University of Technology, Jamaica

Faculty of Engineering and Computing

School of Computing and Information Technology

Analysis of Programming Languages (CIT4004)

Group Project: Design A Programming Language Interpreter/Compiler

Jesse Nelson (1803868)

Tajae Anglin (1803630)

Andre Grant (1908921)

Prince Walker (1802393)

Tutorial Day and Time: Monday 5-8 pm

Name of Tutor: David W. White

July 5, 2024

Introduction

Name of Interpreter: PLambda, (Python Lambda). It is modeled after the Python programming language developed for Lambda calculus.

This project provides an interpreter for a programming language based on Lambda Calculus. The interpreter supports applicative-order reduction and performs lexical, syntactic, and semantic analysis on input source code. Additionally, it includes a graphical user interface (GUI) for user - friendly interaction and can provide detailed explanations of each reduction step using the ChatGPT APi. The language we developed belongs to the imperative paradigm and is a domain - specific language. It is designed to be versatile and capable of executing a variety of lambda calculus expressions, which includes performing the three reduction types: Alpha, Beta, and Eta. Furthermore, the language that we have developed is categorized as a high-level programming language due to its characteristics such as abstraction from machine code, readable and understandable syntax, portability, Higher-Level abstractions, and ease of use.

Grammar

Our grammar is represented in both Ebnf (Extend Backus - Naur Form) and BNF (Backus - Naur Form).

Ebnf

```
program = expr ;

expr = var | num | func, arg | ‘#’, var, ‘.’, expr | func | arg;

func = var | num | (‘#’, var, ‘.’, expr) | func, arg ;

arg  = var | num | (‘#’, var, ‘.’, expr) | (func, arg) ;

var  = ‘a’ | ‘b’ | ... | ‘z’

num = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

Bnf

```
<program> ::= <expr>

<expr> ::= <var> | <num> | <func> <arg> | "#" <var> "." <expr> | <func> | <arg>

<func> ::= <var> | <num> | "(" "#" <var> "." <expr> ")" | <func> <arg>

<arg> ::= <var> | <num> | "(" "#" <var> "." <expr> ")" | "(" <func> <arg> ")"

<var> ::= "a" | "b" | ... | "z"

<num> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Explanation

1. <program> ::= <expr>

This line defines the starting point of the grammar. Any valid string according to this grammar must match the <expr> rule.

2. <expr> ::= <var> | num | <func> <arg> | "#" <var> "." <expr> | <func> | <arg>

This rule defines what an <expr> (expression) can be:

- <var>: A variable which can be any single lowercase letter from "a" to "z".
- <num>: Any single number from “0” to “9”
- <func> <arg>: A function followed by an argument.
- "#" <var> "." <expr>: A lambda abstraction, representing a function with a variable parameter and an expression body.

3. <func> ::= <var> | num | "#" <var> "." <expr> | <func> <arg>

This rule defines what a <func> (function) can be:

- <var>: A variable which can be any single lowercase letter from "a" to "z".
- <num>: Any single number from “0” to “9”
- "#" <var> "." <expr>: A lambda abstraction, representing a function with a variable parameter and an expression body
- <func> <arg>: A function followed by an argument, allowing for function application.

4. <arg> ::= <var> | "#" <var> "." <expr> | <func> <arg>

This rule defines what an <arg> (argument) can be:

- <var>: A variable which can be any single lowercase letter from "a" to "z".

- <num>: Any single number from “0” to “9”
- "#" <var> "." <expr>: A lambda abstraction, similar to the one in <expr>.
- <func> <arg>: A function followed by an argument, similar to <func>.

5. <var> ::= "a" | "b" | ... | "z"

This rule defines what a <var> (variable) can be:

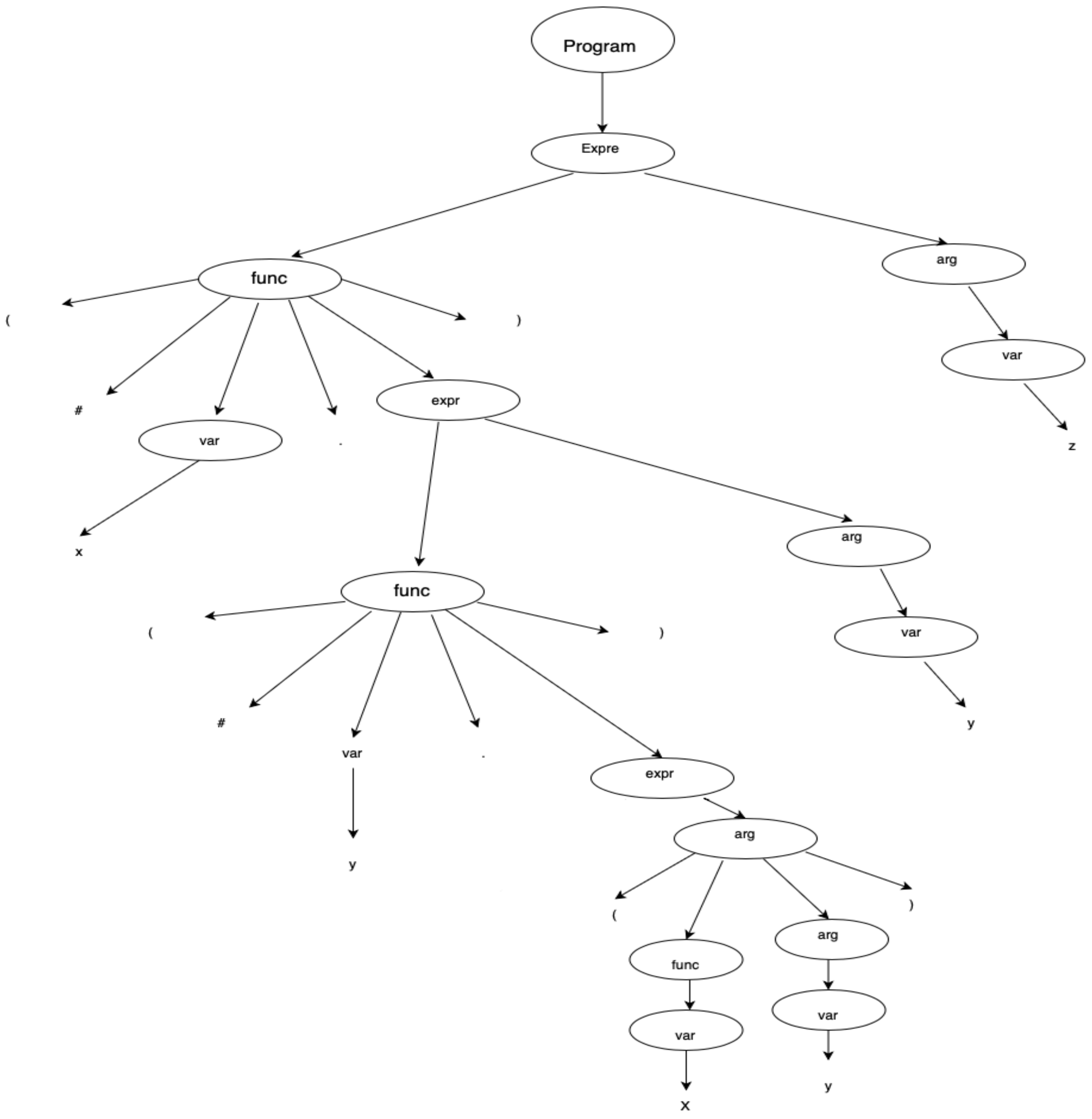
- Any single lowercase letter from "a" to "z".

