CPSC-354 Report

Marc Domingo
Chapman University

October 4, 2022

Contents

1 Introduction

1.1 Week 1

During the first week of Programming Languages, the class lectures went over the concept of *Imperative vs Functional Programming* and *Recursive Programming* in addition to a brief introduction to the programming language **Haskell**.

1.2 Week 2

During the second week of Programming Languages, the class lectures went more into depth regarding techniques in how to use **Recursion** to solve problems, as well as introduced the concept of creating an *Interpreter* in Haskell.

1.3 Week 3

During the third week of Programming Languages, the class lectures went over the concept of Context and Parse Trees in relation to programming a calculator interpreter.

1.4 Week 4

During the fourth week of Programming Languages, the lectures covered the concept of Lambda Calculus, specifically about its syntax and how to parse Lambda equations.

1.5 Week 5

During the fifth week of Programming Languages, the lectures covered the concept of performing substitution in Lambda Calculus through pen-and-paper, and through an interpreter.

2 Homework

2.1 Week 1

C++ Code for Greatest Common Denominator:

```
#include <iostream>
using namespace std;

int gcd(int a, int b)
{
    if (a == 0)
    {
        return b;
    }

    if (b == 0)
    {
        return a;
    }

    if (a == b)
    {
        return a;
    }

    if (a > b)
    {
        return gcd((a - b), b);
    }

    if (b > a)
    {
        return gcd(a, (b - a));
    }
}

int main() {
    // Write C++ code here
    cout << "GCD of 9 and 33 is : " << gcd(9, 33);
    return 0;
}
```

The function for calculating the Greatest Common Divisor (GCD) functions by first taking two integers as inputs, represented by **a** and **b**. In the case of inputs like gcd(9, 33), 9 and 33 are considered to be **a** and **b** in the gcd function respectively. The function first checks to see if either integer is *0*, and in the case of either integer being zero, returns the other integer entered in the function as the gcd. If neither number is zero, the function checks to see if both integers are **equal to each other**. In the case that both integers are equal, the function returns the first integer as the gcd. If neither of the previous cases are met, the function then compares integer **a** and **b**. If integer **a** is larger, the function recursively calls itself, and *integer a is*

replaced with (a - b). If integer **b** is larger, the function recursively calls itself, and *integer b is replaced with (b - a).* The function *continues making recursive calls* with modified numbers until a case where gcd is found.

2.2 Week 2

```
len [] = 0
len (x:xs) = 1 + len xs

select_evens [] = []
select_evens [a] = []
select_evens (x:y:list) = y:(select_evens list)

select_odds [] = []
select_odds [a] = [a]
select_odds (x:y:list) = x:(select_odds list)

member _ [] = False
member n (x:xs)
  | x == n = True
  | otherwise = member n xs

append [] list_original = list_original
append (x:list_add) list_original = x:(append list_add list_original)

revert [] = []
revert (item:xs) = append (revert xs) [item]

less_equal [] [] = True
less_equal (x:list_one) (y:list_two) = if x <= y
                            then less_equal list_one list_two
                            else False
```

In the case of **select_evens** [**"a","b","c","d","e"**]:

```
select_evens ["a","b","c","d","e"] =
     "b":(select_evens ["c","d","e"]) =
     "b":"d":(select_evens ["e"]) =
     "b":"d":[] =
     ["b","d"]
```

In the case of **select_odds** [**"a","b","c","d","e"**]:

```
select_odds ["a","b","c","d","e"] =
     "a":(select_odds ["c","d","e"]) =
     "a":"c":(select_odds ["e"]) =
     "a":"c":["e"] =
     ["a","c","e"]
```

In the case of **member 2** [**5,2,6**]:

```
member 2 [5,2,6] =
       member 2 [2,6] =       5 != 2
       True                   2 == 2
```

In the case of **member 3 [5,2,6]**:

```
member 3 [5,2,6] =
     member 3 [2,6] =       5 != 3
     member 3 [6] =         2 != 3
     member 3 [] =          6 != 3
     False
```

In the case of **append [1,2] [3,4,5]**:

```
append [1,2] [3,4,5] =
     1:(append [2] [3,4,5]) =
     1:(2:(append [] [3,4,5])) =
     1:(2:([3:4:5])) =
     [1,2,3,4,5]
```

In the case of **revert [1,2,3]**:

```
revert [1,2,3] =
     append (revert [2,3]) [1] =
     append (append (revert [3]) [2]) [1] =
     append (append (append (revert []) [3]) [2]) [1] =
     append (append (append [] [3]) [2]) [1] =
     append (append ([3]) [2]) [1] =
     append (3:(append [] ([2]))) [1] =
     append (3:([2]) [1] =
     append [3,2] [1] =
     3:(append [2] [1]) =
     3:(2:(append [] [1])) =
     3:(2:([1])) =
     [3,2,1]
```

