Lambda Calculus Syntax

Group 41
https://github.com/richa-kd/csi3120

Problem Statement:

The goal of the program is to check if lambda calculus expressions are written correctly and to break them down into smaller parts, called tokens. If the expression is correct, the program creates a tree-like structure called a parse tree. Lambda calculus expressions are mathematical ways to define functions and how they apply to other expressions. This parser helps identify common mistakes, such as missing parentheses, incorrect variable names, or incomplete lambda expressions.

Parsing Method:

This program uses a top-down parsing approach, where it starts with the whole expression and breaks it down into smaller parts step by step. It goes from the largest part of the expression (like a lambda expression or a group in parentheses) and keeps breaking it down until it reaches individual components like variables or symbols.

Top-down parsing works well with lambda calculus because the expressions are simple and follow a straightforward structure. This method is also easier to code since we can use recursion—a function that calls itself to handle nested expressions, like parentheses inside other parentheses.

Examples with Parsing Techniques:

In top-down parsing, we start at the top (the main expression) and move downward, breaking the expression into smaller pieces. Here's how it works for some examples:
Lambda Expression \x.x:
=> Start with the lambda (\), which means we are defining a function.
=> Expect a variable (in this case, x).
=> Parse the body of the lambda (in this case, another x).
Result: The expression is valid and tokenized as: ['\\', 'x', 'x'].

Parentheses Expression (a (b c)):
=> Start with the opening parenthesis (().
=> Read the inner expression (a (b c)).
=> Read the nested parentheses (b c) and close them.
=> Close the outer parentheses.
Result: The expression is valid and tokenized as: ['(', 'a', '(', 'b', 'c', ')', ')'].