

CPSC-354 Report

Marc Domingo
Chapman University

November 27, 2022

Contents

1	Introduction	2
1.1	Week 1	2
1.2	Week 2	2
1.3	Week 3	2
1.4	Week 4	2
1.5	Week 5	2
1.6	Week 6	2
1.7	Week 7	2
1.8	Week 8	2
1.9	Week 9	2
1.10	Week 10	3
1.11	Week 11	3
1.12	Week 12	3
2	Homework	3
2.1	Week 1	3
2.2	Week 2	4
2.3	Week 3	6
2.4	Week 4	7
2.5	Week 5	10
2.6	Week 6	15
2.7	Week 7	16
2.8	Week 8	22
2.9	Week 9	23
2.10	Week 10	24
2.11	Week 11	25
2.12	Week 12	25
3	Project	27
3.1	Specification	27
3.2	Prototype	27
3.3	Documentation	27
3.4	Critical Appraisal	27
4	Conclusions	27

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.

1.6 Week 6

During the sixth week of Programming Languages, the lectures covered the concept of how to substitute and rewrite variables within functions, before applying the concept to Lambda Calculus.

1.7 Week 7

During the seventh week of Programming Languages, the lectures provided an introduction to the concept of Abstract Reduction Systems (ARS).

1.8 Week 8

During the eighth, week of Programming Languages, the lectures covered the concepts of Confluence and Termination in how they relate to ARS.

1.9 Week 9

During the ninth week of Programming Languages, the lectures covered the concepts of how to extend the definitions of Lambda calculus to if-then-else statements, defining variables through *let*, and recursion, in addition to starting Assignment 2.

1.10 Week 10

During the tenth week of Programming Languages, the lectures covered the concepts of Operational and Denotational Semantics, and how they apply to Natural Numbers. In addition to this, the lecture also involved the discussion about the design of a domain-specific language within the context of financial engineering.

1.11 Week 11

During the eleventh week of Programming Languages, the lectures covered the introduction to Part 2 of Assignment 2, where we implement the concept of Lists into our Lambda Calculus language. The lectures also introduced the concept of Hoare Logic.

1.12 Week 12

During the twelfth week of Programming Languages, the lectures covered the concept of Finitely Branching Systems, and eventually its application to ARS and proofs.

2 Homework

2.1 Week 1

The objective for this week's homework is to implement code to calculate the Greatest Common Denominator between two integers in a coding language of our choice.

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

The objective of this week's homework is to implement the following functions in Haskell code—a function *len* that takes in a list and calculates the length of a list, a function *select_evens* that takes in a list and returns only the even-indexed elements of that list, a function *select_odds* that takes in a list and returns only the odd-indexed elements of that list, a function *member* that takes in an element of a list and a list and returns whether that element exists within the list, a function *append* that takes in two lists and adds the elements of the second list to the first list, a function *revert* that takes in a list and returns a list with its elements in the reverse order of the original list, and a function *less_equal* that takes in two lists and returns whether the first list is less than the second list or not.

```

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
```

For this section of the homework, the objective is to show the steps that Haskell would take in order to achieve the result of a function given specific inputs.

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
```

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:

