CS141 Coursework 2

Lambda Calculator

For this coursework, I decided to implement a program that parses and evaluates untyped lambda expressions. The program uses alpha and beta substitution to reduce the lambda expression to its simplest form.

All interaction with the user is performed in the command line, including inputting the expression and displaying the result of the lambda calculus
The program can be used by running `stack run`:

Enter an untyped λ expression:
Please separate lambda variables with () e.g. `(\x.(\y.x))x` instead of `(\x.\y.x)x`
```
>>= (((\x.(\y.x))(\a.a))(\b.b))
```
Result:
```
(((\x.(\y.x))(\a.a))(\b.b))
((\y.(\a.a))(\b.b))
(\a.a)
```
Above is an example of how the program would run if a valid lambda expression was given. The program first prints out the given expression in a formatted form, often adding more brackets to make the order of operations clear to the computer and the user. It then continues to print each step of the lambda calculus for the user to follow.

A limitation of the program is that the input string must separate the lambda variables, or the following error will appear:
```
cswk-program-exe: "<stdin>" (line 1, column 6):
unexpected '\\'
expecting end of "."
```

This is due to the method that the program parses lambda variables, and if they are not separated the second lambda variable would not be parsed.

The program is also able to perform alpha substitution when needed, this is when the lambda variable has the same name as the free variable. For example, `(((\x.(\y.(y x)))y) (\z.z)).` Here the free variable y is the same as `\y.(y x)` and this can be confusing when y is beta-substituted into `\x.(\y.y x)` as it will result in `\y.y y`. It is ambiguous which is the original y that needs to be substituted and which was the free variable. Therefore, alpha substitution is needed to distinguish between the two.

Enter an untyped λ expression:
Please separate lambda variables with () e.g. `(\x.(\y.x))x` instead of `(\x.\y.x)x`
```
>>=(((\x.(\y.(y x)))y) (\z.z))
```
Result:
```
(((\x.(\y.(y x))) y) (\z.z))
((\!y.(!y y)) (\z.z))
((\z.z) y)
y
```
When the alpha substitution is performed, the lambda variable y is renamed to !y making the difference clear in the expression.
The program is also able to perform lambda expressions with numbers as free variables, and include numbers and operators in the expressions.

Enter an untyped λ expression:
Please separate lambda variables with () e.g. `(\x.(\y.x))x` instead of `(\x.\y.x)x`
```
>>= (\y.(\x.x-y))5 y
```
Result:
```
(((\y.(\x.(x-y))) 5) y)
```

```
((\!x.(!x-5)) y)
(y-5)
```

Architectural Choices

For the program, I split the computation into 2 aspects: the parser which reads and formats the inputs and the calculator which evaluates and performs the calculus on the expression. The separation of functionality makes finding specific methods quicker and the program easier to follow.
The two files are called from the main when needed, this keeps the main program uncluttered and the code more modular.

In my `parser.hs`, I used records[4] to make a token parser that defines lexical parsers using the GenTokenParser[1] fields. I decided to use a record to reduce repeated code as I could pattern match using record syntax[4]. The records also grouped the lexers in one unit, making it clearer and more cohesive in the program compared to individually defining each lexer[1].

Libraries

The primary library I used was parsec, which has a primarily monadic interface. The library included functionalities to write parses that turns the λ expression inputted into a more structured representation,[5] that's the program could understand how to evaluate.

The main functions I used included `parse`[3], from Text.Parsec, which evaluated the input for any errors (ParseError) and displayed this to the user before any parsing was performed.
The function `makeTokenParser` [1] from Text.Parsec.Token was particularly important in creating the GenTokenParser record which defined the lexical parses. Lexical parsers determined what can be parsed and which cannot. For example, the `identifier` [1] lexer only allows for legal identifiers which are not in the reserved words. The lexers, including `parens`[1] and `reservedOp`[1], helped distinguish variables from expression and therefore, defined the Var, Lambda and Num datatypes.
The makeExprParser[7] and Operator(InfixL)[7] functions were important in defining the mathematical operators for them to be used in the lambda calculus.

Another library I used was Data.List to have access to set operations, such as union and nub, which were important in calculating the free variables in the expression.

Personal Experience

To begin with, I found the parsec library hard to navigate and understand due to a lot of new terminologies such as lexers and tokens. As well as this, there are many parsec libraries like megaparsec and attoparsec and I was unsure on which library was the best for my application. I found there were many ways to do the same functionality as there are so many libraries and trying to stick to one library, for cohesion, was a challenge.

However, the documentation for parsec was excellent and had many guided examples of using the functions. Megaparsec also had a very good tutorial page[2], which was useful for adding in mathematical operators as it was the same method for parsec. Overall, after getting my understanding of the library and reading documentation[7], implementing it was smoother than expected.

I also have discovered a great appreciation and liking towards lambda function. At the beginning of the coursework, I spent time understanding their purpose, history, and the method in calculating them. This was very useful in ensuring the program was outputting the correct answer by comparing my answer to the program's.
Despite not implementing many structures, I have a great understanding of data types and defining new types to perform separate functionalities when needed.

Online I found lambda expressions[6] with their respective answers to test my calculator. This gave me a rough idea if my program was working correctly but I also created my own more specific tests to ensure edge cases were covered.

References

[1] Text.parsec.token. https://hackage.haskell.org/package/parsec-3.1.15.0/docs/Text-Parsec-Token.html#v:makeTokenParser. (Accessed on 03/24/2022).

[2] Megaparsec tutorial. https://markkarpov.com/tutorial/megaparsec.html#parsing-expressions. (Accessed on 03/24/2022).

[3] Text.megaparsec. https://hackage.haskell.org/package/megaparsec-5.2.0/docs/Text-Megaparsec.html. (Accessed on 03/24/2022).

[4] haskell - avoid repetition in lexing when using parsec - stack overflow. https://stackoverflow.com/questions/38572507/avoid-repetition-in-lexing-when-using-parsec. (Accessed on 03/24/2022).

[5] Writing a lambda calculus evaluator in haskell – bor0's blog. https://bor0.wordpress.com/2019/03/19/writing-a-lambda-calculus-evaluator-in-haskell/. (Accessed on 03/24/2022).

[6] code golf - write an interpreter for the untyped lambda calculus - code golf stack exchange. https://codegolf.stackexchange.com/questions/284/write-an-interpreter-for-the-untyped-lambda-calculus. (Accessed on 03/24/2022).

[7] Write you a haskell. http://dev.stephendiehl.com/fun/WYAH.pdf. (Accessed on 03/24/2022).

[8] Control.monad.combinators.expr. https://hackage.haskell.org/package/parser-combinators-1.3.0/docs/Control-Monad-Combinators-Expr.html#v:makeExprParser. (Accessed on 03/24/2022).