CPSC-354 Report

Eleas Vrahnos
Chapman University

November 6, 2022

Abstract

To be written at a later date.

Contents

1 Introduction

This report is written by Eleas Vrahnos. It details all assignments and progress made in the Programming Languages course at Chapman University. It includes weekly homework assigments, programming assignments, and a final project that demonstrate understanding and application in various class topics.

2 Homework

This section will contain my solutions to the weekly homework assignments.

2.1 Week 1

The following is a Python implementation of the Euclidean algorithm:

```python
def gcd(a,b):
    while a != b:
        if a > b:
            a = a-b
        else:
            b = b-a
    return a
```

We can test this code by going through the function with a sample input `gcd(9, 33)`, step by step.

1. `gcd(9, 33)`
   - The function is called, assigning 9 to variable `a` and 33 to variable `b`.
2. `while a != b:`
   - The while loop condition returns True, so the loop starts.
3. `else:`
   - `a > b` ($9 > 33$) returns False, so the else block executes.
4. `b = b-a`
   - `b` is now assigned to $33 - 9$, which is 24.
5. `while a != b:`
   - The while loop condition returns True, so the loop starts.
6. `else:`
   - `a > b` ($9 > 24$) returns False, so the else block executes.
7. `b = b-a`
   - `b` is now assigned to $24 - 9$, which is 15.
8. `while a != b:`
   - The while loop condition returns True, so the loop starts.
9. `else:`
   - `a > b` ($9 > 15$) returns False, so the else block executes.
10. `b = b-a`
    - `b` is now assigned to $15 - 9$, which is 6.
11. `while a != b:`
    - The while loop condition returns True, so the loop starts.
12. `if a > b:`
    - `a > b` ($9 > 6$) returns True, so the first block executes.
13. `a = a-b`
    - `a` is now assigned to $9 - 6$, which is 3.
14. `while a != b:`
    - The while loop condition returns True, so the loop starts.
15. `else:`
    - `a > b` ($3 > 6$) returns False, so the else block executes.
16. `b = b-a`
    - `b` is now assigned to $6 - 3$, which is 3.
17. `while a != b:`
    - The while loop condition returns False (`3 == 3`), so the loop ends.
18. `return a`
    - `a` is returned from the function, giving the correct greatest common divisor of **3**.

2.2 Week 2

The following are implementations of various functions in Haskell.

select_evens, lists the even-indexed elements of a given list:

```haskell
-- Implementation
select_evens [] = [] -- in the case of a list with even number elements
select_evens (x:[]) = [] -- in the case of a list with odd number elements
select_evens (x:y:xs) = y : select_evens (xs)

-- Execution Sequence with example ["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"]
```

select_odds, lists the odd-indexed elements of a given list:

```haskell
-- Implementation
select_odds [] = [] -- in the case of a list with even number elements
select_odds (x:[]) = [x] -- in the case of a list with odd number elements
select_odds (x:y:xs) = x : select_odds (xs)

-- Execution Sequence with example ["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"]
```

member, determines whether an element is part of a given list:

```haskell
-- Implementation
member a [] = False
member a (x:xs)
    | a==x = True
    | otherwise = member a (xs)

-- Execution Sequence with example 2 [5,2,6]
member 2 [5,2,6] =
    member 2 [2,6] =
    True
```

append, appends a list to another list:

```
-- Implementation
append [] ys = ys
append (x:xs) ys = x : append xs ys

-- Execution Sequence with example [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]
```

revert, reverses a list:

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

-- Execution Sequence with example [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]
```

less_equal, checks if the element in a list is less than or equal to the same-indexed element in another list:

```
-- Implementation
less_equal [] [] = True
less_equal (x:xs) (y:ys)
    | x > y = False
    | otherwise = less_equal (xs) (ys)

-- Execution Sequence with example [1,2,3] [2,3,2]
less_equal [1,2,3] [2,3,2] =
    less_equal [2,3] [3,2] =
    less_equal [3] [2] =
    False
```

2.3   Week 3

The following investigates the Tower of Hanoi problem. Here is a given Haskell implementation describing moves in the game, as well as the execution sequence for the test input `hanoi 5 0 2`.

