CPSC-354 Report

Eleas Vrahnos
Chapman University

October 2, 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:

```
-- 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:

```
-- 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:

```
-- 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 $2 + 1$:
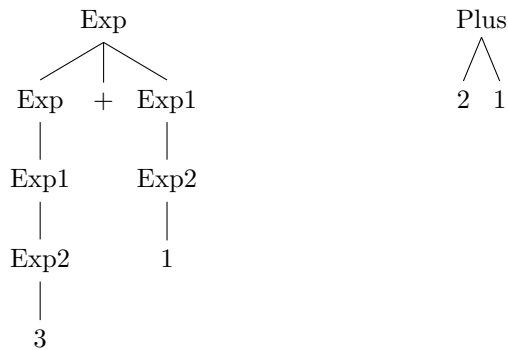
```
              Exp                        Plus
          /    |    \                    /  \
        Exp   +   Exp1                  2    1
         |          |
        Exp1       Exp2
         |          |
        Exp2        1
         |
         3
```

The expression $1 + 2 * 3$:

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

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
```

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

Exp
 ├─ Exp
 │   ├─ Exp
 │   │   ├─ Exp + Exp1
 │   │   │   ├─ Exp1   * Exp2
 │   │   │   └─ Exp2   3
 │   │   Exp1   Exp2
 │   │   Exp2   2
 │   │   1
 │   + Exp1
 │   ├─ Exp1 * Exp2
 │   Exp2   5
 │   4
 + Exp1
    Exp2
     6

