CPSC-354 Report

Marc Domingo
Chapman University

October 13, 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:

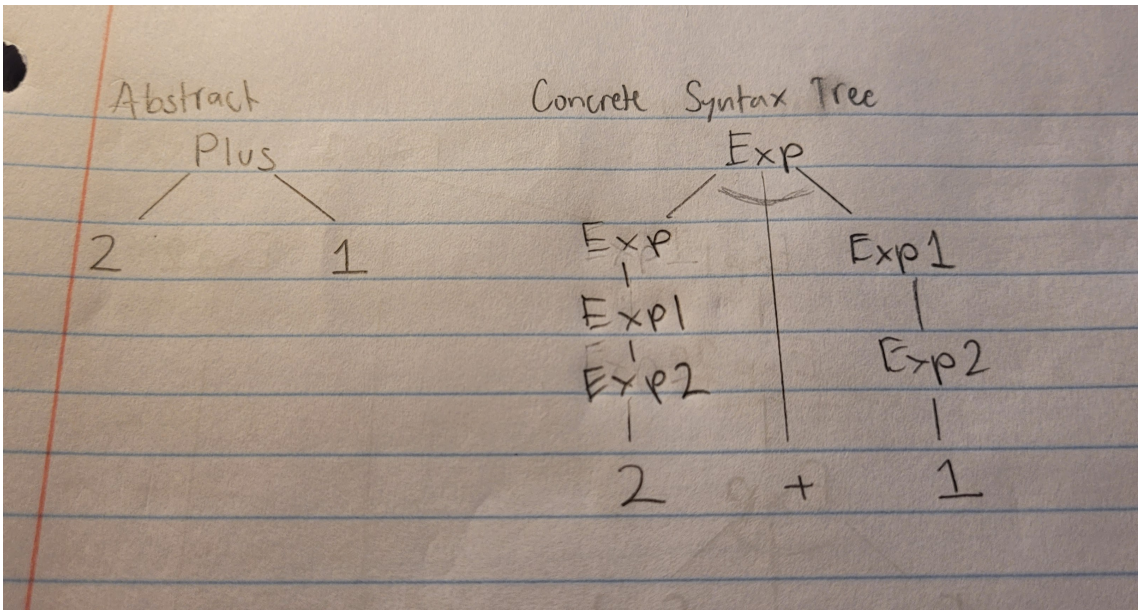
```
0->2
```

```
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.

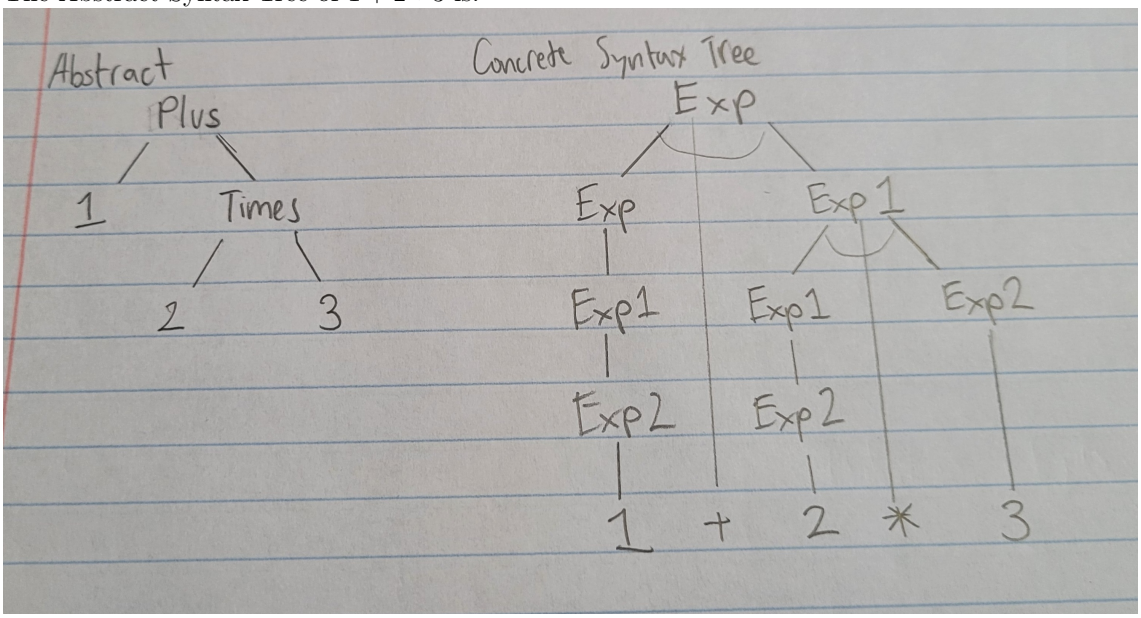
2.4 Week 4

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



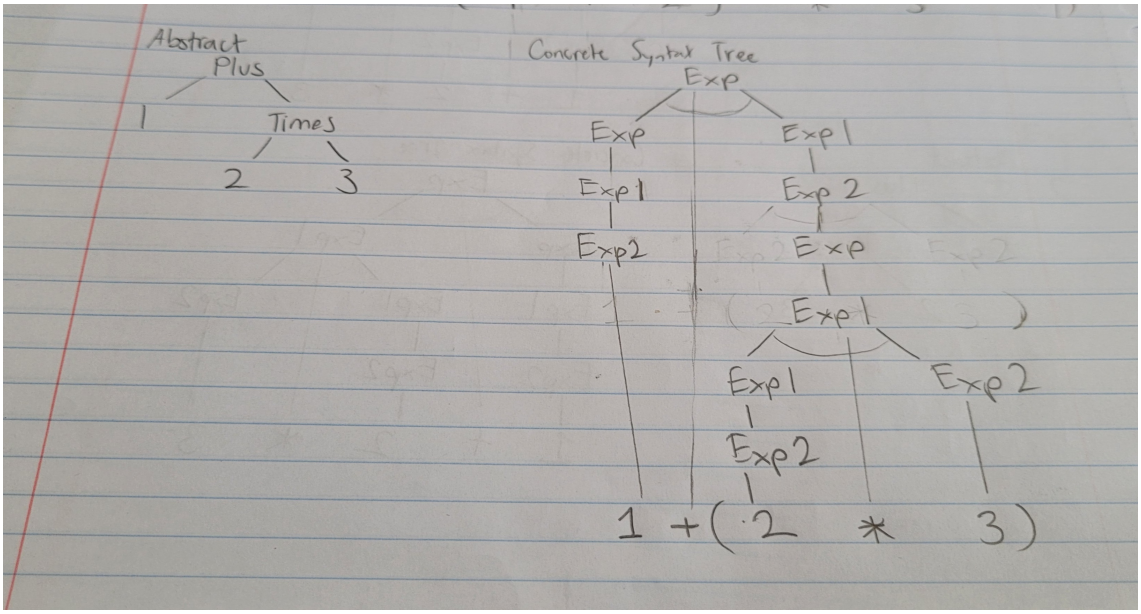

```
Plus (Num 2) (Num 1)
```

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



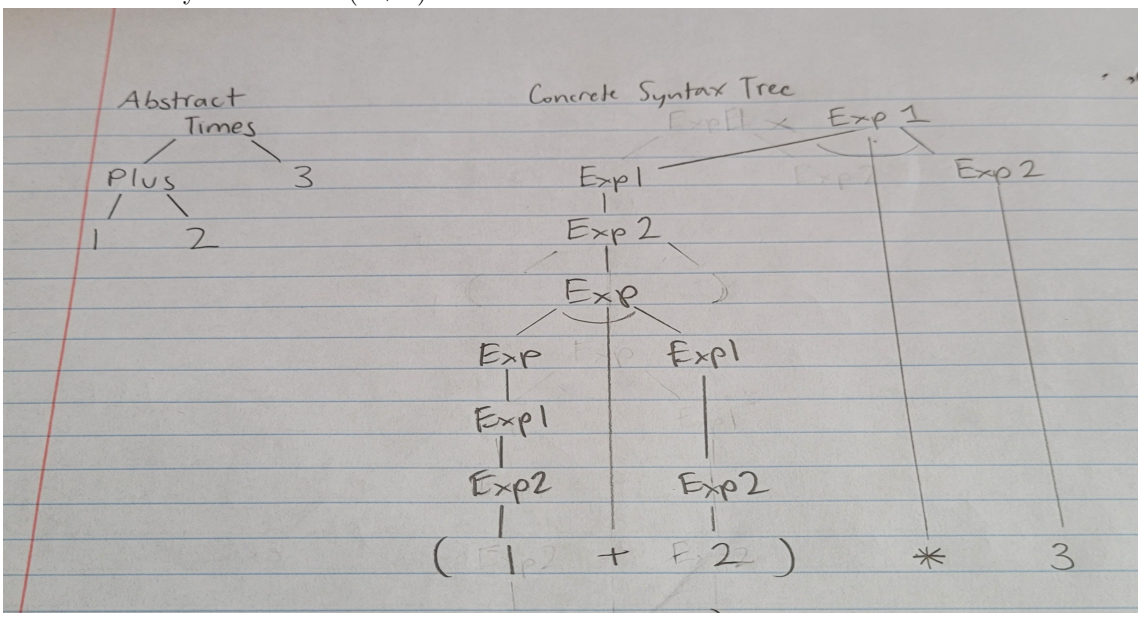

