Design of the code (including data-structures)

High level description of approach

Firstly, I distinguished certain keywords that appear throughout the test cases (e.g. the λ and . character in lambda parsers; + - * operators in arithmetic parsers; if, and, or in logic parsers) and built their respective church encoding, which is of type Builder (I haven't built it into a Lambda, since I have to add more Builders to the it). Then I try to parse the keywords in the input string and return the corresponding Builder to construct the final Lambda. The values (i.e. characters for lambda parsers, integers for arithmetic parsers, True and False for logic parsers) are also parsed using their own parsers (the char parser returns the character passed into a lam or term function; the integer is passed into the intToLam function; and the Booleans are passed into the boolToLam function). After these values are parsed, I combined them all together using the ap function to connect all the pieces of Builder types together, then built them into a Lambda.

High level structure of code

I broke down the lambda expressions into two different parts, namely the input and the output. I parse the inputs (which will be constructed into lambda using the lam function) and the outputs (which will be constructed into lambda using the term function) separately into a list, to easily combines them together using fold functions. Similarly, I broke down the arithmetic strings and logic expression strings into two different parts, namely the first value and the continuous operations. I parsed the first value using the normal parser, and parsed the remaining operations part by part and stored them into a list. Then by using the first value as an initial value, I used fold functions to collapse the list into a single value, which is the lambda expression of the whole input.

Code architecture choices

The code architecture I used is to parse strings into various Builders (Parser Builder), then combined the parsers into a more general parser (parser combinator of Parser Builders), then used the build function after combining all the results, thus converting Builder types into Lambda types. Ultimately, I will get a parser of type Parser Lambda. I defined a datatype Expr to aid with parsing arithmetic operations. It consists of two parts: the operator (op) and the second value (value). Since an arithmetic operation is in the form <value1> <operator> <value2>, but the lambda expression is constructed of <operator> <value1> <value2>, I need some way to reconstruct the solution, which I thought of using record syntax to obtain the data stored in Expr for the function in the fold functions.

Parsing

BNF Grammar

The BNF Grammar of Exercise 1 is defined as follows:

```
<longLambdaP> ::= <input> <output> | <input> <longLambdaP> <close>
<input> ::= <open> <lambdaChar> <char> <dotChar>
<output> ::= <nextChar> <argl> <close> | <nextChar> <close>
<argl> ::= <open> <nextChar> <close>
<nextChar> ::= <char> <nextChar> | <char>
<open> ::= "("
<close> ::= ")"
<lambdaChar> ::= "λ" | "/"
<dotChar> ::= "."
<char> ::= "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"

<shortLambdaP> ::= <open> <inp> <outp> <close> | <open> <inp> <outp> <close>
<shortLambdaP> | <inp> <outp>
<inp> ::= <lambdaChar> <nextChar>
<outp> ::= <dotChar> <nextChar> | <dotChar> <args>
<args> ::= <nextChar> <open> <nextChar> <close> | <nextChar> <shortLambdaP>
<nextChar> ::= <char> <nextChar> | <char>
<open> ::= "("
<close> ::= ")"
<lambdaChar> ::= "/"
<dotChar> ::= "."
<char> ::= "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"

<lambdaP> ::= <longLambdaP> | <shortLambdaP>
```

Usage of parser combinators

By breaking down the input string into different parts, I made parsers for each part and then finally combined them together to form a parser combinator. E.g. longLambdaP consists of two parts, namely the input (arguments) and output (result). By splitting the parsing into these two parts, I can build the Lambda of the result fairly easily instead of trying to parse the whole string by itself.

Choices made in creating parsers and parser combinators

I tried to create short parsers that are solely responsible for parsing a keyword (e.g. andP parser to parser the “and” keyword in an input string), and returns a result of type Builder. I did the same to create short parsers for different values (characters, integers and Booleans) as well. Eventually, I would have pieces of Builders that can be combined into one complete Builder, which I can apply the build function to obtain the Lambda.

