

Programming Languages Report

Evelyn Lawrie
Chapman University

October 23, 2023

Abstract

This report is composed of weekly homework assignments for the Programming Languages course at Chapman University. First, an introduction describes the objectives and contents of this report. Then, further sections explore each assigned problem, week by week. The report ends in a conclusion that summarizes the goals of this project and the larger context it belongs to.

Contents

1	Introduction	2
2	Week 1	2
2.1	GCD Introduction	2
2.2	GCD Code	2
2.3	Example Computation	3
3	Week 2	3
3.1	Towers of Hanoi Introduction	3
3.2	Example Computation	3
3.3	Towers of Hanoi Formula	5
4	Week 3	6
4.1	Parsing Introduction	6
4.2	Concrete Syntax Trees	6
4.3	Abstract Syntax Trees	8
5	Week 4	10
5.1	Lambda Calculus Introduction	10
5.2	Syntax	10
5.3	Semantics	15
6	Week 5	16
6.1	Variables, Binding, Scope, and Substitution	16
6.2	Interpreting Lambda Calculus	17
7	Week 6	18
7.1	Rewriting: Introduction	18
7.2	String Rewriting Exercises	21
8	Week 7	22

1 Introduction

This report's goal is to encourage problem solving and original thinking through the implementation of several self-contained weekly homework assignments. The task entails implementing a solution to each problem in a programming language of choice and then working out a given example of the algorithm's logic. By thinking through and thoroughly understanding each step of the algorithm, a deeper understanding of the logic behind why algorithms work the way that they do is cultivated.

2 Week 1

2.1 GCD Introduction

The greatest common divisor algorithm can be understood from a geometric, algebraic, and programming perspective. The basis of this algorithm is the idea that whatever divides two numbers n and m where $n > m$ must also divide $n - m$. This can be explored geometrically by visualizing small rectangles dividing up bigger rectangles until there is no smaller common divisor that will evenly fit in rectangles. This algorithm is also referred to as Euclid's algorithm, stemming from the Greek mathematician known as the "father of geometry" [GCD].

When exploring how algebra can solve the greatest common divisor problem, this entails breaking up the problem into three cases:

- $n > m$
- $n < m$
- $n = m$

Here is the algebraic definition of Euclid's algorithm using the three cases as described above [LTX]:

$$gcd(n, m) = \begin{cases} gcd(m, m - n), & \text{if } n > m \\ gcd(n, n - m), & \text{if } n < m \\ n, & \text{if } n = m \end{cases} \quad (1)$$

2.2 GCD Code

Below is a recursive implementation of the GCD function in Java.

```
// recursive function to implement the greatest common divisor of two integers
int gcdCalc(int n, int m) {
    if (n > m) {
        return gcdCalc(m, n - m); // subtract m from n when n is bigger
    }
    else if (n < m) {
        return gcdCalc(n, m - n); // subtract n from m when m is bigger
    }
    else {
        return n; // return n when m and n are equal
    }
}
```

2.3 Example Computation

Below is an implementation of the greatest common divisor algorithm using the example numbers 9 and 33:

$$\gcd(9, 33) = \gcd(9, 33 - 9) \quad (2)$$

$$= \gcd(9, 24) \quad (3)$$

$$= \gcd(9, 24 - 9) \quad (4)$$

$$= \gcd(9, 15) \quad (5)$$

$$= \gcd(9, 15 - 9) \quad (6)$$

$$= \gcd(9, 6) \quad (7)$$

$$= \gcd(9 - 6, 6) \quad (8)$$

$$= \gcd(3, 6) \quad (9)$$

$$= \gcd(3, 6 - 3) \quad (10)$$

$$= \gcd(3, 3) \quad (11)$$

$$= 3 \quad (12)$$

3 Week 2

3.1 Towers of Hanoi Introduction

Towers of Hanoi is a mathematical game in which disks are stacked in ascending order on the far left of three rods. The aim of this puzzle is to move these disks to the far right rod. For each move of one rod at a time, the rule must be obeyed that no disk can be placed onto a disk that is smaller than itself [WIK]. The minimal number of moves required to find a solution given n number of disks is $2^n - 1$ [WIK].

In the calculation below, *hanoi n x y* is describing a function call to move n disks from rod x to rod y . The far right rod is denoted with a 0, the middle as 1, and the right with a 2. This problem can be approached through recursion, with the function defined as:

$$\text{hanoi } 1 \ x \ y = \text{move } x \ y \quad (13)$$

$$\text{hanoi}(n + 1) \ x \ y = \text{hanoi } n \ x \ (\text{other } x \ y) \quad (14)$$

$$\text{move } x \ y \quad (15)$$

$$\text{hanoi } n \ (\text{other } x \ y) \ y \quad (16)$$

The function *move x y* is a function that takes the top disk from position x and moves it to position y . The term *other x y* is the third rod, neither x nor y .

