CSCI 4430/6430 Programming Languages Fall 2024 Programming Assignment #1

Due Date: 7:00 PM September 26th (Thursday)

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 Submitty Discussion Forum 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. It must then perform eta reduction until no longer possible.

In the above grammar, `<atom>` is defined as a lowercase alphanumeric string that starts with a letter. 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 available, along with supporting code to handle input and output.

Your program must accept two command line arguments. The first argument is the name of a file containing a list of lambda calculus expressions. The second argument is the name of the file you will write your reduced lambda calculus expressions into.

Your parser should write out the reduced expressions to a file, again one per line. If something goes wrong and you cannot reduce an expression, you should write a single new line and then continue with the next lambda calculus expression. Writing more than one line, or writing nothing at all, will interfere with autograding. Do not write anything else, such as debug printouts, test numbers, etc.

You may (and should!) 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.

You may choose whatever names you wish when alpha-renaming, as long as they do not violate the definition of `<atom>`. Your code will be judged correct if we could make your solution match our solution by renaming zero or more of your variable names.

Sample Interpretations

Below are some lambda calculus interpretation test cases:

Expression	Result	Comment
`(\x.x y)`	`y`	Identity combinator
`\x.(y x)`	`y`	Eta reduction
`(\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
`((\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 (input.lambda), where each line contains one lambda expression.

Notes for Haskell Programmers

We provide a main file (main.hs) and a helper file (PA1Helper.hs). The helper file gets a list of lambda calculus expressions from an input file and parses them into the `Lexp` datatype (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 in the 'main.hs' file that takes an `Lexp` value as input and returns a `Lexp` value as output.

The provided helper code will both write to stdout and to the output file. The former is intended to help you during the development process. **Only the lines written to the file will matter when the assignment is graded.**

Note: Please keep your main file name as 'main.hs'. It should accept two arguments when compiled and executed. The provided main.hs has this functionality provided. **Please edit only the main.hs file as it is the only Haskell file you will submit.**

Sample Interaction

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

$ runghc main.hs sample.lambda output.lambda

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

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.

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

Grading: The assignment will be graded mostly on correctness, but code clarity and readability will also be a factor. If something is unclear, explain it with comments!. **You are expected to comment every function. It is good practice to provide type signatures as well.** For example, `"reducer :: Lexp -> Lexp"` for the reducer function.

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).

You are expected to write a lambda calculus interpreter. Any hard-coding attempts (e.g. if-else the given inputs) will result in a severe point reduction.

Submission Requirements: Please submit only the main file (main.hs) and a README file. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific known features/bugs in your solution.

Team Creation in Submitty: You are required to create a team in Submitty whether you are completing the assignment individually or in pairs. **Team creation will be locked on September 19th (Thursday) at 7pm which is a week before the assignment is due.**