CPSC-354 Report

Everett Prussak
Chapman University

November 26, 2022

Abstract

Consisting of CPSC 354 material

Contents

1 Introduction

This Report contains mainly consists of the Homework for each week, and the main project. In section 1, you can find the stuff done by Professor Kurz. Section 2 consists of Homework for each week. Section 3 consists of the project.

2 Homework

2.1 Week 1

Homework 1: Using Euclid's Elements Proposition 2 Algorithm on finding the Greatest Common Divisor amongst two numbers.

2.1.1 Python

In python, this algorithm can be written as:

```
a = 9
b = 33

while(a!=b):
    if(a>b):
         a = a-b
     else:
         b = b-a

print(a)
```

Using the sample given, 9 and 33, the Greatest Common Divisor is found in a simple way. Using the Euclid's Elements Proposition 2 Algorithm, we have two variables: **a** and **b**. To start, our samples are manually entered from the start of the program, and will begin the while loop. The while loop will continue until the **a** is the same value as **b**. The first line of code in the loop is an **if-statement**. This will compare **a** and **b** values, and will continue inside the **if-statement** if **a** is a large value than **b**. The Euclid's Elements Algorithm says: If

$$a > b$$

then replace **a** by

$$a - b$$

This is what happens in this first **if-statement**, as the variable **a** is replaced with a - b

If the **if-statement** is not executed, then the else statement will be preformed. Since our while loop tells us that it will continue until **a** is the same value as **b**, then we know that this else statement is:

$$b > a$$

This is the second part of the Euclid's Elements Proposition 2 Algorithm. It says that when:

$$b > a$$

to replace **b** with

$$b - a$$

This is what happens in this line of python code. The variable **b** is clearly replaced with b - a.

This while loop will continue until the values of **a** and **b** are the same. Once they are, the program will print the value of **a**. In this particular example, the value **3** would be printed.

2.1.2 C++

In C++, the algorithm can be written as:

```
#include <iostream>

using namespace std;

int main(int charc, char** argv){
    int a = 9;
    int b = 33;

    while(a!=b){
        if(a>b){
            a = a-b;
        }
        else{
            b = b-a;
        }
    }
    cout << a << endl;
}
```

Very similar code to the code in 2.1.1. The same process is being used, with two variables being created before the while loop. The while loop will continue until the values of the two variables are the same. Then the two conditions of

$$a > b$$

and

$$b > a$$

, are evaluated using an **if-statement** and **else-statement**. Once the correct condition is identified, the corresponding calculation done of each variable takes place. This will continue until the Greatest Common Divisor is found, and is printed to the screen. In this place, 3 is again printed.

2.2 Week 2

Homework 2: Create six Functions using Haskell: Select Evens, Select Odds, Member, Append, Revert, and Less Equal.

2.2.1 Select Evens and Select Odds

The task of these two functions could be used with each other. For Select Evens, the user would write the function name, then a list. The output would be the even element indices (Starting with 1 not 0). Here is the code for both.

```
select_evens [] = []
select_evens (x:xs) = select_odds xs

select_odds (x:xs) = x : select_evens xs
select_odds [] = []
```

These two functions are connected. Without one, the other will not work.

Going through the program, we will start with Select Odds. Using recursion, the head element will be split off first. Select Evens will be called with the other elements left in the list. Select Evens will then split the head and other elements again. This process will continue until the tail is an empty list. These methods will allow for only the odd indexed elements to be printed, or only the even indexed elements.

Here are a few outputs from the terminal:

```
ghci> select_evens ["a","b","c","d","e"]
["b","d"]
ghci> select_odds ["a","b","c"]
["a","c"]
ghci> select_odds ["a","b","c","d","e"]
["a","c","e"]
ghci> select_odds [1,2,3,4,5]
[1,3,5]
ghci> select_evens[432,34,543,2334,23]
[34,2334]
```

Here is the Task 2 Equational Reasoning for the Select Evens and Odds Function:

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

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

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

select_evens ("c" : ["d", "e"]) = select_odds["d", "e"]

select_odds ("d" : ["e"]) = "b" : ("d" : select_evens ["e"])

select_evens ("e" : []) = select_odds []
```

```
select_odds [] =  "b" : ("d" : ([]))

["b", "d"]
```

Note that if Select Odds was called, the same procedure would occur, but would start with Select Odds first and would be the opposite of this output.

2.2.2 Member

In the member function, the user would ask for a ”True” or ”False” about if a list consisted of a particular element. If the list consisted of the element, then True would be returned. Otherwise, False is returned.

```
member y (x:xs) = if y==x
    then True
    else if len(xs) < 1
        then False
        else member y xs
```

The element that the user wants to know is the y. It starts off by comparing y to x, which is the head of the current list. If y is the same as x, then the element that the user is looking for is indeed in the list and True is returned. However, if it is not, then we will use the len function to see the size of the rest of the list. If it is not above 1, then False is returned. False is returned because there is nothing left to compare to the user element. However if it is above 1, then member is called again with the user element and the rest of the elements. This simulates the element being compared to each element in the list.

Here are some outputs for this function:

```
ghci> member "a" ["a", "b"]
True
ghci> member "a" ["c", "d", "e"]
False
ghci> member 4 [2,3,5]
False
ghci> member 4 [2,3,3,4,5]
True
ghci> member 4 [4,1,6,2]
True
```

Here is the Task 2 Equational Reasoning for the Member Function:

```
member 4 [2,3,4] =
    4 == 2
    else if len([3,4]) < 1

    len[3,4] = 3

    else if 3 < 1
        then False
        else member 4 [3,4]

member 4 [3,4]
```

```
    4 == 3
    else if len([4]) < 1

    len[4] = 2

    else if 2 < 1
        then False
        else member 4 [4]

member 4 [4]
    4 == 4
    then True
```

2.2.3 Append

The append function takes two lists from the user, and appends all of the second lists elements to the back of the first list. This function was difficult for me at first, but I realized that the first list stays at the front of the list, and that only the second list needed elements to ”move”.

Below is my code for the Append Function:

```
append [] (y:ys) = if len(ys) > 0
    then
        y : append [] ys
    else
        [y]
append (x:xs) (y:ys) = x : append xs (y:ys)
```

The last line will be the first line that executes. The user will have the two lists, but the function will have to iterate the entire (x:xs) list first. This is because these elements will be in the front of the new list regardless. After this line is recursively called, the first line will be called with an empty list and all of the second list. The len function is also used in this function as well. The if statement looks at the size of the tail elements of the second list. If it is greater than 0, then it will recursively add the elements to the list that already contains the first list elements. Once the len(ys) is not greater than 0, the else statement will have [y] which will basically not recall any of the append functions, and will not append anything to the list. This will let the recursion end.

Here are some outputs of the append function:

```
ghci> append [1,2] [3,4,5]
[1,2,3,4,5]
ghci> append [1,2,3,4,5] [7,8,9]
[1,2,3,4,5,7,8,9]
```

Here is the Task 2 Equational Reasoning for the Append Function:

```
append [1,2,3] [7,8,9] =
    1 : (append [2,3] [7,8,9])
    1 : (2 : (append [3] [7,8,9]))
    1 : (2 : (3 : (append [] [7,8,9])))
    1 : (2 : (3 : (7 : (append [8,9]))))
    1 : (2 : (3 : (7 : (8 : (append [9])))))
    1 : (2 : (3 : (7 : (8 : (9 : (append[]))))))
    1 : (2 : (3 : (7 : (8 : (9 : [])))))
```

2.2.4 Revert

The Revert function will take a list from the user, and output the list with the elements in reversed order.

Below is the code to the Revert Function:

```
revert [] = []
revert (x:xs) = append (revert xs) [x]
```

This function uses the previously created function Append. Revert will call the append method recursively. It will append the first element of the list to the back, then continue with the rest of the elements. This was designed with some thought of a Stack.
Here are some outputs for the Revert Function:

```
ghci> revert [1,2,3]
[3,2,1]
ghci> revert [8,1,2,4]
[4,2,1,8]
```

Here is the Task 2 Equational Reasoning for the Revert Function:

```
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 [3,2] [1]
    [3,2,1]
```

2.2.5 Less Equal

The last function that I created was Less Equal. This simply compared the elements from two lists to see if first list was less than or equal to the same indexed element of the other list. For example [1,2] vs [5,6] would be true because 1 is less than or equal to 5 and 2 is less than or equal to 6.

Below is the code to the Less equal Function

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

The beginning of this function will start with the second line. If compares the head elements of list 1 and list 2. If x is less than or equal to, the Less Equal function will be called again, but this time with the tail elements. If at any point the list 1 element "x" is greater than list 2 element "y", False will be returned.

If all of the elements are compared, then base case is called. This is because there are no elements left, so they are empty lists. Since nothing was flagged, then True will be returned, meaning that all of the list 1 elements are either less than or equal to the list 2 elements at the same index.

Here are some outputs for the Less Equal Function:

```
ghci> less_equal [1,2,3] [2,3,4]
True
ghci> less_equal [1,2,3] [2,3,2]
False
ghci> less_equal [1,2,3] [2,3,3]
True
```

Here is the Task 2 Equational Reasoning for the Less Equal Function:

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

    2 <= 3
    then
        less_equal [3] [4]

    3 <= 4
    then
        less_equal [] []

    True
```

If the left value was ever larger, than False would be the value.

2.2.6 Len

I did not create this function. I took this from Hackmd.io, thanks to Professor Kurz. Below is the code used that my other functions also used.

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

2.3 Week 3

Week 3 Homework was about the Hanoi Tower. The task was to complete the execution from the dots down. Here that is:

```
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
```

I followed this exact sequence with the Tower of Hanoi website. It took 31 turns, which is the minimum moves possible for a Hanoi Tower of size 5.

The word "Hanoi" appears 31 times. The word "move" also appears 31 times.
A simple equation can be expressed for the number of blocks in the Hanoi Tower. Assuming n is the number of blocks/rings in the tower, the minimum number of moves can be expressed as

$$2^n - 1$$

In this case, n was 5. Thus,

$$2^5 = 32$$

and then

$$32 - 1 = 31$$

Here are some other minimum moves required for other ring heights.

$$2^1 - 1 = 1$$

$$2^2 - 1 = 3$$

$$2^3 - 1 = 7$$

$$2^4 - 1 = 15$$

$$2^6 - 1 = 63$$

2.4 Week 4

Week 4 consisted of making Parse Tree and Abstract Syntax Tree of math 5 problems.

2.4.1 Parse Tree Diagrams

This is Part 1 of Week 4's Homework. The task was to write out the derivation trees for the following:

$$2 + 1$$

$$1 + 2 * 3$$

$$1 + (2 * 3)$$

$$(1 + 2) * 3$$

$$1 + 2 * 3 + 4 * 5 + 6$$

while using this Context-Free Grammar:

```
Exp -> Exp '+' Exp1
Exp1 -> Exp1 '*' Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
Exp -> Exp1
Exp1 -> Exp2
```

For

$$2 + 1$$

I got the following Parse Tree:

```
         Exp
       /  |  \
      Exp '+' Exp1
     /         \
    Exp1        Exp2
   /             \
  Exp2           Integer
 /                 |
Integer            1
  |
  2
```

Here this tree starts with Exp. It expands for addition with Exp and Exp1. After this, I simply converted Exp to Exp1 to Exp2 then the number 2, and Exp1 to Exp2 then the number 1.

For

$$1 + 2 * 3$$

I got the following Parse Tree:

```
         __ Exp__
        /    |   \
      Exp   ’+’   __Exp1__
      /          /    |    \
    Exp1       Exp1  ’*’   Exp2
    /           |           |
   Exp2        Exp2      Integer
    |           |           |
 Integer     Integer        3
    |           |
    1           2
```

This Tree starts similar to last time with Exp. Exp will expand to Exp and Exp1. Exp will expand directly until it is the integer 1. Exp1 will expand to Exp1 and Exp2 for multiplication. Exp1 and Exp2 from this will become 2 and 3. This will allow the tree to calculate 2 times 3 to become 6. Then the root will have the two nodes of 1 and 6 to add to become 7.

For

$$1 + (2 * 3)$$

I got the following Parse Tree:

```
        ___Exp____
       /    |      \
     Exp   ’+’    Exp1
    /               |
   Exp1            Exp2
   |             /  |  \
   Exp2        ’(’ Exp ’)’
   |                |
 Integer           Exp1
   |              /    \
   1           Exp1   Exp2
                |       |
               Exp2   Integer
                |       |
             Integer    3
                |
                2
```

This is a little more complex than the prior example, but will end with the same answer of 7. Exp will expand to Exp and Exp1 for addition. Exp will expand until it becomes the integer of 1. Exp1 will become Exp2 then Exp because this had parenthesis. Exp will expand to Exp1, then Exp1 will expand to Exp1 and Exp2. These will become 2 and 3 so it will be multiplied together. Then this will finally become 7 if calculated.