How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses

I constructed most of the parser combinators using the do notation, which is syntactic sugar for various bind functions in the Monad typeclass. In some short parsers, I used the left apply (<*) and right apply (*>) to negate unwanted results (such as the '.' character in lambda parsers). In some cases where I had to combine parsers of type Builder and type Lambda in future questions, I made use of the Functor fmap (<$>) to aid with typechecking (mapping build over a parse result of type Builder).

Functional Programming

Small modular functions

Short and concise functions such as open, close, lambdaChar, dotChar and char are created to help parse individual characters that are guaranteed to appear in the string. Some small helper functions such as input and argl for the longLambdaP parser are also created to help break down the string into smaller parts before recombining the results.

Composing small functions together

By using do notation, I combined small functions together to form a general parser (e.g. addP and minusP parse the "+" or "-" sign in a string and returns the respective lambdas, whereas number parses an integer and returns the lambda representation of it, I combined the two to form a basicOperator parser which parses simple arithmetic expressions.

Declarative style

No temporary variables are declared, nor is there any functions that may cause side effects are used. All of the parsers are chained together using do notation, bind, fmap and apply functions, which follows a declarative style of programming.

Haskell Language Features Used

Typeclasses and Custom Types

I created a datatype Expr that consists of two attributes, op and value. The op attribute stores the lambda expression for the operator (in Builder type) and the value attribute stores lambda expression for the value (in Builder type). This is to facilitate the reduction of the list of operations after parsing the input via record syntax.

fmap, apply, bind

Since parsers are a subclass of Functors, Applicatives and Monads, I used fmap (<$>), apply (<*) and bind (>>) functions to combine the effects and results of various parsers together to achieve the intended result of a certain parser.

Higher order functions

Some functions are created (e.g. expr and bracketed) to reduce redundant code implementation. They are higher order functions that take in a parser as an argument and gives back a more robust parser, such as bracketed gives back a parser that can take in both bracketed and unbracketed inputs.

Function composition

I composed various functions together to form a general parser. For example eqP can only parse equalities for integers, whereas eqBoolP can only parse equalities for booleans. By combining the two into complexCalcP (along with a few other parsers for various inequalities and logic operators), I get a general parser that can parse arithmetic operations, inequalities and Boolean logic.

Leveraging built in functions

I used built-in functions foldl, foldr and flip to aid with my construction of the solutions. The list parser stores the parsed results as a list, so I made use of the fold functions to collapse the results into a single value (i.e. a Lambda).

Description of Extensions

The factorial parser

I implemented a parser that can take in a string of a factorial and returns the lambda expression for its result.

E.g. lamToInt <$> parser factP “4!”

Result >< Just 24

Originally I wanted to implement a recursive version of the parser by using the Y combinator for the church encoding, but I can’t quite get the recursive part working, so I turned to the U combinator, which is essentially applying the lambda to itself.

The factorial function is : if n == 0 then return 1 else return n * factorial (n-1)

Converting it into lambda would be

fact := λn. isZero (intToLam 1) (multiply n (**fact** (pred n)))
Making use of the U combinator,
U := λf. f f
fact := U (λf. λn. isZero (intToLam 1) (mult n (**f f** (pred n))))

The map parser

I tried to implement a parser than can take in a string of formula “map <f> <l>” where f is a function of a basic arithmetic expression (+ or -) and l is a list of integers.

The expression for map I came up with is of the form:

map f l = if l isNull then null else map f (cons (f (head l)) (rest l))

To prevent getting stuck in an infinite recursion for repeated calls to the map function itself, I tried using the U combinator to process the function, but I could not get the anticipated result.

The quicksort parser

I tried to implement a parser that can take in a astring of formula “quicksort <l>” where l is a list of integers.

After researching online, I came across the lambda church encoding for the sorting function and implemented the various church encoding into Builders. I then constructed them together to form the quicksort function.

However, it does not seem to work properly as trying to obtain the integer representation (using lamToInt) of the head of the list after sorting takes a long time (or even stuck in an infinite loop).