6. <num> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

This rule defines what a <num> (number can be:

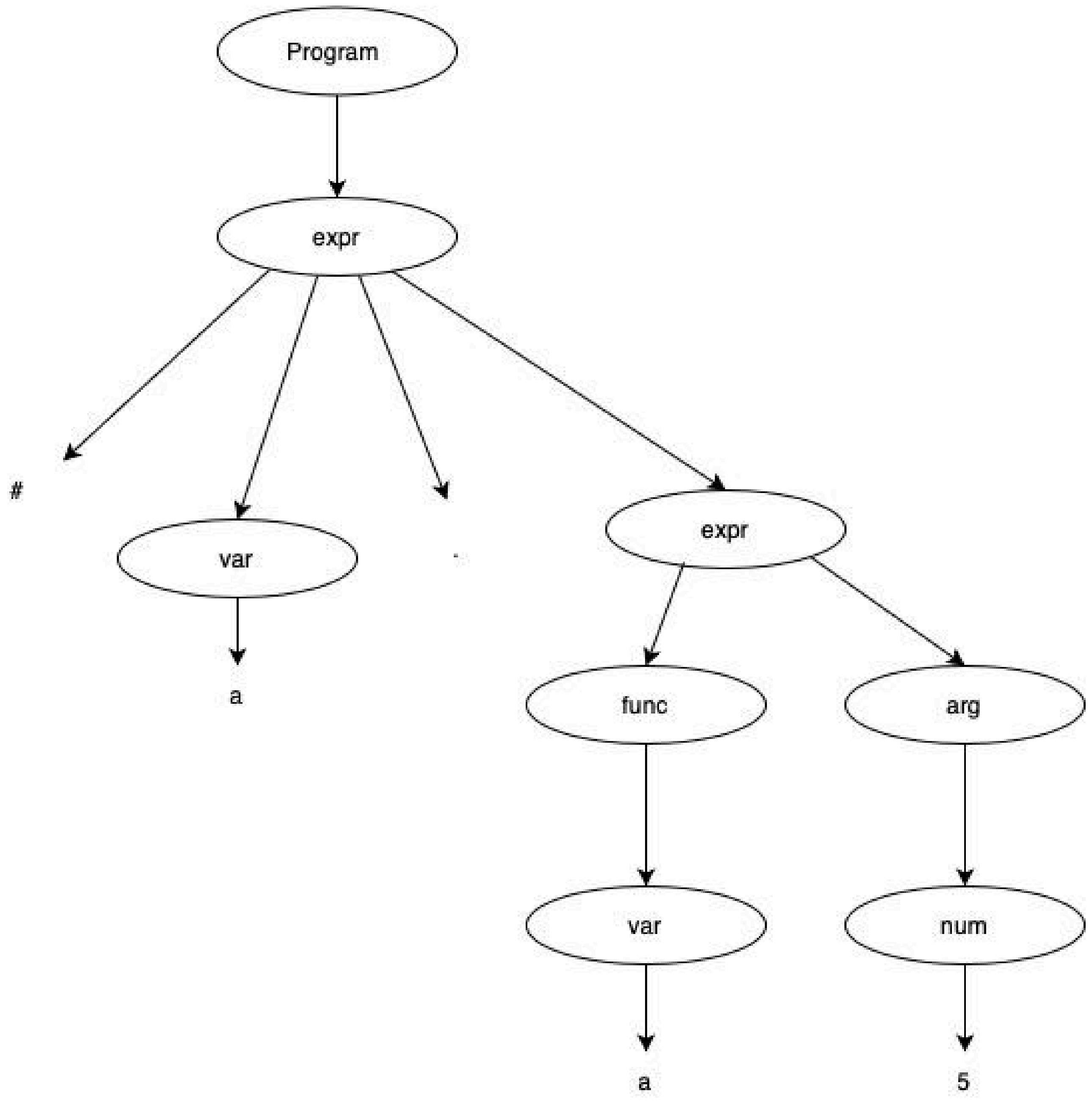
- Any single number from “0” to “9”