In the case of **less_equal [1,2,3] [2,3,4]**:

```
less_equal [1,2,3] [2,3,4] =
     less_equal [2,3] [3,4] =      1 <= 2
     less_equal [3] [4] =          2 <= 3
     less_equal [] [] =            3 <= 4
     True
```

In the case of **less_equal [1,2,3] [2,3,2]**:

```
less_equal [1,2,3] [2,3,2] =
     less_equal [2,3] [3,2] =      1 <= 2
     less_equal [3] [2] =          2 <= 3
     False                         3 > 2
```

2.3 Week 3

```
hanoi 5 0 2
   hanoi 4 0 1
      hanoi 3 0 2
         hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
```

```
            move 0 1
            hanoi 1 2 1 = move 2 1
        move 0 2
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
    move 0 1
    hanoi 3 2 1
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
        move 2 1
        hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
            move 0 1
            hanoi 1 2 1 = move 2 1
move 0 2
hanoi 4 1 2
    hanoi 3 1 0
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
        move 1 0
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
    move 1 2
    hanoi 3 0 2
        hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
            move 0 1
            hanoi 1 2 1 = move 2 1
        move 0 2
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
```

Movement solution for Towers of Hanoi with n Disks = 5:

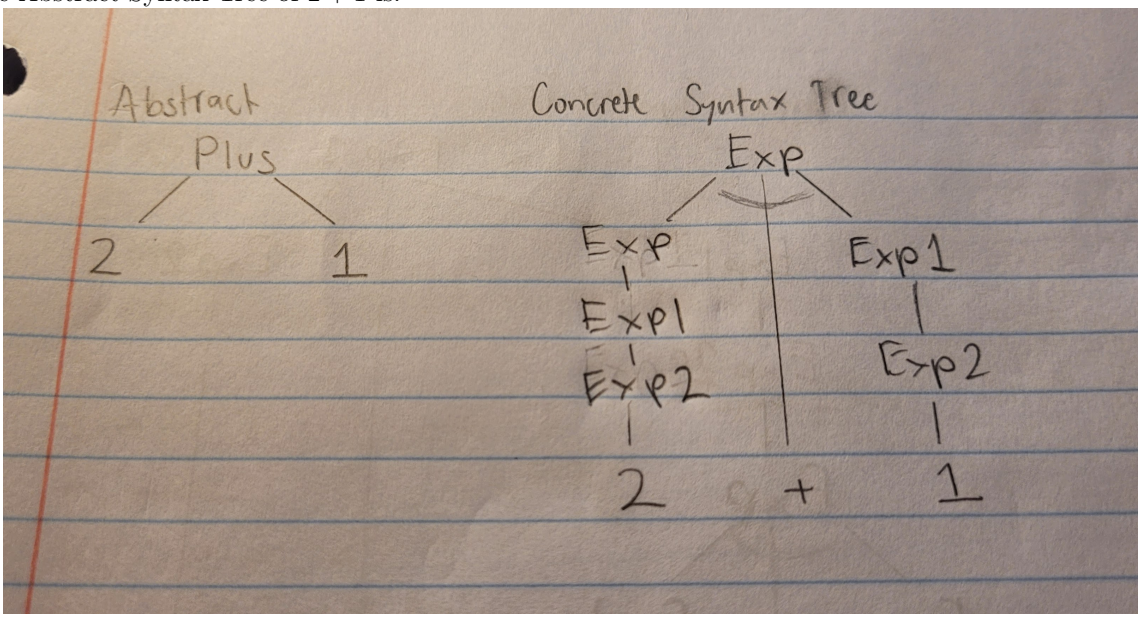
```
0->2
0->1
2->1
0->2
1->0
1->2
0->2
0->1
2->1
2->0
1->0
2->1
```

```
0->2
0->1
2->1
0->2
1->0
1->2
0->2
1->0
2->1
2->0
1->0
1->2
0->2
0->1
2->1
0->2
1->0
1->2
0->2
```

Within the computation for the Towers of Hanoi with n *Disks* $= 5$, the word "Hanoi" appears a total of **31 times**. This can be expressed with the following equation: $Hanoi = 2^N - 1$, where N is the number of disks from the initial tower.

