Assignment 1: Lambda Calculus Syntax Report

CSI3120-B (Fall 2024)
Professor: Karim Alghoul
Student Name: John-Adrian, Zachary Sikka
Student Number: 300262408, 300305350

Program Overview

This report provides an overview of the implementation for Assignment 1, which involves implementing a lexical analyzer and syntax analyzer for lambda calculus expressions based on a set of given grammar rules. Lambda calculus is utilized to express calculus expressions through abstraction. The program will read multiple text files (some filled with valid examples and others filled with invalid examples of lambda expressions), tokenize these lambda expressions, then parse these tokens and generate a parse tree. The goal of this program is to ensure the syntax of the lambda expressions are correct, detect any errors, and generate a parse tree that reflects the structure of the input.

Achieving these objectives allows us to validate lambda calculus expressions.

Grammar rules

The following rules are what we must abide by when evaluating lambda expressions

Program Structure:

- `<expr> ::= <var> | '(' <expr> ')' | '\' <var> <expr> | <expr> <expr>`

Variable Names: Consist of letters a-z, A-Z, and digits 0-9, but must start with a letter.

Whitespace: Whitespace insensitive, but whitespace should separate variables

The Dot (.) after lambda abstraction: The dot (.) can be used after a lambda abstraction variable to clarify grouping, acting like parentheses. When a dot is used, it groups everything to the right of the dot with the closest lambda abstraction. It’s as if you're inserting parentheses around the expression following the dot.

Features Implemented

Here are the following features implemented:

1. Lexical Analysis (John)

    a. The program breaks down the input expression into tokens, which includes \, (,), and variable names.
    b. The program ensures that any variable name tokenized is valid, by following the grammar rules above
    c. Whitespace is ignored but used to separate tokens like variables
    d. Parentheses and lambda expressions are tokenized to represent grouping in parsed expressions
2. Parsing (Zach):
    a. The program uses the top down parsing technique to parse each expression. Specifically it uses recursive descent parsing to handle nested structures such as lambda expressions or parentheses
    b. This method was chosen as it was explained in class and works well for simple grammar rules. This is because each non terminal in the grammar rules corresponds to a function, which makes the process clear to build and follow.
    c. The parsing always starts from the first grammar definition(an expression in this case). It then works from left to right to parse each token and match it to the corresponding grammar rule.
    d. Once each non-terminal is matched to a grammar rule and parsed it is added to the parse tree
3. Error Handling (John):
    a. During the tokenizing process, we handle improper expressions, such as missing parentheses, invalid variable names, and improper lambda abstraction usage.
    b. The error messages include the position in the input string where the error occurred and a small description of the issue.

Examples with Parsing Techniques:

Invalid_examples.txt:

Input: \x(

Output: Error: Bracket '(' at index 2 is not matched with a closing bracket ')'

The program encounters the \, indicating a lambda expression, then parses x as a variable. However, when we finish processing the entire expression, we check to see firstly if there is an open parenthesis (which we keep track via variable open_parens). In this scenario, we have an open parenthesis, so there is an unmatched bracket error

Input: \x

Output: Error: Missing expression after lambda abstraction at index 0

We process \x as a valid lambda expression, but after parsing the two, we check if we reached the end of the string. If we do, it means that there is no valid lambda expression, so we raise an error

Input: \\x\\

Output: Error: Backslash '\' not followed by a valid variable name at index 0

We start with parsing \, but when we check to find the variable name (as per the grammar rules of lambda calculus), we find another \, thus raising an error for invalid variable (as that is what is expected next)

Input: ((x

Output: Error: Bracket '(' at index 0 is not matched with a closing bracket ')'

We process the parentheses and the variable x, but when its time to return the tokens, we check to see if there are any open parentheses. If there are, we return false.

Input: ()

Output: Error: Missing expression inside parentheses at index 1

When we check for ‘(‘, we make sure that the next character is not a ‘)’. If it is the case, our program returns a missing expression error.