Complete parse tree / AST for a sample program in your language.
The sample program / expression that we chose is:
(#x.(#y.(xy))y)z



The second sample program / expression that we chose is:

#a.a5



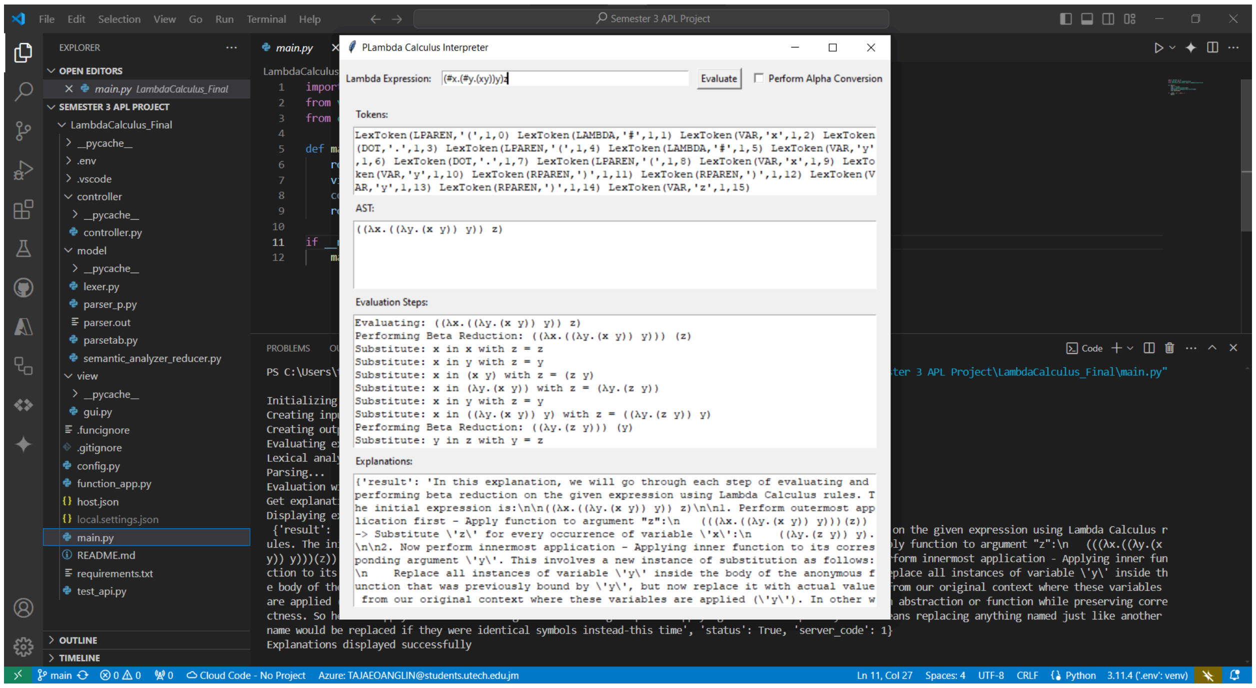
Tokens

Token	**Symbol/ Initial**
t_LAMBDA	LAMBDA
t_DOT	DOT
t_LPAREN	LPAREN
t_RPAREN	RPAREN
t_VAR	VAR

Regular Expressions

Token	**Regular Expression**
t_LAMBDA	r’\#’
t_DOT	r’\.’
t_LPAREN	r’\(‘
t_RPAREN	r’\)’
t_VAR	r’[a-zA-Z]’

Demonstration of scope and binding in sample code



Lambda Expression: (#x.(#y.(xy))y)z

Tokens:

```
LexToken(LPAREN,'(',1,0) LexToken(LAMBDA,'#',1,1) LexToken(VAR,'x',1,2) LexToken
(DOT,'.',1,3) LexToken(LPAREN,'(',1,4) LexToken(LAMBDA,'#',1,5) LexToken(VAR,'y'
,1,6) LexToken(DOT,'.',1,7) LexToken(LPAREN,'(',1,8) LexToken(VAR,'x',1,9) LexTo
ken(VAR,'y',1,10) LexToken(RPAREN,')',1,11) LexToken(RPAREN,')',1,12) LexToken(V
AR,'y',1,13) LexToken(RPAREN,')',1,14) LexToken(VAR,'z',1,15)
```

AST:

```
((λx.((λy.(x y)) y)) z)
```

Evaluation Steps:

```
Evaluating: ((λx.((λy.(x y)) y)) z)
Performing Beta Reduction: ((λx.((λy.(x y)) y))) (z)
Substitute: x in x with z = z
Substitute: x in y with z = y
Substitute: x in (x y) with z = (z y)
Substitute: x in (λy.(x y)) with z = (λy.(z y))
Substitute: x in y with z = y
Substitute: x in ((λy.(x y)) y) with z = ((λy.(z y)) y)
Performing Beta Reduction: ((λy.(z y))) (y)
Substitute: y in z with y = z
```

Explanations:

```
{'result': 'In this explanation, we will go through each step of evaluating and
performing beta reduction on the given expression using Lambda Calculus rules. T
he initial expression is:\n\n((λx.((λy.(x y)) y)) z)\n\n1. Perform outermost app
lication first - Apply function to argument "z":\n   (((λx.((λy.(x y)) y)))(z))
-> Substitute \'z\' for every occurrence of variable \'x\':\n    ((λy.(z y)) y).
\n\n2. Now perform innermost application - Applying inner function to its corres
ponding argument \'y\'. This involves a new instance of substitution as follows:
\n    Replace all instances of variable \'y\' inside the body of the anonymous f
unction that was previously bound by \'y\', but now replace it with actual value
 from our original context where these variables are applied (\'y\'). In other w
```

```
name would be replaced if they were identical symbols instead-this time', 'status': True, 'server_code': 1}
Explanations displayed successfully
```

In Lambda Calculus, scope and binding refers to the rules that govern the visibility and lifetime of variables. A variable is said to be bound if it is declared within a lambda abstraction; otherwise, it is free. In the above snippet, one prime example of scope is between the bounded variables and their inputs. In the lambda expression (#x.(#y.(xy))y)z, we have two inputs which are #x and #y and there bounded variables of (xy). The y variable highlighted in red is said to be a free variable as it is out of scope of the bounded variable. Our code is able to distinguish between this and evaluate our expression to normal form by first executing an alpha reduction and then executing a beta reduction to give us the final answer of zy.