For

$$(1 + 2) * 3$$

I got the following Parse Tree:

```
        ____Exp1___
       /     |     \
     Exp1   '*'     Exp2
    /            /  |   \
   Exp2        '('  Exp ')'
  /               /  |  \
Integer         Exp '+'  Exp1
  |             |         |
  3            Exp1      Exp2
                |         |
              Exp2      Integer
                |         |
             Interger     2
                |
                1
```

This time the parenthesis is with the addition of 1 and 2. I started with multiplication (Exp1) because it is 3 being multiplied with the entire parenthesis. Exp1 will expand to Exp1 and Exp2. Exp1 will become the integer 3. Exp2 will first go back to Exp because this is a parenthesis and our rule allows to go back to Exp when Parenthesis is present. Exp will then expand to Exp and Exp1 for the addition. These will expand until the integers 1 and 2 are present. Then 1 added to 2 will become 3. 3 multiple with 3 will become 9. This answer is 9.

For

$$1 + 2 * 3 + 4 * 5 + 6$$

I got the following Parse Tree:

```
                ____________Exp__________
               /             |            \
            _____Exp_____   '+'          Exp1
          /       |      \                  \
          Exp    '+'       Exp1            Exp2
       /   |  \          /   |   \          |
      Exp '+' Exp1     Exp1 '*'  Exp2     Integer
     /      /   |  \    |            |     |
    Exp1  Exp1 '* Exp2  Exp2      Integer  6
     |      |      |      |          |
    Exp2   Exp2 Integer  Integer     5
     |      |       \     |
    Integer Integer  3    4
    |       |
    1       2
```

This was by far the most complicated of the problems. To start, I expanded with the addition of 1 with everything else. This means Exp would be the root with Exp and Exp1 being the child nodes. Exp1 will become Exp2 then the integer 1. Now I must account for the other 5 numbers in the equation. Starting with Exp again I decided for the sum of 2 multiple with 3 being added to entire sum of 4 multiplied with 5 and then added with 1. Exp will expand to Exp and Exp1 for this addition. Exp1 will become the 2 times 3 side of the equation. Exp1 will expand to Exp1 and Exp2. Exp1 will expand until integer 2 is reached, and Exp2 is expanded until integer 3 is reached. Now back to the Exp that is being added with 2 times 3. This Exp will be expanded to Exp and Exp1. Exp will be expanded until integer 6 is reached. Exp1 is the multiplication formula. Exp1 will expand to Exp1 and Exp2. These two nodes will then be converted until integer 4 and 5 are reached. Now the tree is complete. First 1 will be added with the rest of the equation. The rest of the equation will have 2 multiplied with 3 become 6. 4 is multipled by 5 to become 20 then added with 6 to become 26. This 26 is added with 6 from 2 times 3. Then our 1 is added to become 33.

2.4.2 Abstract Syntax Tree

The same 5 problems (Equations) that were used for the Parse Tree's are now used to be made into Abstract Syntax Trees.

These AST are to be made using this BNFC Grammar:

```
Plus.   Exp ::= Exp "+" Exp1 ;
Times.  Exp1 ::= Exp1 "*" Exp2 ;
Num.    Exp2 ::= Integer ;

coercions Exp 2 ;
```

For

$$2 + 1$$

I got the following Abstract Syntax Tree:

```
         Plus
         /  \
        Num  Num
        |     |
        2     1
```

This Abstract Syntax Tree is easy to read. The only math being used is an addition sign. Simply Plus is used then expanded to its two terms of 2 and 1.

For

$$1 + 2 * 3$$

I got the following Abstract Syntax Tree:

```
   Plus
  /    \
 Num   Times
  |   /   \
  1  Num  Num
      |    |
      2    3
```

This Abstract Syntax Tree is bigger than the prior equation/problem. I once again started with Plus then expanded 1 and the product of 2 times 3. This works with our Grammar Rules.

For

$$1 + (2 * 3)$$

I got the following Abstract Syntax Tree:

```
   Plus
  /    \
 Num   Times
  |   /   \
  1  Num  Num
      |    |
      2    3
```

This is the same exact AST as the prior example. Since the parenthesis is around the multiplication, which would be first in this instance anyways, nothing changes. Other than the parenthesis being added, nothing changes, so the rest of the AST is the same

For

$$(1 + 2) * 3$$

I got the following Abstract Syntax Tree:

```
  Times
  /   \
 Plus  Num
 /  \    |
Num Num  3
|    |
1    2
```

This AST is different than the prior examples. Since the parenthesis are around the 1 plus 2, then this part must be done first in order to be multiplied with the three. I had Times be the root with the expansion of integer 3, and Plus. The Plus would then expand to 1 and 2. By this tree, 1 Plus 2 is 3 then 3 times 3 is 9.

For

$$1 + 2 * 3 + 4 * 5 + 6$$

I got the following Abstract Syntax Tree:

```
          ______Plus______
        /                  \
       __Plus__            Num
     /         \             |
    Plus       Times         6
   /   \       /   \
  Num  Times Num   Num
   |   /  \   |     |
   1  Num Num 4     5
      |    |
      2    3
```

This AST is much more complex than the other examples. To begin, I had Plus become the root, as 1 Plus other side of equation will be the child nodes. The nodes will be integer 1 and Plus. Plus will expand to Plus and Times. The Times will expand to the first part of the other equation of 2 Times 3. The Plus will expand to Times and integer 6. The Times will become 4 and 5. Doing this AST from bottom up, will have 4 Times 5 become 20. Then 20 Plus 6 becoming 26. 2 Times 3 becoming 6. 6 plus 26 becoming 32. Then finally 32 will be added with 1 to have 33 as the grand total, the same as the Parse Tree example.

Is the abstract syntax tree of 1+2+3 identical to the one of (1+2)+3 or the one of 1+(2+3)?
Yes, the AST are the same for these three equations.

2.5 Week 5

For this week's homework, the task was to create Abstract Syntax Trees and evaluate lambda expressions.

2.5.1 Abstract Syntax Trees of Lambda Calculus

The following expressions were given to create 2-D Abstract Syntax Trees and Linearized Abstract Syntax Trees.

