser into simpler parsers, improving the code's readability.

Another combinator used is the 'chain' combinator. The chain function takes in two parsers, p and op. It will chain 0 or more p, all separated by op. One example is the arithmetic parser, which will use the chain combinator to chain numbers separated by a mathematical operator (+, -, *, **). Due to this, I can chain multiple arithmetic operations of unlimited length together with ease without employing recursion.

In addition, I simulated a parser tree using the chain function. This is shown within my logicExpr parser. The logicExpr parser will first call orExpr, entering the first layer of the tree. In orExpr, I chain multiple instances of the andExpr parser, separated by the 'or' operator

parser. The andExpr is the second layer, calling the baseExpr while chaining it separated by 'and'. At baseExpr, it's at the end of the tree, calling the parsers with the highest precedence. Through this, a parser tree is created, with or having the lowest precedence and brackets having the highest precedence.

For flexibility and robustness, every parser, besides logicP and listOpP, can ignore whitespaces in their input string, increasing the range of acceptable inputs.

As mentioned above, I utilised Monad and Foldable typeclasses to assist in parsing. Another typeclass that came in use was the Functor typeclass. Fmap (<$>) is used, for example, to apply the 'term' function to each character in a string of variables. As the characters are wrapped in a context (a list), fmap permits me to partially apply the term function to each variable without extracting the character one by one.

Part 3: Functional Programming

As iterated multiple times earlier, I decomposed enormous, complicated problems into more straightforward ones. This is a core feature of Functional Programming, as it allows easier debugging and enhances the readability of code. All functions are pure with no side effects. As a result, bugs can be minimised or isolated as one function will not affect other parts of the code. The compose (.) allows me to chain functions and operators one after another while limiting the usage of messy brackets. Another use of the declarative style use is recursion. Recursion helps to emulate the imperative-style loops with all the benefits and none of the drawbacks. Imperative loops have side effects, potentially causing a domino effect of bugs and causing the whole program to fail. Thus, recursion is superior in this way, as it allows loops with no side effects. Point-free style is used, which utilises Eta-conversion, operator sectioning and composition to reduce a function to remove their arguments. One example of this is my numberBuilder function which takes in a number and converts it to its church encoding form. The input 'n' has been reduced and removed, thus making the code more concise.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Part 4: Haskell Language Features Used

Although no Applicative typeclass was used, my usage of Monads does the same as what the "apply" (<*>) function does. As some functions are within a Parser context, I used the Monad bind to pull it out and apply it to another item within a Parser context. Similar to the Applicative, the context is rewrapped when I return the Monad using pure.

Additionally, higher-order functions served a useful purpose in my code. They allow me to chain multiple function calls together while conserving space. Furthermore, my parse tree in the allExpr employs the idea of higher-order functions to simulate precedence. Each function in the allExpr will call another until it reaches the function with the highest precedence. Then the last function to be called is evaluated first, then the previous one. Without high-order functions, this would have been impossible to implement. Some built-in functions I used were map, which I used to apply functions to a list of Builders. Some functions provided in the Parser file were also used to great effect, such as spaces, list/list1 and satisfy, which were used instead of creating new functions that accomplish the same effect, leading to additional redundancy. Some features were also imported from Data.Char such as isDigit and isLower.

Part 5: Description of Extensions

Some supplementary features aded were nested if statements for my logicP. Furthermore, to demonstrate the flexibility of my listP, it can parse lists with arithmetic operations, logical expressions, comparison expressions and nested lists within them. Combined with listOpP, it can lead to some interesting outputs.

For example, head cons not True [3 < 5, not (False or False] will be able to evaluate the logical expressions inside and the output will be False.

Potential features I intended to implement were for logicP to be able to parse comparison expressions and evaluate them to booleans. For example, if 3 < 4 then 5 == 5 else 4 >= 4. This demonstrates the reusability of my code if implemented as the comparisons will also evaluate to booleans. For part 3 exercise 2, the function of my chose was Factorial, Fibonacci and Foldr.

(1194 words excluding header and BNF)