```
-- Implementation
hanoi 1 x y = move x y

hanoi (n+1) x y =
    hanoi n x (other x y)
    move x y
    hanoi n (other x y) y

-- Execution Sequence
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
```

From this execution, the moves for a 5-ring Tower of Hanoi game can be seen as follows:

```
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
```

Analysis: From this computation, the word `hanoi` appears exactly 31 times in the execution. Based on executions of the game with a different number of starting rings, the formula $2^n - 1$ can be derived to determine how many times `hanoi` will appear, with `n` being the number of disks in the game.
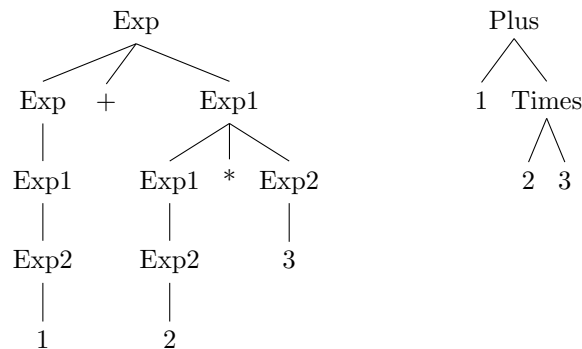
2.4   Week 4

The following compares concrete and abstract syntax trees of various mathematical expressions.
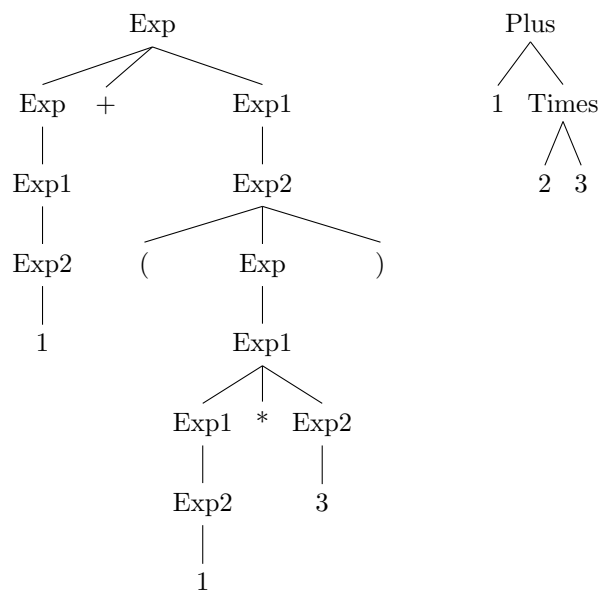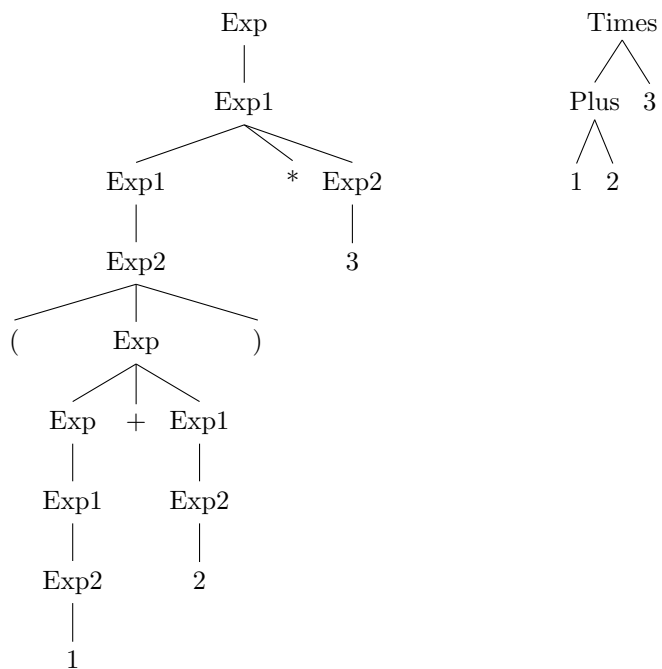
The expression $2 + 1$:
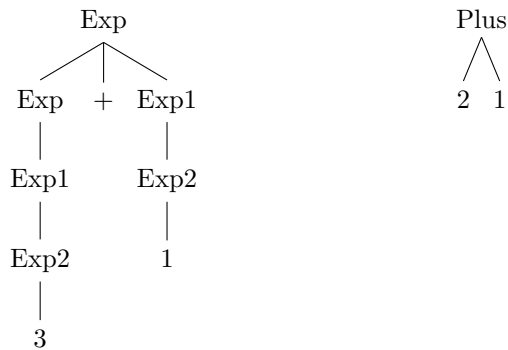
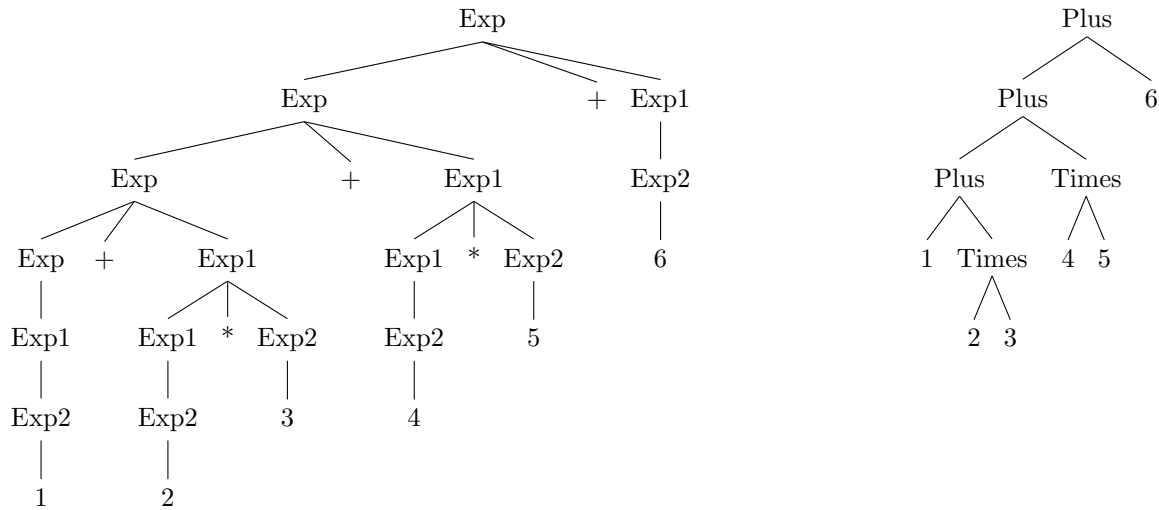
The expression $1 + 2 * 3$:

The expression $1 + (2 * 3)$:

```
                        Exp                              Plus
               /         |         \                     /  \
           Exp    +           Exp1                      1   Times
            |                  |                              /  \
          Exp1              Exp2                             2    3
            |              /    |    \
          Exp2          (     Exp     )
            |                  |
            1               Exp1
                          /   |   \
                      Exp1    *   Exp2
                       |            |
                     Exp2           3
                       |
                       1
```

The expression $(1 + 2) * 3$:

```
                       Exp                            Times
                        |                              /  \
                      Exp1                          Plus   3
                   /    |    \                       /\
               Exp1     *   Exp2                    1  2
                |              |
              Exp2             3
             /   |   \
           (    Exp    )
               / | \
           Exp  +  Exp1
            |        |
          Exp1     Exp2
            |        |
          Exp2       2
            |
            1
```

8

The expression $1 + 2 * 3 + 4 * 5 + 6$:

```
                          Exp                                                    Plus
              Exp                    +   Exp1                              Plus          6
        Exp          +    Exp1           Exp2                        Plus        Times
   Exp   +    Exp1      Exp1  *  Exp2      6                      1    Times     4   5
   Exp1     Exp1  * Exp2  Exp2    5                                   2   3
   Exp2     Exp2   3       4
    1        2
```

Analysis of the abstract syntax tree of $1 + 2 + 3$: The abstract syntax tree of $1 + 2 + 3$ would match the tree of $(1 + 2) + 3$. This is because the first breakdown of $+$ separates it to `Exp` and `Exp1`, and `Exp1` cannot reduce down to another sum. Therefore, the right side of the tree must become an integer, while the left side reduces down to a sum. The resulting tree would be as follows, which matches $(1 + 2) + 3$ and not $1 + (2 + 3)$.

```
                    Plus(Exp)
         Plus(Exp)            3(Exp1)
   1(Exp)   2(Exp1)
```
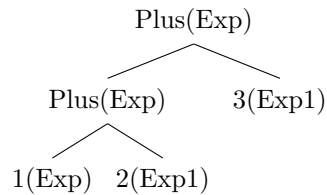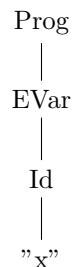
2.5   Week 5

After generating a working parser demonstrating lambda calculus, linearized abstract syntax trees and 2-dimensional notation abstract syntax trees can be generated for the below expressions.
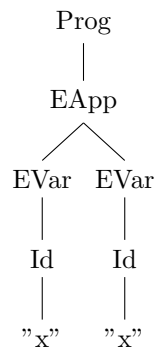
```
-- x
x
Prog (EVar (Id "x"))
```

```
        Prog
         |
        EVar
         |
         Id
         |
        "x"
```

```
-- x x
x x
Prog (EApp (EVar (Id "x")) (EVar (Id "x")))
```
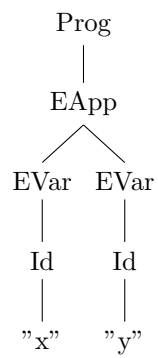
```
                        Prog
                         |
                        EApp
                        /  \
                    EVar    EVar
                     |       |
                     Id      Id
                     |       |
                    "x"     "x"
```

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

```
                        Prog
                         |
                        EApp
                        /  \
                    EVar    EVar
                     |       |
                     Id      Id
                     |       |
                    "x"     "y"
```
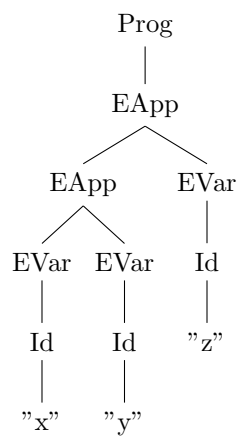
```
-- x y z
x y z
Prog (EApp (EApp (EVar (Id "x")) (EVar (Id "y"))) (EVar (Id "z")))
```

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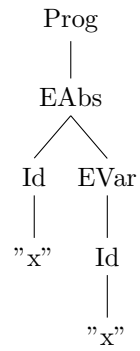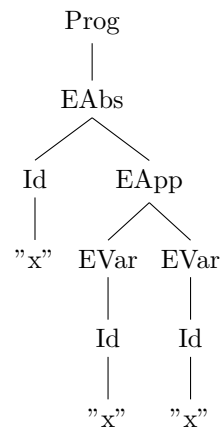
```
-- \ x.x
\ x . x
Prog (EAbs (Id "x") (EVar (Id "x")))
```