Plus
 ├─ Plus   6
 │   ├─ Plus   Times
 │   │   1 Times 4 5
 │   │   2 3
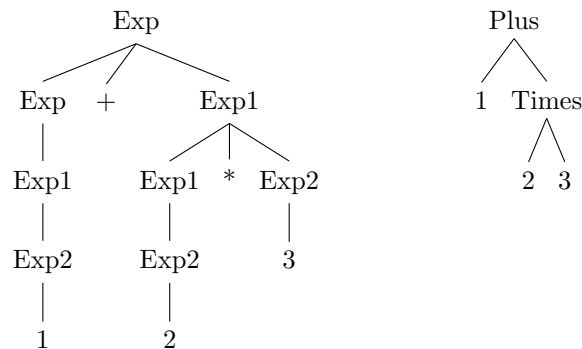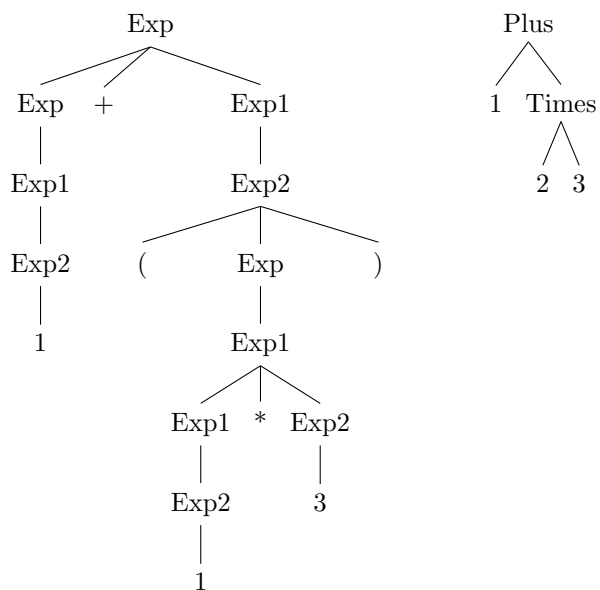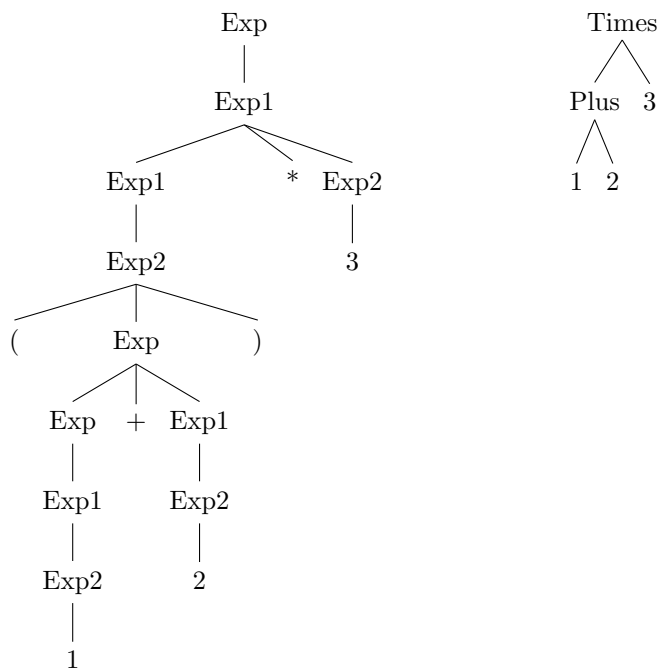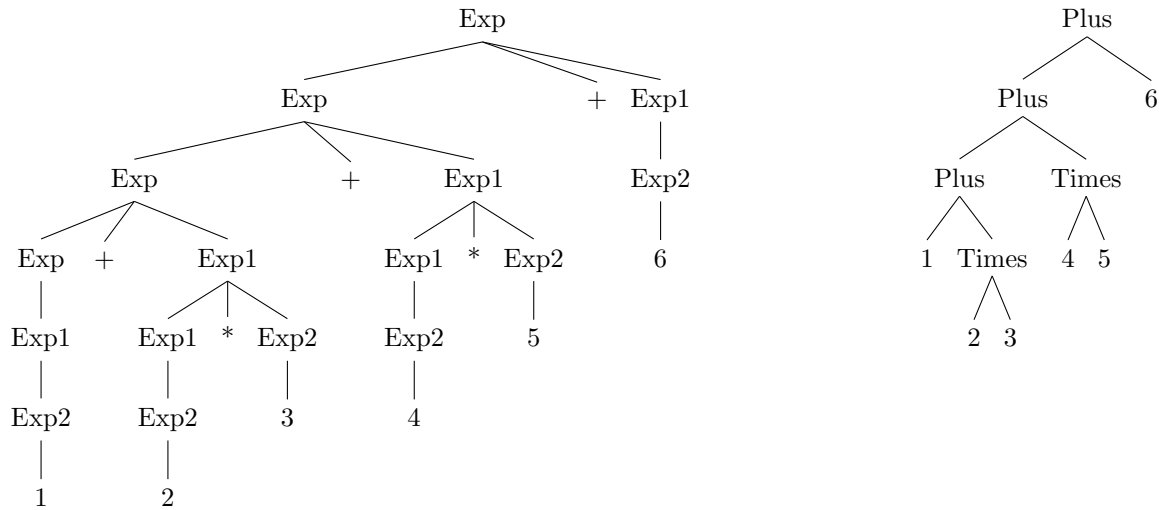
Analysis of the abstract syntax tree of $1 + 2 + 3$: The abstract syntax tree of $1 + 2 + 3$ would match the tree of $(1 + 2) + 3$. This is because the first breakdown of $+$ separates it to `Exp` and `Exp1`, and `Exp1` cannot reduce down to another sum. Therefore, the right side of the tree must become 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)
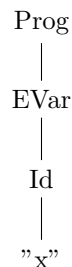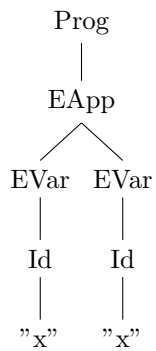 │   ├─ 1(Exp) 2(Exp1)
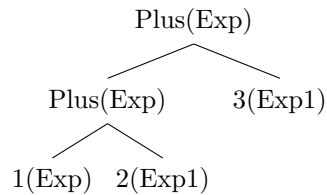
2.5 Week 5

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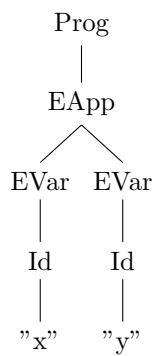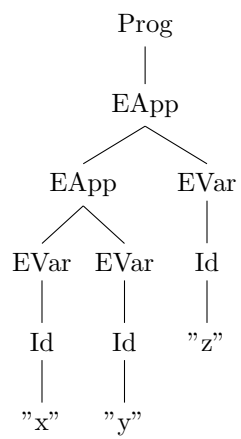
```
-- x
x
Prog (EVar (Id "x"))
```

