CSCI.4430 Programming Languages Fall 2021
Programming Assignment #1
Due Date: 7:00 PM September 27 (Monday)

*This assignment is to be done either **individually** or **in pairs**. **Do not show your code to any other group** and **do not look at any other group's code**. Do not put your code in a public directory or otherwise make it public. However, you may get help from the mentors, TAs, or the instructor. You are encouraged to use the LMS Discussions page to post problems so that other students can also answer/see the answers.*

Lambda Calculus Interpreter

The goal of this assignment is to write a lambda calculus interpreter in a functional programming language to reduce lambda calculus expressions in a call-by-value (applicative order) manner.

You are to use the following grammar for the lambda calculus:

```
<expression> ::= <atom>
               |   "\" <atom> "." <expression>
               |   "(" <expression> " " <expression> ")"
```

Your interpreter is expected to take each lambda calculus expression and repeatedly perform beta reduction until no longer possible (a value expression that can no longer be beta-reduced) and then eta reduction until no longer possible.

In the above grammar, `<atom>` is defined as a lower case letter followed by a sequence of zero or more alphanumeric characters, excluding Oz language keywords. A full listing of Oz language keywords can be found on P. 839 Table C.8 of "Concepts, Techniques, and Models of Computer Programming". Your interpreter is to take lambda calculus expressions from a text file (one expression per line) and reduce them sequentially. To enable you to focus on the lambda calculus semantics, a parser is provided both in Haskell and Oz.

Hints: You may define auxiliary procedures for alpha-renaming, beta-reduction, and eta-conversion. For beta reduction, you may want to write an auxiliary procedure that substitutes all occurrences of a variable in an expression for another expression. Be sure that the replacing expression does not include free variables that would become captured in the substitution. Remember that in call-by-value, the argument to a function is evaluated **before** the function is called.

Sample Interpretations

Below are some lambda calculus interpretation test cases:

Expression	Result	Comment
`(\x.\y.(y x) (y w))`	`\z.(z (y w))`	Avoid capturing the free variable `y` in `(y w)`
`(\x.\y.(x y) (y w))`	`(y w)`	Avoid capturing the free variable `y` in `(y w)`, and perform eta reduction
`(\x.x y)`	`y`	Identity combinator
`\x.(y x)`	`y`	Eta reduction
`((\y.\x.(y x) \x.(x x)) y)`	`(y y)`	Application combinator
`(((\b.\t.\e.((b t) e) \x.\y.x) x) y)`	`x`	If-then-else combinator
`\x.((\x.(y x) \x.(z x)) x)`	`(y z)`	Eta reductions
`(\y.(\x.\y.(x y) y) (y w))`	`(y w)`	Alpha renaming, beta reduction and eta reduction all involved

For your convenience, these have been given in a sample input file, where each line contains one lambda calculus expression.

Notes for Oz Programmers

You should remove any other Oz installations and use Mozart 2. You can download the binaries from the project's GitHub repository Note for Windows users: the interactive Oz editor may fail to launch if you install Mozart on a path containing spaces. It may also fail to launch due to problems with your environment (causing Mozart to be unable to find its bundled copy of Emacs). If you are having problems, please post about it on LMS Discussions or come to office hours for help.

A parser is provided. Use this starter code and this provided main file. `main.oz` receives command line arguments and invokes `PA1Helper`'s `RunProgram` procedure, which handles reading the input and writing output (by default, these files will be `input.lambda` and `output.lambda` unless otherwise specified). You will need to implement the `ReduceExp` method in `main.oz`. The program will write reduced expressions to the output file **and** print out both the original and reduced expressions to `stdout` as demonstrated in the sample interaction below. Only the lines written to the file will be considered when grading.

Sample Interaction

```
$ cat input.lambda
(\x.x y)
(\x.\y.(x y) (y w))

$ ozc -c parser.oz

$ ozc -c PA1Helper.oz

$ ozc -c main.oz

$ ozengine main.ozf input.lambda output.lambda

Original: (\x.x y)
Reduced: y
Original: (\x.\y.(x y) (y w))
Reduced: (y w)

$ cat output.lambda
y
(y w)
```

Notes for Haskell Programmers

Use this parser to get a list of lambda calculus expressions from an input file. See also the sample main.hs file (in particular, see the `runProgram` function.) Specifically, type constructors for the `Lexp` datatype have been exported from the module. This datatype is used to represent lambda calculus expressions in Haskell, and the type constructors should be used to pattern match a lambda calculus expression. Your goal is to create a `reducer` function that takes a `Lexp` value as input and returns a `Lexp` value as output.

Note: Please name your main file `main.hs`. It should take a filename as an argument (defaulting to `input.lambda`); the file itself will consist of multiple lines, as described earlier, each with a single lambda calculus expression on it. The provided code will parse this file, write to the output file, and print out both the original and reduced expressions to `stdout`. You must output the **reduced** lambda calculus expression to an output file (defaulting to `output.lambda`), as described in the sample interaction below. Like the Oz implementation, only the lines in the file will be considered when grading.

Sample Interaction

```
$ cat sample.lambda
(\x.x y)
(\x.\y.(x y) (y w))

$ runghc main.hs input.lambda output.lambda

Input  1: (\x.x y)
Result 1: y

Input  2: (\x.\y.(x y) (y w))
Result 2: (y w)

$ cat output.lambda
y
(y w)
```

If you get an error about Text.Parsec being undefined, run `stack install parsec` (or `cabal install parsec`).

Further Haskell Hints: It may be useful to consider Map and Set, which can be found in the Data.Map and Data.Set modules, respectively. It is also recommended to use Hoogle, a search engine for looking up Haskell documentation.

Notes for Other Programming Languages

You may use another functional programming language, or a multi-paradigm programming language such as Python to complete this assignment, but you should only use that language's functional programming aspects. Be certain you have a strong understanding of the language and do not go beyond the rules of the assignment (e.g. using imperative or object oriented behavior in Python.) **If you intend to use another programming language, please get explicit approval from Professor Varela first.**

Due Date: 7:00 PM September 27 (Monday)

Grading: The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor. If something is unclear, explain it with comments!

Correctness is judged by testing if your lambda calculus expressions are alpha-equivalent to the expected expressions. Therefore, your choice of names when performing alpha renaming will not affect correctness, as long as your chosen names are valid (i.e. lowercase alphanumeric strings that start with a letter and are not Oz keywords).

Submission Requirements: Please submit a ZIP file with your code, including a `README` file. Your ZIP file should be named with your LMS user name(s) as the filename underscore the language you chose.

Examples: `userid1_oz.zip, userid1_userid2_hs.zip, userid1_lisp.zip`.

Only submit one assignment per pair via LMS. In the `README` file, place the names of each group member (up to two). It should also have a list of specific features/known bugs in your solution.