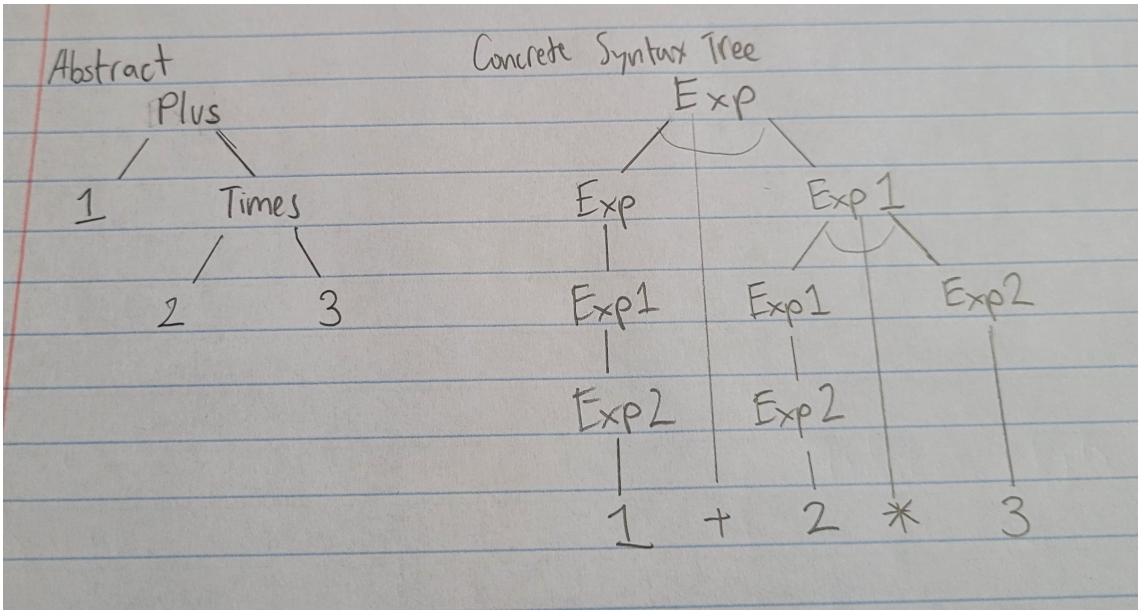
2.4 Week 4

The Abstract Syntax Tree of $2 + 1$ is:



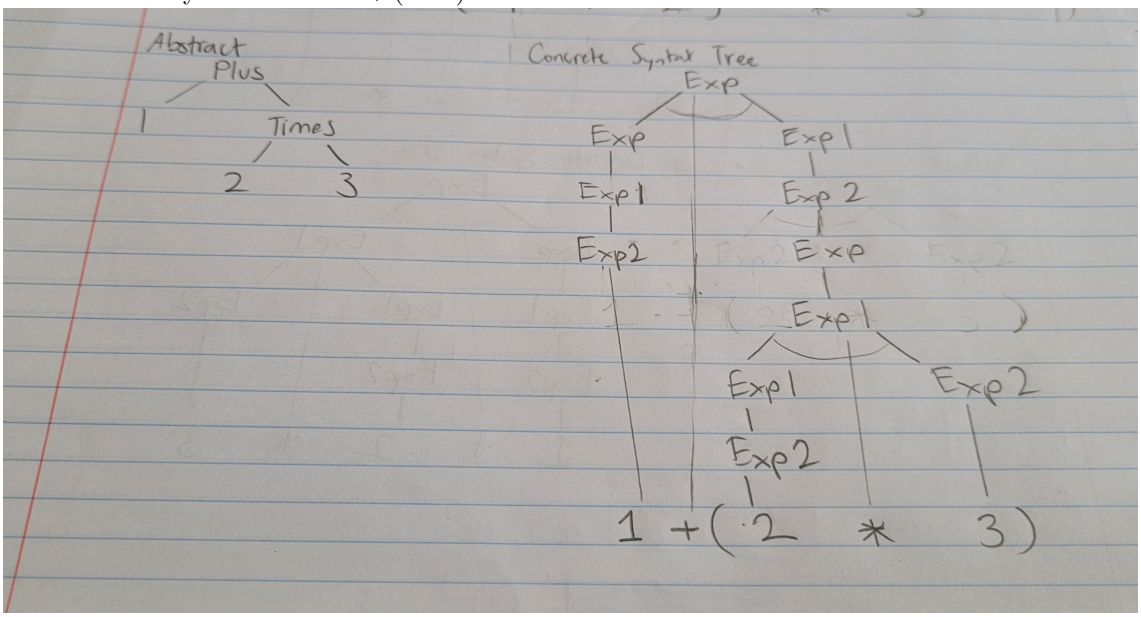
```
Plus (Num 2) (Num 1)
```

The Abstract Syntax Tree of $1 + 2 * 3$ is:



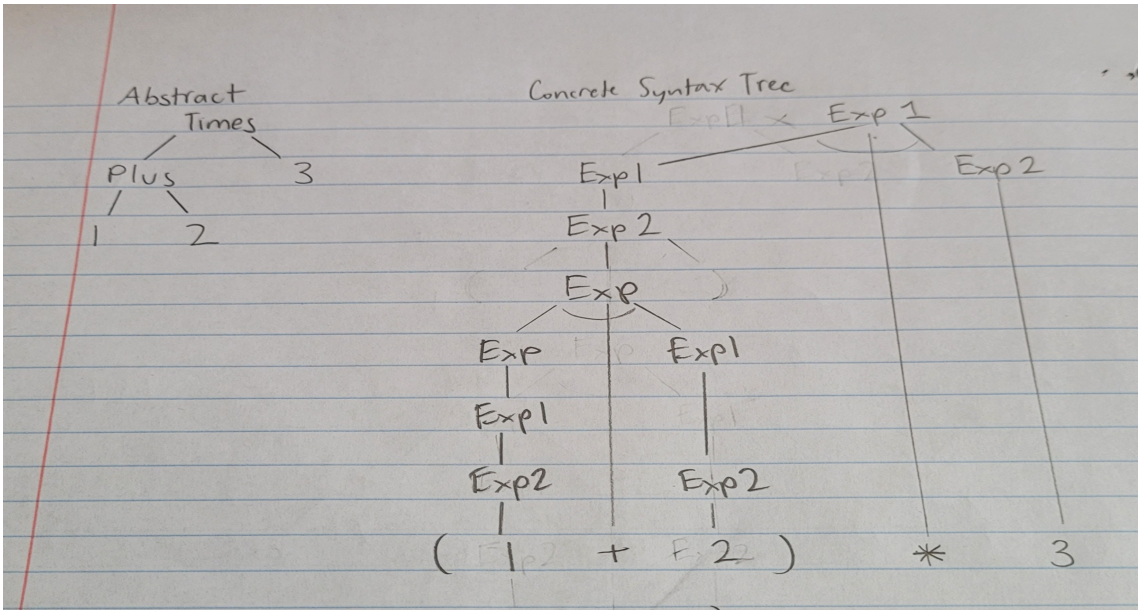

```
Plus (Num 1) (Times (Num 2) (Num 3))
```

The Abstract Syntax Tree of $1 + (2 * 3)$ is:



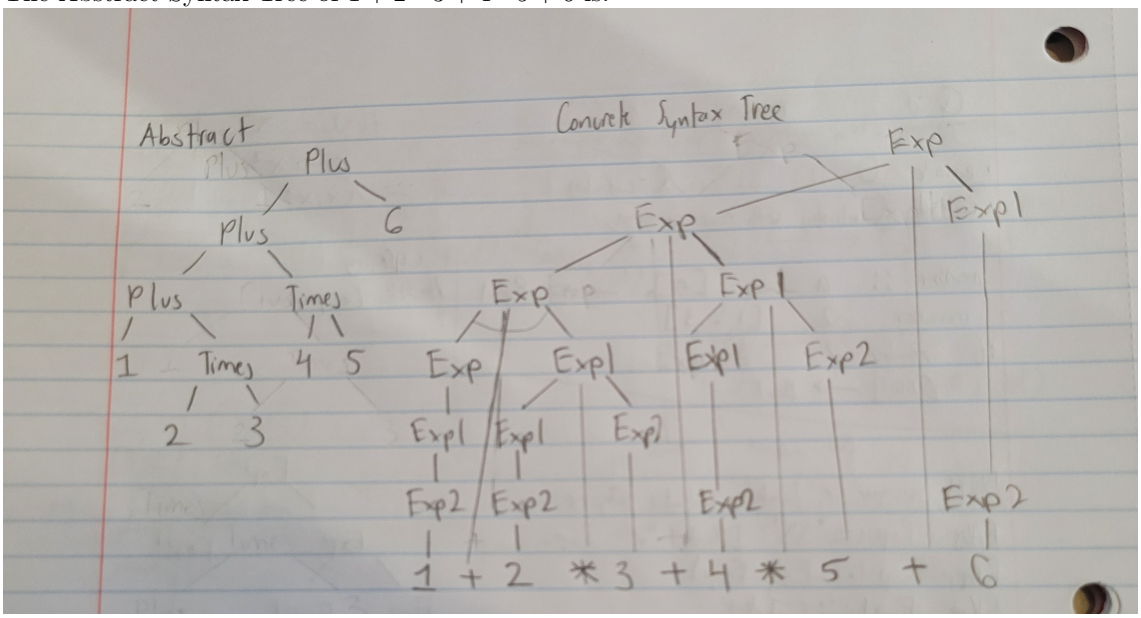

```
Plus (Num 1) (Times (Num 2) (Num 3))
```

The Abstract Syntax Tree of $(1 + 2) * 3$ is:




```
Times (Plus (Num 1) (Num 2)) (Num 3)
```

The Abstract Syntax Tree of $1 + 2 * 3 + 4 * 5 + 6$ is:




```
Plus (Plus (Plus (Num 1) (Times (Num 2) (Num 3))) (Times (Num 4) (Num 5))) (Num 6)
```

The abstract syntax tree of $1 + 2 + 3$ would be identical to the one of $(1 + 2) + 3$ as the compiler would calculate from left to right as there are no other higher-order operations aside from addition.

