Lambda calculus syntax

Assignment 1

Group 24

Connor states, 300254333

Andres ackland-snow, 300081300

Professor:

Karim alghoul

Csi3120-b

University of ottawa

October 4th, 2024

**Problem statement**

The main goal of this project was to create a program that lexically analyzes and parses lambda calculus expressions. The program takes input from a text file, tokenizes the expressions, and uses recursive parsing to determine if the syntax is valid based on the lambda calculus grammar.

Also, the program constructs and outputs a structured representation of the expression in the form of an abstract syntax tree (ast). The parser is needed to handle expressions that involve variables, parentheses, and lambda abstractions (\). It also gives error messages with details when syntax errors are detected, ensuring the programs reliability in analyzing complex lambda expressions.

The more broad goal is to make the understanding of lambda calculus syntax easier, find errors in the structure of lambda expressions, and give a structured output that clarifies relationships between the individual parts of the expression. This is useful for academic purposes and practical applications in functional programming languages, where lambda calculus is very important.

**Parsing method:**

Our program uses recursive descent parsing, which is a type of top-down parsing method. Recursive descent parsing involves breaking down an expression into its components by recursively parsing each part of the expression based on the defined grammar rules. We chose this method because it matches well with the grammar in the examples included with the assignment, which can be more easily divided into smaller sub-expressions.

How the parsing works:

1. The program starts by reading the input string and breaking it down into tokens (variables, parentheses, backslashes for lambda abstractions, etc.).
2. Once tokenized, the program recursively processes each part of the expression, starting from the outermost components and working inward.
   1. Ex: if the expression is wrapped in parentheses, the parser recognizes the parentheses, then recursively parses the sub-expression inside the parentheses.
3. Lambda abstractions (\x) are handled by identifying the backslash, then ensuring the following part of the expression is a valid variable, and then parsing the rest of the expression.

The reason we chose recursive descent parsing is that the grammar for lambda calculus is simple and can be broken down into smaller parts. Also, top-down parsing allows for clear and readable error messages, because the program processes the expression step by step, identifying where the issue occurs.

**Examples with parsing techniques:**

Our code made use of the top-down parsing method.

Valid examples:

Example 1: (a b)

1. Top-down parsing steps:

* Recognize the outermost parentheses: ( and ).
* Inside the parentheses, there are two tokens: a and b.
* Break the expression into smaller parts:
  + A: identify this as a variable or an identifier.
  + B: similarly, identify this as a variable or identifier.

1. Bottom-up parsing steps:

* Begin by identifying the smallest tokens: a and b
* Look at the structure that contains these tokens: a and b are enclosed in parentheses.
* Recognize that the combination of tokens (a b) forms a valid expression.
* Merge the tokens and the parentheses into a valid expression node.

1. How the program handles the input:

* The program begins by reading the input line (a b).
* The parse\_tokens function is called, which:
  + Skips the opening parenthesis (.
  + Recognizes a as a valid variable name.
  + Recognizes b as a valid variable name.
  + Detects the closing parenthesis ) and confirms that the expression inside the parentheses is valid.
* A list of tokens [(, a, b, )] is constructed.
* The build\_parse\_tree function then:
  + Creates a root node for the entire expression.
  + Adds a and b as children of the root node.
  + Successfully constructs the parse tree and outputs it using the print\_tree function.

Example 2: abc

1. Top-down parsing steps:

* Start by looking for a valid variable name: the entire string abc is a valid variable (starts with a letter, followed by more valid characters).
* Since no parentheses or other characters are present, the entire string is treated as a single token.

1. Bottom-up parsing steps:

* Begin with identifying the smallest tokens: recognize abc as a valid variable.
* No further tokens to process, so the expression is valid, as it is just a single valid variable name.

