CSCI.4430/6430 Programming Languages Fall 2016
Programming Assignment #1

*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 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 in Oz and in Haskell.

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 expression. Lines are separated with exactly one '\n' character, and there should be no '\n' following the last line for the parser to work properly.

Notes for Oz Programmers

Use this parser to get a list of lambda calculus expressions from an input file. See also the sample usage of the parser. The answers should be printed in the Browser window with the `Browse` function, one line per expression. Lambda calculus expressions are parsed as atoms for variables, as two-element `lambda` tuples for functional abstractions, and as two-element lists for function applications. Your goal is to create a `Run` function to interpret lambda calculus expressions.

Further Oz Hints: Make sure oz is started in the folder containing parser.ozf and input.lambda (in Windows, the easiest way is by putting the parser, input and test files in one folder and double clicking test.oz), or changing the absolute paths or relative paths of `'parser.ozf'` and `'input.lambda'` in `test.oz`.

See the source code for the parser if you are interested, and feel free to report problems and provide comments.

Notes for Haskell Programmers

Use this parser to get a list of lambda calculus expressions from an input file. See also the sample usage of the parser (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 expression. Your goal is to create a `reducer` function that takes an `Lexp` value as input and returns a `Lexp` value as output.
Note: Please name your main files 'main.hs'. It should take as input a single line consisting of a filename. The file itself will contain multiple lines. Make sure that the output consists of each step on a separate line with a blank line between each expression.

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.

Due Date: Thursday, 09/22, 7:00PM

Grading: The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor (comment, comment, comment!).

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). Your README file should also have a list of specific features/bugs in your solution.