```

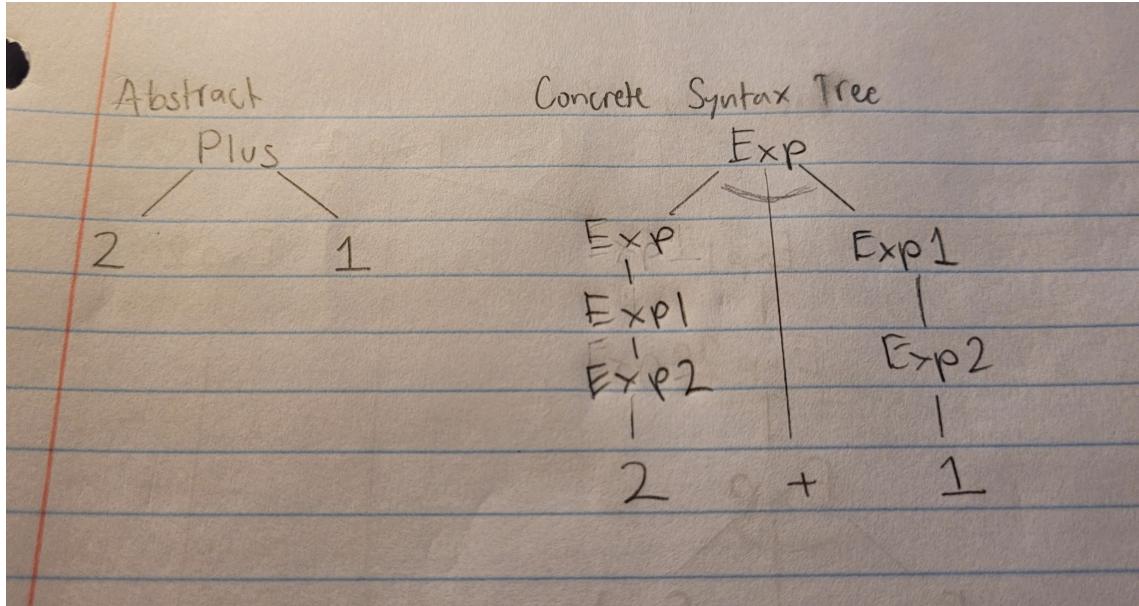
0->2
0->1
2->1
0->2
1->0
1->2
0->2
0->1
2->1
2->0
1->0
2->1
0->2
0->1
2->1
0->2
1->0
1->2
0->2
1->0
2->1
2->0
1->0
1->2
0->2
0->1
2->1
0->2
1->0
1->2
0->2

```

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.

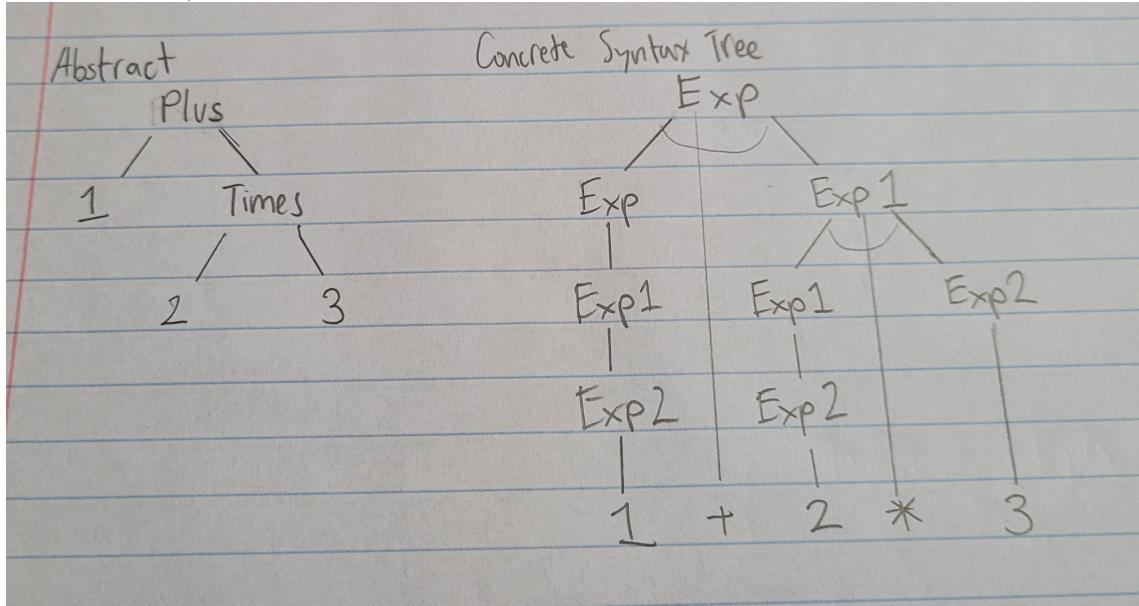
2.4 Week 4

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



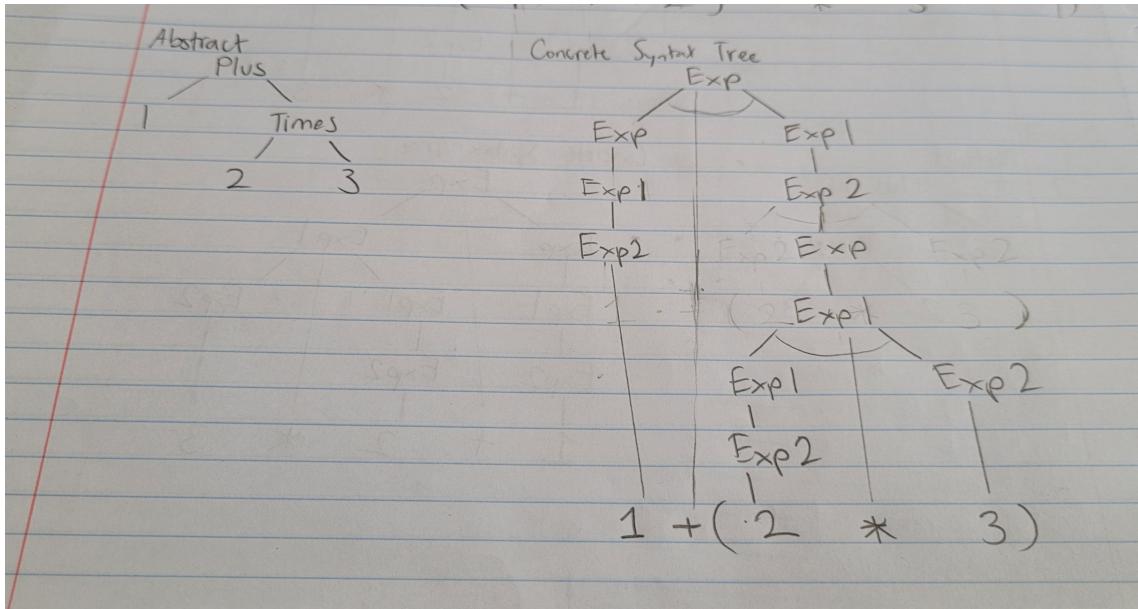
Plus (Num 2) (Num 1)

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



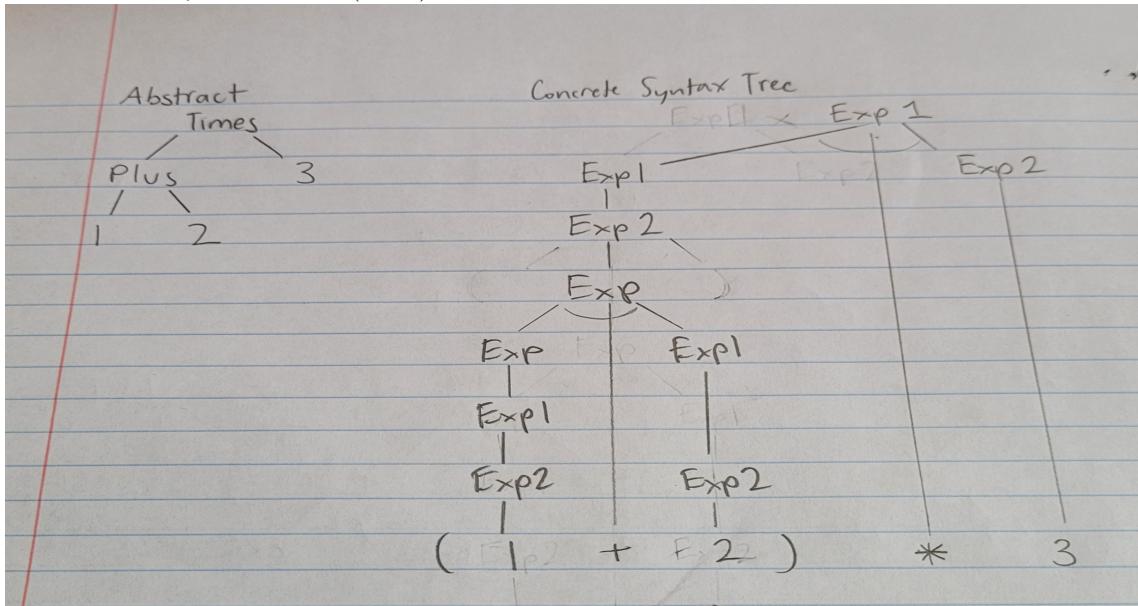
Plus (Num 1) (Times (Num 2) (Num 3))

