CSI 3120 Assignment 1 Report

Problem Statement:

The program developed is a lambda calculus parser designed to parse expressions in lambda calculus. The program processes and analyzes lambda expressions, ensuring that the syntax rules of lambda calculus are correctly followed and then it generates both a tokenized string and the corresponding parse tree. This is done by inputting the text files containing valid and invalid lambda expressions which are then checked for correct syntax for each individual symbol/element (such as variables, operators, parentheses and dots). The program is designed to handle various edge cases presented in the lambda expressions that will print out informative error messages based on the type of syntax error and its index position. For valid expressions, the parser will construct a parse tree representing the hierarchical structure of the expression which is done by breaking down the expression into components and then organized in a tree structure where each node represents a specific part of the expression. In summary, this is a tool to interpret, debug and evaluate if a given lambda expression is valid or invalid based on lambda calculus.

Parsing Method:

(Note that this is what the program should of been implemented like but there was an error producing the correct parse tree in the finished code)

The parsing method used in the program is recursive descent parsing, a type of top-down parsing. The parsing method works by recursively calling the *build_parse_tree_rec* function upon itself to handle the nested structure of the lambda expression such as the backslashes and the parentheses. The tokens are processed left to right (top = starting root and down is the leaves) in which the nodes are created based on what type of value the token shows, for example: parentheses, backslashes and variables. The recursion is implemented to process nested expressions such as when an open parentheses is the current value, it will build subtrees until it finds the next closing parenthesis. This is important to handle multiple levels of nesting where an expression can have various subtrees based on multiple token values.

The reason why this method is used is that it would make sense for the parse tree to be built from each individual token of the expression. Iterating from the start of the token while recursively calling using the next token will eventually build the full parse tree including the subtrees whenever the syntax detects a need for nested expressions.

Parsing examples:

(Assume all root nodes in the top down is shown as the top and the second step will always be a child node to the root node – The recursive descent parsing method used in the program acts similarly to the top down approach)

<u>Valid examples:</u>

1. (A B)
    - Top-Down:
        - Create root node
        - Check (and create the node
        - Check A and create node for A, but add as child node to (
        - Check B and create node for B, but add as child node to (
        - Check) which matches (and close the (node to complete tree
    - Bottom-Up:
        - Check A and create leaf node
        - Check B and create leaf node
        - Check (to create the non-terminal node
        - Add A and B as the child nodes to (
        - Check) which matches (and close the (node
2. abc
    - Top-Down:
        - Create root node
        - Check a and create node
        - Check b and create node
        - Check c and create node
        - **All added to root node

- Bottom-Up:
    - Check a b and c and create the leaf nodes
    - Add to the parent node that contains the leaf nodes

3. a b c
    - Top-Down:
        - Same as ex: 2 (whitespaces are skipped)
    - Bottom-Up:
        - Same as ex: 2 (whitespaces are skipped)
4. a (b c)
    - Top-Down:
        - Create root node
        - Check a and create node for a
        - Check (and create node for (
        - Check b and create node for b while adding to (as child node
        - Check c and create node for c while adding to (as child node
        - Check) which matches (and close the (node
    - Bottom-Up:
        - Check b and c to create the leaf nodes
        - Check (to create non-terminal
        - Add b and c node to non-terminal node (
        - Check) which matches and close the (node
5. (\x a b)
    - Top-Down:
        - Create root node
        - Check (and create node for (
        - Check \ and create node for \ while adding it to (node as a child node
        - Check x and create child node for x and add to \ node
        - Check a and create a node for a while adding to the \ node (part of the lambda expression)
        - Check b and create a node for b while adding to the \ node (part of the lambda expression)
        - Check) which matches and close the (node
    - Bottom-Up:
        - Check a and b and create leaf nodes

- Check \ to become parent node (lambda abstraction)
- Check x and create child node to add to \ node
- Add a and b leaf nodes to \ node
- Check (and) to match and close the expression

6. \x. a b
    - Top-Down:
        - Create root node
        - Check \ and create a child node to root node
        - Check x and create child node for x to the \ node (bound)
        - Check . node and create . node (indicates start of lambda expression)
        - Check a and create a child node to add to . node
        - Check b and create a child node to add to . node
    - Bottom-Up:
        - Create leaf nodes for a and b
        - Check . and create a non-terminal node for the .
        - Add a and b leaf nodes as the child nodes to the . node
        - Check x and create node for x (bound)
        - Check \ and create node for \
        - Add x as the child node to \
        - Add the . node as the child of the \ node (body of the lambda expression)
7. (\x((a) (b)))
    - Top-Down:
        - Create root node
        - Check (and create (node
        - Check \ node and add as a child node to the (node
        - Check x and create x node and add as the child node to \ node
        - Check (and create node for (and add as the child to the \ node
        - Check (and create node for (and add as the child to the previous (node
        - Check a and create node for a and add as the child node for the first (node
        - Check) and match with the last (node, closing the node for a
        - Check (and create node for (, adding it as a sibling to a
        - Check b and create a node for b and add as the child node to the second (node

        - Check) and match with last (node closing the node for b
        - Check) and match with the previous (node closing the node for a and b
        - Check) to match with the initial (node to close the entire expression
    - Bottom-Up:
        - Create leaf node for a
        - Check) and matches for a to close the a node
        - Create leaf node for b
        - Check) and matches for b to close the b node
        - Check (and create a non-terminal node for a and b
        - Add a and b leaf nodes as the child nodes of the non-terminal (node
        - Check (and create node for (
        - Check x and create node for x (bound)
        - Check \ and create node for \
        - Add x node as the child node to \
        - Check (and add the non-terminal node containing a and b as the child node to the \ node
        - Check) to match (for the lambda expression ((a)(b))
        - Check) to match with initial (node to close the entire expression

<u>Invalid examples:</u>

1. \ (missing variable after lambda).
    - Top-Down:
        - Check \ and create \ node → error occurs
    - Bottom-Up:
        - No tokens to evaluate → error occurs
2. \x (missing expression after lambda abstraction).
    - Top-Down:
        - Check \ and create \ node
        - Check x and create x node → error occurs
    - Bottom-Up:
        - Create leaf node for x → error occurs
3. ((x (missing closing parenthesis).
    - Top-Down:

        - Check (and create (node
        - Check (and create (node
        - Check x and create x node → error occurs
    - Bottom-Up:
        - Create x leaf node → error occurs
4. () (missing expression).
    - Top-Down:
        - Check (and create (node
        - Check) which matches and close the (node → error occurs
    - Bottom-Up:
        - No tokens to evaluate → error occurs
5. a (b (missing closing parenthesis).
    - Top-Down:
        - Check a and create a node
        - Check (and create (node
        - Check b and create b node → error occurs
    - Bottom-Up:
        - Create b leaf node → error occurs here
6. a (b c)) (input not fully parsed)
    - Top-Down:
        - Check a and create a node
        - Check (and crate (node
        - Check b and create b node\
        - Check c and create c node
        - Check) which matches (and close the node for (
        - Check) node → error occurs
    - Bottom-Up:
        - Create leaf nodes for b and c
        - Make the (b c) node
        - Check) and close the (node
        - Check) → error occurs