CPSC-354 Report

Eleas Vrahnos
Chapman University

December 16, 2022

Abstract

Every computer needs their own set of instructions and syntax in order to communicate with users and other computers. Programming languages are these sets of instructions that instruct a computer how to operate. Programming languages operate all kinds of computers in the modern world, including those found in automobiles and microwaves. The most popular programming languages used by developers and programmers today include Python, JavaScript, and C++. Even though these languages are different in many ways, they all originated from their own basic building blocks. These building blocks make each programming language unique in their use and functionality. For example, the difference between interpreted vs. compiled languages can lead to tradeoffs between syntax simplicity and performance issues. Each programming language has their own advantages and disadvantages, and the way they are structured determines what strengths and weaknesses it will have. This report aims to highlight what these building blocks are and how they can be used to build a fully functional programming language. This will be done through explorations of various concepts such as syntax trees, abstract reduction systems, and recursion applications. A final software project will also be showcased at the end, investigating how the Ruby programming language compares to other languages.

Contents

1   Introduction

This report is written by Eleas Vrahnos, a third-year undergraduate at Chapman University at the time of this report's completion. It details all assignments and progress made in the Programming Languages course, CPSC 354. It includes weekly homework assigments and a final project that demonstrate understanding and application in various class topics covered that week. Almost all of these exercises are coded in the functional programming language Haskell, which is used to help make creating an example programming language interpreter as simple as possible.

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, a method for computing the greatest common divisor between two integers. A sample input `gcd(9, 33)` is tested step by step below.

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

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 and corresponding examples. For the execution sequences, equational reasoning is shown in the comments by line number of the implementation.

`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"]) -- line 3
    = "b" : ("d" : (select_evens["e"])) -- line 3
    = "b" : ("d" : ([])) -- line 2
    = ["b","d"] -- line 1
```

`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"]) -- line 3
    = "a" : ("c" : (select_odds["e"])) -- line 3
    = "a" : ("c" : ("e")) -- line 2
    = ["a","c","e"] -- syntax of lists
```

`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] -- lines 2 -> 4
    = True -- lines 2 -> 3
```

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]) -- line 2
    = 1 : (2 : (append [] [3,4,5])) -- line 2
    = 1 : (2 : ([3,4,5])) -- line 1
    = [1,2,3,4,5] -- syntax of lists
```

revert, reverses a list (makes use of append):

```
-- 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] -- line 2
    = append (append (revert [3]) [2]) [1] -- line 2
    = append (append (append (revert []) [3]) [2]) [1] -- line 2
    = append (append (append [] [3]) [2]) [1] -- line 1
    = append (append [3] [2]) [1] -- append line 1
    = append (3 : (append [] [2])) [1] -- append line 2
    = append (3 : [2]) [1] -- append line 1
    = append [3,2] [1] -- syntax of lists
    = 3 : (append [2] [1]) -- append line 2
    = 3 : (2 : (append [] [1])) -- append line 2
    = 3 : (2 : [1]) -- append line 1
    = [3,2,1] -- syntax of lists
```

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] -- lines 2 -> 4
    = less_equal [3] [2] -- lines 2 -> 4
    = False -- lines 2 -> 3
```

2.3   Week 3

The following investigates the Tower of Hanoi problem in Haskell and the execution of a 5-ring game.

```
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
```

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 rings in the game. This is also seen to be the optimal amount of moves to solve the Tower of Hanoi problem with `n` starting rings.

2.4 Week 4

The following compares concrete and abstract syntax trees of various mathematical expressions.

Shown below is the context-free grammar used for the concrete syntax trees.

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

The expression $2 + 1$:

```
            Exp                           Plus
          / |  \                         /    \
       Exp  +   Exp1                   Num     Num
        |        |                      |       |
       Exp1     Exp2                     2       1
        |        |
       Exp2    Integer
        |        |
     Integer     1
        |
        2
```

The expression $1 + 2 * 3$:

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

8

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