Details on the programming language used to develop PLambda

To create PLambda, Python was used because of its simplicity and readability which makes it easier to understand and maintain code, which is crucial for a complex project like a compiler. We also utilized Python because it has an in-built module Ply which was integral in completing our project. PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. The main goal of PLY is to stay fairly faithful to the way in which traditional lex/yacc tools work. This includes supporting LALR(1) parsing as well as providing extensive input validation, error reporting, and diagnostics. PLY consists of two separate modules; lex.py and yacc.py, both of which are found in a Python package called ply. The lex.py module is used to break input text into a collection of tokens specified by a collection of regular expression rules. yacc.py is used to recognize language syntax that has been specified in the form of a context free grammar.

Characteristics of a good programming language evident in PLambda

Simplicity

This refers to the ease of understanding, learning, and usage of a language. With PLambda, simplicity is one of its strongest and most prominent features, where the language was designed to execute the reductions of a lambda expression until it is in normal form. In contrast with other programming languages, PLambda does not utilize common keywords such as IF, FOR and WHILE. However, the syntax of the language is minimalistic, using only essential elements such as variables, function abstractions, and function applications. The simplicity of our languages makes our language easy to read and understand, as there are fewer syntactic elements to process. Additionally, the simplicity of our project affects its writability as writing code in this language is straightforward due to the limited number of constructs. Finally, because our project is simple our project can be said to be reliable. The reduced complexity of our project, by utilizing consistent patterns which will decrease the likelihood of syntax errors, contributes to more reliable code execution.

Syntax Design

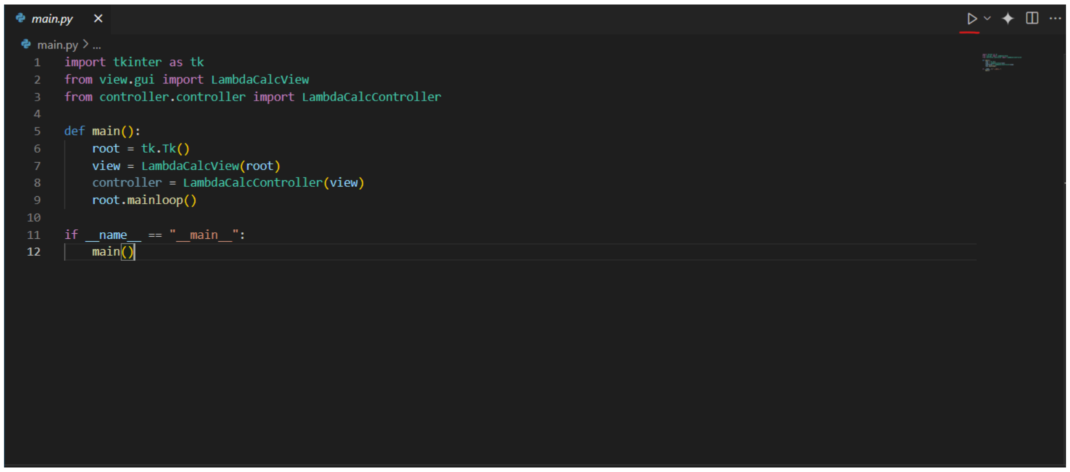
A computer language's syntax is a crucial component that affects readability, writability, and maintainability. For the interpreter to correctly read the instructions, the syntax makes sure that the code is written consistently and in a comprehensible manner which is provided by PLambda.. Code may be read and understood more easily because of our project’s syntactic design, which also increases the language's dependability. This facilitates both the process of writing error-free, proper code and the upkeep of already-written code. The language employs a minimal syntax, primarily using parentheses for grouping and a small set of symbols for operations, this symbol includes the # (hash-tag) which represents our lambda (λ) symbol, it is used for lambda abstractions, function applications, and variable representation. The advantages of our syntax design ensures that there is less visual clutter, making it easier to identify the structure and flow of the code. Additionally, programmers can write code quickly and with fewer mistakes due to the simplicity of the syntax. Furthermore, fewer syntactic rules mean less chances for syntax errors, leading to a more reliable code.

Expressivity

Expressivity as a characteristic of programming languages means the extent to which computation, data, and concepts can be represented and communicated. Despite its simplicity, PLambda allows for complex function definitions and applications, supporting recursive and high - order functions. The expressive power allows complex ideas to be represented concisely, making it easier to understand the programmer’s intent. Additionally, the ability to write complex computations with minimal syntax makes the language efficient to use.