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



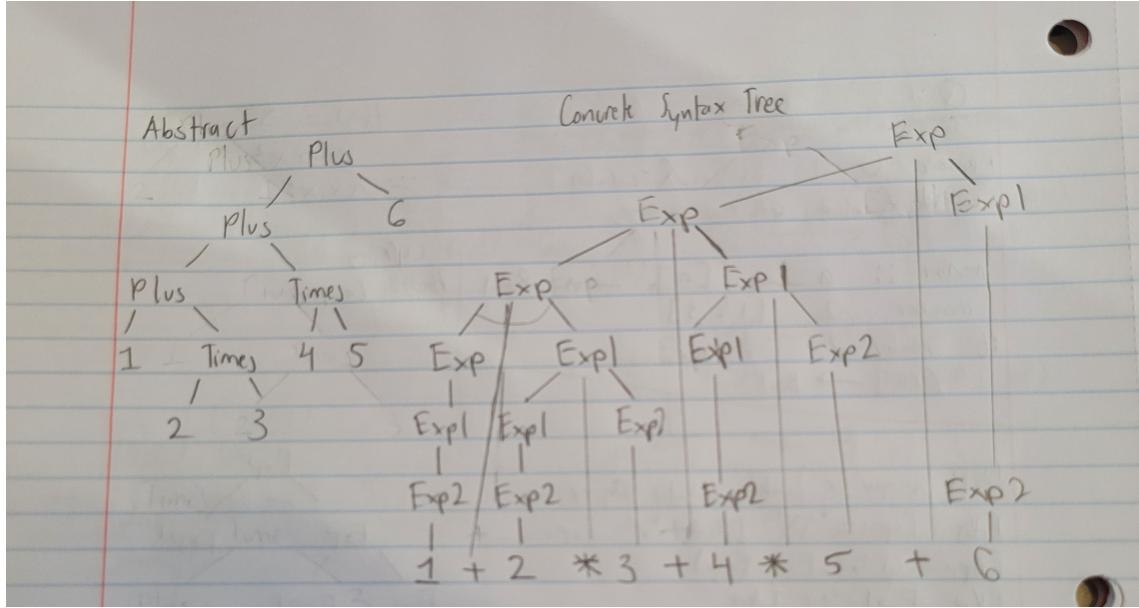
Plus (Num 1) (Times (Num 2) (Num 3))

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.

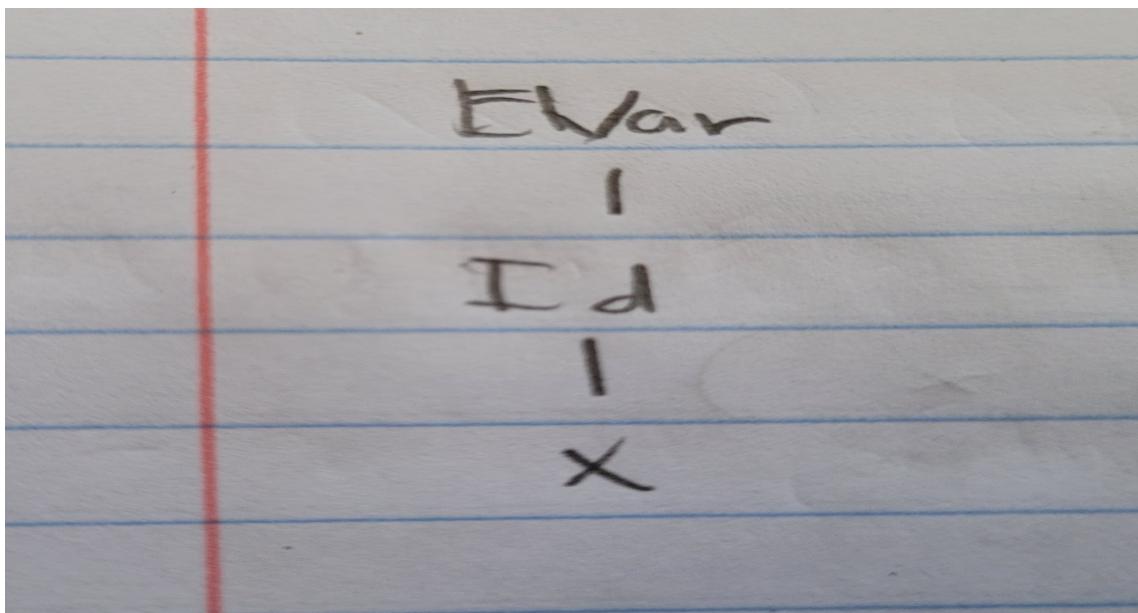
2.5 Week 5

Part 1:

The Linearized Tree for x is:

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

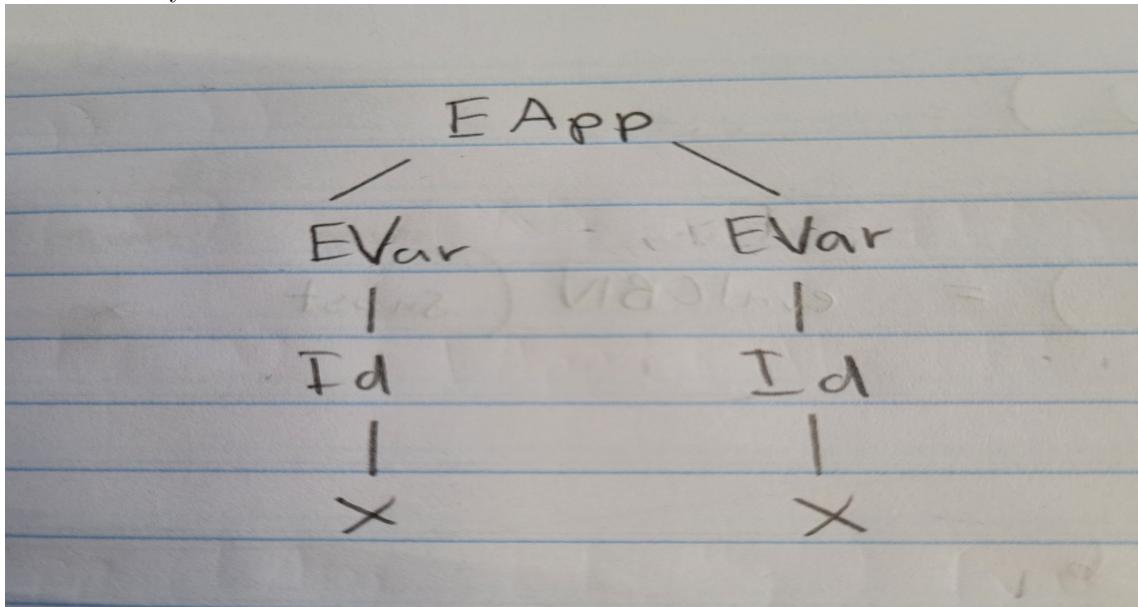
Its Abstract Syntax Tree:



The Linearized Tree for x x is:

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

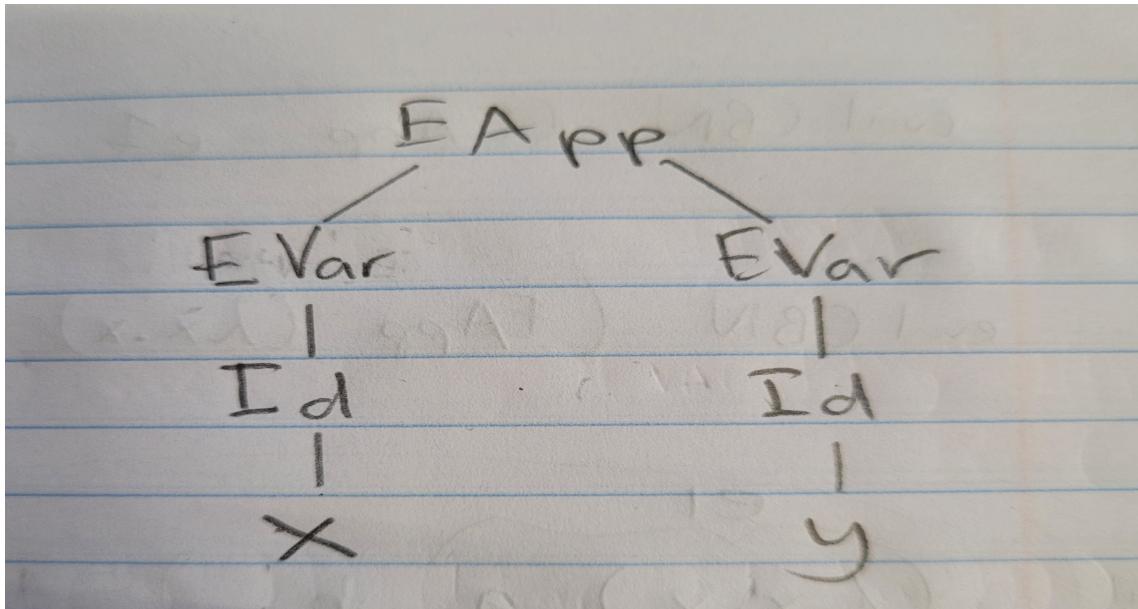