```
                    Exp                              Plus
           ┌─────────┼─────────┐                ┌─────┴─────┐
          Exp        +        Exp1             Num        Times
           │                   │                │       ┌───┴───┐
          Exp1                Exp2              1       Num     Num
           │            ┌──────┼──────┐                 │       │
          Exp2         (      Exp      )                2       3
           │                   │
        Integer               Exp1
           │              ┌────┼────┐
           1            Exp1   *   Exp2
                          │          │
                        Exp2      Integer
                          │          │
                       Integer       3
                          │
                          1
```

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

```
                    Exp                              Times
                     │                          ┌─────┴─────┐
                    Exp1                        Plus        Num
           ┌─────────┼─────────┐            ┌────┴────┐      │
          Exp1       *        Exp2         Num       Num     3
           │                   │            │         │
          Exp2              Integer         1         2
     ┌─────┼─────┐             │
    (     Exp     )            3
           │
      ┌────┼────┐
     Exp   +   Exp1
      │          │
     Exp1      Exp2
      │          │
     Exp2     Integer
      │          │
   Integer       2
      │
      1
```

9

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

```
                          Exp                                                 Plus
                 _____|_____                                   _____|_____
                Exp         +    Exp1                             Plus            Num
           _____|_____          |                          ____|____            |
          Exp     +    Exp1      Exp2                      Plus     Times          6
        ___|___        _|_        |                      __|__      __|__
       Exp + Exp1   Exp1 * Exp2 Integer                 Num  Times Num Num
        |          __|__    |      |                     |   __|__  |   |
      Exp1        Exp1 * Exp2  Integer                   1  Num Num 4   5
        |          |     |     |                             |   |
      Exp2        Exp2 Integer                               2   3
        |          |     |
    Integer     Integer  3
        |          |
        1          2
```

(tree structure showing: Exp → Exp + Exp1; Exp1 → Exp2 → Integer 6; Exp → Exp + Exp1; Exp1 → Exp1 * Exp2 → Integer 5, Exp2 → Integer 4; Exp → Exp + Exp1; Exp1 → Exp1 * Exp2 → Integer 3, Exp2 → Integer 2; Exp → Exp1 → Exp2 → Integer 1)

(second tree: Plus → Plus Num(6); Plus → Plus Times; Plus → Num(1) Times(Num 2, Num 3); Times → Num(4) Num(5))

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)      Num3(Exp1)
          _____|_____
      Num1(Exp)   Num2(Exp1)
```
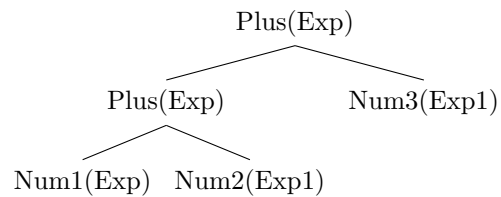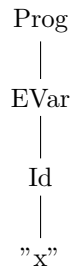
2.5 Week 5

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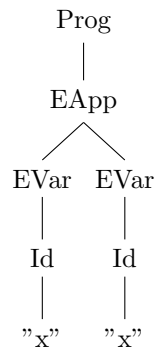
```
-- x
x
Prog (EVar (Id "x"))
```

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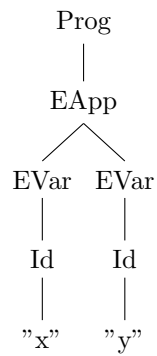
```
-- x x
x x
Prog (EApp (EVar (Id "x")) (EVar (Id "x")))
```