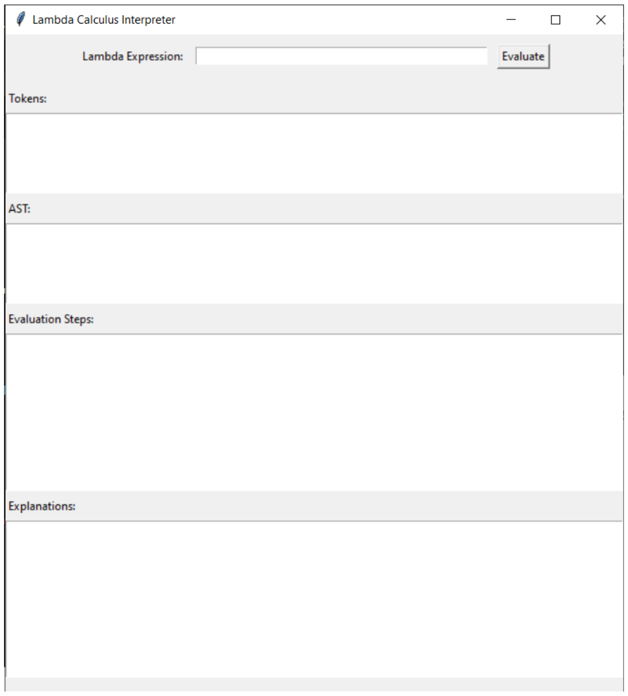
How to Run

The program is executed from clicking the run code button (High - lighted in red) in our main.py file



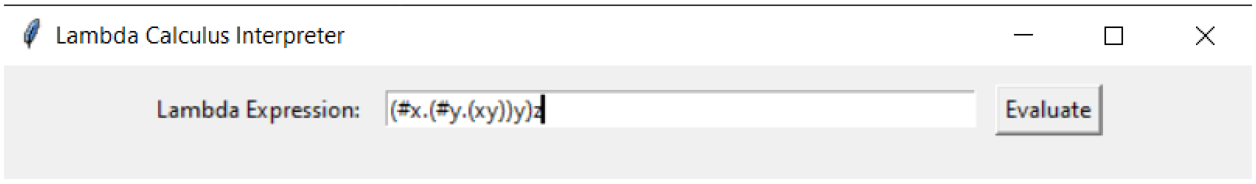
```python
import tkinter as tk
from view.gui import LambdaCalcView
from controller.controller import LambdaCalcController

def main():
    root = tk.Tk()
    view = LambdaCalcView(root)
    controller = LambdaCalcController(view)
    root.mainloop()

if __name__ == "__main__":
    main()
```

After the code is executed our graphical user interface (GUI) will be displayed.



The user now has the option to enter the lambda expression they want to evaluate in the Lambda Expression input box. After they have entered the expression they have to click the Evaluate button for the expression to execute.

Aft



(#x.(#y.(xy))y)z

er the user has clicked the evaluate button, our interpreter will process the expression and display the results in the output sections (View Tokens, View AST, View Evaluation Steps and View Explanation).

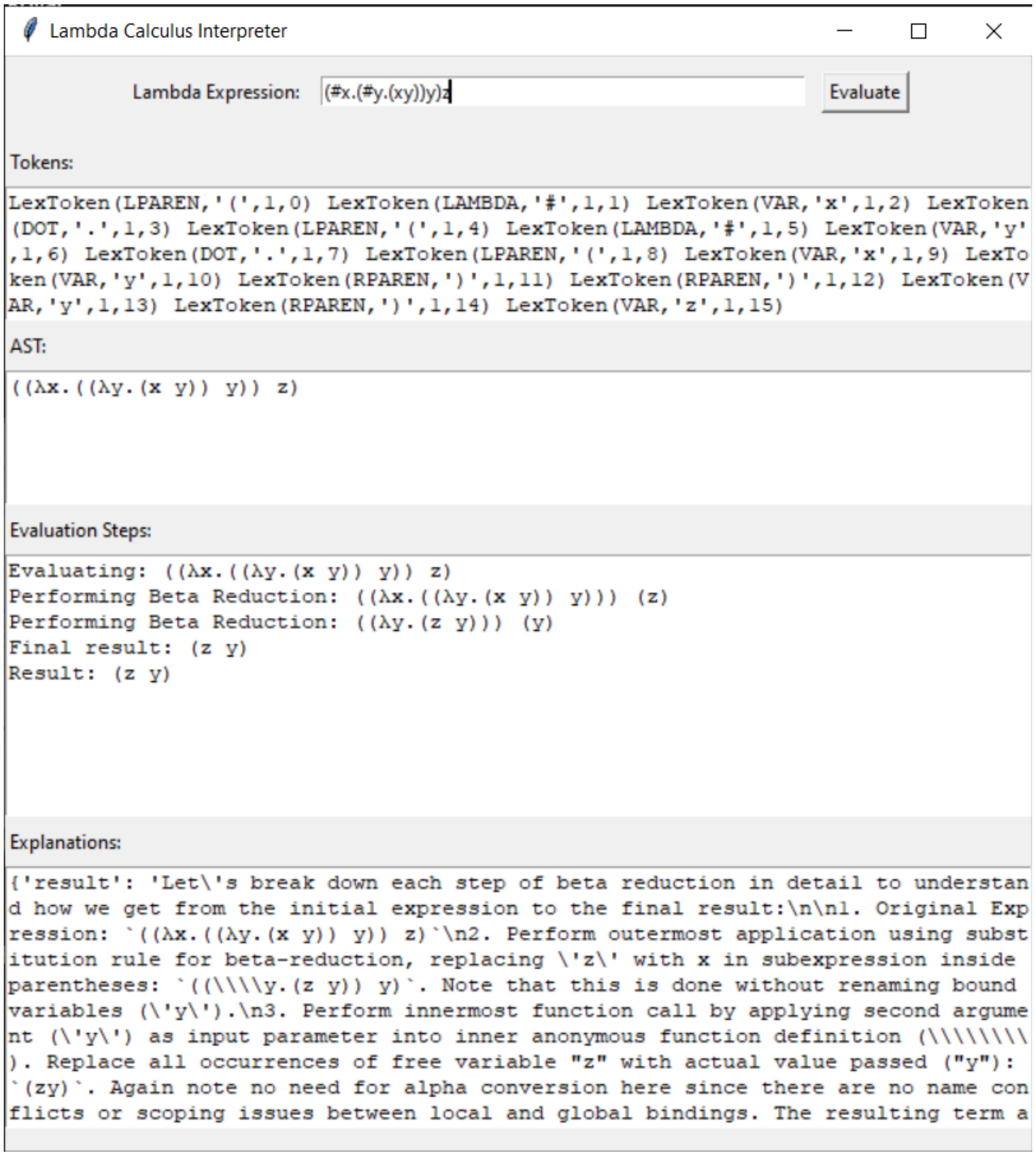
View Tokens - As the name suggests the lexical analyzer takes the source code which in this case would be the expression that we entered and outputs the tokens of the expression based on our semantics.