```
Plus (Num 1) (Times (Num 2) (Num 3))
```

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



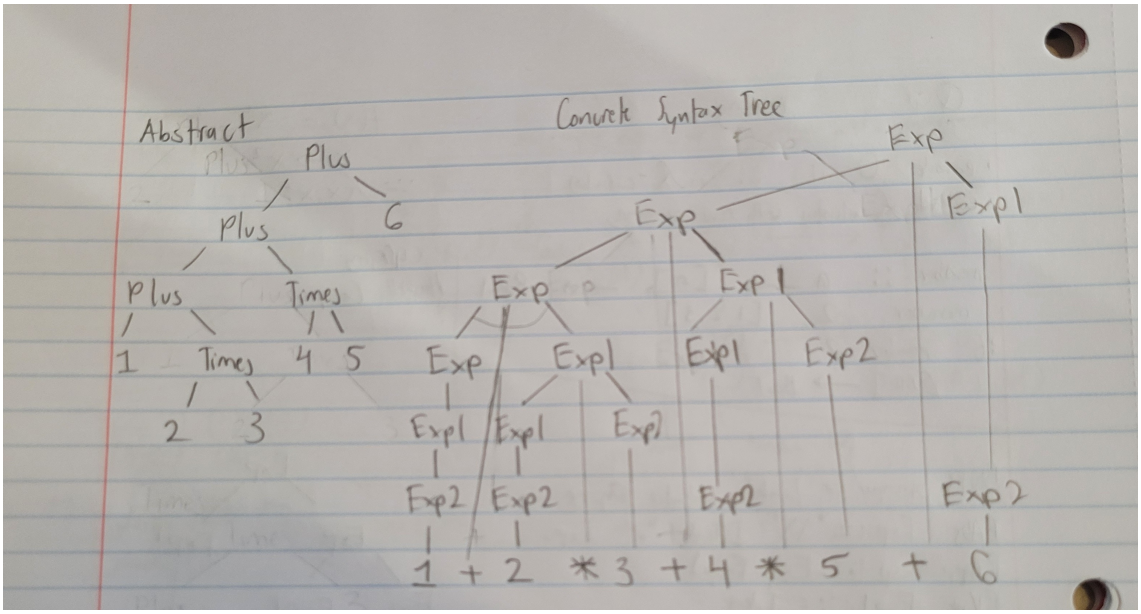

```
Plus (Num 1) (Times (Num 2) (Num 3))
```

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.

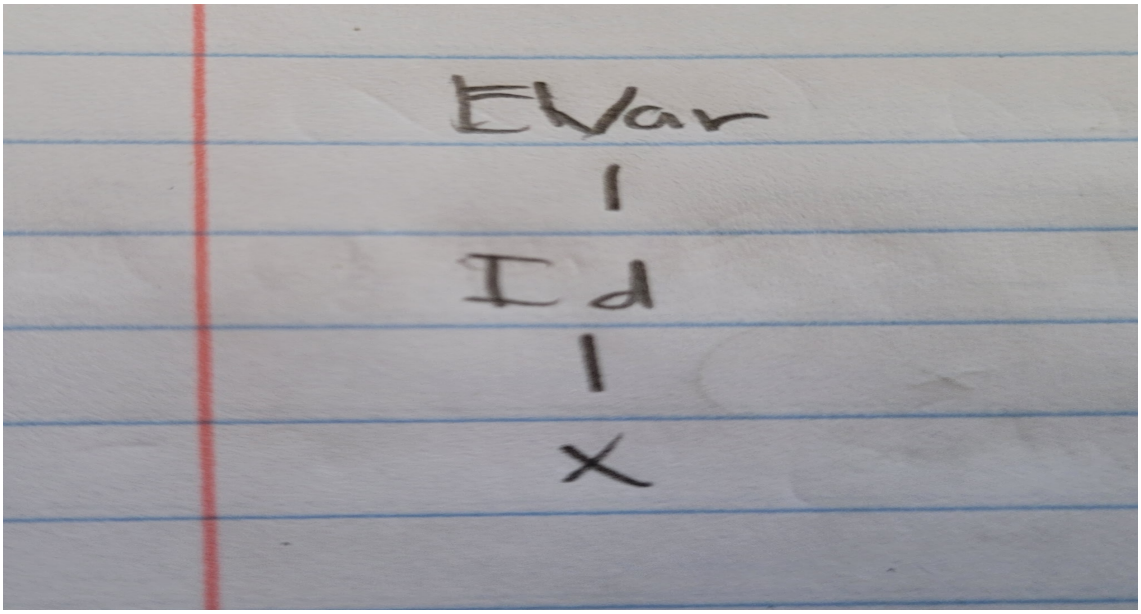
2.5 Week 5

Part 1:

The Linearized Tree for x is:

```
x
Prog (EVar (Id "x"))
```

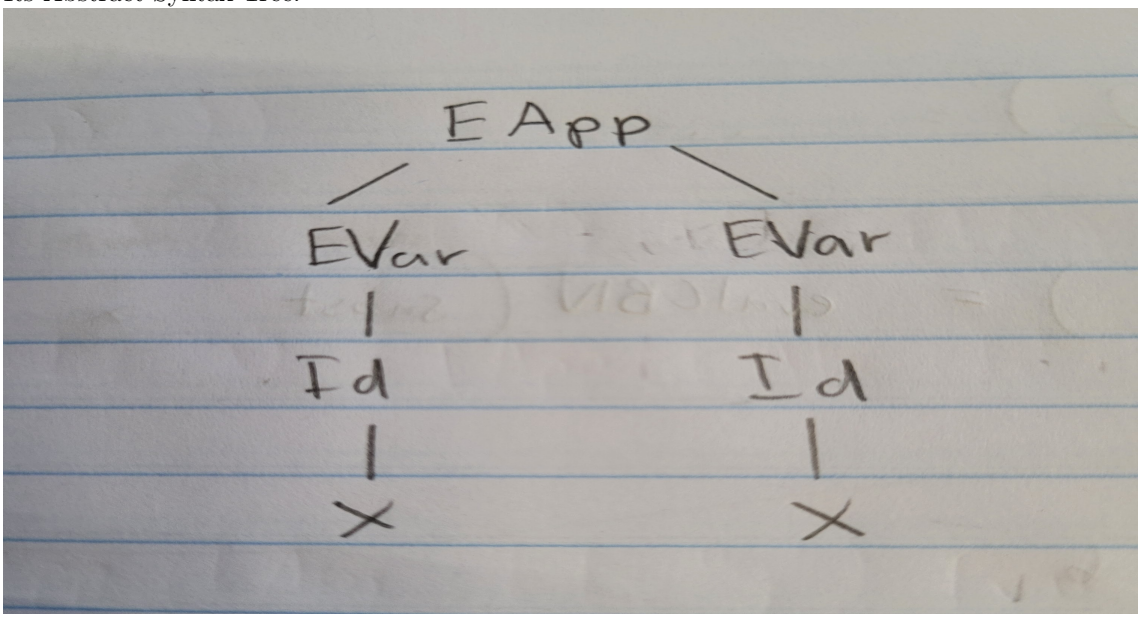
Its Abstract Syntax Tree:



The Linearized Tree for x x is:

```
x x
Prog (EApp (EVar (Id "x")) (EVar (Id "x")))
```

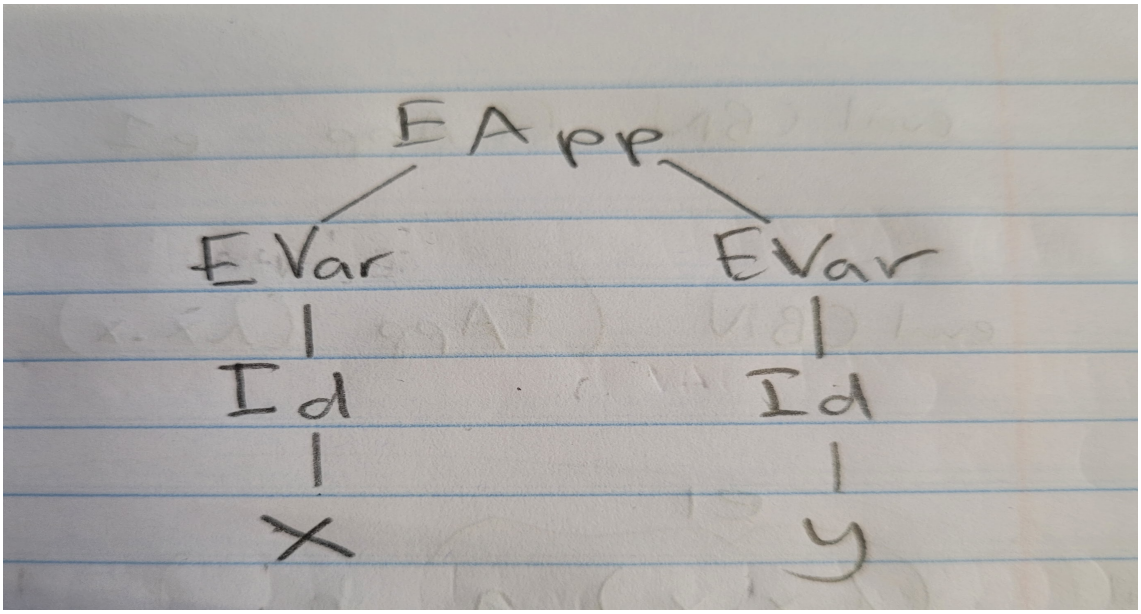
Its Abstract Syntax Tree:



The Linearized Tree for x y is:

```
x y
Prog (EApp (EVar (Id "x")) (EVar (Id "y")))
```

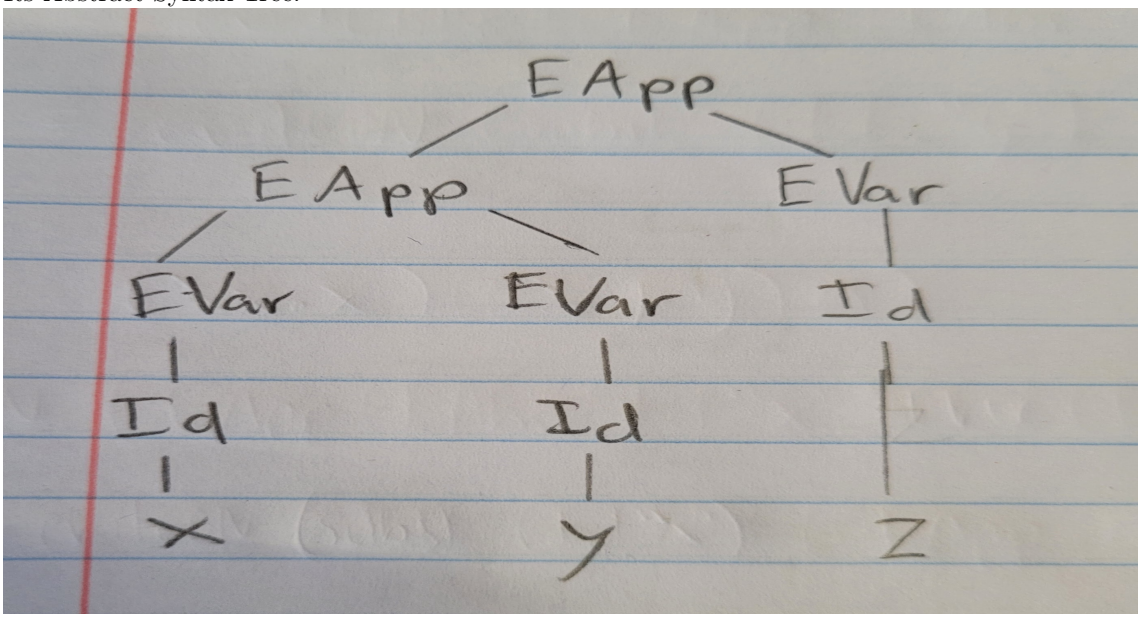
Its Abstract Syntax Tree:



The Linearized Tree for x y z is:

```
x y z
Prog (EApp (EApp (EVar (Id "x")) (EVar (Id "y"))) (EVar (Id "z")))
```

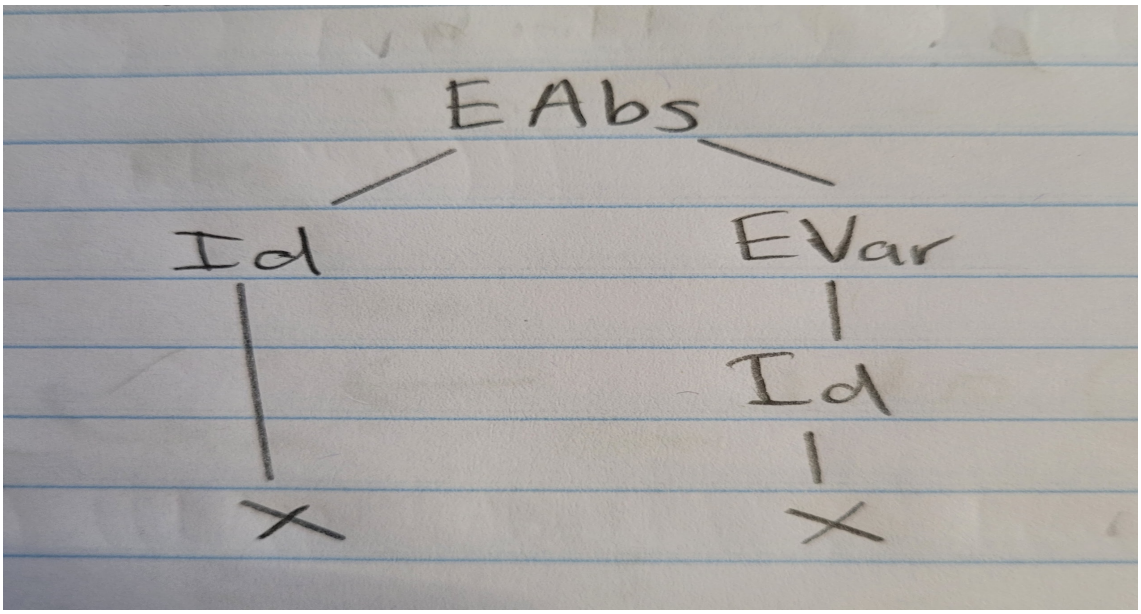
Its Abstract Syntax Tree:



The Linearized Tree for \x.x is:

```
\ x . x
Prog (EAbs (Id "x") (EVar (Id "x")))
```

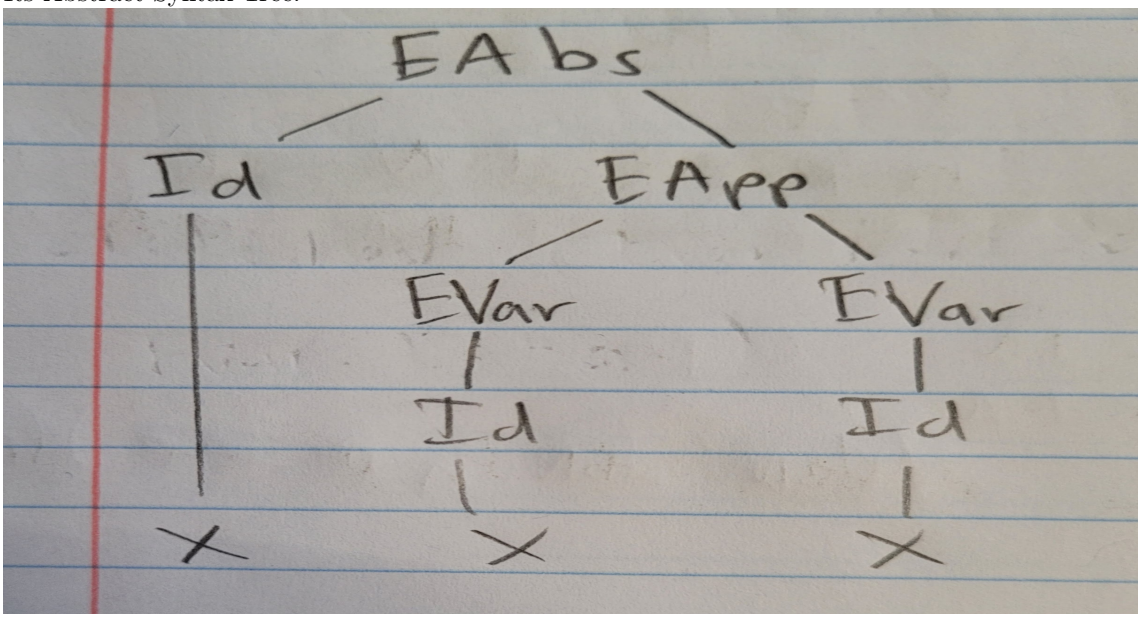
Its Abstract Syntax Tree:



The Linearized Tree for \x.x x is:

```
\ x . x x
Prog (EAbs (Id "x") (EApp (EVar (Id "x")) (EVar (Id "x"))))
```

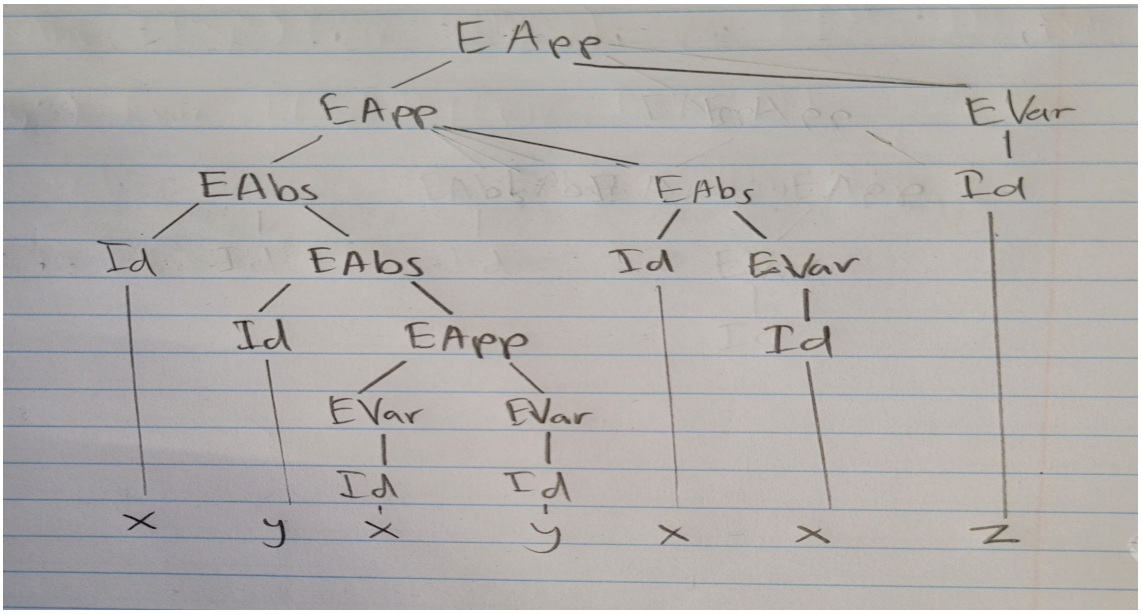
Its Abstract Syntax Tree:



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

```
\ x . \ y . x y (\ x . x)z
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EVar (Id "x")) (EVar (Id "y"))))) (EAbs (Id
    "x") (EVar (Id "x")))) (EVar (Id "z")))
```

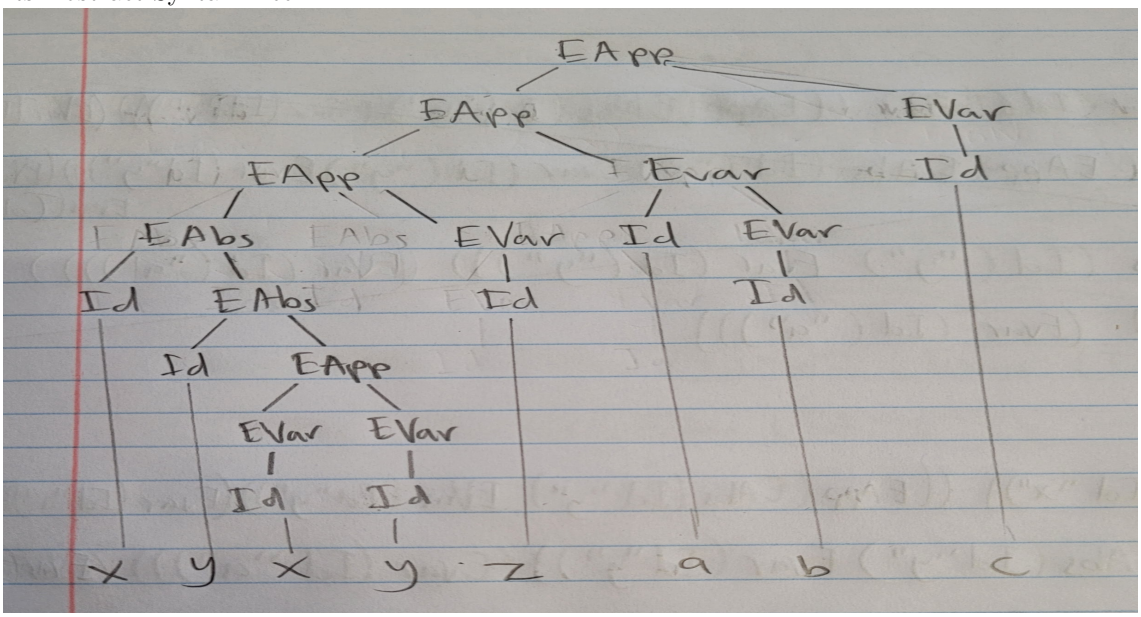
Its Abstract Syntax Tree:



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

```
\ x . \ y . x y z a b c
Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EApp (EVar (Id "x")) (EVar (Id "y")))
    (EVar (Id "z"))))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))
```

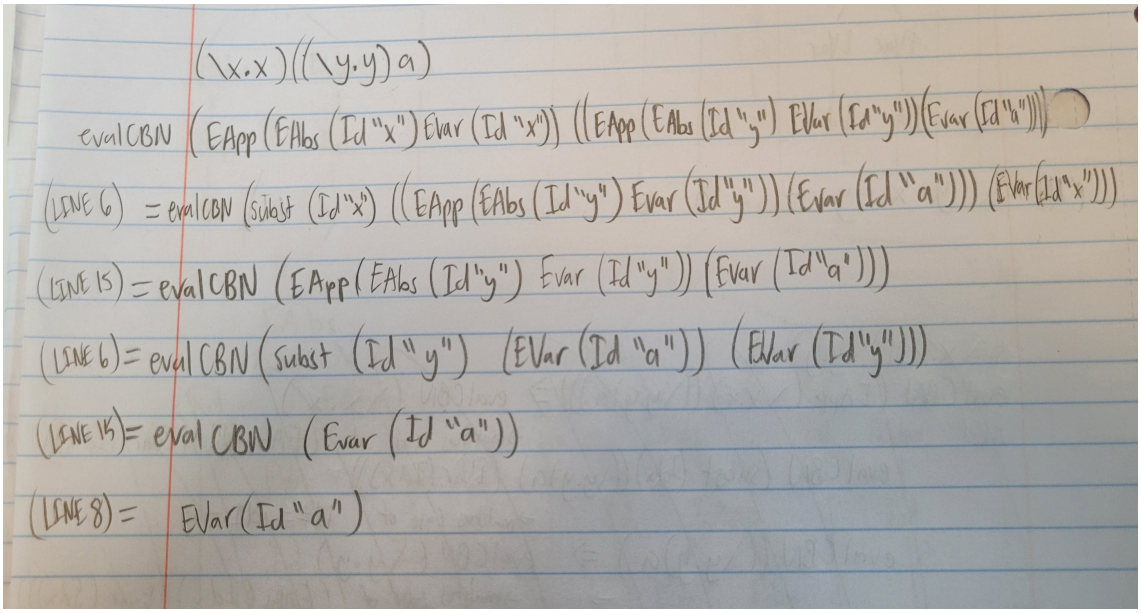
Its Abstract Syntax Tree:



Part 2:
Evaluating of Equations:

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



2.6 Week 6

```
(\exp . \two . \three . exp two three)
(\m.\n. m n)
(\f.\x. f (f x))
(\f.\x. f (f (f x)))

=
((\m.\n. m n) (\f2.\x2. f2 (f2 x2)) (\f3.\x3. f3 (f3 (f3 x3))))
=
((\n. (\f2.\x2. f2 (f2 x2)) n) (\f3.\x3. f3 (f3 (f3 x3))))
=
((\f2.\x2. f2 (f2 x2)) (\f3.\x3. f3 (f3 (f3 x3))))
=
(\x2. (\f3.\x3. f3 (f3 (f3 x3))) ((\f3.\x3. f3 (f3 (f3 x3))) x2))
=
(\x2. (\f3.\x3. f3 (f3 (f3 x3))) (\x3. x2 (x2 (x2 x3))))
=
(\x2. (\x3. (\x3. x2 (x2 (x2 x3))) ((\x3. x2 (x2 (x2 x3))) ((\x3. x2 (x2 (x2 x3))) x3))))
=
(\x2. (\x3. (\x3. x2 (x2 (x2 x3))) ((\x3. x2 (x2 (x2 x3))) ((x2 (x2 (x2 x3)))))))
=
(\x2. (\x3. (\x3. x2 (x2 (x2 x3))) ((x2 (x2 (x2 (x2 (x2 (x2 x3))))))))))
=
```

```
(\x2. (\x3. (x2 (x2 (x2 (x2 (x2 (x2 (x2 (x2 (x2 x3)))))))))))
```

2.7 Week 7

For lines 5-7 as shown through:

```
evalCBN (EApp e1 e2) = case (evalCBN e1) of
    (EAbs i e3) -> evalCBN (subst i e2 e3)
    e3 -> EApp e3 e2
```

The variables e1, e2, i, and e3 are all bound variables. In the case of e1, the binder is *evalCBN (EApp e1 e2)*, while its scope is *case (evalCBN e1)*. In the case of e2, it shares the same binder as e1 with *evalCBN (EApp e1 e2)*, but has two scopes– *evalCBN (subst i e2 e3)* and *EApp e3 e2*. Meanwhile, the binder for i is *(EAbs i e3* and its scope is *evalCBN (subst i e2 e3)*. Lastly, e3 is binded by *(EAbs i e3)*, but has two scopes– *evalCBN (subst i e2 e3)* and *EApp e3 e2*.

For lines 18-22 as shown through:

```
subst id s (EAbs id1 e1) =
    -- to avoid variable capture, we first substitute id1 with a fresh name inside the body of the
        Lambda-abstraction, obtaining e2. Only then do we proceed to apply substitution of the
        original s for id in the body e2.
    let f = fresh (EAbs id1 e1)
        e2 = subst id1 (EVar f) e1 in
        EAbs f (subst id s e2)
```

The variables f and e2 are both free variables as they are derived from the code block itself, while the variables id, s id1, and e1 are bound variables. In the case of id, it is bound by *subst id s (EAbs id1 e1)* and its scope is *e2 = subst id1 (EVar f) e1 in EAbs f (subst id s e2)*. In the case of s, it is bound by *subst id s (EAbs id1 e1)* and its scope is *e2 = subst id1 (EVar f) e1 in EAbs f (subst id s e2)*. In the case of id1, it is bound by *subst id s (EAbs id1 e1)* with the scope of *let f = fresh (EAbs id1 e1)*. Lastly, e1 is binded by *subst id s (EAbs id1 e1)* in the scope of *let f = fresh (EAbs id1 e1)* and *e2 = subst id1 (EVar f) e1 in EAbs f (subst id s e2)*.

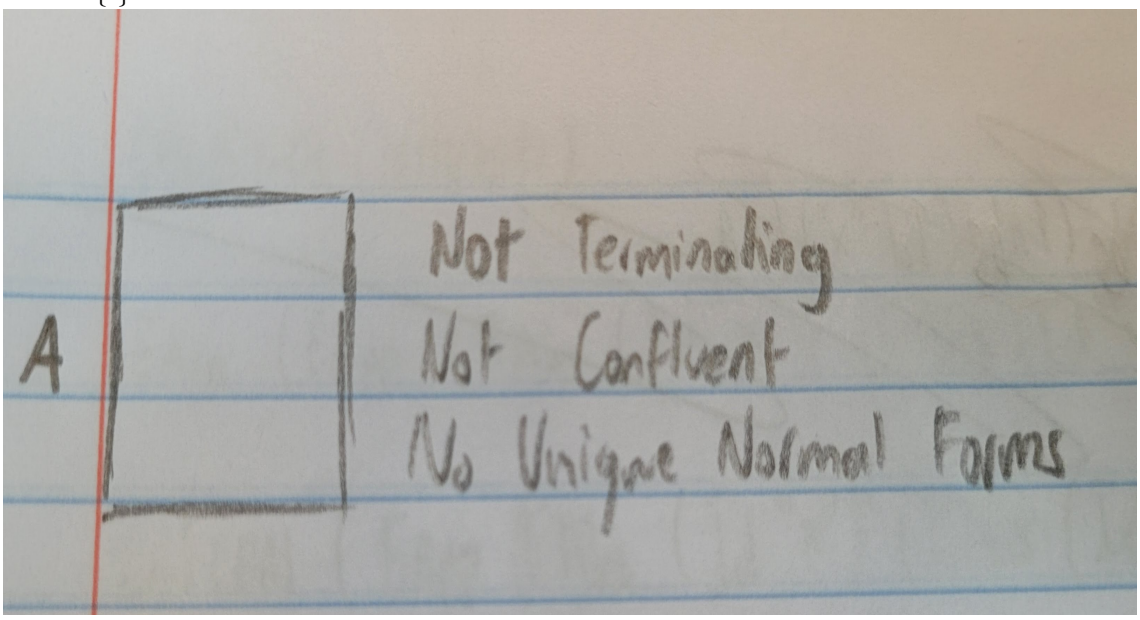
Item 3:

```
--Idea:
(\x.\y.x) y z = (\fresh.y) z = y

(EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "y"))) (EVar (Id "z")))

-- In first EApp, (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "y"))) is e1,
    (EVar (Id "z")) is e2
-- In second EApp, (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) is e1, (EVar (Id "y")) is e2

= -- Line 5
evalCBN ( ( evalCBN (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "y"))) ) (EVar
    (Id "z")) )

    -- inner evalCBN
    -- Line 6
    evalCBN (subst ((EAbs (Id "x")) (EVar (Id "y")) (EAbs (Id "y") (EVar (Id "x")))))

        -- Line 18
        -- Line 20
        f = fresh ((EAbs (Id "y") (EVar (Id "x"))))
```

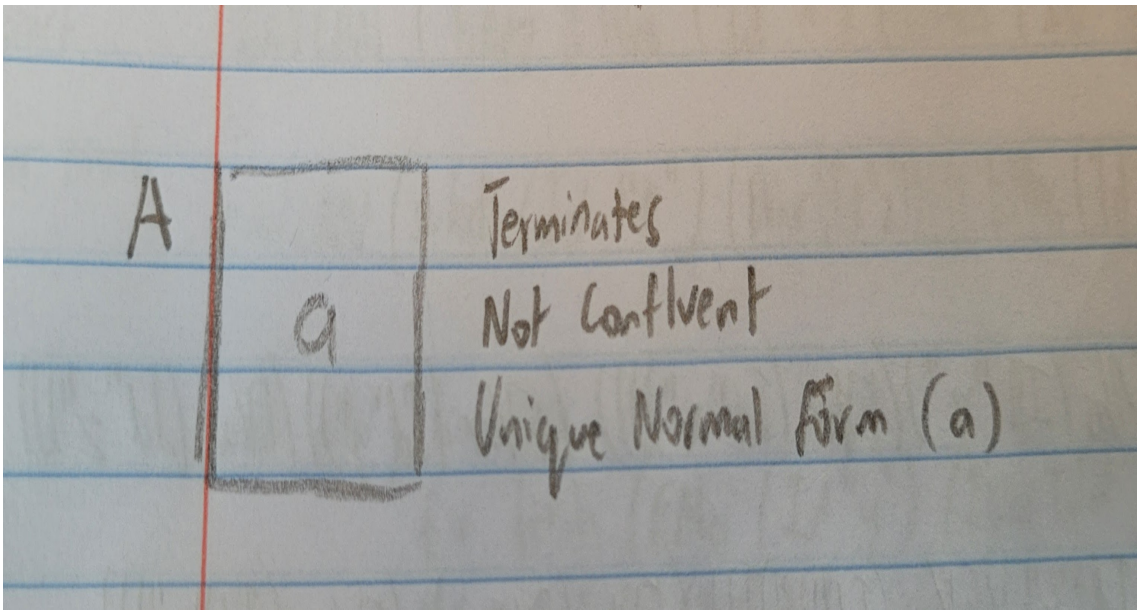
```
        -- consider \y now \y0
        -- Line 21
        e2 = subst (Id "y") (EVar "y0") (EVar (Id "x"))
        -- Line 16
        EAbs (Id "y0") (subst (Id "y") (EVar (Id "y")) (EVar (Id "x")))
        -- Line 16

    evalCBN (EAbs (Id "y0") (EVar (Id "y")))