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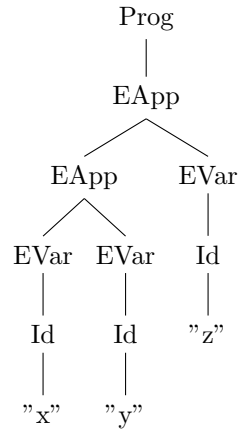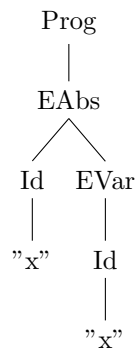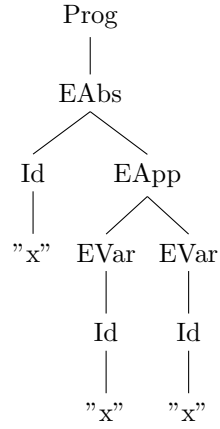
```
-- \ x.x
\ x . x
Prog (EAbs (Id "x") (EVar (Id "x")))
```

```
                   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"
```

13
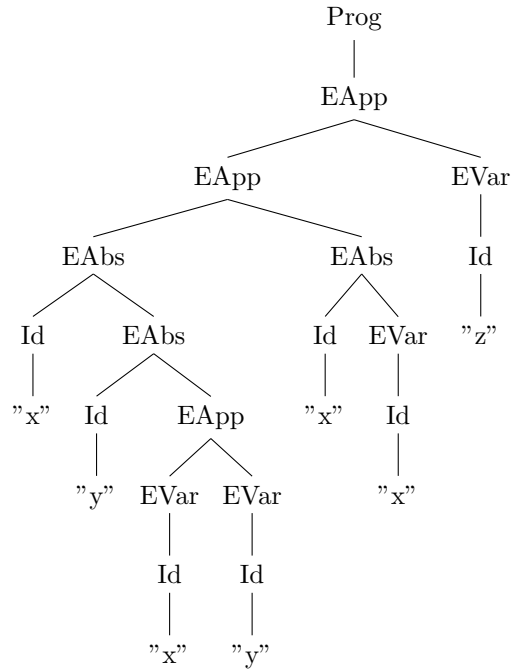
```
-- (\ x . \ y . x y z) a b c
\ x . \ y . x y z a b c
Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EApp (EVar (Id "x")) (EVar (Id "y")))
     (EVar (Id "z")))))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))
```

```
                                        Prog
                                          |
                                        EApp
                                       /      \
                                  EApp          EVar
                                 /     \           |
                            EApp        EVar       Id
                           /    \         |         |
                       EAbs      EVar    Id       "c"
                      /    \       |      |
                    Id     EAbs   Id    "b"
                     |     /   \    |
                   "x"   Id    EApp "a"
                          |    /    \
                        "y"  EApp    EVar
                            /   \      |
                       EVar    EVar   Id
                         |       |     |
                        Id      Id   "z"
                         |       |
                       "x"     "y"
```
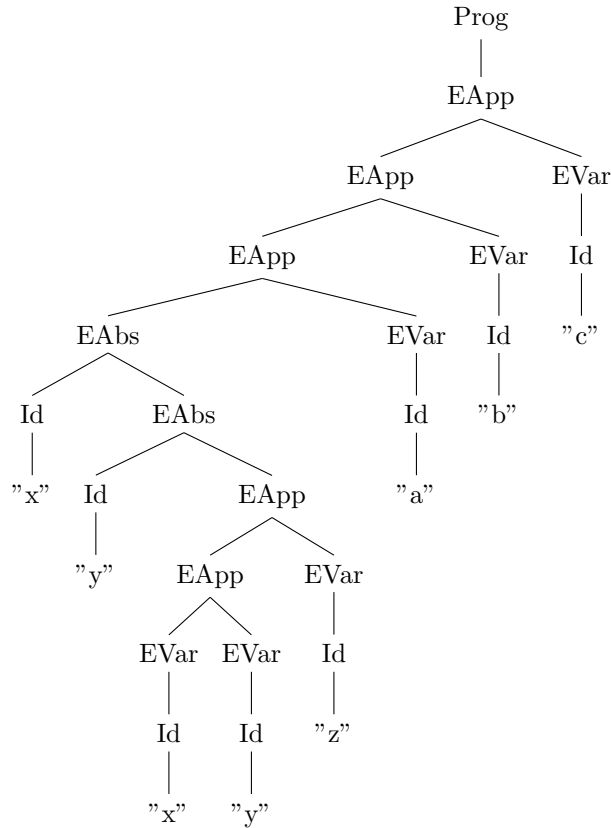
The following will show the reduction of several lambda calculus expressions, with the last reduction using the `evalCBN` method from provided interpreter code.

```
(\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"))) -- provided interpreter 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"))) -- provided interpreter line 27
    = evalCBN (EVar (Id "a")) -- reduction of subst in one step
    = a -- line 32
```