View AST - The AST section displays the parsed Abstract Syntax Tree

View Evaluation Steps - In this section, our interpreter displays and lists each step of the evaluation process, detailing the reductions being performed (Alpha, Beta, and Eta.)

View Explanation - Finally, in this section the user is provided additional information about each of the reductions being performed on the expression they entered, using the ChatGPT Api to explain the process.

Below is a sample of our code



```
LexToken(LPAREN,'(',1,0) LexToken(LAMBDA,'#',1,1) LexToken(VAR,'x',1,2) LexToken
(DOT,'.',1,3) LexToken(LPAREN,'(',1,4) LexToken(LAMBDA,'#',1,5) LexToken(VAR,'y'
,1,6) LexToken(DOT,'.',1,7) LexToken(LPAREN,'(',1,8) LexToken(VAR,'x',1,9) LexTo
ken(VAR,'y',1,10) LexToken(RPAREN,')',1,11) LexToken(RPAREN,')',1,12) LexToken(V
AR,'y',1,13) LexToken(RPAREN,')',1,14) LexToken(VAR,'z',1,15)
```

```
((λx.((λy.(x y)) y)) z)
```

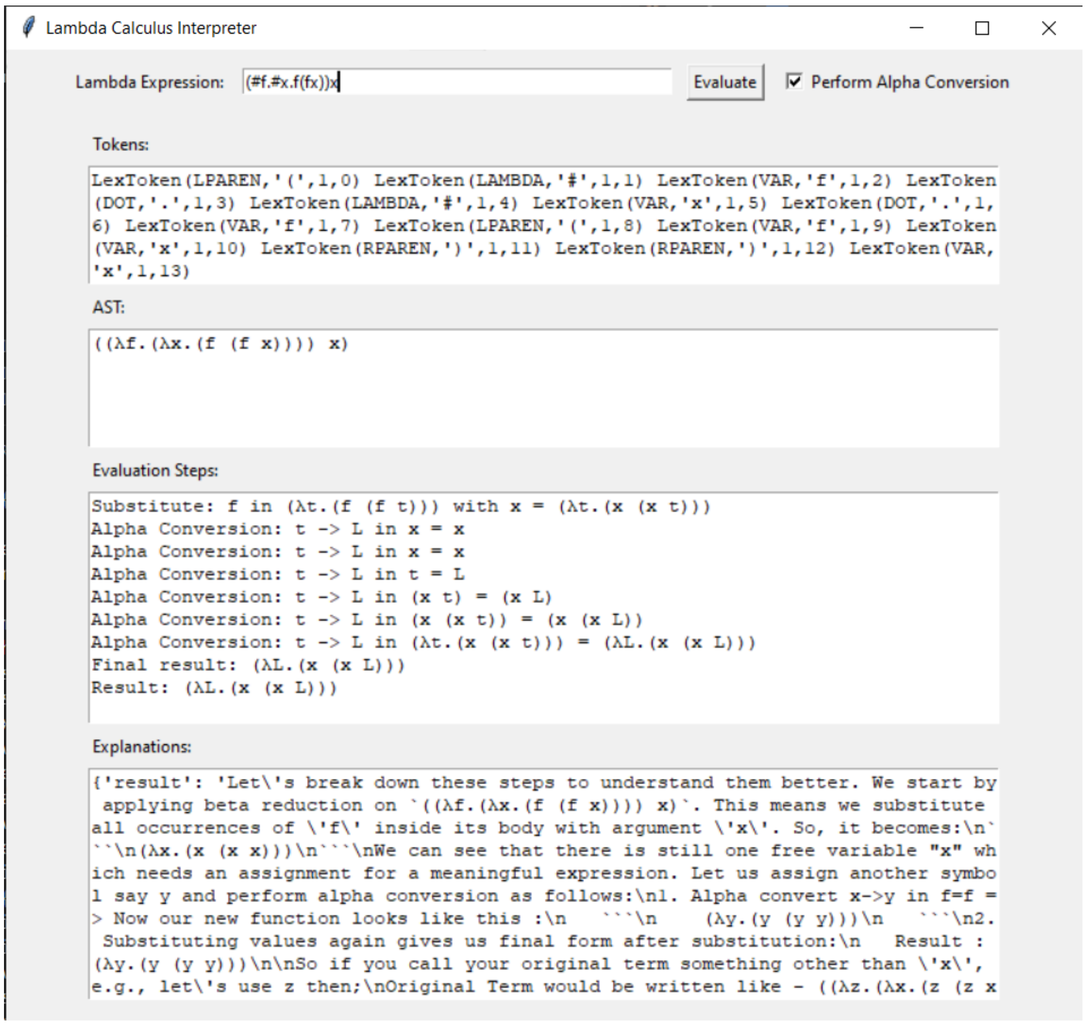
```
Evaluating: ((λx.((λy.(x y)) y)) z)
Performing Beta Reduction: ((λx.((λy.(x y)) y))) (z)
Performing Beta Reduction: ((λy.(z y))) (y)
Final result: (z y)
Result: (z y)
```

```
{'result': 'Let\'s break down each step of beta reduction in detail to understan
d how we get from the initial expression to the final result:\n\n1. Original Exp
ression: `((λx.((λy.(x y)) y)) z)`\n2. Perform outermost application using subst
itution rule for beta-reduction, replacing \'z\' with x in subexpression inside
parentheses: `((\\\\y.(z y)) y)`. Note that this is done without renaming bound
variables (\'y\').\n3. Perform innermost function call by applying second argume
nt (\'y\') as input parameter into inner anonymous function definition (\\\\\\\\
). Replace all occurrences of free variable "z" with actual value passed ("y"):
`(zy)`. Again note no need for alpha conversion here since there are no name con
flicts or scoping issues between local and global bindings. The resulting term a
```