Input: a (b

Output: Error: Bracket '(' at index 2 is not matched with a closing bracket ')'

Our program processes the variables and skip the whitespace, but at the end, it does a check for open parentheses and returns an error.

Input: a (b c))

Output: Bracket ')' at index 7 is not matched with an opening bracket '('.

Our program processes the variables and skip the whitespace, but when it processes ‘)’. It checks if the open_parens counter is 0. If its the case, there is no open parentheses in the expression that can match our closing parentheses.

Input: .

Output: Error: Must have a variable name before character '.' at index 0

When we process the ‘.’ character, we check if the dot variable follows a valid variable (since theoretically, we would have already processed the variable if we did a pass by it, we merely check if the previous (or i-1) character was a valid character. In this case, we had no valid variable before the dot, so we raise an error.

Input: (

Output: Error: Bracket '(' at index 0 is not matched with a closing bracket ')'

As said before, we check to see if there are any unmatched parentheses at the end of our program. In this case, we raise an error.

Input:)

Output: Bracket ')' at index 0 is not matched with an opening bracket '('.

Whenever we process the ‘)’ character, we check if the number of open_parens equals 0. If it is the case, like in this input, we return 0 as there are no open parentheses to match our ‘)’.

Input: 1ab

Output: Error: Invalid character '1' at index 0

When we come across an alphanumeric character, we check to see if its a valid variable. In our situation, it is not as it starts with a number.

Input: \ x. (a b)

Output: Error: Invalid space inserted after '\' at index 0

When our program processes backslashes, we make sure that there are no space characters after it. If it is the case, then the string is invalid and cannot be parsed.

Input: \ x.(x z)

Output: Error: Invalid space inserted after '\' at index 0

When our program processes backslashes, we make sure that there are no space characters after it. If it is the case, then the string is invalid and cannot be parsed.

Input: \

Output: Error: Backslash '\' not followed by a valid variable name at index 0

When we process backslashes, we check if the character after the backslash is not alphanumeric or if the index has reached the length of the string. In this scenario, our index has reached the length of the string, and therefore returns invalid variable as there is no variable after the backslash

Input: (.

Output: Error: Must have a variable name before character '.' at index 1

When we process dots, we check if there is a valid token before the . (from the list of valid characters excluding parentheses, backslashes, and dots.

Input:).

Output: Bracket ')' at index 0 is not matched with an opening bracket '('.

We process the ‘)’ and realize that our open_parens counter is 0, which indicates that there is no open parentheses that can match our closing one

Input: (.)

Output: Error: Must have a variable name before character '.' at index 1

When we process dots, we check if there is a valid token before the . (from the list of valid characters excluding parentheses, backslashes, and dots.

Input: \x .(x z)

Output: Error: Must have a variable name before character '.' at index 3

When we process dots, we check if there is a valid token before the . (from the list of valid characters excluding parentheses, backslashes, and dots.) In this scenario, we find a space, which makes the expression invalid in the eyes of our program.

Input: \ x.(x z)

Output: Error: Invalid space inserted after '\' at index 0

Our program, whenever it processes a \, checks if the next character in the string is not a space. If it is, it returns an error.

Input: ++

Output: Error: Invalid character '+' at index 0

Our program checks for any character that is not in the list of all valid characters in the program. If this check is true, it returns an error.

Input: \ab

Output: Error: Missing expression after lambda abstraction at index 0

Our program first processes the backslash, then iterates through the string to find the entire variable. Once it confirms that the backslash and the variable are valid, it does a check to see if the next character is either part of the valid characters in lambda calculus, or if the index has reached the end of the string. If that's the case, it returns an invalid expression error.

Input: \ a b

Output: Error: Invalid space inserted after '\' at index 0

Our program, whenever it processes a \, checks for any spaces after the \. If that is the case, it returns an invalid space error.