3.2 Example Computation

Below are the calculations of the hanoi function with the example of 5 rings moving from left to right:

```
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

```

Here are the 31 moves present in the above calculation:

1. move 0 2
2. move 0 1
3. move 2 1
4. move 0 2
5. move 1 0
6. move 1 2
7. move 0 2
8. move 0 1

9. move 2 1
10. move 2 0
11. move 1 0
12. move 2 1
13. move 0 2
14. move 0 1
15. move 2 1
16. move 0 2
17. move 1 0
18. move 1 2
19. move 0 2
20. move 1 0
21. move 2 1
22. move 2 0
23. move 1 0
24. move 1 2
25. move 0 2
26. move 0 1
27. move 2 1
28. move 0 2
29. move 1 0
30. move 1 2
31. move 0 2

3.3 Towers of Hanoi Formula

The word *hanoi* appears 31 times in the computation above. This is the same frequency as the number of moves per n amount of disks. Therefore, the formula for amount of function calls necessary for the computation of a given number of disks is:

$$numFunctionCalls(numDisks) = numMoves(numDisks) = 2^{numDisks} - 1 \quad (17)$$

4 Week 3

4.1 Parsing Introduction

Concrete syntax is defined as written code while abstract syntax is the representation of code, which is what the code means to the machine. Abstract syntax can be visualized with trees, and it is made up of algebraic/recursive data types. Parsing is the process of translating concrete syntax into abstract syntax. Some programming languages skip the parsing step and are composed of abstract syntax rather than concrete syntax. Parsing analyzes symbols using a parser generator to see whether they follow formal grammar rules, like the ones outlined by BNFC or other formal grammar structures.

4.2 Concrete Syntax Trees

Given the context-free grammar:

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

The task is to implement concrete syntax trees for the example strings given. The grammar above is a set of rules that defines a language. In this case, that language is the set of arithmetic expressions using times and plus. Below are the written out parse trees:

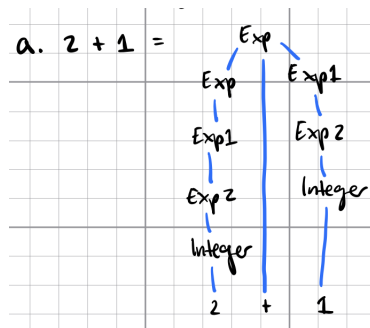


Figure 1: $2 + 1$

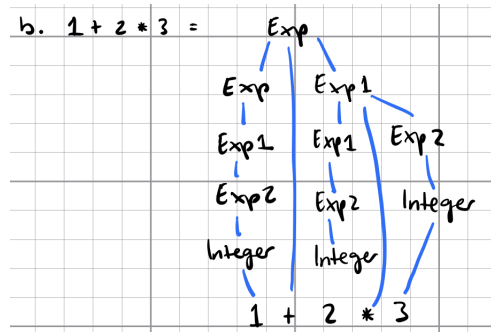


Figure 2: $1 + 2 * 3$

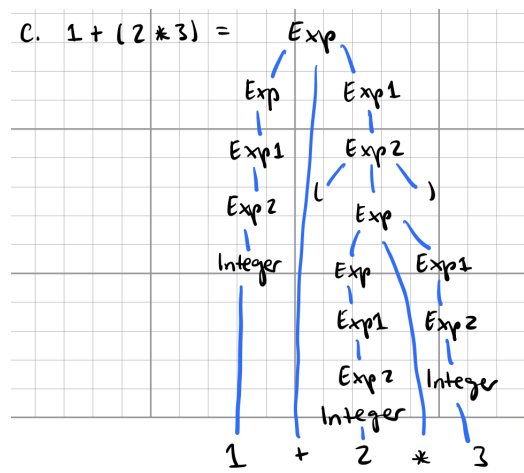


Figure 3: $1 + (2 * 3)$

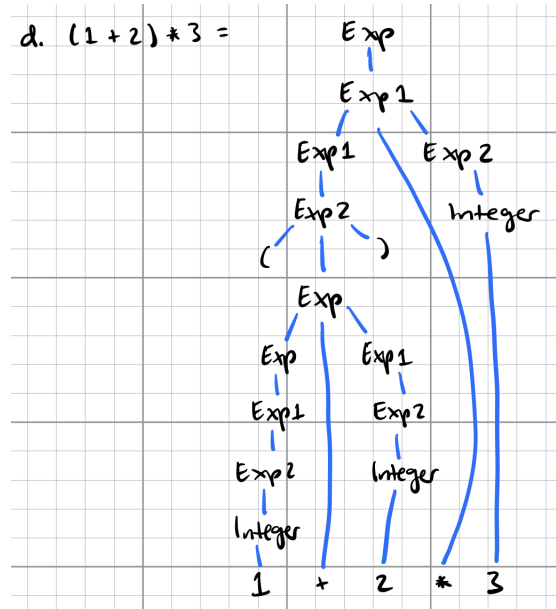


Figure 4: $(1 + 2) * 3$

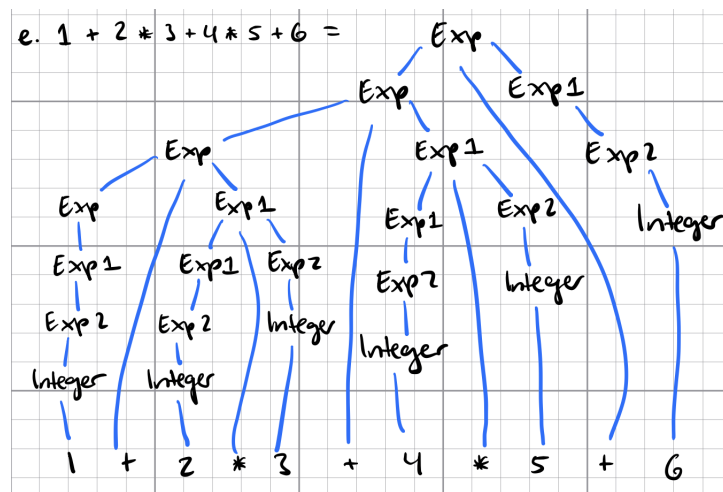


Figure 5: $1 + 2 * 3 + 4 * 5 + 6$

4.3 Abstract Syntax Trees

Given the BNFC grammar:

```

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

```

```

coercions Exp 2 ;

```

We are tasked with writing out the abstract syntax trees for the example arithmetic equations that follow the rules of the BNFC grammar defined above. BNFC comes with its own language, LNBF, which is a language for writing context-free grammars. The above set of rules is an example of what grammar looks like in BNFC and it represents the structure for the set of arithmetic equations that use plus and times. The following trees are written as abstract syntax trees. These trees differ from concrete syntax trees in three ways: the unnamed rules are eliminated, parent nodes are labeled with the names of the corresponding rules, and concrete syntax (such as symbols like "+" or "*") are eliminated.