= -- Line 7
evalCBN (EApp (EAbs (Id "y0")) (EVar (Id "y"))) (EVar (Id "z"))
= -- Line 6
evalCBN (subst (Id "y0") (EVar (Id "z")) (EVar (Id "y")))
= -- Line 16
evalCBN (EVar (Id "y"))
= -- Line 8
EVar (Id "y")
```

---

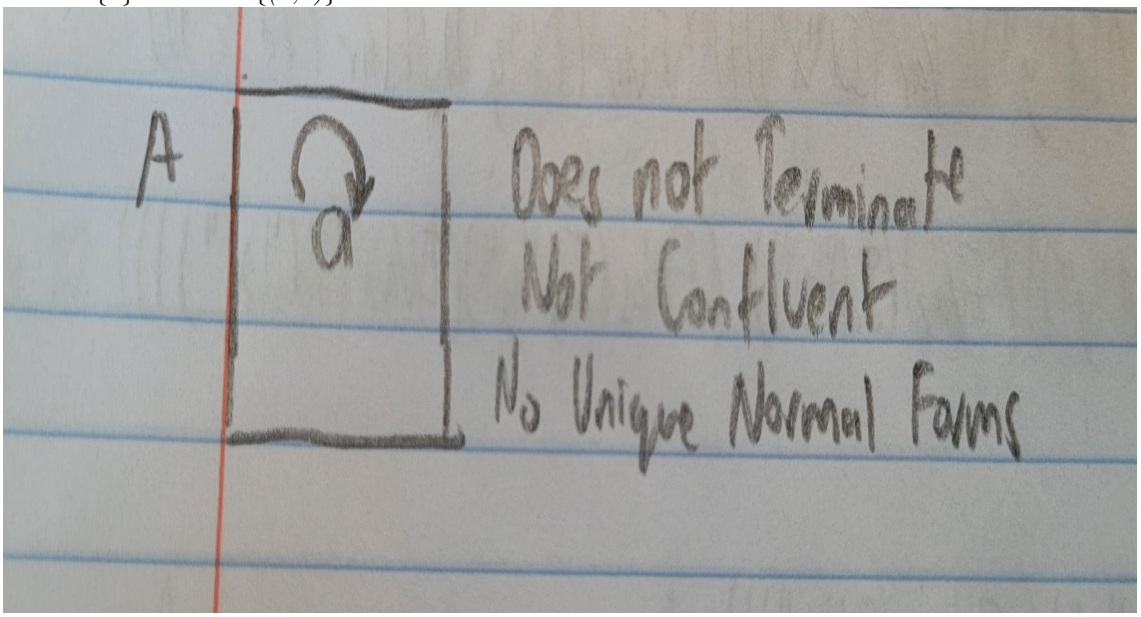
1. $A = \{\ \}$



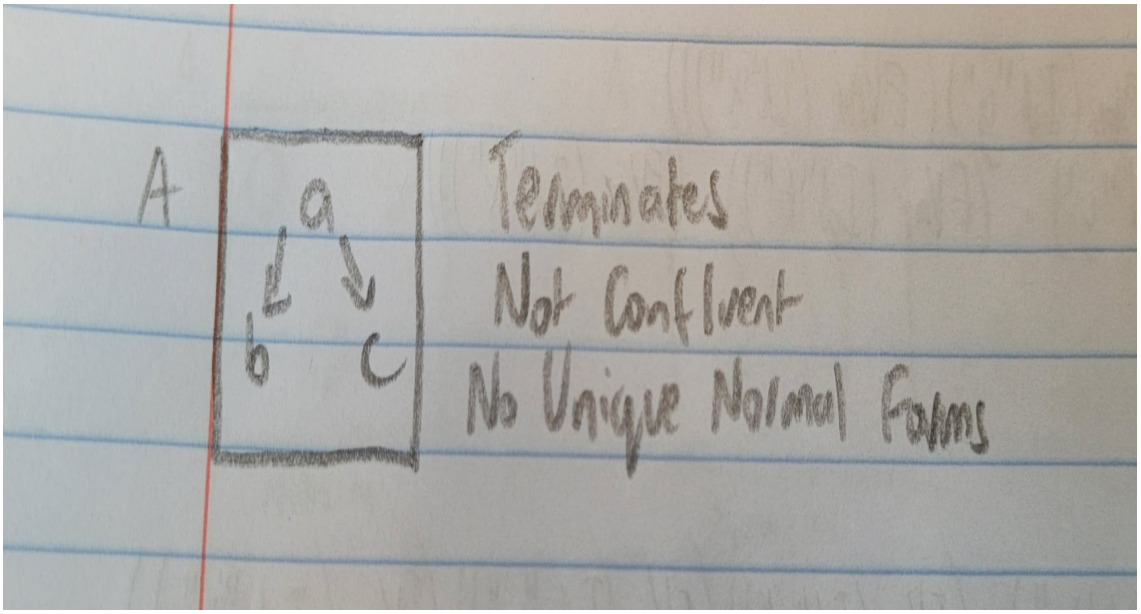
2. $A = \{a\}$ and $R = \{\}$



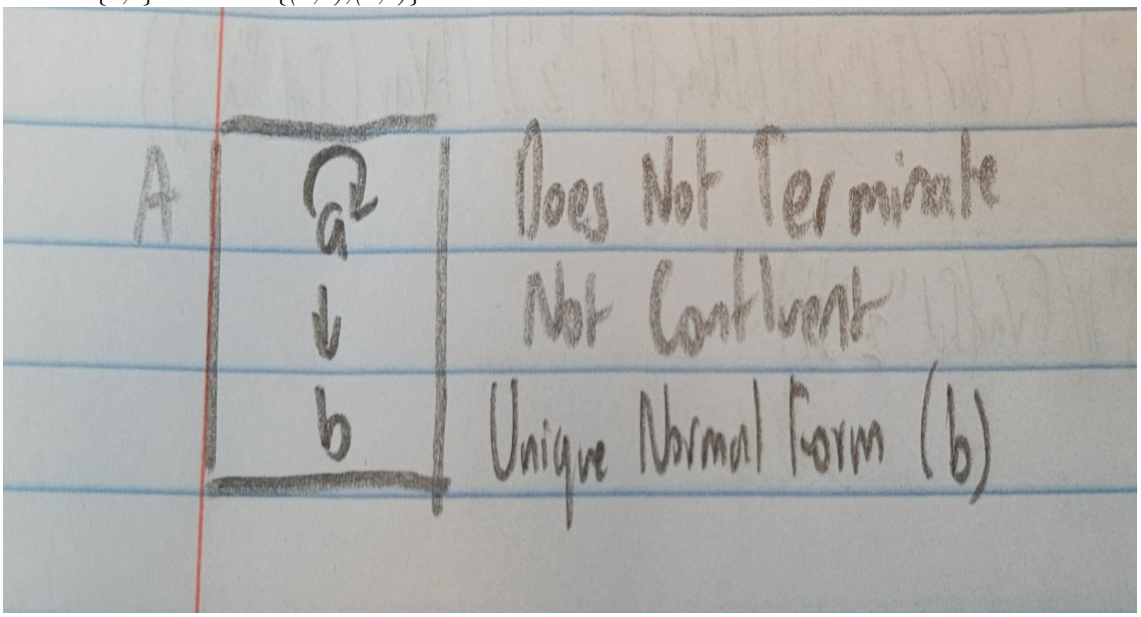
3. $A = \{a\}$ and $R = \{(a,a)\}$



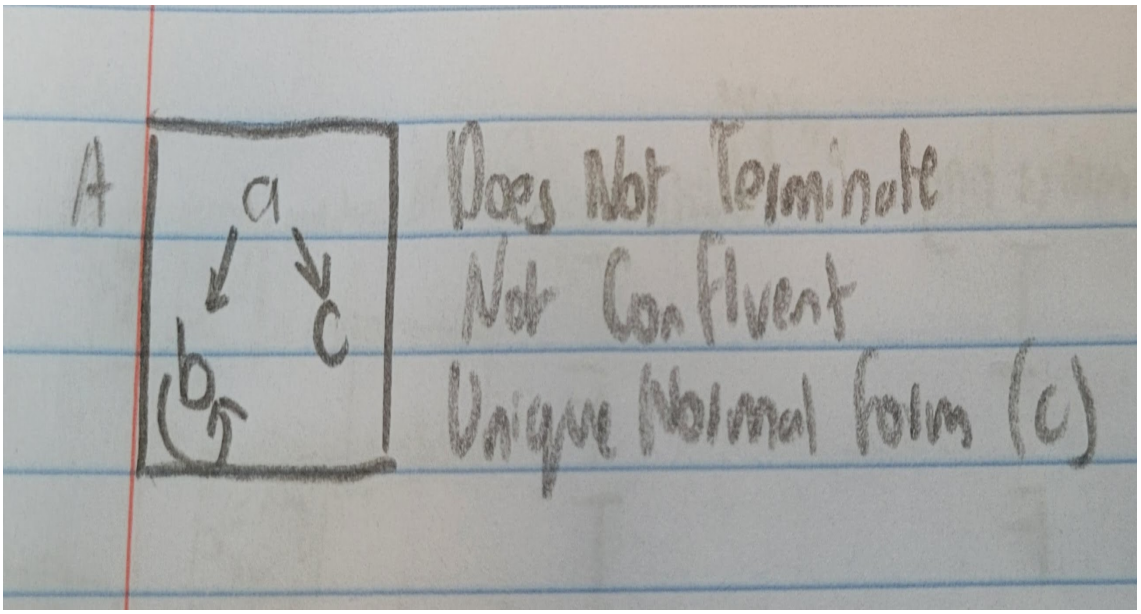
4. $A = \{a,\ b,\ c\}$ and $R = \{(a,b),\ (a,c)\}$



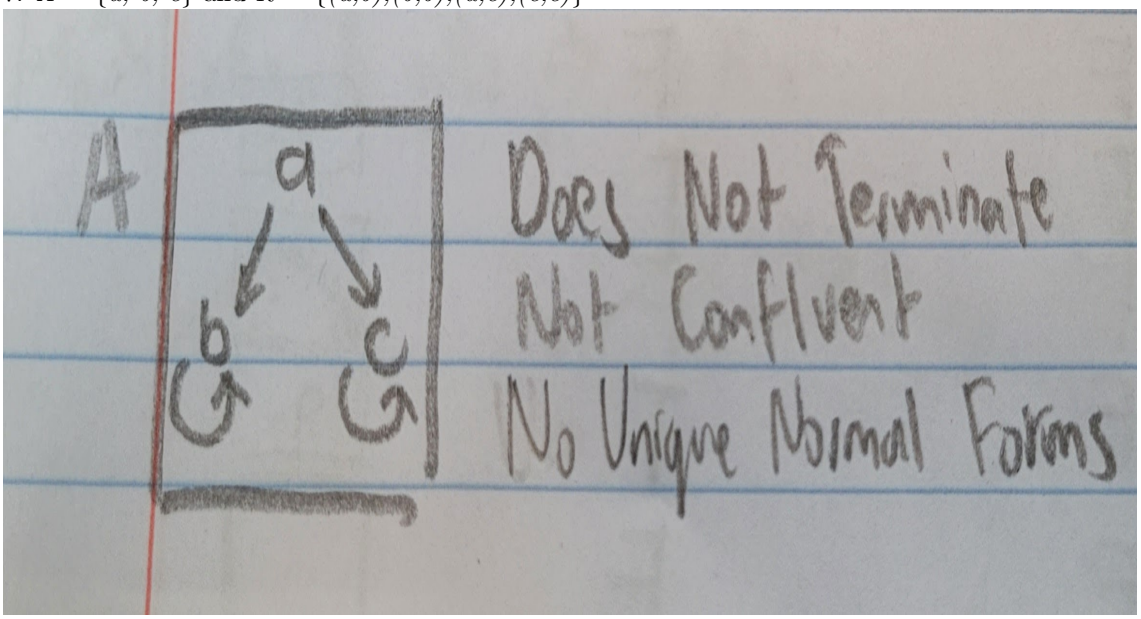
5. $A = \{a,b\}$ and $R = \{(a,a),(a,b)\}$



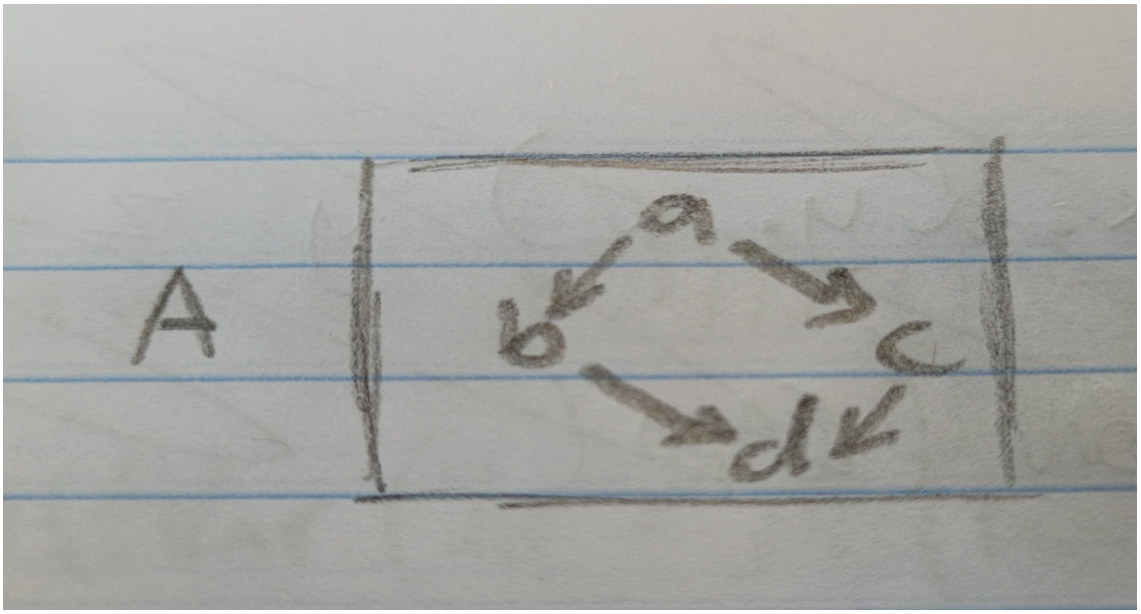
6. $A = \{a,\ b,\ c\}$ and $R = \{(a,b),(b,b),(a,c)\}$



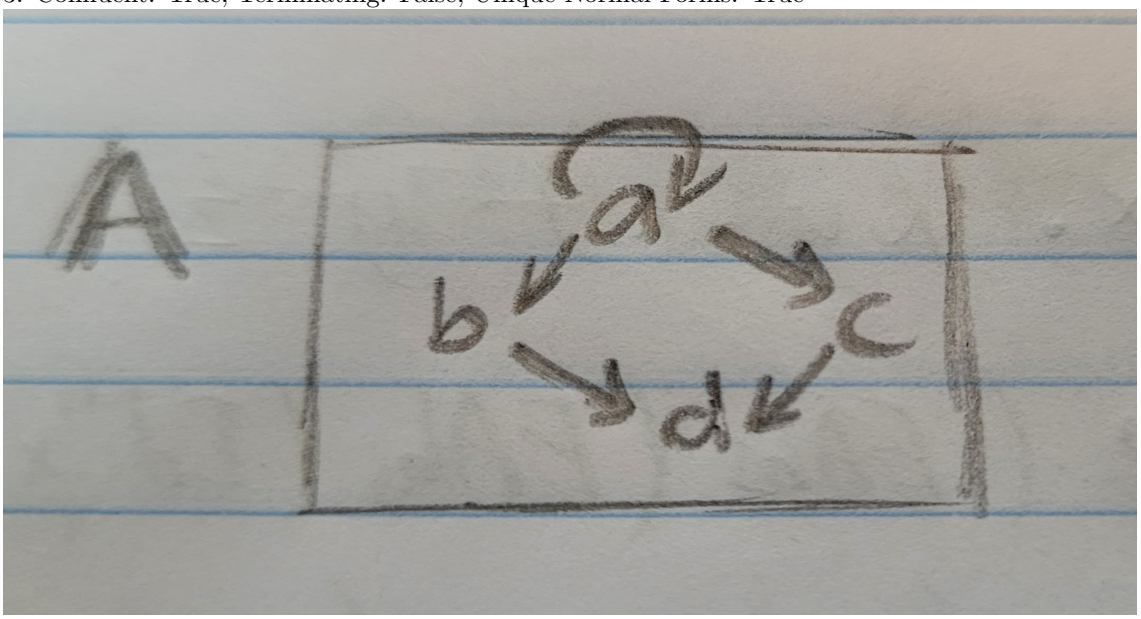
7. $A = \{a,\ b,\ c\}$ and $R = \{(a,b),(b,b),(a,c),(c,c)\}$



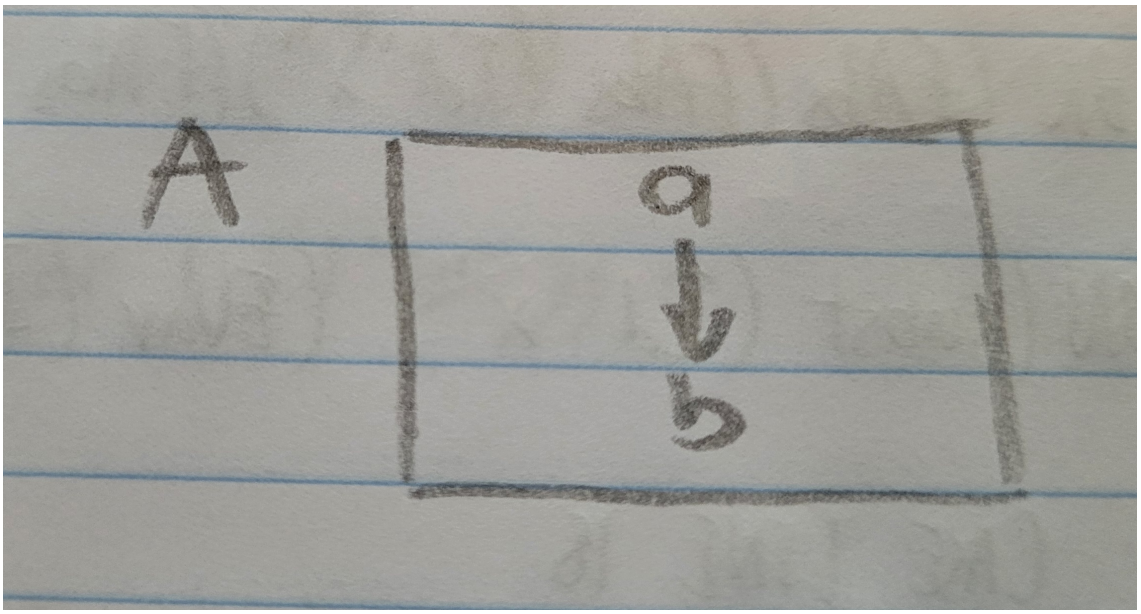
Example for cases
1. Confluent: True, Terminating: True, Unique Normal Forms: True



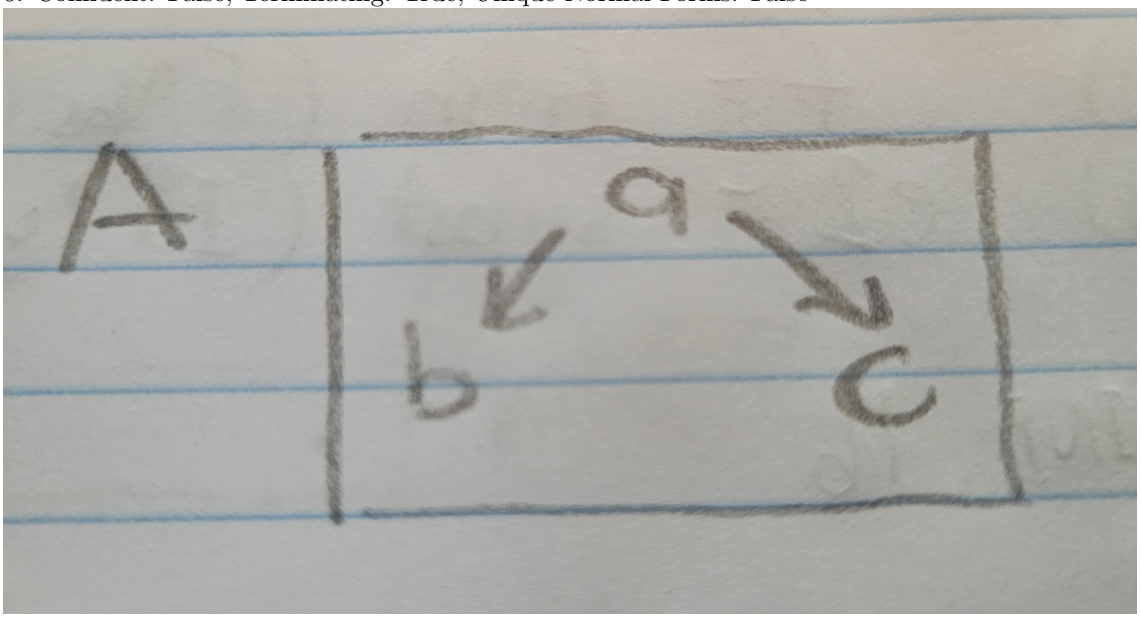
2. Confluent: True, Terminating: True, Unique Normal Forms: False
No example of this would exist, as in order for an ARS to be Terminating and Confluent it would need an Unique Normal Form as if one of the normal forms of the initial term reduces to itself, while the other normal form reduces to that same normal form, the ARS would no longer be Terminating as it can infinitely reduce to itself.