```
                    Prog
                     |
                    EAbs
                    /  \
                  Id    EVar
                  |      |
                 "x"     Id
                         |
                        "x"
```

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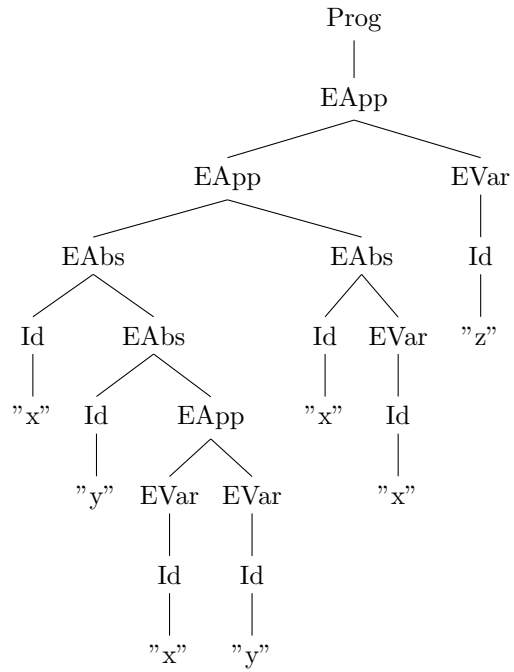
```
                    Prog
                     |
                    EAbs
                    /  \
                  Id    EApp
                  |     /  \
                 "x" EVar  EVar
                       |     |
                      Id    Id
                       |     |
               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

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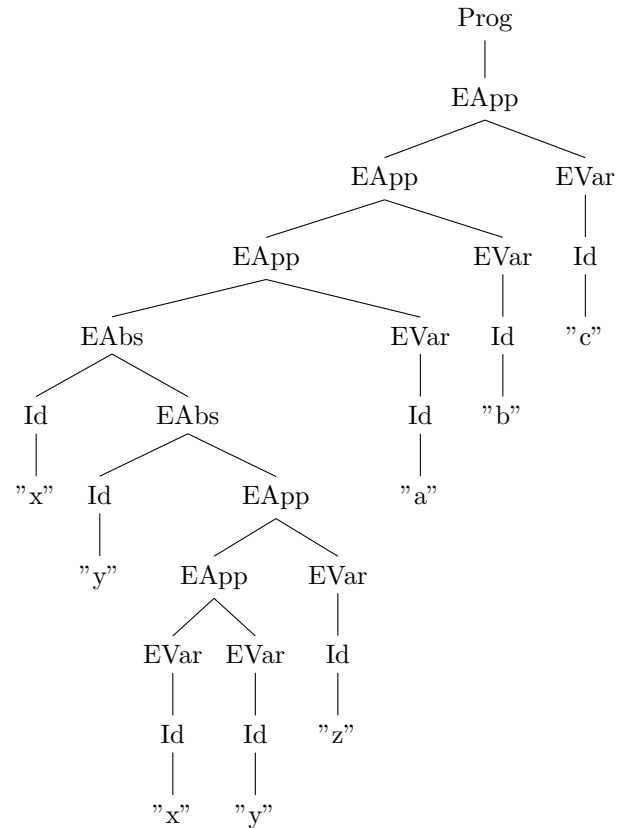
```
                                    Prog
                                     |
                                    EApp
                                   /    \
                              EApp       EVar
                            /      \        \
                        EAbs        EAbs 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

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


The following will show the reduction of several lambda calculus expressions.

```
(\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
```

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

```
evalCBN (\x.x)((\y.y)a) =
    evalCBN (EApp (EAbs (Id "x") (EVar (Id "x"))) (EApp (EAbs (Id "y") (EVar (Id "y"))) (EVar (Id
        "a")))) = -- converted to concrete format
    evalCBN (subst (Id "x") (EApp (EAbs (Id "y") (EVar (Id "y"))) (EVar (Id "a"))) (EVar (Id
        "x"))) = -- line 27
    evalCBN (EApp (EAbs (Id "y") (EVar (Id "y"))) (EVar (Id "a"))) = -- reduction of subst in one
        step
    evalCBN (subst (Id "y") (EVar (Id "a")) (EVar (Id "y"))) = -- line 27
    EVar (Id "a") -- reduction of subst in one step
```

2.6 Week 6

The following is an evaluation of a longer lambda calculus expression.