1. How the program handles the input:

* The program reads the input abc.
* The parse\_tokens function:
  + Identifies abc as a valid variable name.
  + Adds it to the token list.
* The tokens list [abc] is created.
* The build\_parse\_tree function:
  + Creates a root node with the token abc.
  + Since there are no further tokens, the tree consists only of this root node.
* The parse tree is output, showing abc as the only node.

Example 3: a b c

1. Top-down parsing steps:

* Parse each individual token: identify a, b, and c as valid variable names.
* No parentheses or complex structures are present, so each token is treated individually.
* Parsing completes successfully as all tokens are valid.

1. Bottom-up parsing steps:

* Identify the smallest tokens: a, b, and c: each of these tokens is recognized as a valid variable.
* No further structure is needed.
* The expression is valid as it consists of a sequence of valid variables.

1. How the program handles the input:

* The program reads the input a b c.
* The parse\_tokens function:
  + Recognizes a, b, and c as valid variable names.
  + Adds each of them to the token list.
* The tokens list [a, b, c] is created.
* The build\_parse\_tree function:
  + Creates a root node with the token list [a, b, c].
  + Each token is added as a child node of the root.
* The parse tree is printed, showing a, b, and c as separate child nodes under the root.

Example 4: a (b c)

1. Top-down parsing steps:

* Start by parsing the first token a, which is a valid variable.
* Encounter the opening parenthesis (, indicating the start of a sub-expression.
* Inside the parentheses, identify b and c as valid variables.
* Conclude the sub-expression with the closing parenthesis ).
* Parsing completes successfully, with ‘a’ followed by a valid parenthesized expression (b c).

1. Bottom-up parsing steps:

* Identify the smallest tokens: a, b, and c.
* Recognize a as a valid variable.
* Encounter the parentheses ( and ) that enclose b and c.
* Group b and c into a valid sub-expression.
* Combine the tokens into a complete valid expression: a (b c).

1. How the program handles the input:

* The program reads the input a (b c).
* The parse\_tokens function:
  + Recognizes a as a valid variable.
  + Detects the opening parenthesis (.
  + Identifies b and c as valid variables inside the parentheses.
  + Detects the closing parenthesis ) and confirms that the sub-expression is valid.
* The tokens list [a, (, b, c, )] is created.
* The build\_parse\_tree function:
  + Creates a root node with a and the sub-expression (b c) as children.
  + Adds b and c as child nodes within the sub-expression.
* The parse tree is printed, showing a and the sub-expression (b c) as separate branches.

Example 5: (\x a b)

1. Top-down parsing steps:

* Start by recognizing the lambda abstraction \x.
* Identify a and b as valid variables following the lambda abstraction.
* The parentheses indicate that \x a b is a single expression.
* Parsing completes successfully.

1. Bottom-up parsing steps:

* Identify the smallest tokens: \x, a, and b.
* Recognize \x as a valid lambda abstraction.
* Identify a and b as valid variables.
* Combine the tokens into the complete expression (\x a b).

1. How the program handles the input:

* The program reads the input (\x a b).
* The parse\_tokens function:
  + Recognizes the lambda abstraction \x.
  + Identifies a and b as valid variables following the lambda.
  + Confirms the parentheses enclose a valid expression.
* The tokens list [(, \\x, a, b, )] is created.
* The build\_parse\_tree function:
  + Creates a root node with the lambda expression as a child.
  + Adds a and b as child nodes of the lambda expression.
* The parse tree is printed, showing the lambda expression and its variables as separate branches.

Example 6: \x. A b

1. Top-down parsing steps:

* Start with the lambda abstraction \x.
* The dot . Indicates the separation between the lambda parameter and the body of the expression.
* Identify a and b as valid variables in the body of the lambda expression.
* Parsing completes successfully, as the expression is in the form of a valid lambda abstraction followed by valid variables.

1. Bottom-up parsing steps:

* Identify the smallest tokens: \x, ., a, and b.
* Recognize \x as a valid lambda abstraction.
* The dot . Indicates the start of the lambda body.
* Identify a and b as valid variables in the body of the lambda expression.
* Combine all tokens into the complete valid lambda expression \x. A b.