Its Abstract Syntax Tree:



The Linearized Tree for x y is:

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

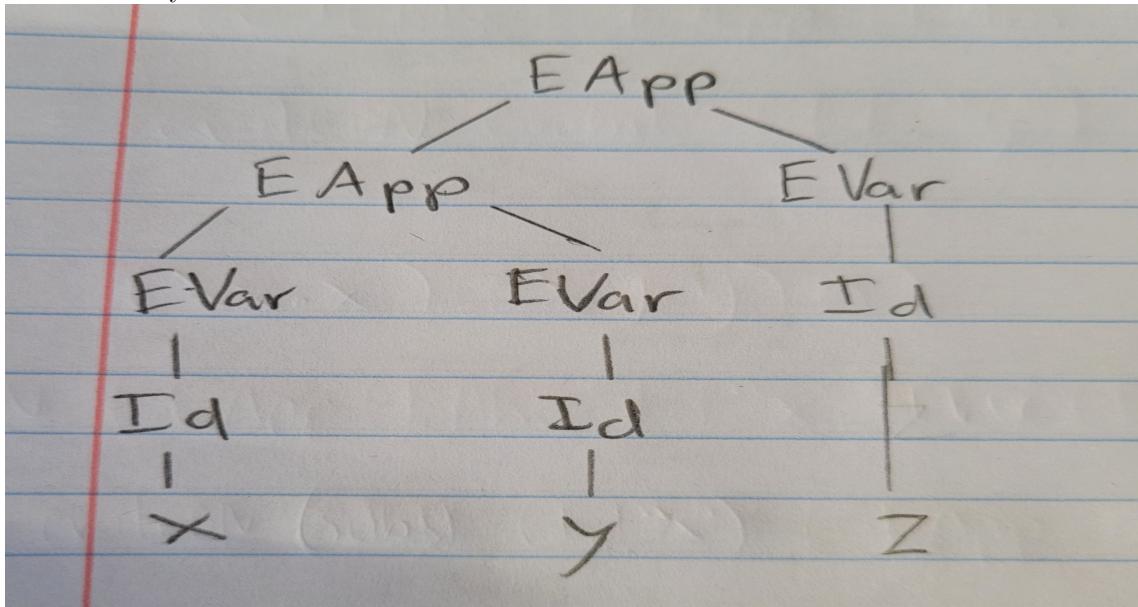
Its Abstract Syntax Tree:



The Linearized Tree for x y z is:

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

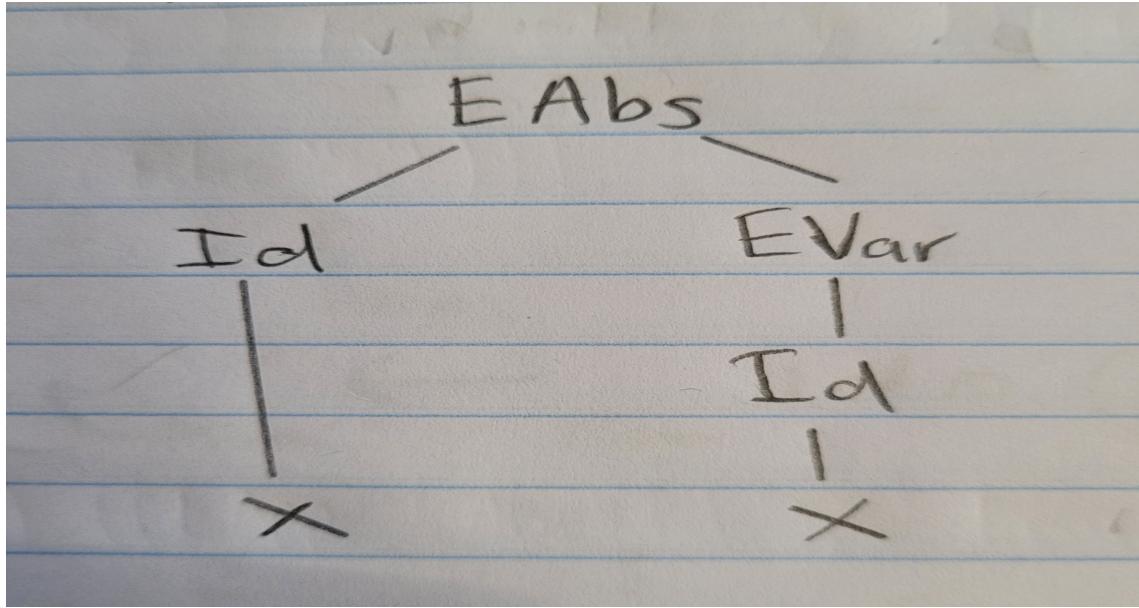
Its Abstract Syntax Tree:



The Linearized Tree for \x.x is:

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

Its Abstract Syntax Tree:



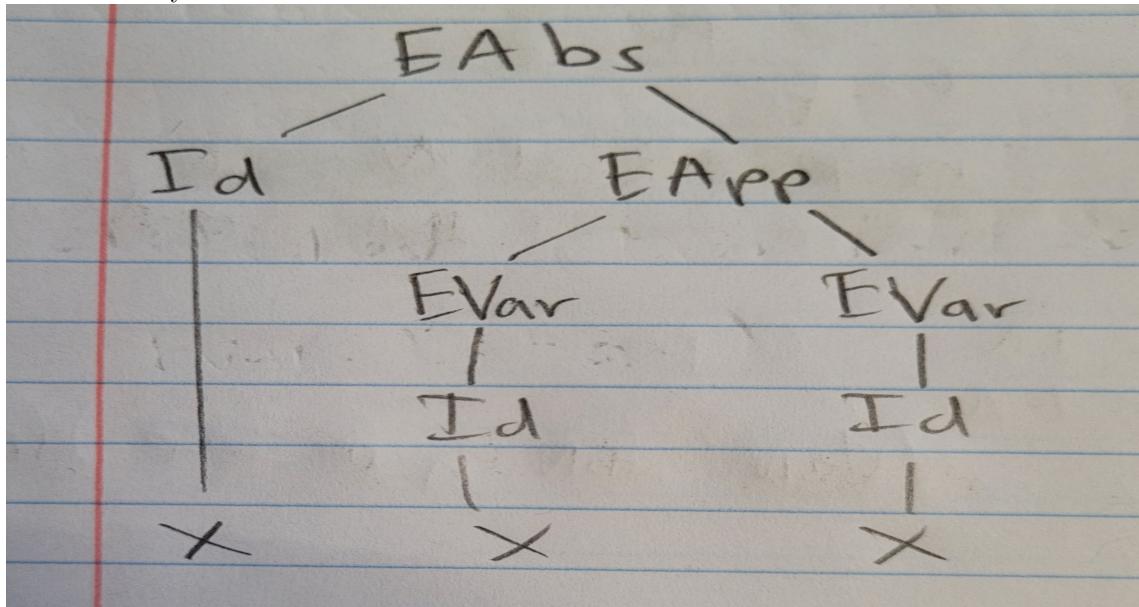
The Linearized Tree for $\lambda x. x x$ is:

```

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

```

Its Abstract Syntax Tree:



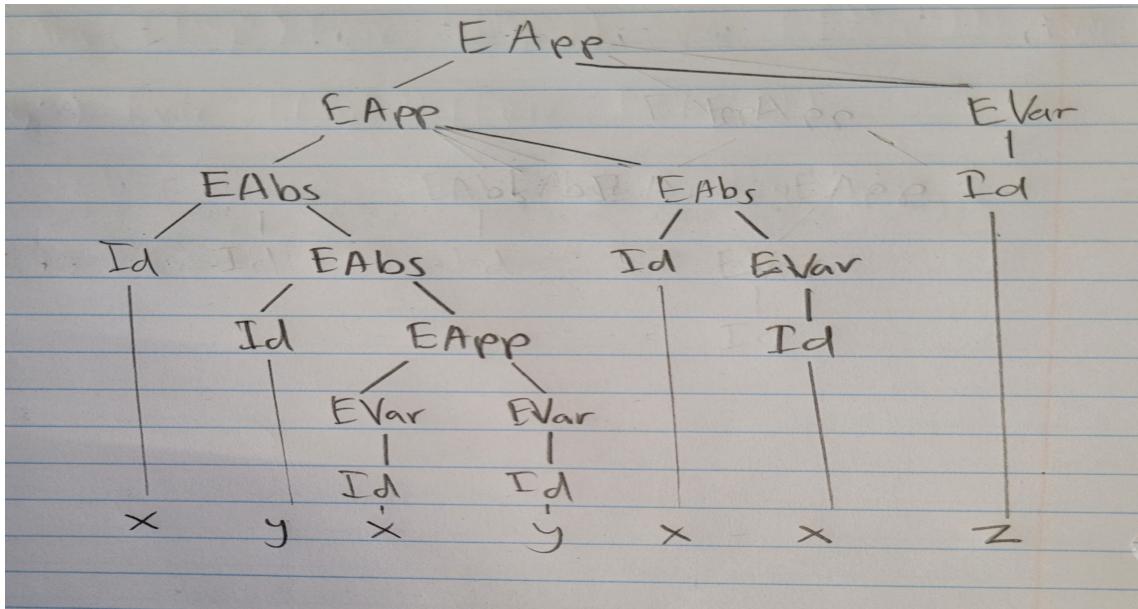
The Linearized Tree for $(\lambda x . (\lambda y . x y)) (\lambda x. x) z$ is:

```

\ 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")))

```

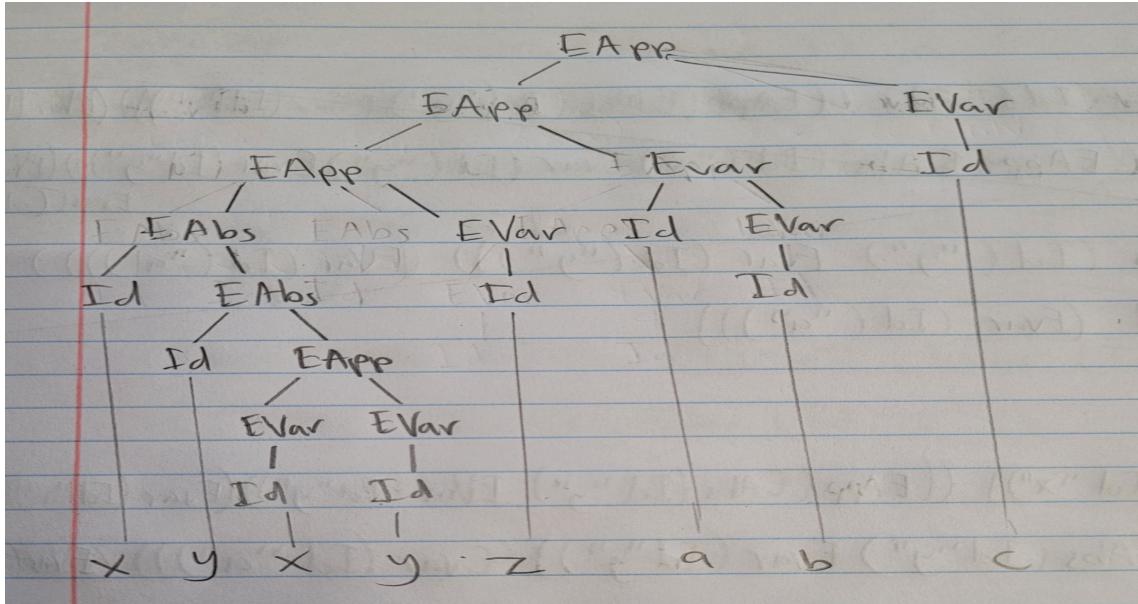
Its Abstract Syntax Tree:



The Linearized Tree for $(\lambda x . \lambda y . x y z) a b c$ 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
(\lambda x . \lambda y . x) a b = (\lambda y . a) b = a
(\lambda x . \lambda y . y) a b = (\lambda 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)
```

$(\lambda x.x)((\lambda y.y)a)$

$\text{evalCBN } (\text{EApp}(\text{EAbs}(\text{Id}^{\text{"x"}})\text{Evar}(\text{Id}^{\text{"x"}})) ((\text{EApp}(\text{EAbs}(\text{Id}^{\text{"y"}})\text{Evar}(\text{Id}^{\text{"y"}}))(\text{Evar}(\text{Id}^{\text{"a"}})))$

$(\text{LINE 6}) = \text{evalCBN } (\text{subst}(\text{Id}^{\text{"x"}})(\text{EApp}(\text{EAbs}(\text{Id}^{\text{"y"}})\text{Evar}(\text{Id}^{\text{"y"}}))(\text{Evar}(\text{Id}^{\text{"a"}}))) (\text{Evar}(\text{Id}^{\text{"x"}})))$

$(\text{LINE 15}) = \text{evalCBN } (\text{EApp}(\text{EAbs}(\text{Id}^{\text{"y"}})\text{Evar}(\text{Id}^{\text{"y"}}))(\text{Evar}(\text{Id}^{\text{"a"}})))$

$(\text{LINE 6}) = \text{evalCBN } (\text{subst}(\text{Id}^{\text{"y"}})(\text{EVar}(\text{Id}^{\text{"a"}})) (\text{EVar}(\text{Id}^{\text{"y"}})))$

$(\text{LINE 15}) = \text{evalCBN } (\text{Evar}(\text{Id}^{\text{"a"}}))$

$(\text{LINE 8}) = \text{EVar}(\text{Id}^{\text{"a"}})$

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 x3)) ((\f3.\x3. f3 (f3 (f3 x3))) x2))
=
(\x2. (\f3.\x3. 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 x3))))))))
```

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

2.7 Week 7

For lines 5-8 as shown through:

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

In *Line 5*, e_1 and e_2 are bound to the left of $=$, and the scope is the end of *Line 7*. For *Line 6*, the variables i and e_3 are bound to the left of $-_j$, and the scope is the end of the same line. Meanwhile, in *Line 7*, the variable e_3 is bound to the left of $-_j$, and its scope is the end of the same line. Finally, for *Line 8*, the variable x is bound to the left of $=$, and its scope is the end of the same line.

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)
```

In *Line 18*, the variables id , s , $id1$, and $e1$ are all bound to the left of $=$, and the scope is the end of *Line 22*. Meanwhile, in *Line 20*, the variable f is bound to the left of $=$, and the scope is also the end of *Line 22*. Lastly, on *Line 21*, the variable $e2$ is bound by $=$, and its scope is the end of *Line 22* as well.

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
```

```

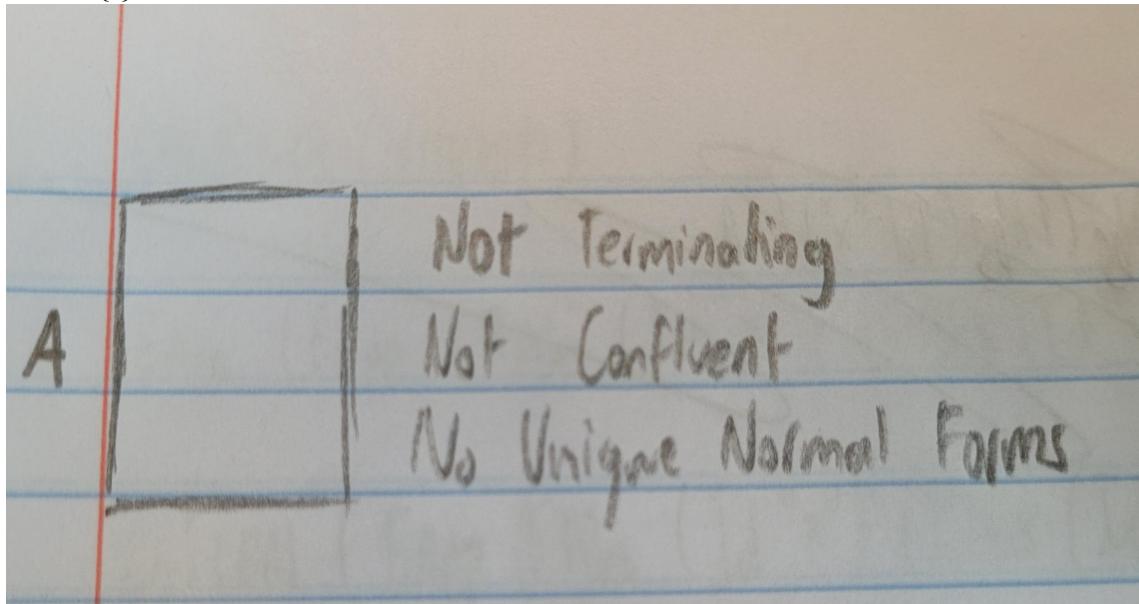
e2 = subst (Id "y") (EVar "y0") (EVar (Id "x"))
-- Line 16
EAbs (Id "y0") (subst (Id "y") (EVar (Id "y")) (EVar (Id "x")))
-- Line 16