```
-- (\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)) (\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. (\x2. ((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2
       (f2 x2))) x) x2))))
   = (\x. (\x2. (x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
   = (\x. (\x2. (x (x (x (((\x2. x (x (x x2)))) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
   = (\x. (\x2. (x (x (x ((\x2. x (x (x x2))) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
   = (\x. (\x2. (x (x (x (x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) x2))))))))
   = (\x. (\x2. (x (x (x (x (x (x ((\x2. x (x (x x2))) x2)))))))))
   = \x. \x2. x (x (x (x (x (x (x (x (x x2)))))))))
```

2.7 Week 7

This section will investigate bound and free variables in a lambda-calculus interpreter.
The following is a section of the interpreter code.

```
evalCBN (EApp e1 e2) = case (evalCBN e1) of -- line 5
(EAbs i e3) -> evalCBN (subst i e2 e3)      -- line 6
e3 -> EApp e3 e2                            -- line 7
```

In this code, all variables (`e1`, `e2`, `e3`, and `i`) are **bound variables**. This is because the variable names can be changed without changing the function `evalCBN`.

```
e1:
   Binder: EApp e1 e2
   Scope: case (evalCBN e1) of | (EAbs i e3) -> evalCBN (subst i e2 e3) | e3 -> EApp e3 e2
e2:
   Binder: EApp e1 e2
   Scope: case (evalCBN e1) of | (EAbs i e3) -> evalCBN (subst i e2 e3) | e3 -> EApp e3 e2
e3:
   Binder (line 6): EAbs i e3
   Scope (line 6): subst i e2 e3
   Binder (line 7): e3
   Scope (line 7): EApp e3 e2
i:
   Binder: EAbs i e3
   Scope: subst i e2 e3
```

The following is another section of the interpreter code.

```
subst id s (EAbs id1 e1) =          -- line 18
let f = fresh (EAbs id1 e1)         -- line 20
    e2 = subst id1 (EVar f) e1 in -- line 21
    EAbs f (subst id s e2)          -- line 22
```

In this code, all variables (`id`, `s`, `id1`, `e1`, `f`, and `e2`) are **bound variables**. This is because the variable names can be changed without changing the function `subst`.

`id`:

    Binder: `subst id s (EAbs id1 e1)`

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

`s`:

    Binder: `subst id s (EAbs id1 e1)`

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

`id1`:

    Binder: `EAbs id1 e1`

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

`e1`:

    Binder: `EAbs id1 e1`

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

`f`:

    Binder: `f = fresh (EAbs id1 e1)`

    Scope: `| e2 = subst id1 (EVar f) e1 in | EAbs f (subst id s e2)`

`e2`:

    Binder: `e2 = subst id1 (EVar f) e1`

    Scope: `EAbs f (subst id s e2)`

Another example of evalCBN is demonstrated here:

```
evalCBN (\x.\y.x) y z =
   evalCBN (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "y"))) (EVar (Id
      "z"))) = -- converted to concrete format
   evalCBN (EApp (EApp (EAbs (Id "x") (EAbs (Id "y0") (EVar (Id "x")))) (EVar (Id "y"))) (EVar
      (Id "z"))) = -- fresh applied in one step
   evalCBN (EApp (EApp (evalCBN (subst (Id "y0") (EVar (Id "y")) (EVar (Id "x")))) (EVar (Id
      "y"))) (EVar (Id "z"))) = -- line 6
   evalCBN (EApp (EAbs (Id "y0") (EVar (Id "y"))) (EVar (Id "z"))) = -- subst reduction
   evalCBN (EApp (evalCBN (subst (Id "y0") (EVar (Id "z")) (EVar (Id "y")))) (EVar (Id "z"))) =
      -- line 6
   evalCBN (EVar (Id "y")) = -- subst reduction
   EVar (Id "y") -- evalCBN x = x
```

The following is an analysis of several abstract reduction systems.

`A={}`

A picture will not be drawn for this, for it is just an empty set. This ARS is terminating because it cannot reduce any further. It is confluent because every peak converges to the same result. It has a unique normal form (which is the empty set).

`A={a} and R={}`

A picture will not be drawn for this, for it is just the letter "a". This ARS is terminating because it cannot reduce any further. It is confluent because every peak converges to the same result. It has a unique normal form (which is "a").

`A={a} and R={(a,a)}`

This ARS is not terminating because it can always reduce further. It is confluent because every peak converges to the same result. It has a unique normal form (which is "a").

`A={a,b,c} and R={(a,b),(a,c)}`
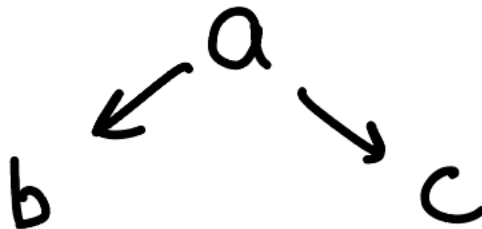
This ARS is terminating because it cannot reduce any further. It is not confluent since the computations do not converge back. It does not have a unique normal form.
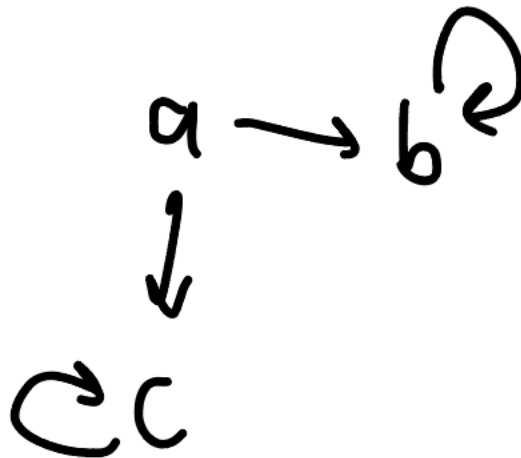
`A={a,b} and R={(a,a),(a,b)}`
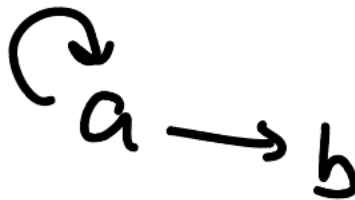
This ARS is not terminating because it can always reduce further. It is not confluent since the computations do not converge back. It does not have a unique normal form.
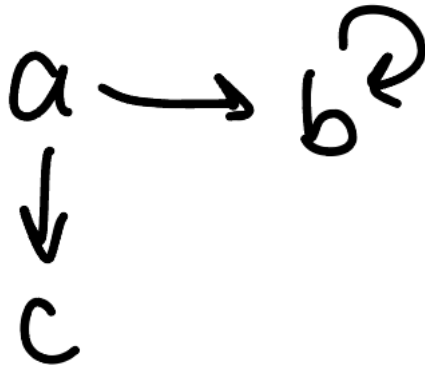
A={a,b,c} and R={(a,b),(b,b),(a,c)}

This ARS is not terminating because it can always reduce further. It is not confluent since the computations do not converge back. It does not have a unique normal form.

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

This ARS is not terminating because it can always reduce further. It is not confluent since the computations do not converge back. It does not have a unique normal form.
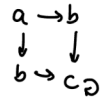
Lastly, the 8 different cases of confluence, termination, and presence of unique normal forms will be shown.
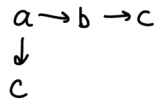
True, True, True:

$$
\begin{array}{ccc}
a & \to & b \\
\downarrow & & \downarrow \\
b & \to & c
\end{array}
$$
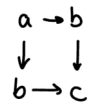
True, True, False: Congruence implies that there are unique normal forms, so there is no example of this case.

True, False, True:

$$
\begin{array}{ccc}
a & \to & b \\
\downarrow & & \downarrow \\
b & \to & c \circlearrowright
\end{array}
$$

True, False, False: Congruence implies that there are unique normal forms, so there is no example of this case.

False, True, True:

$$
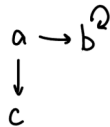\begin{array}{ccc}
a & \to & b \to c \\
\downarrow & & \\
c & &
\end{array}
$$

False, True, False:

$$
\begin{array}{cc}
a & \to b \\
\downarrow & \\
c &
\end{array}
$$

False, False, True:

$$
a \longrightarrow b \circlearrowright
$$

False, False, False:

$$
\begin{array}{cc}
a & \to b \circlearrowright \\
\downarrow & \\
c &
\end{array}
$$

2.8 Week 8

The following will investigate properties of the following rewrite system:

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

Termination: This ARS does not terminate because of the final two rules. There is a cycle between the strings `ab` and `ba`, meaning that there is no end to the cycle if there exists either of these strings.

Normal forms: The normal forms of this ARS are therefore an empty string, `a`, and `b`. Any string longer than one character with just `ab` and `ba` will be rewritten as another same-length string, as per the rewrite rules.