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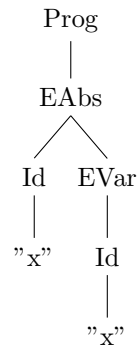
```
-- \ x.x x
\ x . x x
Prog (EAbs (Id "x") (EApp (EVar (Id "x")) (EVar (Id "x"))))
```
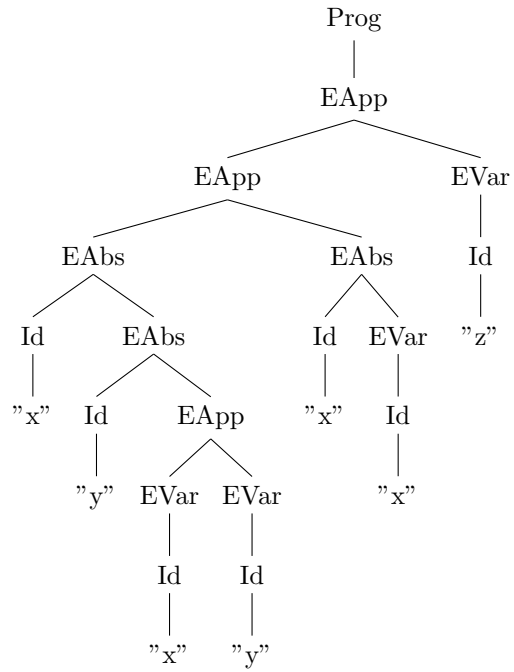
```
                        Prog
                         |
                        EAbs
                        /    \
                      Id      EApp
                       |      /    \
                      "x"  EVar    EVar
                            |        |
                           Id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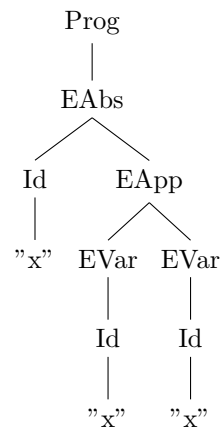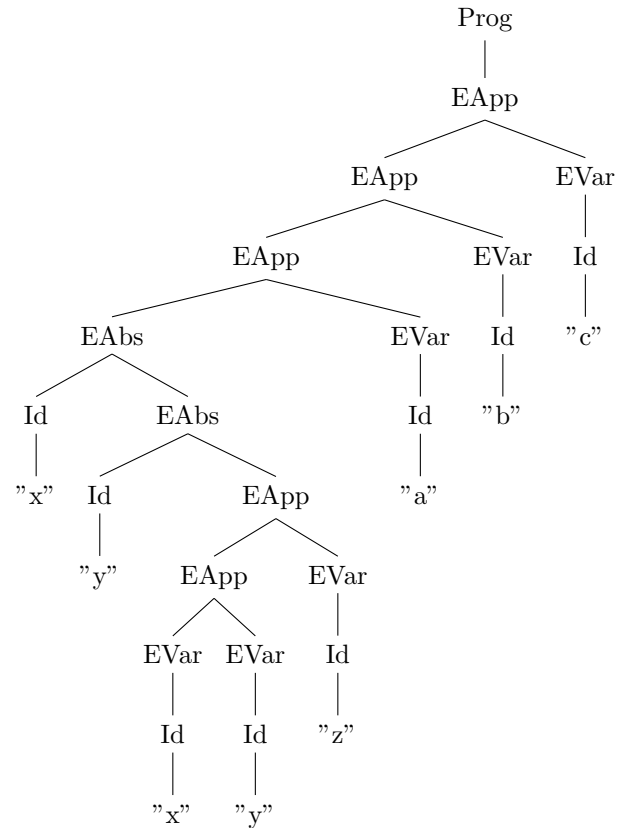
```
-- (\ x . (\ y . x y)) (\ x.x) z
\ x . \ y . x y (\ x . x)z
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EVar (Id "x")) (EVar (Id "y"))))) (EAbs (Id
    "x") (EVar (Id "x")))) (EVar (Id "z")))
```

```
                                    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")))) =
   evalCBN (subst (Id "x") (EApp (EAbs (Id "y") (EVar (Id "y"))) (EVar (Id "a"))) (EVar (Id
      "x"))) =
   evalCBN (EApp (EAbs (Id "y") (EVar (Id "y"))) (EVar (Id "a"))) =
   evalCBN (subst (Id "y") (EVar (Id "a")) (EVar (Id "y"))) =
   EVar (Id "a")
```

3   Project

To be written at a later date.

4   Conclusions

To be written at a later date.