Input: \(c c)

Output: Error: Backslash '\' not followed by a valid variable name at index 0

Whenever we process backslashes, we check if the next character is part of a variable. In order to do this check we process the character vs a list of all the valid characters within a variable. If the character is not in this list, we return an invalid variable error.

extra_Invalid_examples.txt:

Input: \x y.

Output: Missing expression after dot at index 5

Error: We check for backslash first, then process both \ and x. Afterwards our program passes the check for a valid expression and continues. Finally, it checks the dot character, where it checks if there is a valid expression after the . If there is not, then it returns an error

Input: (\x x) \

Output: Backslash '\' not followed by a valid variable name at index 9

Firstly, the program approves the characters (, \, x, x,), as the parentheses are part of the grammar rule (expr), and the backslash is followed by a valid variable, and x is part of the list of valid variables

Input: a..b

Output: Must have a variable name before character '.' at index 2

Whenever we check a dot character, our program makes sure that a valid variable precedes it. For the first dot character, it passes the test as we have a before the '.'. However, the second '.' returns an error as the character that proceeds it ('.') is not a valid character for variable names.

Input: \\ x (\y.) z

Output: Backslash '\' not followed by a valid variable name at index 0

Whenever we check a backslash, our program checks the next characters for a variable name, once it defines what the variable name is, it checks via a function to determine whether the variable is valid aka, if it follows the grammar rules of starting with an alphabetic letter and contains letters and numbers only. In this case, the variable the program found after '\' was another '\', which is invalid

Input: (a b (c d e)))

Output: Bracket ')' at index 14 is not matched with an opening bracket '('.'.

Each time we encounter a parentheses, our program increases a open parentheses counter, and checks if the next character is a ')'. If it is, it returns an error as this is an empty parentheses, which goes against the grammar rules of lambda calculus. Next, it will check all the closing parentheses and decrement our open parentheses counter. However, if it encounters a closing parentheses and sees that the counter is 0, it returns an error as there is no more open parentheses to match the closing one.

Extra_valid_examples.txt:

Input: (\x. x (y z)) (\a. a b)

Tokenized output: (__x_(_x_(_y_z_)_)_(__a_(_a_b_)_)_)

Start Parsing Parenthesized Expression: The parser encounters '(' and calls parse_paren_expr().

- This begins parsing the first part of the expression.

Lambda Abstraction \x.:

- The parser encounters \\ and calls parse_lambda_expr().
- Inside parse_lambda_expr(), it creates a lambda node for x and advances past the dot (.).
- parse_expr() is then called to parse the body of the lambda, which starts with x.
    - parse_var() is called to parse the variable x.

Application x (y z):

- The parser sees (and calls parse_paren_expr() again.
- Inside the parentheses, parse_expr() calls parse_var() for y and z, building an application node for y z.

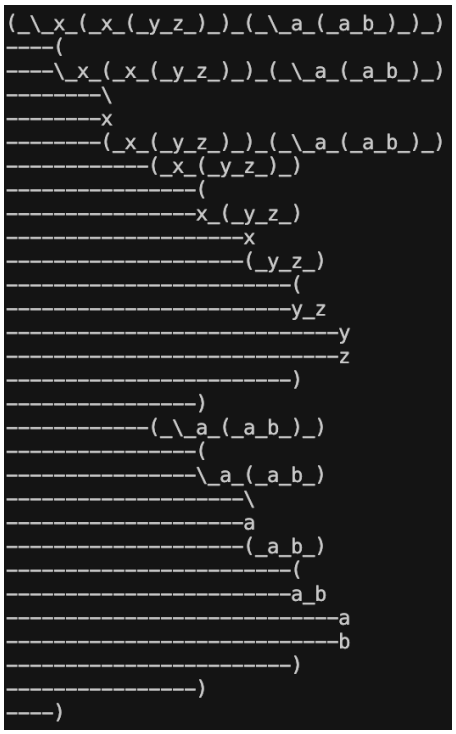
Second Lambda Abstraction \a. a b:

- After parsing the first lambda expression, the parser encounters the second lambda abstraction \a. a b.
- It calls parse_lambda_expr() for a and processes the body (a b) using parse_expr() and parse_var().