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 (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
evalCBN x = x                                -- line 8
```

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

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

`e2`:
Binder: `evalCBN (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): `evalCBN (subst i e2 e3)`
Binder (line 7): `e3`
Scope (line 7): `EApp e3 e2`

`i`:
Binder: `EAbs i e3`
Scope: `evalCBN (subst i e2 e3)`

`x`:
Binder: `evalCBN x`
Scope: `x`

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: 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)

e1:
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)

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

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

Another example of `evalCBN` from a provided interpreter snippet is demonstrated here, now including the above snippet for `fresh`:

```
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 (evalCBN (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id
       "y")))) (EVar (Id "z"))) -- line 5
   = evalCBN (EApp (evalCBN (subst (Id "x") (EVar (Id "y")) (EAbs (Id "y") (EVar (Id "x")))))
       (EVar (Id "z"))) -- line 6
   = evalCBN (EApp (evalCBN (EAbs (Id "y0") (subst (Id "x") (EVar (Id "y")) (subst (Id "y") (EVar
       (Id "y0")) (EVar (Id "x")))))) (EVar (Id "z"))) -- line 22
   = evalCBN (EApp (evalCBN (EAbs (Id "y0") (subst (Id "x") (EVar (Id "y")) (EVar (Id "x")))))
       (EVar (Id "z"))) -- line 16
   = evalCBN (EApp (evalCBN (EAbs (Id "y0") (EVar (Id "y")))) (EVar (Id "z"))) -- line 15
   = evalCBN (EApp (EAbs (Id "y0") (EVar (Id "y"))) (EVar (Id "z"))) -- line 8
   = evalCBN (subst (Id "y0") (EVar (Id "z")) (EVar (Id "y"))) -- line 6
   = evalCBN (EVar (Id "y")) -- line 16
   = y -- line 8
```

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 an empty set cannot reduce any further. It is confluent because there is no divergence of elements. It has unique normal forms (which is the empty set).


`A={a} and R={}`
A picture will not be drawn for this, for it is just `a`. This ARS is terminating because `a` cannot reduce any further. It is confluent because there is no divergence of elements. It has unique normal forms (which is `a`).

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

This ARS is not terminating because `a` can always reduce to itself. It is confluent because the divergence of `a` can converge back to itself. It does not have unique normal forms, as `a` can always reduce to itself.

`A={a,b,c} and R={(a,b),(a,c)}`
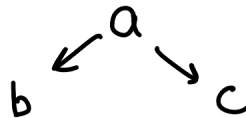
This ARS is terminating because it cannot reduce any further than `b` or `c`. It is not confluent since the divergence to `b` and `c` cannot converge back. It does not have unique normal forms, as there are two normal forms of `a` (`b` and `c`).

`A={a,b} and R={(a,a),(a,b)}`
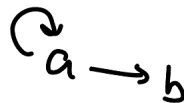
This ARS is not terminating because `a` can always reduce further. It is confluent because the divergence to `a` and `b` can converge back, with the recursive `a` converging back to the original `b`. It has unique normal forms, as `a` has the unique normal form of `b`.

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

$$a \longrightarrow b \circlearrowright$$
$$\downarrow$$
$$c$$
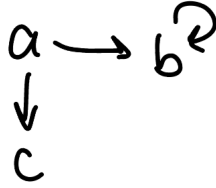
This ARS is not terminating because b can always reduce further. It is not confluent since the divergence to b and c cannot converge back. It does not have unique normal forms, as there are two normal forms of a (b and c).

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

$$a \longrightarrow b \circlearrowright$$
$$\downarrow$$
$$\circlearrowright c$$
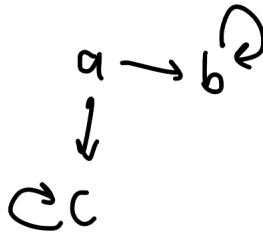
This ARS is not terminating because b and c can always reduce further. It is not confluent since the divergence to b and c cannot converge back. It does not have unique normal forms, as there are two normal forms of a (b and c).

Lastly, the 8 different cases of confluence, termination, and presence of unique normal forms are shown below.

Confluent	Terminating	Has unique normal forms	Example
True	True	True	a → b
True	True	False	If an ARS is confluent and terminating then all elements reduce to a unique normal form.
True	False	True	a ↻ → b
True	False	False	a → b ↻