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

Here is the Linearized and 2-D AST for x:

```
EVar(Ident "x")
```

```
        EVar
         |
         x
```

Here is the Linearized and 2-D AST for x x:

```
EApp(EVar(Ident "x") EVar(Ident "x"))
         EApp
        /    \
       EVar  EVar
        |     |
        x     x
```

Here is the Linearized and 2-D AST for x y:

```
EApp(EVar(Ident "x") EVar(Ident "y"))
        EApp
       /    \
      EVar  EVar
       |     |
       x     y
```

Here is the Linearized and 2-D AST for x y z:

```
EApp (EApp (EVar (Ident "x")) (EVar (Ident "y"))) (EVar (Ident "z"))
       EApp
      /    \
    EVar   EApp
     |    /    \
     z   EVar  EVar
          |     |
          y     x
```

Here is the Linearized and 2-D AST for

$$\backslash x.x$$

```
EAbs (Ident "x") (EVar (Ident "x"))
            EAbs
          /     \
         x      EVar
                 |
                 x
```

Here is the Linearized and 2-D AST for

$$\backslash x.xx$$

```
EAbs (Ident "x") (EApp (EVar (Ident "x")) (EVar (Ident "x")))
             EAbs
           /     \
        EApp     EVar
        /   \      |
    EVar    EVar   x
      |       |
      x       x
```

Here is the Linearized and 2-D AST for

$$(\backslash x.(\backslash y.xy))(\backslash x.x)z$$

```
EApp (EApp (EAbs (Ident "x") (EAbs (Ident "y") (EApp (EVar (Ident "x")) (EVar (Ident "y"))))) (EAbs (Ide
```

```
           EApp
        /        \
       EApp       EVar
    /      \        |
  EAbs      EAbs    z
  /  \      /  \
 x   EAbs   x   EVar
     /  \        |
    y   EApp     x
       /    \
      EVar  EVar
       |     |
       x     y
```

Here is the Linearized and 2-D AST for

$$(\backslash x.\backslash y.xyz)abc$$

```
EApp (EApp (EApp (EAbs (Ident "x") (EAbs (Ident "y") (EApp (EApp (EVar (Ident "x")) (EVar (Ident "y")))
```

```
                  ________EApp________
                /                      \
              EApp                     EVar
            /     \                      |
          EApp    EVar                   c
        /     \    |
      EAbs   EVar  b
     /   \      \
    x    EAbs    a
        /    \
       y     EApp
            /    \
          EApp   EVar
         /   \    |
        EVar EVar z
        |     |
        x     y
```

These expressions start easy with simple free variables such as x. The problems gradually got harder to include bound variables, and would need to be changed if we were evaluating these expressions.

2.5.2 Lambda Expression Evaluation

This section is to evaluate the following Lambda Expressions and solve:

```
 1.  (\x.x) a
 2.  \x.x a
 3. (\x.\y.x) a b
```

```
4. (\x.\y.y) a b
5. (\x.\y.x) a b c
6. (\x.\y.y) a b c
7. (\x.\y.x) a (b c)
8. (\x.\y.y) a (b c)
9. (\x.\y.x) (a b) c
10. (\x.\y.y) (a b) c
11. (\x.\y.x) (a b c)
12. (\x.\y.y) (a b c)
```

Here is the evaluation for number 1:

```
(\x.x) a = a
```

Here is the evaluation for number 2:

```
\x.x a = a
```

Here is the evaluation for number 3:

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

Here is the evaluation for number 4:

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

Here is the evaluation for number 5:

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

Here is the evaluation for number 6:

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

Here is the evaluation for number 7:

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

Here is the evaluation for number 8:

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

Here is the evaluation for number 9:

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

Here is the evaluation for number 10:

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

Here is the evaluation for number 11:

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

Here is the evaluation for number 12:

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

2.5.3 evalCBN calculation

The task her was to write by hand the calculation of

```
(\x.x)((\y.y)a)
```

using evalCBN in interpreter.hs.

Clearly the answer here would simply be "a", but in terms of what the computer sees will be different

```
evalCBN (EApp (EAbs (Ident "x") (EVar (Ident "x"))) (EApp (EAbs (Ident "y") (EVar (Ident "y"))) (EVar (I
evalCBN (subst (Ident "x") (EApp (EAbs (Ident "y") (EVar (Ident "y"))) (EVar (Ident "a"))) EVar (Ident
evalCBN (EApp (EAbs (Ident "y") (EVar (Ident "y"))) (EVar (Ident "a"))) = Line 47
evalCBN (subst (Ident "y") (EVar (Ident "a")) (EVar (Ident "x"))) = Line 27
evalCBN (EVar (Ident "a")) = Line 47
EVar (Ident "a") = Line 32
```

2.6 Week 6

2.6.1 Evaluation

The goal this week was to Evaluate

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

Here is what I got:

```
(\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) (\f.\x. f (f x)) (\f2.\x2. f2 (f2 (f2 x2))))
=
((\n. (\f.\x. f (f x)) n) (\f2.\x2. f2 (f2 (f2 x2))))
=
(((\f.\x. f (f x)) (\f2.\x2. f2 (f2 (f2 x2)))))
=
(((\x. (\f2.\x2. f2 (f2 (f2 x2))) ((\f2.\x2. f2 (f2 (f2 x2))) x))))
=
(((\x. (\f2.\x2. ((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x
=
(((\x. (\f2.\x2. ((\x2. x (x (x x2)))) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x)
=
(((\x. (\f2.\x2. ((x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2))))))))))
=
(((\x. (\f2.\x2. ((x (x (x (((\x2. x (x (x x2)))) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2))))))))))
=
(((\x. (\f2.\x2. ((x (x (x (((x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) x2))))))))))))
=
(((\x. (\f2.\x2. ((x (x (x (((x (x (x (((\x2. x (x (x x2)))) x2))))))))))))
=
(((\x. (\f2.\x2. ((x (x (x (((x (x (x (((x (x (x x2)))))))))))))))
=
(((\x. (\f2.\x2. ((x (x (x (((x (x (x (((x (x (x x2)))))))))))))))
```

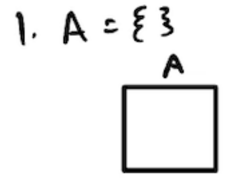
2.7 Week 7

For week 7, there were a variety of tasks. The first was to draw a picture for a number of ARSs. Next was to find an example of an ARS for specific cases. Another task was to the variable for certain lines. evalCBN for hw5 if not done, but in my case is done in Week 5 section. Lastly to use evalCBN for another lambda term.