Our sample code perfectly outputs all the functional requirements of our project especially, the explanation from ChatGPT. The explanation from ChatGPT for the lambda expression we evaluated reads as follows: "{'result': 'Let\'s break down each step of beta reduction in detail to understand how we get from the initial expression to the final result:\n\n1. Original Expression: `((λx.((λy.(x y)) y)) z)`\n2. Perform the outermost application using the substitution rule for beta-reduction, replacing \'z\' with x in subexpression inside parentheses: `((\\\\y.(z y)) y)`. Note that this is done without renaming bound variables (\'y\').\n3. Perform the innermost function call by applying a second argument (\'y\') as input parameter into the inner anonymous function definition (\\\\\\\\). Replace all occurrences of free variable "z" with actual value passed ("y"): `(zy)`. Again note no need for alpha conversion here since there are no name conflicts or scoping issues between local and global bindings. The resulting term after these two reductions becomes our simplified end product which cannot be further reduced - a normal form within Lambda Calculus semantics! So now let us reiterate what was achieved through those successive transformations above:\n\n. Initial Term => Simplification Step #0 [Outer Applying]=>', 'status': True, 'server_code': 1}”

Executing the three types of Lambda Reductions in our code

Alpha Reduction: (#f.#x.f(fx))x



~~~~
LexToken(LPAREN,'(',1,0) LexToken(LAMBDA,'#',1,1) LexToken(VAR,'f',1,2) LexToken
(DOT,'.',1,3) LexToken(LAMBDA,'#',1,4) LexToken(VAR,'x',1,5) LexToken(DOT,'.',1,
6) LexToken(VAR,'f',1,7) LexToken(LPAREN,'(',1,8) LexToken(VAR,'f',1,9) LexToken
(VAR,'x',1,10) LexToken(RPAREN,')',1,11) LexToken(RPAREN,')',1,12) LexToken(VAR,
'x',1,13)
~~~~

~~~~
((λf.(λx.(f (f x)))) x)
~~~~

~~~~
Substitute: f in (λt.(f (f t))) with x = (λt.(x (x t)))
Alpha Conversion: t -> L in x = x
Alpha Conversion: t -> L in x = x
Alpha Conversion: t -> L in t = L
Alpha Conversion: t -> L in (x t) = (x L)
Alpha Conversion: t -> L in (x (x t)) = (x (x L))
Alpha Conversion: t -> L in (λt.(x (x t))) = (λL.(x (x L)))
Final result: (λL.(x (x L)))
Result: (λL.(x (x L)))
~~~~

~~~~
{'result': 'Let\'s break down these steps to understand them better. We start by
 applying beta reduction on `((λf.(λx.(f (f x)))) x)`. This means we substitute
all occurrences of \'f\' inside its body with argument \'x\'. So, it becomes:\n`
``\n(λx.(x (x x)))\n```\nWe can see that there is still one free variable "x" wh
ich needs an assignment for a meaningful expression. Let us assign another symbo
l say y and perform alpha conversion as follows:\n1. Alpha convert x->y in f=f =
> Now our new function looks like this :\n   ```\n    (λy.(y (y y)))\n   ```\n2.
 Substituting values again gives us final form after substitution:\n   Result :
(λy.(y (y y)))\n\nSo if you call your original term something other than \'x\',
e.g., let\'s use z then;\nOriginal Term would be written like - ((λz.(λx.(z (z x
~~~~

For our lambda expression (#f.#x.f(fx))x

- We have to consider the body of the lambda abstraction which is highlighted in red to indicate which variables are free.
- The two occurrences of f occur free since there is no #f within the body to bind it. (The only occurrence of #f in the expression is outside of the body).
- The occurrence of x does not occur free since there is a #x within the body which binds it.