3. Confluent: True, Terminating: False, Unique Normal Forms: True



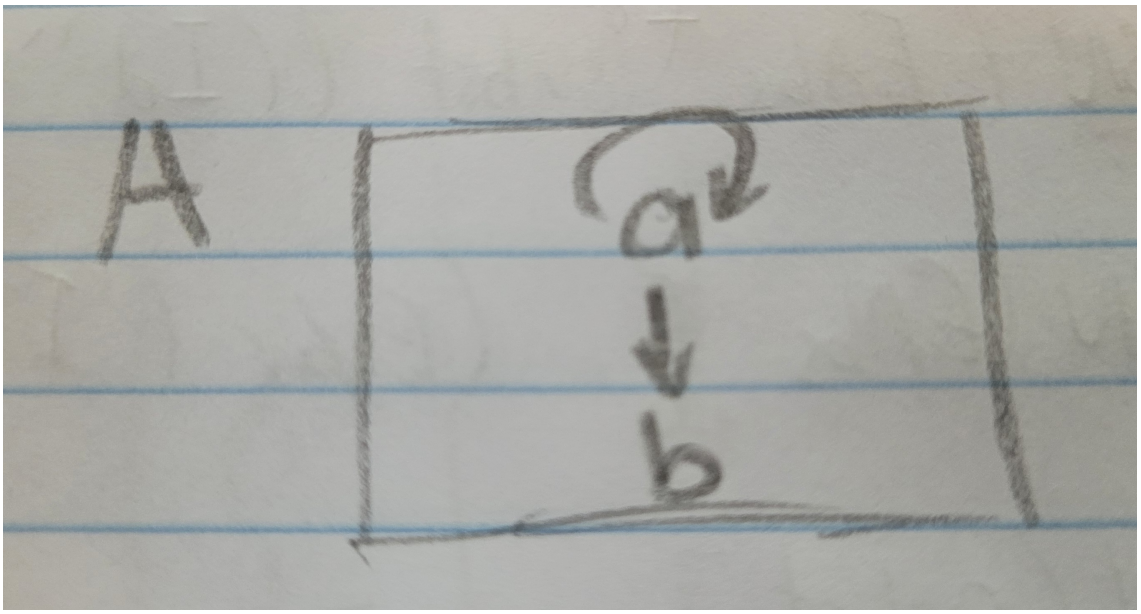
4. Confluent: True, Terminating: False, Unique Normal Forms: False
No example of this would exist, as in order for an ARS to be Confluent, the initial term should be able to be reduced into two different normal forms, and those two normal forms would have to be able to be reduced into another normal form, which would imply a Unique Normal Form.
5. Confluent: False, Terminating: True, Unique Normal Forms: True



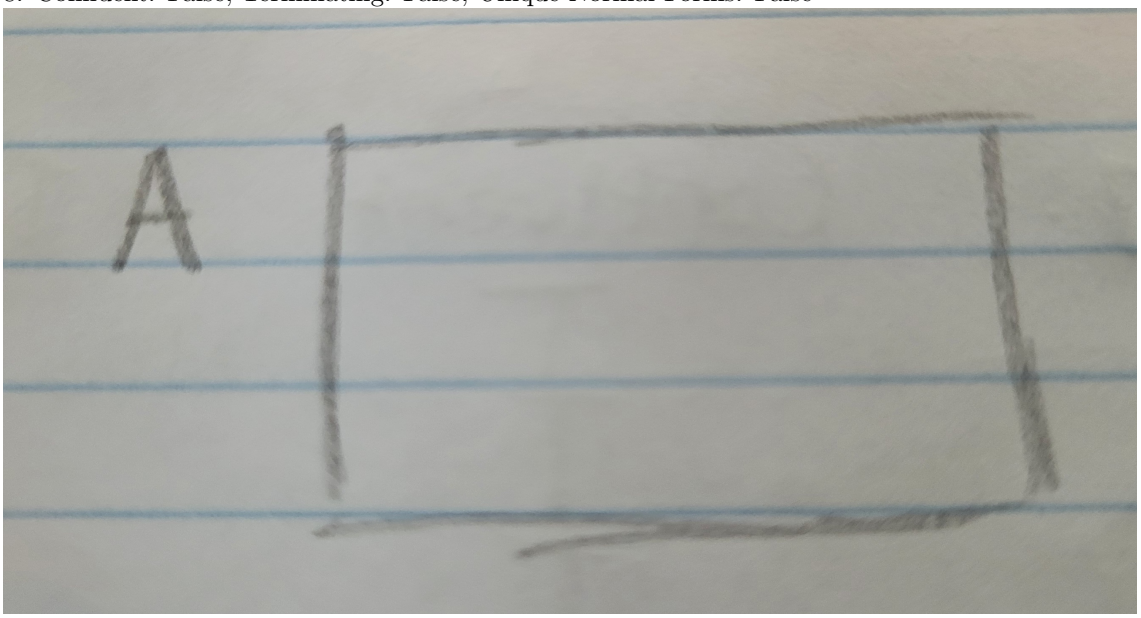
6. Confluent: False, Terminating: True, Unique Normal Forms: False



7. Confluent: False, Terminating: False, Unique Normal Forms: True



8. Confluent: False, Terminating: False, Unique Normal Forms: False



3 Project

3.1 Specification

For my project in Programming Languages (Fall 2022), I intend on performing data analysis on a data set that is publicly available from GitHub. To accomplish this, the following would have to occur (in order):

1. Receive approval from Dr. Kurtz for this specific project idea
2. Come up with specific questions to answer about the Programming Languages data set, the responses to which can be modeled through various statistical models such as Linear/Logistic Regression Graphs,

Data Cluster Graphs, Feature Reduction, LASSO Graphs, and/or Principle Component Analysis (PCA)

3. Receive approval from Dr. Kurtz about the questions and the overall documentation of the data analysis report

4. Download or import the data set from GitHub into Google Colab, this can be accomplished by replicating a specific iteration of the GitHub Programming Languages data set in an Excel sheet

5. Using the Pandas package for Python, create statistical models to respond to each of the questions created for the report

6. By the last day of October, provide a PDF of the Google Colab Notebook containing the compiled statistical models as a proof of implementation for this project. This can also be accomplished by embedding the Python code into the "Prototype" subsection of *report.tex*, with each code block being above the respective model that it generated when compiled

7. Write report for the analysis of the data set itself, explaining the process of selecting which model(s) to accept, reject, or if any model provided any surprising results about the data set in addition to providing a response to each of the questions created from **step 4**. A rough draft of this report should be submitted and present within the "Documentation" subsection of *report.tex*

3.2 Prototype

3.3 Documentation

3.4 Critical Appraisal

4 Conclusions

...