Final Combination:

- The two parts are combined into an application using parse_expr().

Parse Tree:



Input :\x. ((a b) (x y z))

Tokenized output: _x_(_(_(_a_b_)_(_x_y_z_)_)_)

Lambda Abstraction \x.:

- The parser encounters \\ and calls parse_lambda_expr().
- It creates a lambda node for x and calls parse_expr() to parse the body.

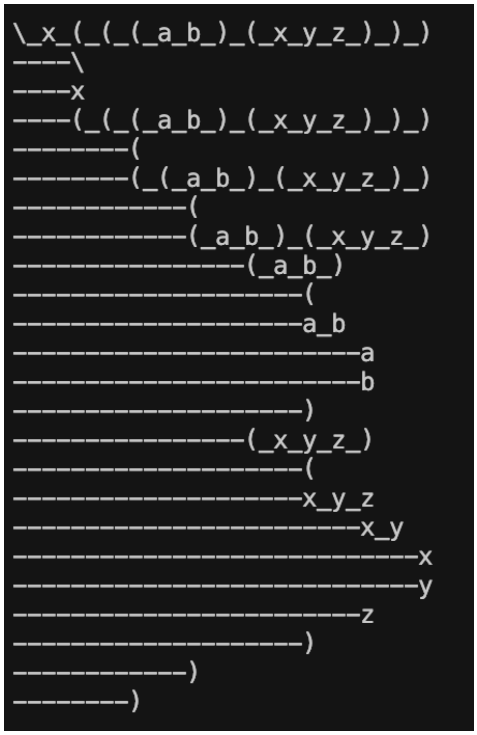
Nested Parentheses ((a b) (x y z)):

- The parser sees (and calls parse_paren_expr().
- It recursively parses both inner parts a b and x y z.
    - For a b, parse_expr() and parse_var() are called for a and b.
    - For x y z, parse_expr() builds an application with parse_var() for x, y, and z.

Final Combination:

- The two parts are combined into an application node in parse_expr().

Parse Tree:



Input: (\x. \y. x (y z)) (\a. a b c)

Tokenized output: (__x_(__y_(_x_(_y_z_)_)_(__a_(_a_b_c_)_)_)_)

First Lambda Abstraction \x.:

- The parser starts with parse_lambda_expr() for \x.
- It creates a lambda node for x and parses the body using parse_expr().

Second Lambda Abstraction \y.:

- Inside the first lambda, it encounters another lambda \y and calls parse_lambda_expr() again.
- This creates a nested lambda node for y.

Application x (y z):

- The parser then parses x (y z) using parse_expr() and parse_var(), building an application node.

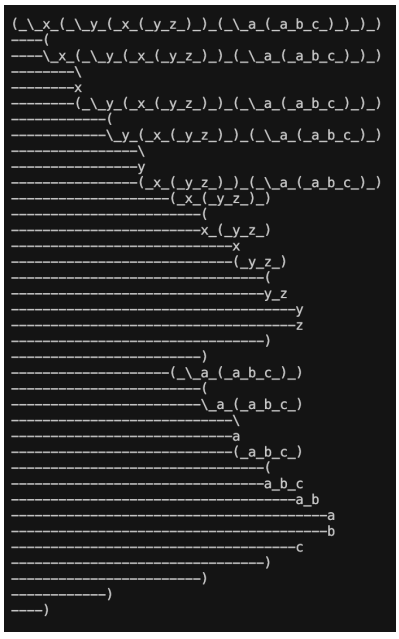
Second Expression \a. a b c:

- The parser processes (\a. a b c) with parse_lambda_expr() for a and parses the body using parse_expr() for a, b, and c.

Final Combination:

- The two lambda expressions are combined into an application node by parse_expr().