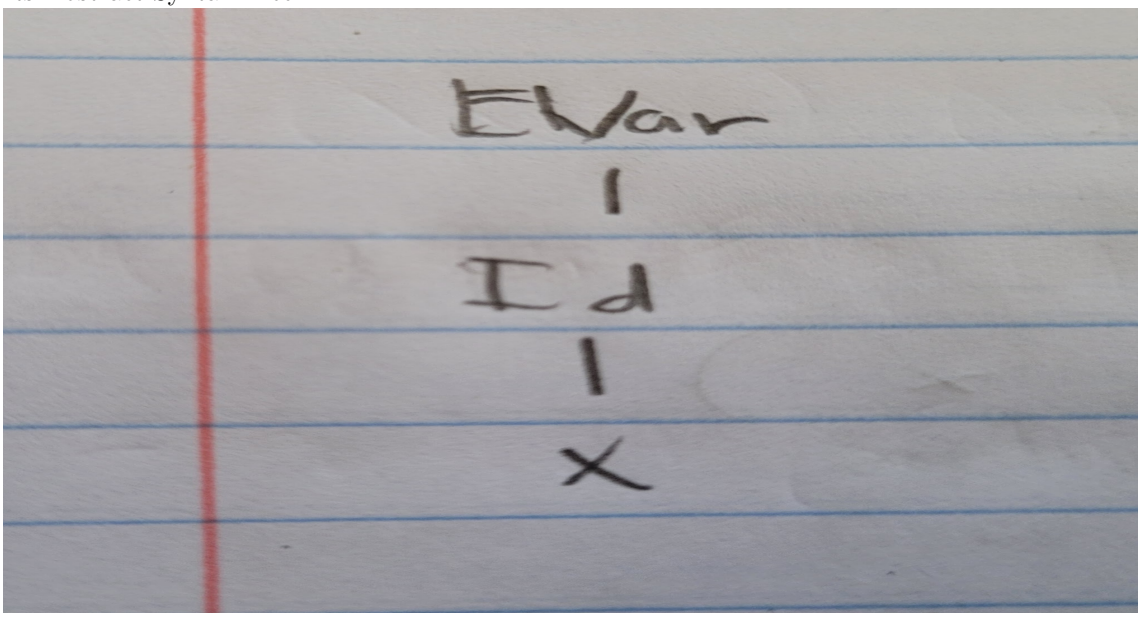
2.5 Week 5

Part 1:

The Linearized Tree for x is:

```
x
```

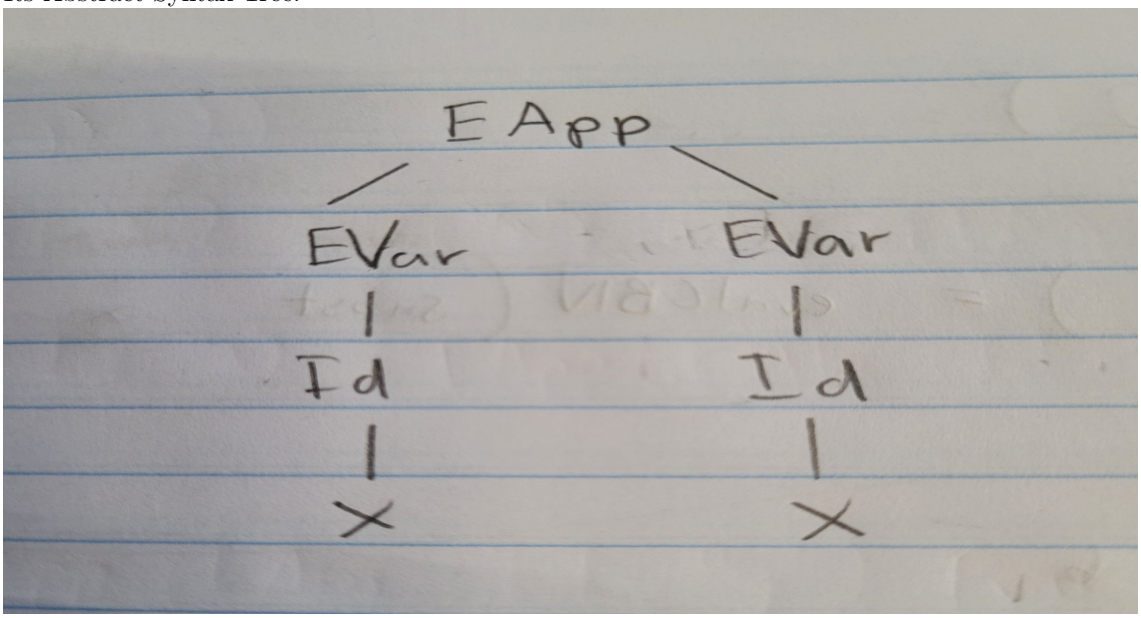
Its Abstract Syntax Tree:



The Linearized Tree for x x is:

```
x x
```

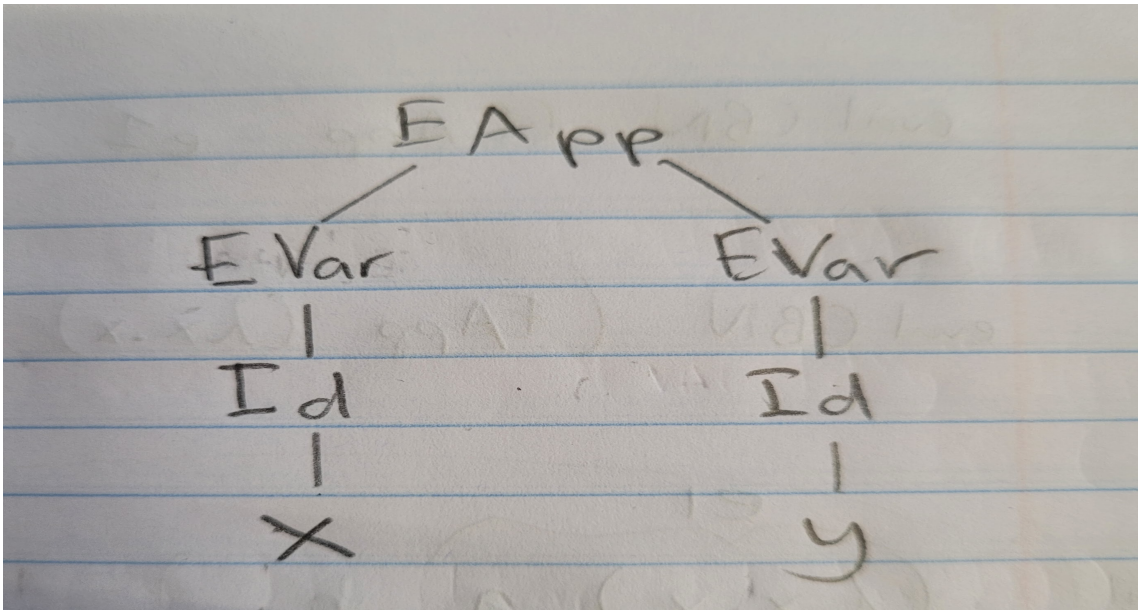
Its Abstract Syntax Tree:



The Linearized Tree for x y is:

```
x y
```

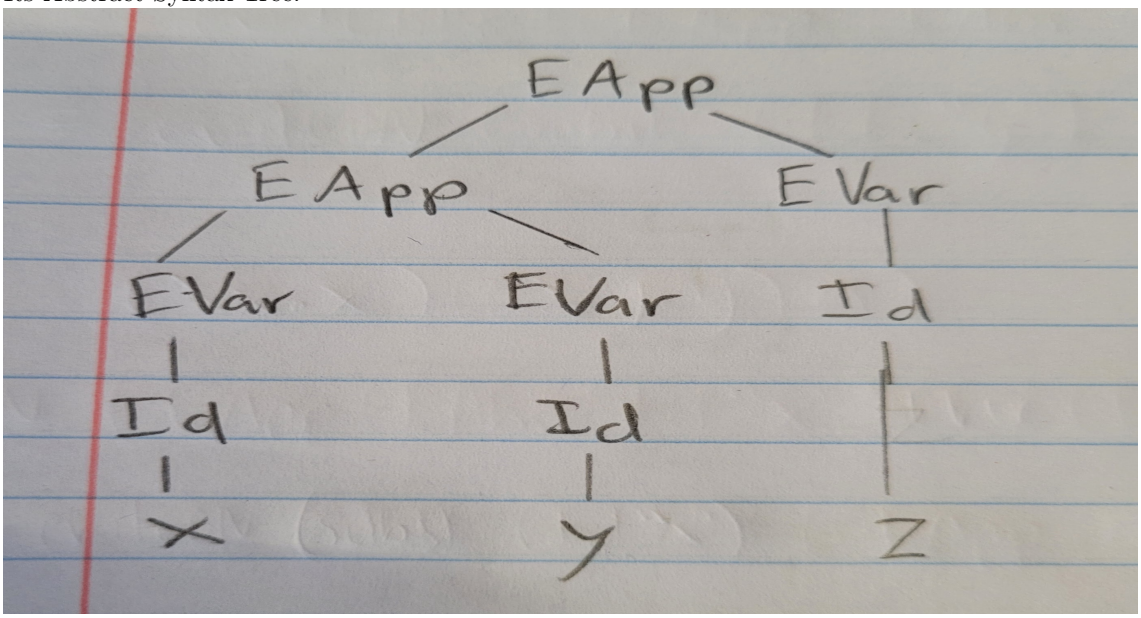
Its Abstract Syntax Tree:



The Linearized Tree for x y z is:

```
x y z
```

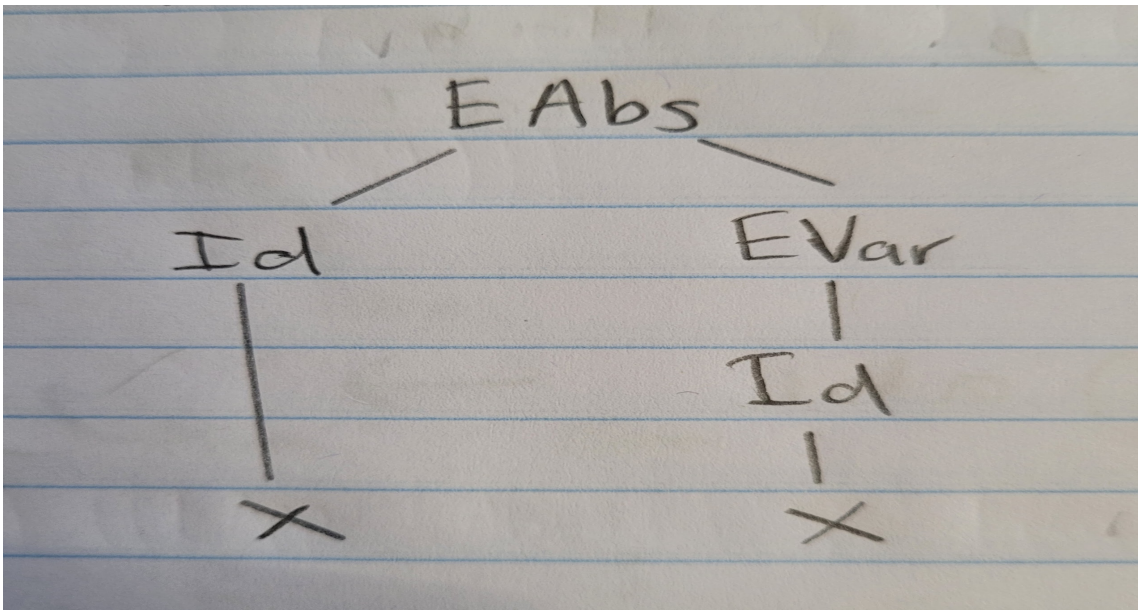
Its Abstract Syntax Tree:



The Linearized Tree for \x.x is:

```
\ x . x
```

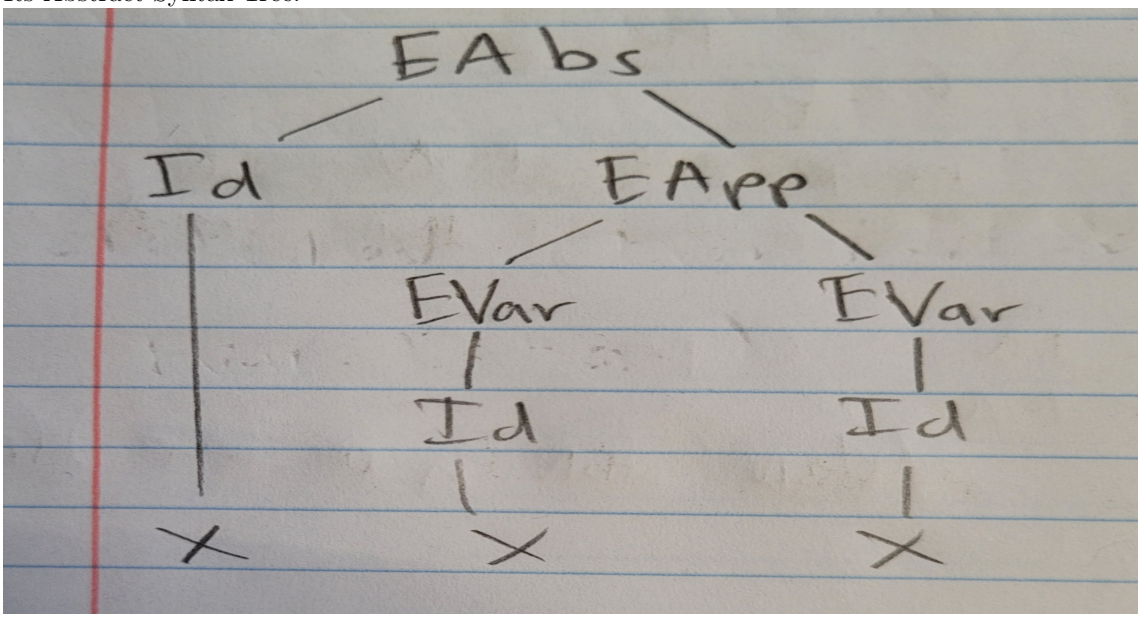
Its Abstract Syntax Tree:



The Linearized Tree for \x.x x is:

```
\ x . x x
```

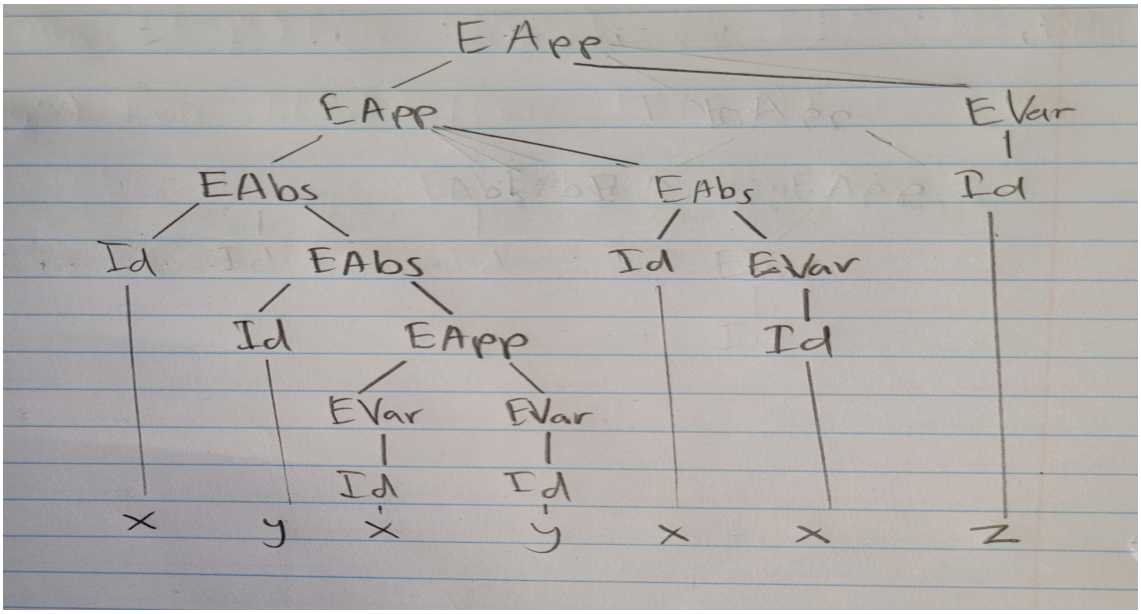
Its Abstract Syntax Tree:



The Linearized Tree for (\x . (\y . x y)) (\x.x) z is:

```
\ x . \ y . x y (\ x . x)z
```

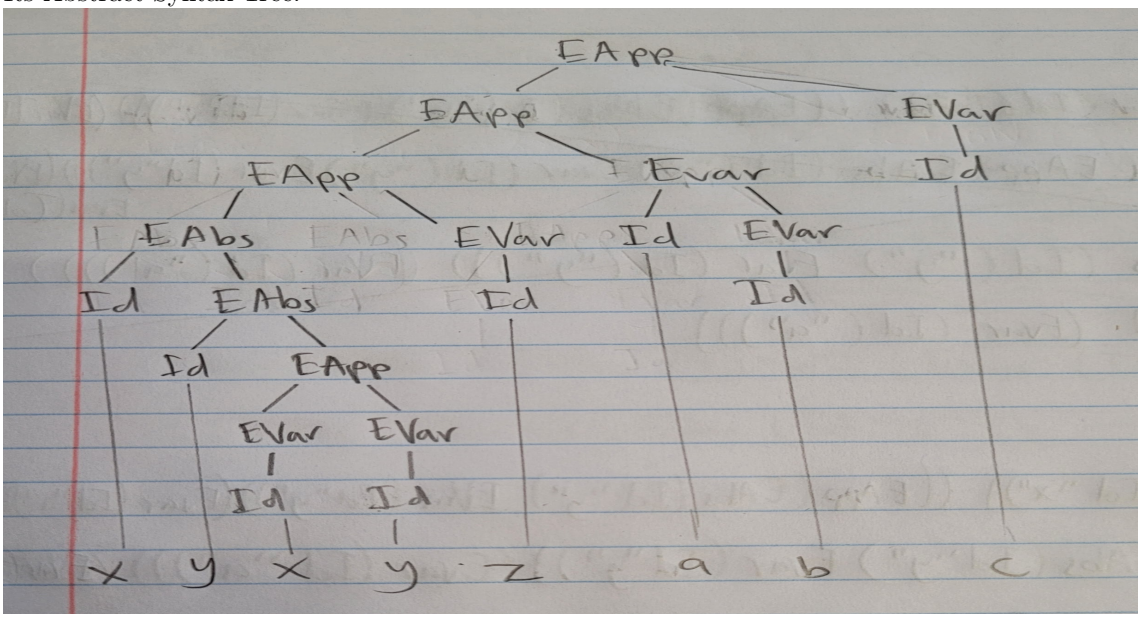
Its Abstract Syntax Tree:



The Linearized Tree for (\x . \y . x y z) a b c is:

```
\ x . \ y . x y z a b c
```

Its Abstract Syntax Tree:



Part 2:
Evaluating of Equations:

```
(\x.x) a = a
\x.x a = a
(\x.\y.x) a b = (\y. a) b = a
(\x.\y.y) a b = (\y. y) b = b
(\x.\y.x) a b c = (\y. a) b c = (a) c = a
(\x.\y.y) a b c = (\y. y) b c = (b) c = b
```

```
(\x.\y.x) a (b c) = (\y. a) (b c) = a
(\x.\y.y) a (b c) = (\y. y) (b c) = b c
(\x.\y.x) (a b) c = (\y. a b) c = a b
(\x.\y.y) (a b) c = (\y. y) c = c
(\x.\y.x) (a b c) = (\y. a b c)
(\x.\y.y) (a b c) = (\y. y)
```