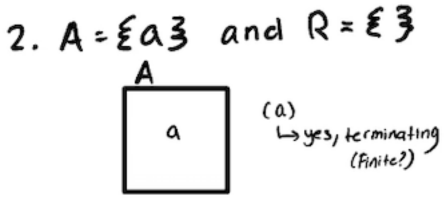
2.7.1 ARS Picture/Determination

The task in this section was to

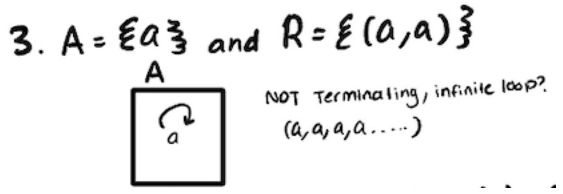
```
Consider the following list of ARSs.
1. A = {}
2. A = {a} and R = {}
3. A= {a} and R = {(a,a)}
4. A = {a,b,c} and R = {(a,b),(a,c)}
5. A = {a,b} and R = {(a,a),(a,b)}
6. A = {a,b,c} and R = {(a,b),(b,b),(a,c)}
7. A = {a,b,c} and R = {(a,b),(b,b),(a,c),(c,c)}

Draw a picture for each of the ARSs.

Is the ARS terminating? Is it confluent? Does it have unique normal forms?
```



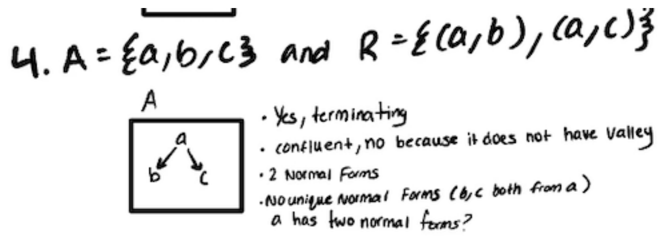
Picture 1:
It is terminating, it is confluent, and it has a unique normal form.



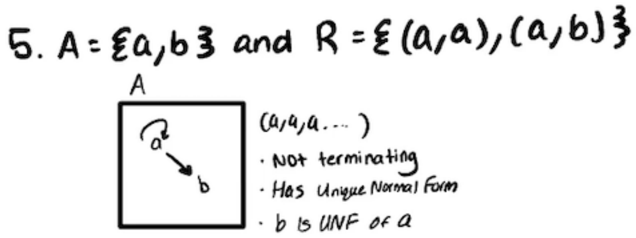
Picture 2:
It is terminating, it is confluent, and it has a unique normal form.



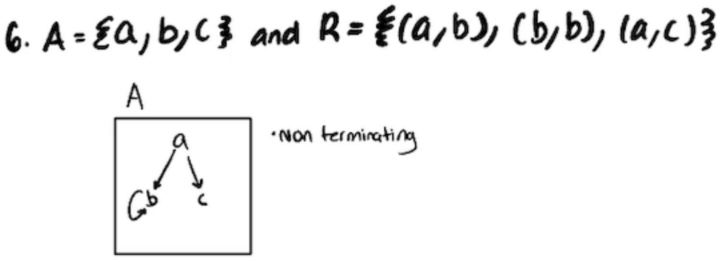
Picture 3:
It is not terminating, it is confluent, and it does not have unique normal forms.


4. A = {a, b, c} and R = {(a, b), (a, c)}
A
a
b
c
· Yes, terminating
· confluent, no because it does not have valley
· 2 Normal Forms
· No unique Normal Forms (b, c both from a)
a has two normal forms?

Picture 4:
It is terminating, it is not confluent, and it does not have unique normal forms.

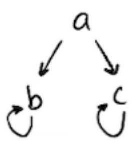


Picture 5:
It is not terminating, it is confluent, and it does have unique normal forms. b is the UNF of a.



Picture 6:
It is not terminating, it is not confluent, and it does not have unique normal forms.

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



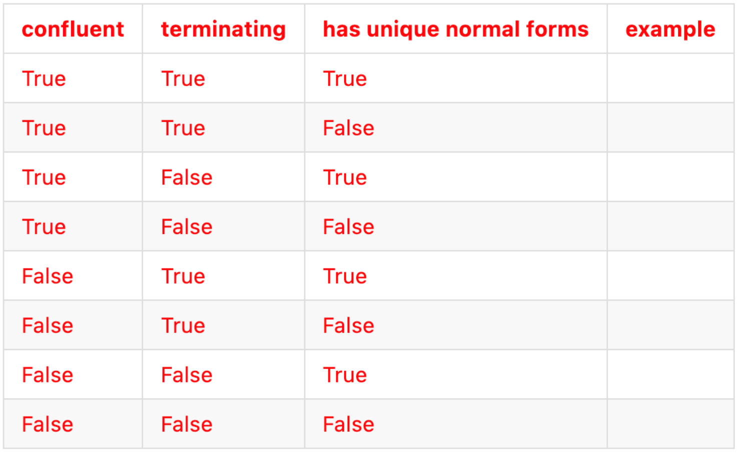
•Normal form you have to stop, so none here so also no UNF

Picture 7:
It is not terminating, it is not confluent, and it does not have unique normal forms.

2.7.2 ARS examples

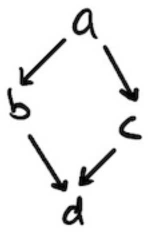
In this section of the homework, the task was to try find an example of an ARS for specific combinations, and to draw the picture of that example.

Here are the combinations that are being tested:



confluent	terminating	has unique normal forms	example
True	True	True	
True	True	False	
True	False	True	
True	False	False	
False	True	True	
False	True	False	
False	False	True	
False	False	False	

For number 1, confluent = True, terminating = True, and Unique Normal Form = True, I got the following ARS:

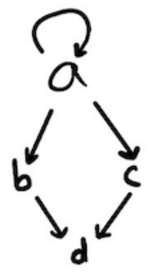


Here, confluence, terminating, and Unique Normal Form are all true. The a is the peak, and b and c reduce into d which is the valley. There is no possibility of having an infinite loop. Lastly, d is the UNF.

For number 2, confluent = True, terminating = True, and Unique Normal Form = False, I got the following ARS:

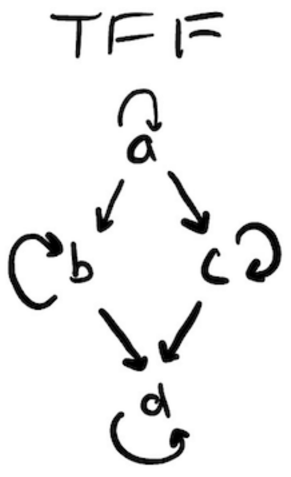
There is not an example for this situation. When Confluence and Terminating are True, that means that UNF must be true. There will not be any situation or diagram where this is not true.