Parse Tree:



Input: \x. ((x (y z)) (a (b c)))

Tokenized output: _x_(_(_(_x_(_y_z_)_)_(_a_(_b_c_)_)_)_)

Lambda Abstraction \x.:

- The parser calls parse_lambda_expr() to handle \x, and creates a lambda node.

First Parentheses (x (y z)):

- Inside the body of the lambda, the parser encounters a parenthesized expression ((x (y z)) and calls parse_paren_expr().
- It builds an application of x to (y z) using parse_expr() and parse_var() for y and z.

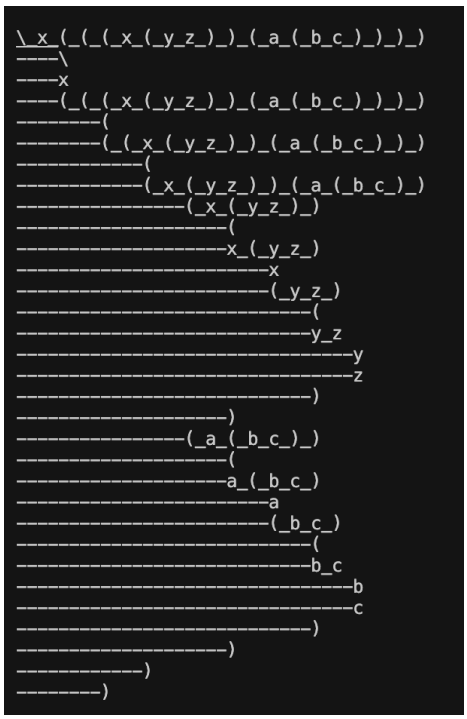
Second Parentheses (a (b c)):

- After parsing the first application, it moves to (a (b c)) and processes it similarly, creating an application of a to (b c) using parse_expr() and parse_var().

Final Combination:

- The two parts are combined into an application node inside parse_expr().

Parse Tree


:

Input: (a (\b. b c)) (d e)

Tokenized output: (_a_(__b_(_b_c_)_)_(_d_e_)_)

Lambda Abstraction \x.:

- The parser calls parse_lambda_expr() to handle \x, and creates a lambda node.

First Parentheses (x (y z)):

- Inside the body of the lambda, the parser encounters a parenthesized expression ((x (y z)) and calls parse_paren_expr().
- It builds an application of x to (y z) using parse_expr() and parse_var() for y and z.

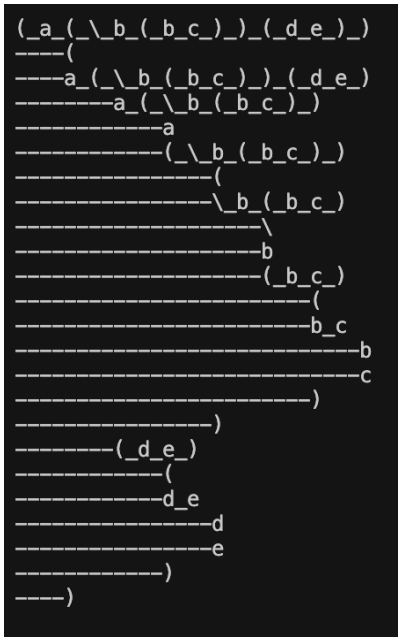
Second Parentheses (a (b c)):

- After parsing the first application, it moves to (a (b c)) and processes it similarly, creating an application of a to (b c) using parse_expr() and parse_var().

Final Combination:

- The two parts are combined into an application node inside parse_expr().

Parse Tree:



Examples In A1 instructions:

General Top Down and Bottom up Explanations

```
<expr> ::= <var> | '(' <expr> ')' | '\' <var> <expr> | <expr> <expr>
```

Example 1: (A B)

- Start with <expr>:
  - Encounter (→ Apply rule <expr> ::= '(' <expr> ')'.
  - Parse the subexpression inside the parentheses (A B).
- Start with <expr>:
  - Encounter A → Apply rule <expr> ::= <var>.
  - Encounter B → Apply rule <expr> ::= <var>.
  - Apply rule <expr> ::= <expr> <expr> to combine A and B.
- Close the parentheses and return the parsed expression.