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

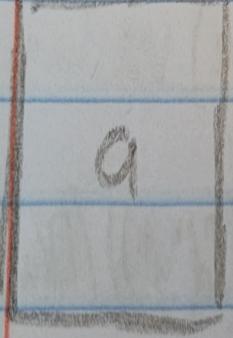
= -- Line 7
evalCBN (EApp (EAbs (Id "y0")) (EVar (Id "y"))) (EVar (Id "z"))
= -- Line 6
evalCBN (subst (Id "y0") (EVar (Id "z")) (EVar (Id "y")))
= -- Line 16
evalCBN (EVar (Id "y"))
= -- Line 8
EVar (Id "y")

```

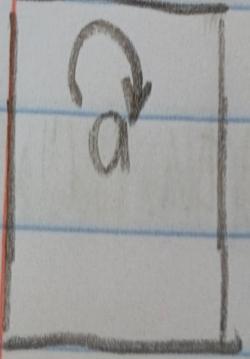
1. $A = \{ \}$



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

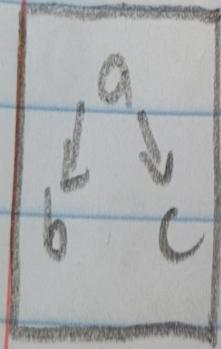
A		Terminates Not Confluent Unique Normal form (a)
---	---	---

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

A		Does not Terminate Not Confluent No Unique Normal Forms
---	--	---

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

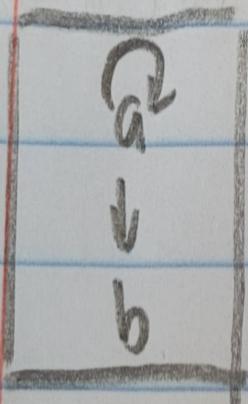
A



Terminates
Not Confluent
No Unique Normal Forms

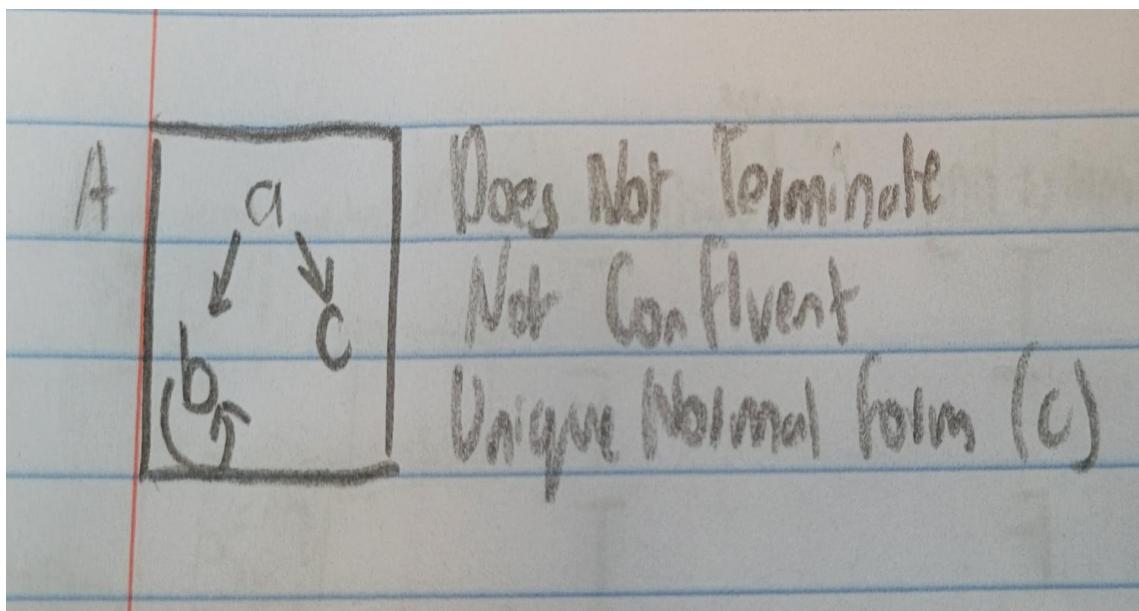
5. $A = \{a, b\}$ and $R = \{(a, a), (a, b)\}$

A

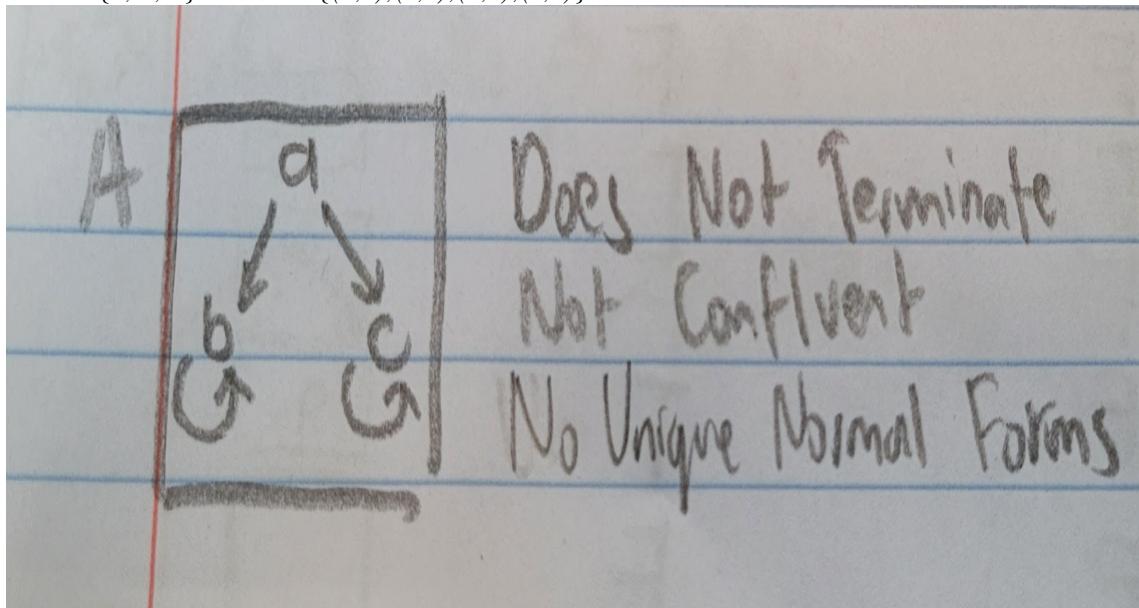


Does Not Terminate
Not Confluent
Unique Normal Form (b)

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

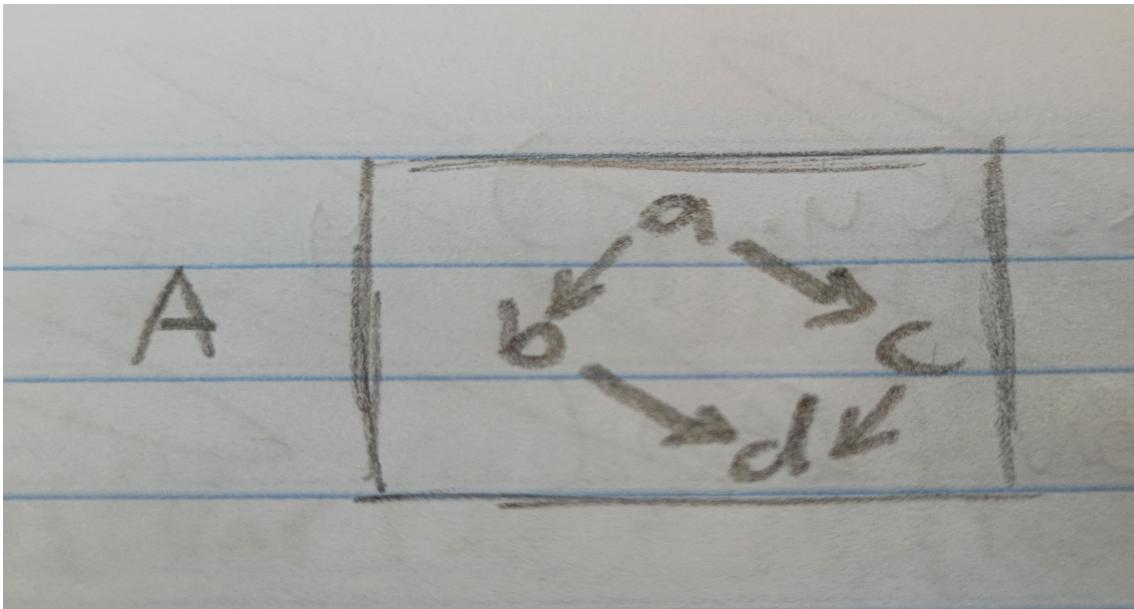


7. $A = \{a, b, c\}$ and $R = \{(a,b), (b,b), (a,c), (c,c)\}$



Example for cases

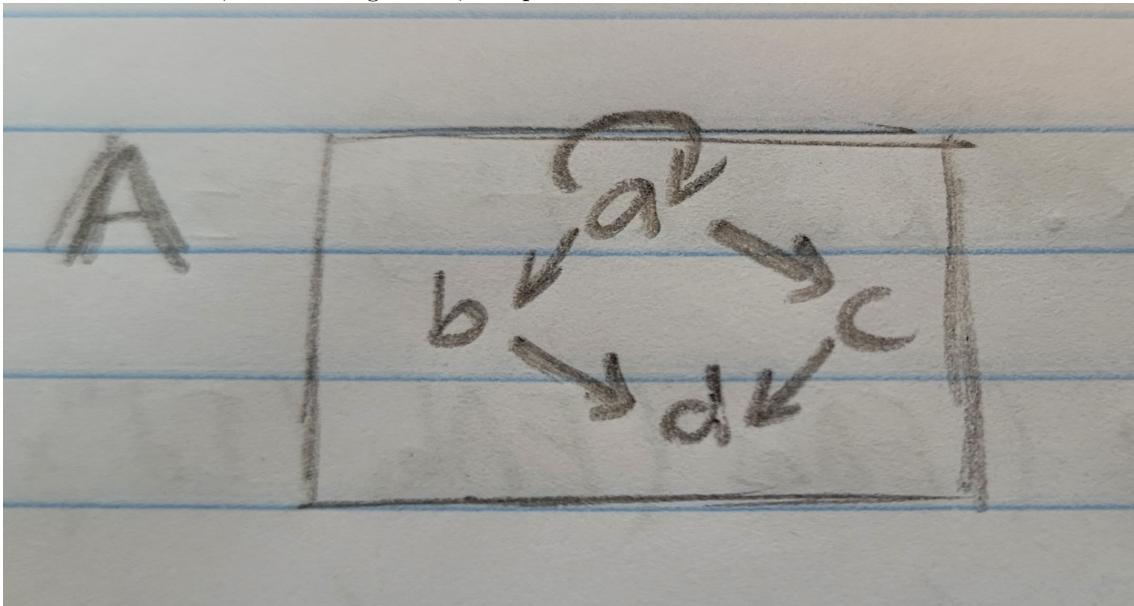
1. Confluent: True, Terminating: True, Unique Normal Forms: True



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



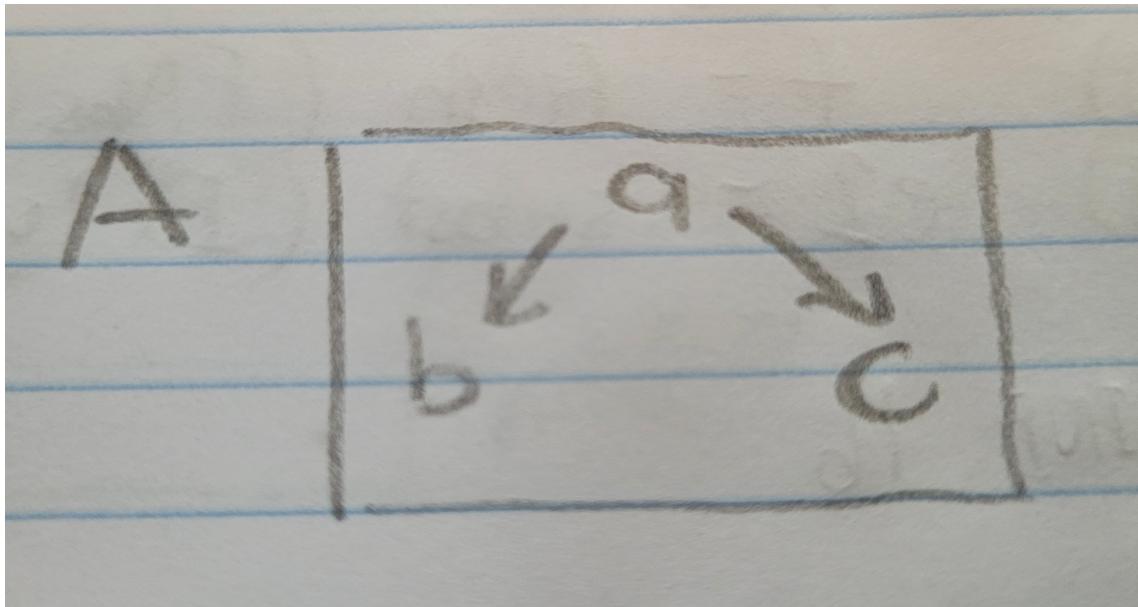
4. Confluent: True, Terminating: False, Unique Normal Forms: False

a points to itself, add picture later

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

There is no example of this as in order for there to be a possibility of having an Unique Normal Form, the ARS must be at the very least Confluent, making "Confluence: False" an impossibility to have alongside "Unique Normal Form: False."

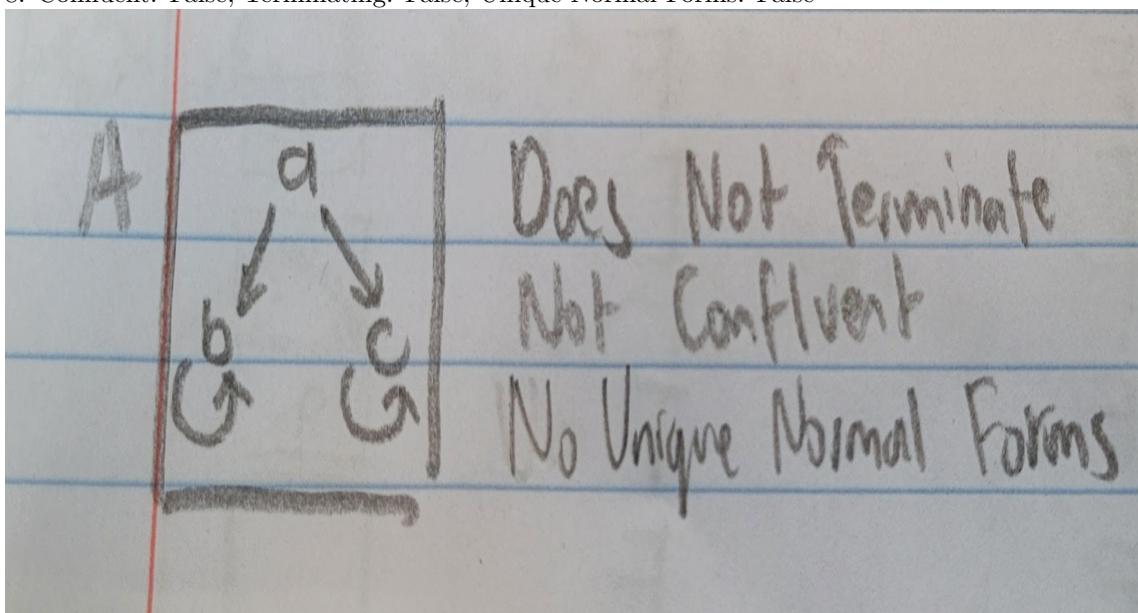
6. Confluent: False, Terminating: True, Unique Normal Forms: False



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

There is no example of this as in order for an ARS to have a Unique Normal Form, at its base, the ARS must be at least Confluent, making "Confluence: False" an impossibility to have alongside "Unique Normal Form: True."

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



2.8 Week 8

For the rewrite rules:

```

aa -> a
bb -> b
ba -> ab
ab -> ba

```

-
1. The ARS does not terminate because the output of the last rewrite rule ($ab \rightarrow ba$) is the input of the third rewrite rule ($ba \rightarrow ab$), and the output of the third rule is the input of the last rule. Because of instances such as: $ba \rightarrow ab \rightarrow ba\dots$, the ARS can be rewritten infinitely and thus the ARS does not terminate.
 2. The normal forms of this ARS are a , b , and $[]$. $[]$ is a normal form as if no characters are in the initial function, the function can only return the empty word $[]$. Meanwhile, the normal forms a and b can be seen in instances such as: $aaa \rightarrow aa \rightarrow a$ and $bbb \rightarrow bb \rightarrow b$.
 3. The main change that would be made to the function would be to eliminate the fourth rule. Thus the ARS would have 3 normal forms: a , b , and ab . This would then make the new ARS look like:

```

aa -> a
bb -> b
ba -> ab

```

This maintains the same equivalence cases as the previous ARS in instances such as: $aaa \rightarrow aa \rightarrow a$, $bbb \rightarrow bb \rightarrow b$, and $[] \rightarrow []$. This does not change the equivalence of the original equation as by the definition of equivalence, the fourth rule $ab \rightarrow ba$ can be also performed backwards $ab \leftarrow ba$, which is the same rule as the third rule. In return, this results in another possible normal form ab .

4. The invariant of this algorithm is that the starting word: is empty $[]$, contains at least one a , contains at least one b , or contains at least one a and at least one b . Through this, the specification of the algorithm determines that for any word, it determines if the word is *blank*, contains the letter a but not b , contains the letter b but not a , or contains both the letters a and b .

2.9 Week 9

Project Milestones

1. By **November 5, 2022**, email Dr. Kurtz the questions regarding the Programming Languages GitHub Data for the report, as well as the Excel sheet based on the data found from Githut.info.
 - There should at least be **three** data analysis-related questions.
 - The questions should relate to different types of statistical models, or involve the modification of a previously used statistical model using another data analysis technique.
 - The questions should have some relation to the nature or ask for some conclusion about Programming Languages based on the resulting statistical model.
 - This can alternatively be found uploaded to the *Programming Languages Project Documents* folder on GitHub.
2. By **November 30, 2022**, create the statistical models associated for each of the questions.
 - This will be accomplished via Google Colab
 - The code and associated models for each of the statistical models should be included within the **Prototype** subsection of this report.
 - Justifications for certain choices when generating the models (i.e. selecting a specific clustering methods or selecting which variables should be excluded from a revised model) should be also included alongside the code for the statistical models as comments.

- The Colab Notebook and a PDF of the compiled Notebook containing the statistical models should be uploaded to the *Programming Languages Project Documents* folder on GitHub.
3. By **December 7, 2022**, have a rough draft of the responses to each of the questions for this report.
- This can be either written in a Word Document uploaded to the *Programming Languages Project Documents* folder on GitHub or written directly to the **Documentation** section of this report.
 - The responses to each of the questions, while still answering the questions, should contain a clear connection and come to a conclusion about trends and observations that can be made regarding the Programming Languages data.
 - Justifications for decisions made regarding why certain modelling techniques or variables get excluded should also be reiterated within the report.

For the ARS (A, \rightarrow) where A is the set of words over the alphabet a, b, c and defined by the rules:

```

ba -> ab
ab -> ba
ac -> ca
ca -> ac
bc -> cb
cb -> bc

aa -> b
ab -> c
ac ->
bb ->
cb -> a
cc -> b

```

The first six rules of the ARS indicate that the order of each member of the alphabet within a word does not matter, as each word can be rewritten into any other word containing the same letters. Consequently, the first six rules also show that the ARS does not necessarily terminate, due to being able to a potential to be rearranged infinitely. The last six rules of the ARS on the other hand, show that the ARS does actually have normal forms, which happen to be a, b, c , and *blank*. While the seventh, eighth, eleventh, and twelfth rules of the ARS indicate that if there are an even number of members within the starting word, that the ARS will result in a single member being the normal form of the ARS, the ninth and tenth rules of the ARS imply that in the event that the starting word has an odd number of members, that the resulting normal form of the ARS would be blank. Because of this, while the ARS may have normal forms, none of them are unique to the ARS. Because this ARS has multiple normal forms, it is not necessarily confluent either as the only case that it would be confluent is where the initial word has an odd number of members.

2.10 Week 10

Given F as:

```
\f. \n. if n == 0 then 1 else f(n-1)*
```

Reduction of $fix_F 2$, which is equivalent to $F f_{fix2}$:

```
f2 =
  fixF 2 =
  F fixF 2 =
  -- Haskell is lazy and evaluates F first
  (\f. \n. if n == 0 then 1 else f(n-1) * n) FixF 2 =
```

```

if 2 == 0 then 1 else fixF (2-1) * 2 =
-- else case
(fixF 1) * 2 =
-- fixF 1 becomes F fixF 1, and Haskell evaluates F first
(\f. \n. if n == 0 then 1 else f(n-1) * n) FixF 1 * 2 =
(if 1 == 0 then 1 else fixF (1-1) * 1) * 2 =
-- else case
(((fixF 0) * 2) * 1) =
-- 1 can be moved to the left-hand side, while fixF 0 becomes F fixF1, and Haskell evaluates F
-- first
(\f. \n. if n == 0 then 1 else f(n-1) * n) FixF 0 * 2 * 1 =
(if 0 == 0 then 1 else fixF(0) * 0) FixF -1 2 * 1 =
-- then case
2 * 1 * 1 = -- arithmetic
2

```

2.11 Week 11

For this section of the homework, we are supposed to leave a copy of our question and two replies to other students regarding the "Composing Contracts" example of an application of DSL to Finances.

2.12 Week 12

The objective of this week's homework is to apply Hoare Logic to a given function.

Given the equation: while ($x \neq 0$) do $z := z * y$; $x := x - 1$ done...

Program variables: x , y , and z

Mathematical variables: m , n , and k

$\{z=n, x=m, y=k\}$ while ($x \neq 0$) do $z := z * y; x := x - 1$ done

t	x	y	z	$t + x = 10$	→ Program does terminate (when $x = 0$)
0	100	2	0		
1	99	2	2		
2	98	2	4		
3	97	2	8	$z = y^t$	→ program appears to calculate y to the power of x
...	$z = y^{(10-x)}$	
9	1	2	512		→ Depends on value of x and z as preconditions
10	0	2	1024		

possible invariants

+ 2 ≠ 0, treat 2 as 1

→ program terminates and yields

$z = y^x$ at the end if
x is positive and $z = 0$ prior

$$y^x * z = k^m * n$$

assumed postcondition

t	x	y	z	$t + x = 5$
0	5	2	0	$z = y^t$
1	4	2	2	
2	3	2	4	$z = y^{(5-x)}$
3	2	2	8	
4	1	2	16	
5	0	2	32	

based on sequential composition rule:

$$\{P \wedge S \wedge Q\} \quad \{Q \wedge T \wedge R\}$$

$$\{P \wedge S; T \wedge R\}$$

$$\{I \wedge B\} \quad S \{L\}$$

$\{I\}$ while B do S done $\{ \rightarrow B \wedge L \}$

Given that I is an Invariant / Precondition

Given that B is the formula's condition

Given that S is the main function

Given that P is a Precondition

Given that F is another function

$$\{y^{(x-1)} * (y^z * y) = k^m * n\} \quad z := z * y \quad \{y^{(x-1)} * (y^z * y)\} \quad \{... \} \quad x := x - 1 \quad \{y^x * z = k^m * n\}$$

$$\{y^{(x-1)} * (y^z * y) = k^m * n\} \quad z := z * y; x := x - 1 \quad \{y^x * z = k^m * n\}$$

$$\{y^{(x-1)} * (y^z * y) = k^m * n\} \quad \text{while } x \neq 0 \text{ do } z := z * y; x := x - 1 \quad \{y^x * z = k^m * n\}$$

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

...