For number 3, confluent = True, terminating = False, and Unique Normal Form = True, I got the following ARS:



Similar to my example in number 1, Confluence is true due to the peak existing with a valley. d would be the UNF in this case. Lastly, it is not terminating because of the possibility of an endless loop on a.

For number 4, confluent = True, terminating = False, and Unique Normal Form = False, I got the following ARS:



Here confluence is true due to the peak and valley being present. Terminating would be false because of the possibility of having an endless loop. Lastly a,b,c, and d all have a possibility of an endless loop, which means that none of them are normal forms, meaning no UNF could be possible.

For number 5, confluent = False, terminating = True, and Unique Normal Form = True, I got the following ARS:

There is no scenario that exists for this situation. When Confluence is false, UNF will never be true.

For number 6, confluent = False, terminating = True, and Unique Normal Form = False, I got the following ARS:

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b & & c \end{tikzcd}$$

Again, a is the peak, but there is no valley. This means that confluent is false for this ARS. This is terminating because a, b, and c, do not have opportunity to be part of an endless loop anywhere. This ARS would simply start at a then go to b or c. This ARS does not have a UNF because b and c are both normal forms, but a must only have one normal form to have a UNF.

For number 7, confluent = False, terminating = False, and Unique Normal Form = True, I got the following ARS:

There is not an example for this scenario. When confluence is false, UNF will always never be True.

For number 8, confluent = False, terminating = False, and Unique Normal Form = False, I got the following ARS:

$$\begin{tikzcd} & a \arrow[loop above] \arrow[dl] \arrow[dr] & \\ b & & c \end{tikzcd}$$

Similar to number 7, but instead of just b from a, there is b and c. That means that both are normal forms, but a must only have one normal form to have an unique normal form. In this case, all 3 are False.

2.7.3 Bound and Free Variables

In this section of Week 7, the task was to

```
in lines 5-7 and also in lines 18-22 explain for each variable

- whether it is bound or free
- if it is bound say what  the binder and the scope of the variable are
```

Here are lines 5-7 that are referred to:

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

The variables e1 and e2 are both bound variables. Their scope goes from the line 5 to line 7. The reason they are bound variables is because the rest of the this function uses e1 and e2 because of there instance in

```
 evalCBN (EApp e1 e2)
```

which tells us that the other lines are bound by the occurrence of e1 and e2.

The binder of the variables e1 and e2 is

```
(EApp e3 e2)
```

The variable i and e3 (on line 6) is a bound variable. The scope is just the line it is on, which is line 6,

```
evalCBN (subst i e2 e3)
```

The binder is

```
(EAbs i e3).
```

The second instance of e3, which is different than the one mentioned on line 6, has a scope of

```
EApp e3 e2
```

and a binder of

```
e3
```

Here are the lines 18-22 that are referred to:

```
subst id s (EAbs id1 e1) =
    let f = fresh (EAbs id1 e1)
        e2 = subst id1 (EVar f) e1 in
        EAbs f (subst id s e2)
```

For the variable id, s, id1, and e1, they are all bound variables. The scope would be lines 18-22, and the binder for s would be

```
id s (EAbs id1 e1)
```

The variable f would also be a bound variable. The scope would be from lines 19 to 22. The binder would be

```
let f
```

the variable e2 would also be a bound variable. The scope would be from line 20-22. The binder would be

```
e2
```

The binder for id1 and e1 would be

```
 (EAbs id1 e1)
```

2.7.4 evalCBN calculation, part 2

The task here was to use evalCBN and subst from Interpreter.hs, and solve by hand showing how to computer processes this information and calculations.

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

This was the lambda expression given. If calculating here, the answer would simply be the second y, which would have to be substituted.

```
evalCBN (EApp (EApp (EAbs (Ident "x") (EAbs (Ident "y") (EVar (Ident "x"))))
EVar (Ident "y"))) (EVar (Ident "z"))) = From Parser

evalCBN ((EAbs (Ident "x") (EAbs (Ident "y") (EVar (Ident "x")))) (EVar (Ident "z"))
(EVar (Ident "y"))) = Line 27

evalCBN (EVar (Ident "y")) = Line 48

EVar(Ident "y") = Line 32
```

2.8 Week 8

This week the task was to examine the rewrite rules of

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

and to answer the following questions:

```
1. Why does the ARS not terminate?
2. What are the normal forms?
3. Can you change the rules so that the new ARS has unique normal forms (but still has
the same equivalence relation)?
4. What do the normal forms mean? Describe the function implemented by the ARS.
```

For **number 1**, the ARS as currently constructed would **not terminate** because the rules

```
  ba -> ab
  ab -> ba
```

These two rules are doing the opposite of each other. Every time ba transforms into ab, ab will transform back to ba. This will continue to happen, and will be an endless loop, meaning that these rewrite rules are not terminating.

For **number 2**, the normal forms would be a and b. Since aa reduces to a, and bb reduces to b, but a and b do not reduce to anything, these would be the normal forms.

number 3: Here are the rules I changed so that the new ARS has unique normal forms, and still has the same equivalence relation.

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

I simply got rid of the last rule. a is the UNF for aa, and b is the UNF for bb. Since the rewriting rules are terminating now, this will allow for UNF to exist.

number 4: The normal forms mean that it does not reduce into anything else.

2.9 Week 9

In week 9, the task were to describe 3 milestones for the project and analysis of ARSs

2.9.1 Project Milestones

For the project, there are going to be 3 main milestones that will allow this project to be done smoothly. The first milestone will be to research strengths and weaknesses about the programming language Swift. This should be fully done by November 8th, or earlier if possible. This task will require research that should take a few days, and some time to write about it in the Report section. The next milestone will be to come up with an idea of an App or Game that will show how these strengths and weaknesses affect coding in this language. This should be done November 13th, as this is a brainstorming section. When I decide, I will write about why I choose to do the App/Game, and how it is related to prior sections of the report. Lastly, I will fully build the App/Game to a working prototype by the early December (December 2nd?). I am not sure how long this will take, so I am not sure what exact deadline I should give myself. I want to have some of the App/Game done before thanksgiving break to show its progress, then to be fully done before Finals week. There will be more explaining on specific things I did in my code, how the app/game works, and links to where I have my code.