Example 2: abc

- Start with <expr>:
  - Encounter abc → Apply rule <expr> ::= <var>.
  - Since there are no parentheses or lambdas, return the variable.

Example 3: a b c

- Start with <expr>:
  - Encounter a → Apply rule <expr> ::= <var>.
  - Encounter b → Apply rule <expr> ::= <var>.
  - Combine a and b into an application using <expr> ::= <expr> <expr>.
  - Encounter c → Apply rule <expr> ::= <var>.
  - Combine the result of a b and c using <expr> ::= <expr> <expr>.

Example 4: a (b c)

- Start with <expr>:
  - Encounter a → Apply rule <expr> ::= <var>.
  - Encounter (→ Apply rule <expr> ::= '(' <expr> ')'.
  - Parse the subexpression inside parentheses (b c).
- Start with <expr>:
  - Encounter b → Apply rule <expr> ::= <var>.
  - Encounter c → Apply rule <expr> ::= <var>.
  - Combine b and c using <expr> ::= <expr> <expr>.
- Combine a and (b c) using <expr> ::= <expr> <expr>.

Example 5: (\x a b)

- Start with <expr>:
  - Encounter (→ Apply rule <expr> ::= '(' <expr> ')'.
  - Inside parentheses, encounter \ → Apply rule <expr> ::= '\' <var> <expr>.
  - Parse the lambda abstraction for x.
- Start with <expr>:
  - Encounter a → Apply rule <expr> ::= <var>.
  - Encounter b → Apply rule <expr> ::= <var>.
  - Combine a and b using <expr> ::= <expr> <expr>.
- Close the parentheses.

Example 6: \x. a b

- Start with <expr>:
  - Encounter \ → Apply rule <expr> ::= '\' <var> <expr>.
  - Parse the lambda abstraction for x.
- Start with <expr>:
  - Encounter a → Apply rule <expr> ::= <var>.
  - Encounter b → Apply rule <expr> ::= <var>.
  - Combine a and b using <expr> ::= <expr> <expr>.

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

- Start with <expr>:
  - Encounter (→ Apply rule <expr> ::= '(' <expr> ')'.
  - Inside parentheses, encounter \ → Apply rule <expr> ::= '\' <var> <expr>.
  - Parse the lambda abstraction for x.
- Start with <expr>:
  - Encounter ((a) (b)).
  - Encounter nested (and recursively parse a and b, applying them to each other
  - Combine the nested expression inside the lambda body

Bottom Up:

Example 1: (A B)

Tokenized: (<var> <var>)

= (<expr> <var>)

= (<expr> <expr>)

= (<expr>)

= <expr>

Example 2: abc

Tokenized: <var>

= <expr>

Example 3: a b c

Tokenized: <var> <var> <var>

= <expr> <expr> <var>

= <expr> <var>

= <expr> <expr>

= <expr>

Example 4: a (b c)

Tokenized: <var> (<var> <var>)

= <expr> (<expr> <expr)

= <expr> <expr>

= <expr>

Example 5: (\x a b)

Tokenized: (\ <var> <var> <var>)

= (\ <var> <expr> <expr>

= (\ <var> <expr>)

= (<expr>)

= <expr>

Example 6: \x. a b

Tokenized: \ <var> (<var> <var>)

= \ <var> (<expr> <expr>)

= \<var> (<expr>)

= \<var> <expr>

= <expr>

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

Tokenized: (\ <var> ((<var>) (<var>)))

= (\ <var> ((<var>) (<expr>)))

= (\ <var> ((<var>) <expr>))

= (\ <var> ((<expr>) <expr>))

= (\ <var> (<expr> <expr>))

= (\ <var> (<expr>))

= (\ <var> <expr>)

= (<expr>)

= <expr>