False	True	True	An ARS has unique normal forms if (and only if) it is confluent and normalising.
False	True	False	b ← a → c
False	False	True	An ARS has unique normal forms if (and only if) it is confluent and normalising.
False	False	False	b ← a ↻ → c

21

2.8 Week 8

The following will investigate properties of the following rewrite system:

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

Why does the ARS not terminate?:   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.

What are the 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.

Modify the ARS so that it is terminating, has unique normal forms: These rewrite rules can be slightly altered so that the ARS is terminating and has unique normal forms. By removing the fourth rule, it eliminates the possibility of the original termination problem. What this does is allow a combination of `a` and `b` through only one permutation. This would end up adding another normal form to the previously mentioned three, which is `ab`.

This works because the functionality of the ARS is preserved. Going through the 4 possible combinations of two-letter strings, `aa` and `bb` reduce in the same fashion. In order to prevent termination, the third or fourth rule should be eliminated. The fourth rule was chosen to be eliminated here to keep the alphabetical orderliness of `ab`. Therefore, the third rule reduces the same and the elimination of the fourth rule causes `ab` to be a normal form.

Describe the specification implemented by the ARS: The specification can be found by seeing the patterns that emerge from reduction using this ARS. This can be done by coming up with possible invariants. Looking at the rules, it can be guessed that any string with the normal form `a` or `b` only contained the letter `a`/`b` in the beginning string, respectively. If a string contained both `a` and `b`, the third rewrite rule would be used eventually, leading to the normal form of `ab`.

Therefore, an invariant that can be concluded is that **the original string only contained a's if the normal form of the string is** `a`. The same invariant applies to `b` as well. Likewise, **the original string contained a's and b's if the normal form of the string is** `ab`. Finally, **the original string contained neither a's nor b's if the normal form of the string is an empty string**. All of these invariants assume that the only possible letters that can appear in the string are `a` and `b`.

The specification can then be constructed to describe the ARS as a function that determines the presence of `a`'s and the presence of `b`'s at the same time. The 4 possible combinations of `a`'s and `b`'s presences can be matched up to the 4 possible normal forms of the ARS.

2.9 Week 9

This section involves an analysis of the following ARS:

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

The first thing to note is that with this ARS, the order of the letters do not matter. Any combination of letters can be reoriented within the entire string, hinting that the specification has something to do with commutativity. The second thing to note is that this ARS is terminating with 4 normal forms: a, b, c, and the empty list. This can be shown since the second half of the rules reduce any two-letter string to a one- or zero-letter string, and this cycle would continue until one of the 4 normal forms are reached.

This can be formalized as an important invariant, which is that every two-letter combination yields a single character. The empty list will be treated as a single character for now, as this is important in the explanation.

Combining these two observations and the invariant, it hints at arithmetic, namely addition and modulo. Using this as a guide, the letters in this ARS can be treated as base-10 numbers instead, with $a = 1$, $b = 2$, and $c = 3$. Pairs of letters together will be interpreted as the addition of those two numbers, with aa acting as $a + a$ as an example. This helps decipher the first half of the ruleset, as it reinforces the fact that adding two letters together can be done in either order (commutativity), and it will yield the same result in the second half of the ruleset.

The 4 normal forms can also be treated as these base-10 numbers, with the empty list acting as 0. Doing this to the second half of the ruleset, we get the following substitution:

```
1 + 1 -> 2
1 + 2 -> 3
1 + 3 -> 0
2 + 2 -> 0
3 + 2 -> 1
3 + 3 -> 2
```

Seeing this substitution, this can be seen to follow the pattern of mod 4. Therefore, the specification for this ARS is that it calculates n mod 4, with n being the sum of a given word's letters, as if they were treated as base-10 numbers. This takes advantage of another hidden invariant, which is the following:

(((x+y) mod 4)+z) mod 4 = (x+y+z) mod 4, $\forall x, y, z \in \mathbb{N}$

To give a longer example, abcb should reduce to the empty list, as $(1 + 2 + 3 + 2) mod 4 = 0$. We can see this does indeed work with the following possible reductions.

```
Reduction 1: abcb -> ccb -> bb -> []
Reduction 2: abcb -> ccb -> ca -> ac -> []
Reduction 3: abcb -> acbb -> aab -> bb -> []
Reduction 4: abcb -> acbb -> aab -> ac -> []
Reduction 5: abcb -> aba -> ca -> ac -> []
Reduction 6: abcb -> aba -> aab -> ac -> []
```

2.10   Week 10

The following is a calculation involving a fixed-point combinator and its computation rule. The calculation solves $fix_F 2$ using equational reasoning.

```
fix_F 2 = F fix_F 2                                      -- computation rule of fix_F
        = (\n. if n == 0 then 1 else n * fix_F (n-1)) 2  -- definition of F and beta reduction
        = if 2 == 0 then 1 else 2 * fix_F (2-1)          -- beta reduction of n
        = 2 * (fix_F 1)                                  -- computation rule of if/then/else
        = 2 * (F fix_F 1)                                -- computation rule of fix_F
        = 2 * (\n. if n == 0 then 1 else n * fix_F (n-1)) 1 -- definition of F and beta reduction
        = 2 * (if 1 == 0 then 1 else 1 * fix_F (1-1))    -- beta reduction of n
        = 2 * (1 * fix_F 0)                              -- computation rule of if/then/else
        = 2 * (1 * F fix_F 0)                            -- computation rule of fix_F
        = 2 * (1 * (\n. if n == 0 then 1 else n * fix_F (n-1)) 0)
                                                         -- definition of F and beta reduction
        = 2 * (1 * (if 0 == 0 then 1 else 0 * fix_F (0-1))) -- beta reduction of n
        = 2 * (1 * (1))                                  -- computation rule of if/then/else
        = 2                                              -- multiplication
```

2.11 Week 11

The following is an essay comprised of my thoughts and analysis on the world of composing financial contracts through a domain specific programming language.

Domain specific programming languages (DSL) are programming languages that are specialized to a specific application or purpose. Common applications today that include the use of DSLs are Microsoft Excel and musical notation software. Certain programming languages also serve as DSLs, such as R (statistics) and HTML (web design). The worlds of programming and finance don't seem to have much relation at first, but the number of ways where they could be co-beneficial in today's technology-driven world is only increasing. Encoding financial contracts through a DSL is seen to have many practical applications. Condensing a wide range of possible contracts into something easier to understand, as well as using these contracts to present information effectively, are some notable advantages that a DSL could have in the financial world.

There are many details that go into creating a generic financial contract. Dates, amounts, conditions, and currency are only a select few of details that make up a contract. By encoding these contracts through a DSL, the format of a contract can be standardized within its own special language. After figuring out what key details can be altered to define a contract, the DSL can describe any contract imaginable with a given set of these important details. A set of variables, or "combinators", would be used to represent these details, but the generic body of a contract would be universal throughout all contracts. This makes it easier for contracts to be conceptualized, with only the key details needing attention.

A key advantage of standardizing these contracts through a programming language is that special operations can be done on these digital contracts. For example, one can combine different contracts if a contract requires two payments at different times. This requires less knowledge and effort than composing a new contract altogether, whether physical or digital. Additionally, one can scale a contract based on another observable combinator. This means that one can define a contract's value solely based on a scaled version of another variable, meaning the value would automatically be calculated given the other variable. One can also see visual interpretations of these contracts to understand them easier, such as "presenting contracts in new graphical ways" and "providing animated simulations" (Source). This might help others to see the value in a contract easier, as well as help financial experts make important decisions based on the contracts.