2.9.2 ARSs Analysis

The first set of rules were

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

aa -> b
```

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

First off, this schema of rules does not terminate. If drawn out, you would see that ba would point to ab, then ab would point back to ba. This would repeat and repeat. This happens for all of these rules, and will be stuck in an infinite loop. The top set of rules would cause issues in many ways.

This a rewrite I would do to put the letters in alphabetical order, and would allow for termination to exist.

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

Here there would be no infinite loops, and would allow for calculation to progress.

Here is an example that this works better for just the top set of rules:

```
cabbabcba
acbbabcba
abcbabcba
abbcabcba
abbacbcba
ababcbcba
aabbcbcba
aabbbccba
aabbbcbca
aabbbbcca
aabbbbcac
aabbbbacc
aabbbabcc
aabbabbcc
aababbbcc
aaabbbbcc
```

To progress further with these rules, we can see:

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

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

we would get the following, starting from the ending of aaabbbbc

```
babbbbcc
babcc
babb
ba
ab
c
```

This will fully terminate now, and even gives us a UNF in the variable c.

The rewrite I created would have confluence because the peaks will have a valley. Lastly, the invariants for the original schema of rules would be

```
 length
 number of ?
```

After the rewrite I created, the new Invariants would be

```
  I don't know
```

2.10 Week 10

The task for Week 10 was to calculate

$\text{fix}_f 2$

These are all the steps I executed:
f 2 =
= fix_f 2
= F fix_f 2
=(\ f. \ n. if n == 0 then 1 else f(n-1)*n) fix_f 2
= (if 2==0 then 1 else fix_f (2-1)*2) fix_f 2
= (fix_f 1) * 2
= (F fix_f 1) * 2
= ((\ f. \ n. if n == 0 then 1 else f(n-1)*n) fix_f 1) * 2
= ((\ f. \ n. if 1 == 0 then 1 else f(1-1) * 1) fix_f 1) * 2
= ((fix_f 0) * 1) * 2
= ((F fix_f 0) * 1) * 2
= (((\ f. \ n. if n == 0 then 1 else f(n-1)*n) fix_f 0) * 1) * 2
= (((\ f. \ n. if 0 == 0 then 1 else f(n-1)*n) fix_f 0) * 1) * 2
= ((1) * 1) * 2
= (1 * 1) * 2
= 1 * 2
= 2

The answer here would simply be 2. The F fix_f n function allows us to use the if/else conditional until n=0. Once n=0, we will have 1, then can compute the rest of the multiplication.

2.11 Week 11

For week 11, the task was to read the article, Composing Contracts: An Adventure in Financial Engineering, and to ask an interesting question and respond to two other classmate questions.

2.11.1 My Question

Q: Figure 4 is connected to our course through its definition of denotational semantics. However, this figure is hard to grasp even with our knowledge of denotational semantics. For example, the first few lines are simply adding contracts, evaluating them, or doing basic arithmetic on them. Once we reach the bottom, we have:

(E8) Ek[[one k2]] = exchk(k2)

And more cases like this. I am very confused on what is happening here. The other definitions like "give", "or", and "scale" seem much easier to understand, and their definitions prior (Figure 2) represents that. "One", "anytime", and "get", have much harder syntax in the actual software side that makes it more confusing to understand what they are trying to accomplish. My question is, could these formulas be simplified for the sake of understanding what these specific contracts are trying to accomplish first, then become more advanced as programmers learn about more and more finance contracts?

2.11.2 Responses

Classmate Eric Phan's Question: If financial experts took the time to learn the language, how beneficial would it really be? In the article it states "identifying the 'right' primitive combinators is quite a challenge." So would it be worth/necessary for financial experts to try?

My Response: I also think that it would be beneficial for the financial experts to learn this language. Overall, I believe that this language will be hard for them to learn at first, but once they understand it, the basis of the language will prove to be beneficial for them. It will make many things that they do by hand or with other calculators, and instead can use this coding language specifically for their purpose.

Classmate Cole Matsueda's Question: How do Combinator Libraries and DSL's take into account for human behavior/constraints? (Late payments, partial payments)

My Response: This is an interesting question. At first I think they had most of the basics of contracts and what not, but as the language gets more advanced as time moves on, these features should be included. Late payments, partial payments, and so on are things that happen all the time in the real world. If this language wants to be implemented by lots of financial experts, then these real-world situations must be included. I am not sure how they would implement this, but I would assume additional time metrics and more would allow for these situations.

2.12 Week 12

Week 12 was to apply the method of analysis from the lecture to

```
while (x != 0) do z:= z*y;   x:= x-1  done
```

To start, I created my pre-condition as

```
{z = n, y = k, x = m}
```

Using Hoare Logic, when a pre-condition is true P, then execute S, then Q. Looking like this:

```
{P} S {Q}
```

Using what I had as my pre-condition, I found the post-condition becomes

```
{z = (y^x) * z}
```

I found this post-condition by solving for z, no matter where in the loop.
Here is an example that this would be correct:
When z = 2, y = 3, and x = 3, the loop would execute 3 times.

```
z = 2 * 3 --> 6
z = 6 * 3 --> 18
z = 18 * 3 --> 54
```

In our case, z would be 54. Using the numbers from the start, 2,3,3 for z,y,x, and equation that would get us to 54 would be

```
(y^x) * z.
```

```
z = (3^3) * (2)
z = 27 * 2
z = 54
```

This is what we would expect. This post-condition works for all other combinations, assuming x,y,z are all 0 or positive.

At this point in the analysis, we have a pre-condition, post-condition, and the program. This is how it would look

```
{z = n, y = k, x = m} while (x != 0) do z:= z*y;   x:= x-1  done {z = (y^x) * z}
```

Thus the invariant becomes

```
{(y^x) * z = (k^m) * n}
```

This can be proven with an example

```
{(y^x) * z = (k^m) * n}
```

```
(3^3) * 2 = (3^3) * 2  --> 27 * 2 = 27 * 2  --> 54 = 54
(3^2) * 6 = (3^3) * 2  --> 9 * 6 = 27 * 2 --> 54 = 54
(3^1) * 18 = (3^3) * 2  --> 3 * 18 = 27 * 2 --> 54 = 54
```

Our final equation would then be:

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

Lastly, I wrote a small python program that confirms the calculations done by hand. Here is the program:

```
x = 3
y = 3
z = 2
while x!=0:
    print(str(z) + " = " + str(z) + " * " + str(y) + " = " + str(z*y))
    z = z * y
    x = x -1


print(z)
```

Here is the output:

```
2 = 2 * 3 = 6
6 = 6 * 3 = 18
18 = 18 * 3 = 54
54
```

3 Project

Swift, created in 2014 by Apple, is a language predominantly to build applications for computers, phones, and more. This programming language allowed for better User-Interface and easier understanding on how to build specific ideas in the application.

3.1 Specification

This project will be learning the programming language Swift. First you will find the History of Swift. Then you will find the Strengths/Weaknesses of using Swift. I will then add a relatively easy ”tutorial” section on Swift. Lastly there will be a coding project done in Swift to show its capabilities. The final coding project has not yet been decided, but I will first download the necessary applications, packages, and more. Then I will follow tutorials and watch videos about how to program in Swift.

3.2 History

The development of Swift began in 2010 by Apple Employees Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, and Joe Groff. They based their original idea off of many existing programming languages such as Haskell, Python, and more. The first release of Swift was in 2014, when Swift 1.0 came to market. Swift had been decided to be developed to replace Apple's earlier creation of a programming language called Objective-C.

Using what many things that Objective-C had created earlier, Swift built many things to advance the programming world. The main intended use for Swift was to create Applications and Games for iOS, computers, and more, mainly for Apple Products. Despite being developed by Apple, Swift can be used across many platforms, such as Windows, iOS, and macOS. Objective-C had lacked many modern languages features, due to being created in the 1980's. Developing an entire new programming language that incorporated many of Objective-C's ideas and code snippets allowed for Swift to be created relatively quick, about 4 years.

When beginning to develop Swift, Chris Lattner had noticed that popular languages such as C++ and

Java had bulit-in common things such as Integers and Strings, but did not include things like arrays and dictionaries. This idea struck Lattner, and he decided that including many built-in types and functionality in the Standard Library would be an important advancement for many future langauges, and allowed for easier programming. Another important thing that the Swift development team wanted to implement was the ability to address common programming errors such as null pointer dereferencing. Many things today are still being added to the growing Swift language, but has proven to be a popular and important innovation in programing languages. In 2015, Swift ranked as the most loved programming language in a survey amongst StackOverflow users.

3.3 Strengths and Weaknesses of Swift

This section will talk about pros and cons of using Swift.

3.3.1 Strengths

Speed
Swift has many strengths across many different programming language areas. One huge strength is the speed of the language, which is faster than Objective-C and Python. Apple even claims that Swift is 8.4 times faster than Python. Having this much of an advantage in the speed category for programming languages is important in computing. This speed superiority exists because of the ability for Swift to detect specific errors, and how their compiler was constructed. Swift was built with LLVM compiler framework, which translates to machine code from the assembly language. The Machine code then optimizes the code, which allows for much faster speeds than other highly popular programming languages.

Apple-Supported
Swift, being developed by Apple, is highly used to create Applications and Games for Apple Products. Apple is a huge brand, so having the ability to be backed by one of the biggest companies in the world, makes the language of Swift to be much more valuable to learn. The language is growing, and will continue to be supported by Apple to make it better. By learning Swift, it is much more likely that it will continue to grow in the future and will not die out. This means that people can learn and develop in Swift, without worrying that coding in Swift could become outdated.

Open-Source
Open-Source programming languages are very important and beneficial for coding. Open-Source allows for flexibility, cost-effective, and allows for better future of the language. It also allows for developers to get access to community resources, debuggers, package managers, and supporting libraries. There are popular langauges such as Java, Python, C++, and as you would expect, Swift. Being Open-Source is a huge advantage over many other programming languages that do not having Open-Source because developers will have an easier understanding on the deeper knowledge of a language, and a more lively community that discuss how to make Swift an even better language.

Simple Syntax
The ability to easily learn and use a programming language is important. Simple syntax languages such as Python are often some of the first languages developers learn due to the simplicity of the code. Swift is very similar in many aspects to Python, but also has its own uniqueness. Swift can easily be ran without having classes, a main method, and importing many things. This allows for first time users to understand the syntax much faster, and thus the language gets much more popular than harder to learn languages. Simple Syntax languages are languages that are easy to read, and do not require extra characters and commands, such as

semi-colons and more.

3.3.2 Weaknesses

Years on Market
Swift is a very young language. Many older languages that can continue to keep its popularity, such as C and C++, have large communities and developers who were exposed to using them early on. Swift has only be available for 8 years, which is quite young. As this is a weakness now, Swift is growing popularity extremely fast and will continue to grow as previously mentioned. The community for C++ is much larger, and thus more users have found bugs, corrections, and more. Swift has more to be discovered about itself, as C++ has discovered plenty. Swift is currently used to develop primarily Apple Applications, but in the future it could be discovered that it is better for other uses that the Apple developers had not thought of.

Backward Compatibilty
As Programming Languages evolve for the better, there are many times when developers are stuck on older versions due to their choice, or not updating before writing their code. However, this could be a real problem for Swift. Apple has made it clear to engage users with their newest OS, iOS, and new versions of anything, without looking for backward compatibility to be their biggest concern. This is very prevelant with Swift, as the language gets new versions, there will be some parts of the new update that will not work great with older versions. This is a problem for now, but currently Apple is creating a back deployment library now to fix these version problems. This weakness will eventually be fixed or at least better than before.

3.4 Easy-To-Learn-Swift

Some examples on Swift.

3.5 Project Overview

The project that I will be making in Swift will be an Application. The Application will consist of tracking data about where Users go during the day. It will track individual points at specific times, and will draw lines on a map about where the User went throughout the day. It will be available so that the User can easily look at a detailed map of their locations. I will use many methods to track location of the User.

3.6 Final Project

3.7 Prototype

3.8 Documentation

3.9 Critical Appraisal

...

4 Conclusions

(approx 400 words)
In the conclusion, I want a critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of programming languages and software engineering?

References

[PL] Programming Languages 2022, Chapman University, 2022.

Homework Sources
Hackmd.io Professor Kurz, Chapman University 2022

Tower of Hanoi Math Is Fun, 2021

Report Sources
History Section
Objective-C. Wikipedia, 2022
Chris Lattner's Homepage Chris Lattner, 2022
Swift About Page Apple, 2022
Introducing Swift on Windows Apple, 2022
Swift on iOS 9 to 5 Mac, Jose Adorno, 2021
Swift on macOS Gavin Wiggins, 2022
Chris Lattner Interview Hacking With Swift, 2020
Programming Languages Survey StackOverflow, 2015

Strengths/Weaknesses Section
Swift Speed Geeks For Geeks, 2021
Swift Speed Comparison Apple, 2022
Swift Compiler Altexsoft, 2021
Open-Source Pro's The Enterprisers Project, Lee Congdon, 2015
Open-Source Programming Languages Analytics Insight, 2021
Swift is now Open-Source OutSource2India
Apple building Backwards Compatibility Medium, Carlos Banos, 2021