1. How the program handles the input:

* The program reads the input \x. A b.
* The parse\_tokens function:
  + Recognizes the lambda abstraction \x.
  + Detects the dot . Separating the lambda parameter from the body.
  + Identifies a and b as valid variables in the lambda body.
* The tokens list ['\\x', '.', 'a', 'b'] is created.
* The build\_parse\_tree function:
  + Creates a root node for the lambda abstraction.
  + Adds a and b as child nodes representing the body of the lambda expression.
* The parse tree is printed, showing the lambda expression and its variables.

Example 7: (\x((a) (b)))

1. Top-down parsing steps:

* Start with the outermost parentheses ( and ).
* Recognize the lambda abstraction \x.
* Inside the lambda abstraction, identify a nested structure:
* Parse the sub-expressions (a) and (b) separately.
* Conclude the parsing by recognizing that both sub-expressions are valid and enclosed within the lambda abstraction.

1. Bottom-up parsing steps:

* Begin by identifying the smallest tokens: \x, a, and b.
* Recognize the parentheses enclosing the expressions: ( around both a and b.
* The parentheses indicate that the expressions (a) and (b) are sub-expressions inside the lambda abstraction.
* Combine the tokens and the parentheses into a complete valid expression: (\x((a) (b))).

1. How the program handles the input:

* The program reads the input (\x((a) (b))).
* The parse\_tokens function:
  + Recognizes the lambda abstraction \x.
  + Detects the nested parentheses enclosing the expressions (a) and (b).
  + Identifies a and b as valid variables inside the parentheses.
  + Confirms the entire expression is valid with the proper parentheses structure.
* The tokens list ['(', '\\x', '(', '(', 'a', ')', '(', 'b', ')', ')', ')'] is created.
* The build\_parse\_tree function:
  + Creates a root node for the lambda abstraction and nested structure.
  + Adds sub-nodes for the expressions (a) and (b) within the lambda expression.
* The parse tree is printed, showing the lambda abstraction and the nested sub-expressions.

Invalid examples**:**

Example 1: \ (missing variable after lambda)

1. Top-down parsing steps:

* Start by recognizing the lambda symbol \.
* Expect a variable name to follow the lambda, but instead encounter a space, which is not valid.
* There is an error, because there is no valid variable after the lambda.

1. Bottom-up parsing steps:

* Begin with identifying the lambda symbol \.
* Expect a variable name to follow, but encounter an invalid space instead.
* There is an error, because the required variable is missing after the lambda.

1. How the program handles the input:

* The program reads the input \ .
* The parse\_tokens function:
  + Recognizes the lambda symbol \.
  + Detects an invalid space after the lambda, which is not allowed.
  + Prints an error message: invalid space inserted after \ at index 0.
  + The function returns false to indicate that the parsing failed.
* The program moves on without building a parse tree since the input is invalid.

Example 2: \x (missing expression after lambda abstraction)

1. Top-down parsing steps:

* Start by recognizing the lambda abstraction \x.
* Expect an expression or variables to follow the lambda, but encounter an empty space.
* There is an error, because there is no valid expression after the lambda abstraction.

1. Bottom-up parsing steps:

* Begin by identifying the lambda abstraction \x.
* Expect a body (expression) to follow the lambda, but encounter nothing.
* There is an error, because the body of the lambda expression is missing.

1. How the program handles the input:

* The program reads the input \x .
* The parse\_tokens function:
  + Recognizes the lambda abstraction \x.
  + Detects that there is no expression following the lambda, which is invalid.
  + Prints an error message: missing complete lambda expression starting at index 2.
  + The function returns false to indicate that the parsing failed.
* No parse tree is constructed due to the invalid input.