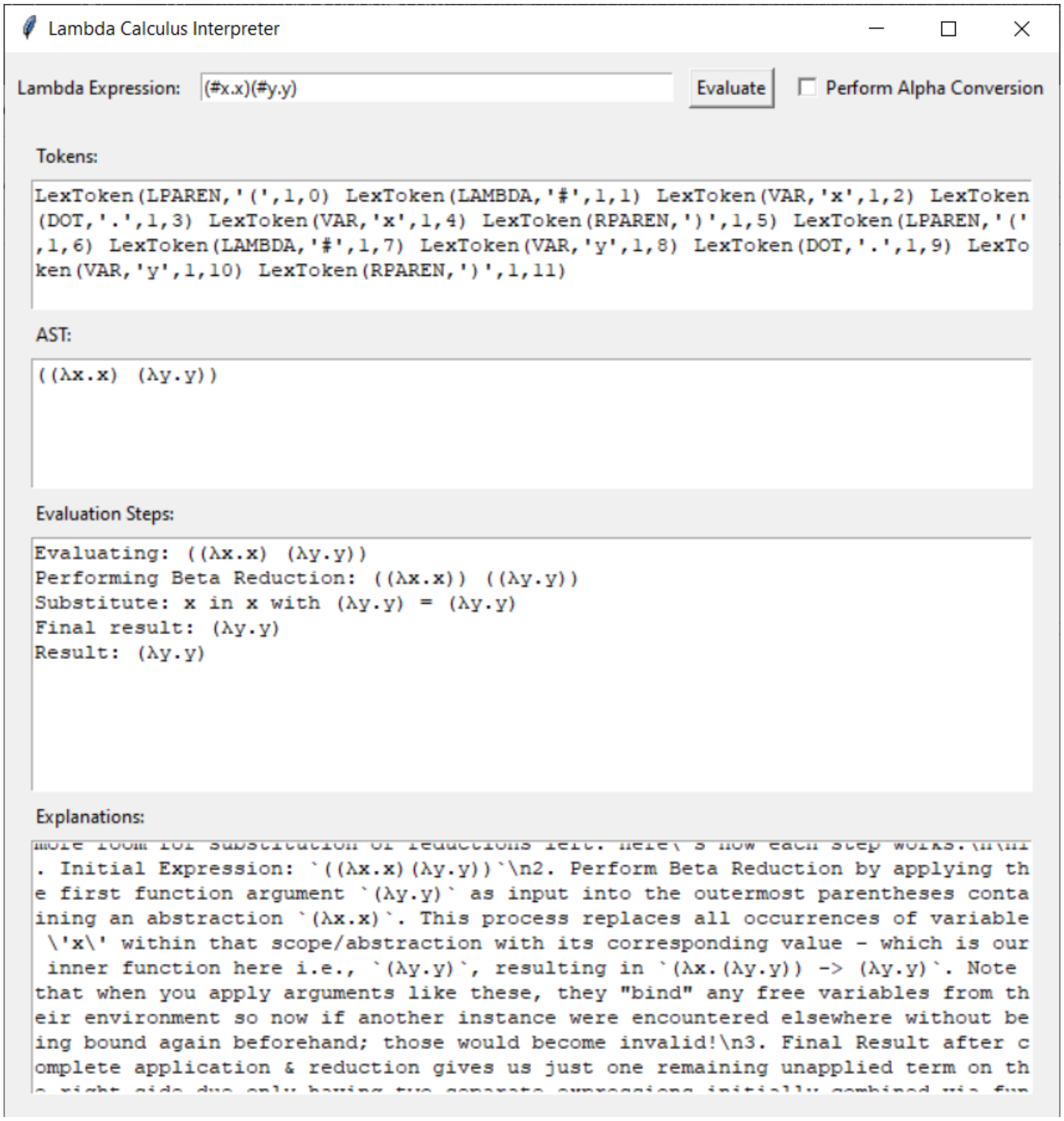
Our code is able to understand that and compute the necessary evaluation steps for alpha reduction. Below are the evaluation steps for our expression.

```
Evaluation steps:
Evaluating: ((λf.(λx.(f (f x)))) x)
Performing Beta Reduction: ((λf.(λx.(f (f x))))) (x)
Alpha Conversion: x -> t in f = f
Alpha Conversion: x -> t in f = f
Alpha Conversion: x -> t in x = t
Alpha Conversion: x -> t in (f x) = (f t)
Alpha Conversion: x -> t in (f (f x)) = (f (f t))
Alpha Conversion: x -> t in (λx.(f (f x))) = (λt.(f (f t)))
Substitute: f in f with x = x
Substitute: f in f with x = x
Substitute: f in t with x = t
Substitute: f in (f t) with x = (x t)
Substitute: f in (f (f t)) with x = (x (x t))
Substitute: f in (λt.(f (f t))) with x = (λt.(x (x t)))
Alpha Conversion: t -> L in x = x
Alpha Conversion: t -> L in x = x
Alpha Conversion: t -> L in t = L
Alpha Conversion: t -> L in (x t) = (x L)
Alpha Conversion: t -> L in (x (x t)) = (x (x L))
Alpha Conversion: t -> L in (λt.(x (x t))) = (λL.(x (x L)))
```

Final result: (λL.(x (x L)))

Result: (λL.(x (x L)))

Beta Reduction: (#x.x)(#y.y)



Lambda Expression: (#x.x)(#y.y)

☐ Perform Alpha Conversion

Tokens:

```
LexToken(LPAREN,'(',1,0) LexToken(LAMBDA,'#',1,1) LexToken(VAR,'x',1,2) LexToken
(DOT,'.',1,3) LexToken(VAR,'x',1,4) LexToken(RPAREN,')',1,5) LexToken(LPAREN,'('
,1,6) LexToken(LAMBDA,'#',1,7) LexToken(VAR,'y',1,8) LexToken(DOT,'.',1,9) LexTo
ken(VAR,'y',1,10) LexToken(RPAREN,')',1,11)
```

AST:

```
((λx.x) (λy.y))
```

Evaluation Steps:

```
Evaluating: ((λx.x) (λy.y))
Performing Beta Reduction: ((λx.x)) ((λy.y))
Substitute: x in x with (λy.y) = (λy.y)
Final result: (λy.y)
Result: (λy.y)
```

Explanations:

```
more room for substitution or reductions left. Here\'s how each step works:\n\n1
. Initial Expression: `((λx.x)(λy.y))`\n2. Perform Beta Reduction by applying th
e first function argument `(λy.y)` as input into the outermost parentheses conta
ining an abstraction `(λx.x)`. This process replaces all occurrences of variable
 \'x\' within that scope/abstraction with its corresponding value - which is our
 inner function here i.e., `(λy.y)`, resulting in `(λx.(λy.y)) -> (λy.y)`. Note
that when you apply arguments like these, they "bind" any free variables from th
eir environment so now if another instance were encountered elsewhere without be
ing bound again beforehand; those would become invalid!\n3. Final Result after c
omplete application & reduction gives us just one remaining unapplied term on th
e right side due only having two separate expressions initially combined via fun
```

For our lambda expression (#x.x)(#y.y)

Our code is able to execute a beta reduction by applying (#y,y) to every instance of x in lambda (#x.x)

Working out

(#x.x)(#y.y)

-> β [(#y.y)/x]x

X

(#y.y)

The evaluation steps from our code are as follow:

```
Evaluating: ((λx.x) (λy.y))

Performing Beta Reduction: ((λx.x)) ((λy.y))

Substitute: x in x with (λy.y) = (λy.y)

Final result: (λy.y)

Result: (λy.y)
```