Unique normal forms: These rewrite rules can be slightly altered so that the ARS can be defined as having unique normal forms. If we treat `a` and `b` as the base-10 digits `0` and `1`, it has a slightly different meaning. Because base-10 numbers inherently have an order (0 before 1), these new rewrite rules will allow constant reduction to a normal form not by length, but by number size.

ARS function: The normal forms of this updated rewrite system would thus be the lowest number possible that can be reduced from the starting string. In terms of `a` and `b`, the resulting string would have all `a`'s to the left and all `b`'s to the right. However, this letter form does not have much meaning, as the letters do not have a predefined comparison order, and thus cannot be counted as a normal form. The normal forms in terms of `0` and `1` has more meaning.

2.9 Week 9

Regarding the final project, the following deadlines will be made to keep track of progress of my programming language exploration:

November 13: Demonstrate full knowledge of the Ruby language by having a prototype of a concise Ruby tutorial copmlete.

November 20: Have an idea for the project portion of the report laid out in detail.

November 27: Have said project complete and full within the report.

December 4: Have the critical appraisal completed, and therefore the entirety of the Project section of the report complete.

The next section will involve an analysis of the following ARS:

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


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

The first thing to note is that with this current version of the ARS, it is not terminating, nor does order of the letters matter. This is because the top half of the ARS does not show the entire ARS to be a complete measure function, and that any combination of letters can be reoriented within the entire string.

Also to note are some normal forms, which include the empty string (from Rules 9 and 10) and the one-character strings (a, b, c). The first thing of notice is that any string larger than 1 character can be reduced to a form without any as. This is because of the reordering rules. Say we order our string so that all as appear first, followed by bs and cs. Generally, This would turn every pair of as into bs, with any lone as therefore joining with newly created bs to form cs. This observation holds for smaller strings as well, of course including those strings that don't contain a to begin with.

After noticing all strings (except for a) can be reduced to a string without as, I noticed that if the resulting string has an even number of bs and $C mod 4 == 0$ (where C is the number of cs in the string), then the string will reduce to an empty string. This is dependent on Rule 10 and Rule 12, saying that if all cs eventually reduce to an even number of bs, and there were originally an even number of bs already present in the string, then everything will reduce to an empty string.

2.10 Week 10

The following is a calculation of $fix_F 2$ using equational reasoning.

```
fix_F 2 = (\n. if n == 0 then 1 else n * fix_F (n-1)) 2
        = if 2 == 0 then 1 else 2 * fix_F (2-1)
        = 2 * (fix_F 1)
        = 2 * (\n. if n == 0 then 1 else n * fix_F (n-1)) 1
        = 2 * (if 1 == 0 then 1 else 1 * fix_F (1-1))
        = 2 * (1 * fix_F 0)
        = 2 * (1 * (\n. if n == 0 then 1 else n * fix_F (n-1)) 0)
        = 2 * (1 * (if 0 == 0 then 1 else 0 * fix_F (0-1)))
        = 2 * (1 * (1))
        = 2
```

3 Project

This section will contain all details for my final project of this course.

3.1 Specification

For my final project, I plan to learn a new programming language, give a concise but informative tutorial on it, and develop a project that ties in to the course material. For this, I plan to learn Ruby. Learning a new popular language could be beneficial for personal purposes. My goal is to both learn a relevant language and utilize some concepts learned in the course to create a meaningful project that represents my knowledge of programming languages. The representative project of choice will be determined in the near future.

4 Conclusions

To be written at a later date.