Example 3: ((x (missing closing parenthesis)

1. Top-down parsing steps:

* Start by recognizing the opening parentheses ((.
* Identify x as a valid variable.
* Expect a closing parenthesis ) to match the opening one, but it is missing.
* There is an error, because the parentheses are not balanced.

1. Bottom-up parsing steps:

* Begin by identifying the variable x.
* Recognize the parentheses structure ((.
* Expect a matching closing parenthesis, but it is not found.
* There is an error, because of the unbalanced parentheses.

1. How the program handles the input:

* The program reads the input ((x.
* The parse\_tokens function:
  + Recognizes the opening parentheses ((.
  + Identifies x as a valid variable.
  + Detects that there is no matching closing parenthesis for the opening (.
  + Prints an error message: bracket ( at index: 0 not matched with a closing bracket ')'.
  + The function returns false due to the unbalanced parentheses.
* No parse tree is built due to the syntax error.

Example 4: () (missing expression)

1. Top-down parsing steps:

* Start by recognizing the opening and closing parentheses ().
* Expect an expression inside the parentheses, but find none.
* There is an error, because the parentheses enclose an empty expression, which is invalid.

1. Bottom-up parsing steps:

* Begin by identifying the parentheses ().
* Expect some tokens or an expression inside the parentheses, but find none.
* There is an error, because of the empty expression within the parentheses.

1. How the program handles the input:

* The program reads the input ().
* The parse\_tokens function:
  + Detects the empty parentheses ().
  + Recognizes that there is no expression inside the parentheses, which is invalid.
  + Prints an error message: missing expression for parenthesis at 0.
  + The function returns false because the parentheses enclose nothing.
* No parse tree is generated due to the missing expression.

Example 5: a (b (missing closing parenthesis)

1. Top-down parsing steps:

* Start by recognizing a as a valid variable.
* Recognize the opening parenthesis (, indicating the start of a sub-expression.
* Identify b as a valid variable inside the parentheses.
* Expect a closing parenthesis ), but it is missing.
* There is an error, because of the unbalanced parentheses.

1. Bottom-up parsing steps:

* Identify the smallest tokens: a and b.
* Recognize the parentheses structure: ( around b.
* Expect a closing parenthesis ) for the outer expression, but it is not present.
* There is an error, because of the unbalanced parentheses.

1. How the program handles the input:

* The program reads the input a (b.
* The parse\_tokens function:
  + Recognizes a as a valid variable.
  + Identifies the opening parenthesis ( for the sub-expression.
  + Recognizes b as a valid variable inside the parentheses.
  + Detects that there is no closing parenthesis for the expression.
  + Prints an error message: bracket ( at index: 2 not matched with a closing bracket ')'.
  + The function returns false due to the unbalanced parentheses.
* No parse tree is created because of the syntax error.

Example 6: a (b c)) (input not fully parsed)

1. Top-down parsing steps:

* Start by recognizing a as a valid variable.
* Recognize the parentheses ( enclosing the sub-expression b c.
* Expect the closing parenthesis ) to match the opening one, but encounter an extra closing parenthesis.
* There is an error, because there is an extra closing parenthesis that does not correspond to any opening one.

1. Bottom-up parsing steps:

* Identify the smallest tokens: a, b, and c.
* Recognize the parentheses structure: ( around b c.
* Encounter an extra closing parenthesis ) that is not part of any valid expression.
* There is an error, because of the unmatched closing parenthesis.

1. How the program handles the input:

* The program reads the input a (b c)).
* The parse\_tokens function:
  + Recognizes a as a valid variable.
  + Detects the opening parenthesis ( for the sub-expression.
  + Identifies b and c as valid variables inside the parentheses.
  + Detects an extra closing parenthesis ) that has no corresponding opening parenthesis.
  + Prints an error message: bracket ) at index: 7 not matched with a opening bracket '('.
  + The function returns false due to the unmatched parenthesis.
* No parse tree is built because the input is not fully parsed.