Although there are many benefits to digitalizing contracts, it comes with a few questions about its practicality. Security and legality are important aspects of physical contracts today, so its digital counterpart would need to be thought out as well. Human behavior would still be an important factor in creating these contracts, so a safeguard to prevent exploitative practices would be necessary. Neutrality in many aspects should be ensured to keep fairness and visibility in this kind of program. For example, a more experienced programmer should not have some kind of advantage over a lesser experienced programmer. Additionally, a DSL's usage in other fields could be investigated more to make simplifying other processes easier. Even if learning this specific DSL perfectly had no beneficial effect to using these contracts, it can be beneficial outside of creating contracts. Programming is widely known as an essential skill in many fields, and it can still prove to be useful at least somewhere else, since financial engineering combines two big seemingly-unrelated fields. In all, a DSL can drastically expand the possibilities of one field's standards, as "a single formal description can drive all manner of automated processes" (Source).

Source: Composing Contracts: An Adventure in Financial Engineering

2.12   Week 12

The following is an analysis of the following code through the lens of Hoare logic:

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

Looking at the short program, the program will terminate only if $x > 0$ before the while loop begins. The program will also calculate $z = y^x$ if $z = 1$ before the while loop.

To find the invariant for this while loop, a table of execution can be made. The example initialization will be $x = 10$, $y = 2$, and $z = 1$. `t` will be used to count the number of iterations so far in the while loop.

t	x	y	z
0	10	2	1
1	9	2	2
2	8	2	4
...	...	...	...
9	1	2	512
10	0	2	1024

These numbers satisfy the following invariants:

- $t + x = 10$

- $z = y^t$

Using algebra, these two equations can be modified to get an invariant consisting of only `x`, `y`, and `z`: $z = y^{(10-x)}$.

This can be formalized through a Hoare triple:

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

This can be proved from the intended purpose of this for loop with the following starting Hoare triple:

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

To prove this, it must be shown that the precondition implies the invariant and that the invariant implies the postcondition.

To show that the precondition implies the invariant, it must be shown that $z = 1 \land x = n$ implies $z = y^{(n-x)}$. If $z = 1$ and $x = n$, the second equation reduces to $1 = y^0$, or $1 = 1$. This is always true.

To show that the invariant implies the postcondition, it must be shown that $z = y^{(n-x)}$ implies $z = y^n$. This is true because of the end assumption that $x = 0$ when the while loop terminates. Therefore, the first equation reduces to $z = y^n$, making both equations the same and thus the implication true.

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.

3.2   Prototype

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

November 13: Create a tutorial that covers the evolution of Ruby and the fundamentals of the programming language. The tutorial part will cover these categories: basics, control structures, and collections. Exercises and their answers will be provided.

November 20: The software project portion will cover an example blog application using the Ruby on Rails framework, going over key aspects of what it takes to create such an application with Rails (for example, show snippets of code demonstrating different functionalities). This milestone will check if a completed first draft is made, which will contain just a walkthrough section.

November 27: Have the software project complete within the tutorial. As per the previous milestone, the project will cover an example blog application using the Ruby on Rails framework. For this milestone, it will include a final version of the walkthrough, as well as final takeaways after the completion of this project.

The protoype tutorial can be found at the following link: https://hackmd.io/@evrahnos/rJyCCKdri

4   Conclusions

To be written at a later date.