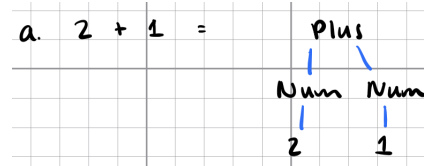


Figure 6: $2 + 1$

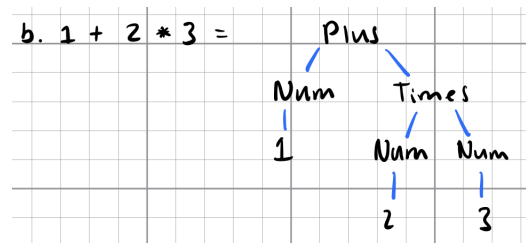


Figure 7: $1 + 2 * 3$

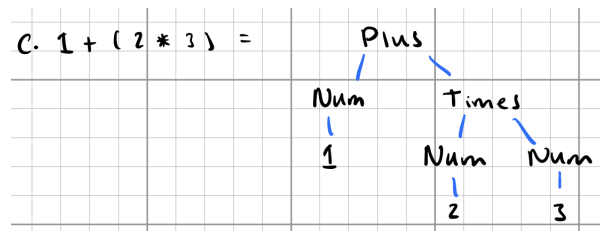


Figure 8: $1 + (2 * 3)$

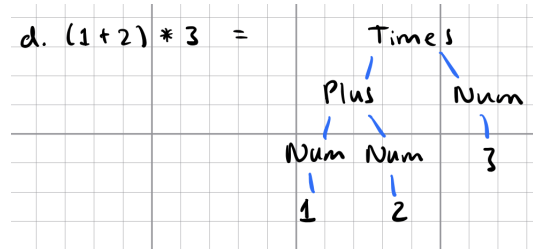


Figure 9: $(1 + 2) * 3$

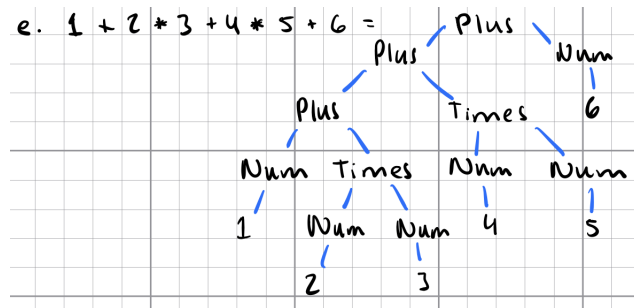


Figure 10: $1 + 2 * 3 + 4 * 5 + 6$

Since parenthesis are not taken into account when writing out the nodes of abstract syntax trees, the tree for the equation $1 + 2 * 3$ is identical to the one for the equation $1 + (2 * 3)$. The first equation follows the order of operations, in which multiplication is evaluated before addition. The second tree puts parenthesis around the multiplication terms, which visually illustrates which part of the equation to evaluate first, even though the equation is evaluated the same without the parenthesis given the order of operations. $(1 + 2) * 3$ manipulates the order of operations, causing the addition to be evaluated first and therefore diverging from the calculation of the first equation without any parenthesis.

5 Week 4

5.1 Lambda Calculus Introduction

Lambda calculus only has three programming constructs. These are abstraction, application, and variables. According to abstraction, a program called e containing the variable x will look like $\lambda x.e$ in Lambda calculus. Application outlines how if $e1$ and $e2$ are programs, $e1e2$ is the program that applies the function $e1$ to argument $e2$. Finally, the most basic programs in Lambda calculus only consist of variables. Lambda calculus abstracts away a lot of the syntax used in most other programming languages. The report this week focuses on the syntax and semantics of Lambda calculus.

5.2 Syntax

For the syntax portion of this week's homework, we are tasked with using the grammar of BNFC outlined for Lambda calculus to output the linearized abstract syntax trees from the following expressions, as well as manually write out the abstract syntax trees for them. Below are the calculations for the given Lambda calculus expressions:

a. $x =$

```

      Var
      |
      Id
      |
      x
  
```

```

(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "x" | ./TestLambda
Parse Successful!
[Abstract Syntax]
Var (Id "x")
[Linearized tree]
x
  
```

Figure 11: x

b. $x \ x =$

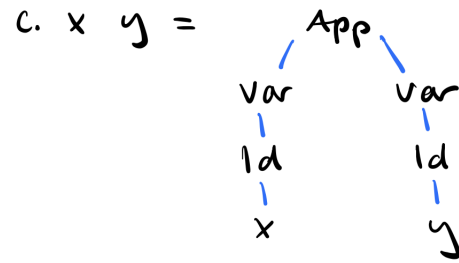
```

      App
     /  \
    Var  Var
    |    |
    Id   Id
    |    |
    x    x
  
```

```

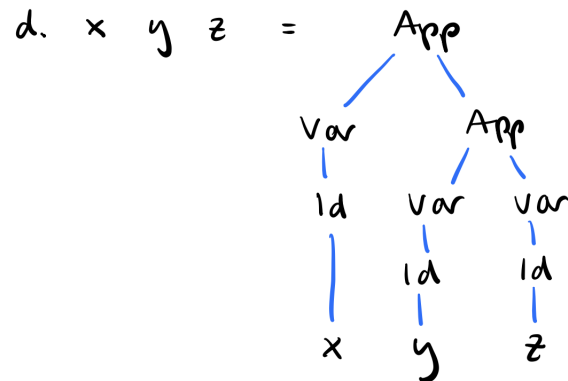
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "x x" | ./TestLambda
Parse Successful!
[Abstract Syntax]
App (Var (Id "x")) (Var (Id "x"))
[Linearized tree]
x x
  
```

Figure 12: x x



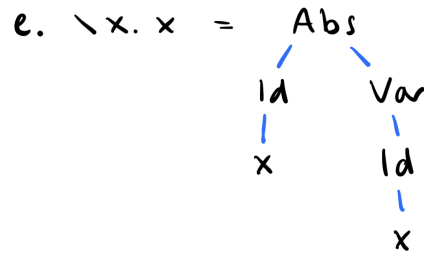
```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "x y" | ./TestLambda
Parse Successful!
[Abstract Syntax]
App (Var (Id "x")) (Var (Id "y"))
[Linearized tree]
x y
```

Figure 13: $x \ y$



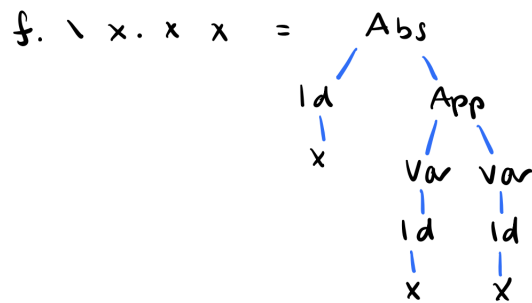
```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "x y z" | ./TestLambda
Parse Successful!
[Abstract Syntax]
App (App (Var (Id "x")) (Var (Id "y"))) (Var (Id "z"))
[Linearized tree]
x y z
```

Figure 14: $x \ y \ z$



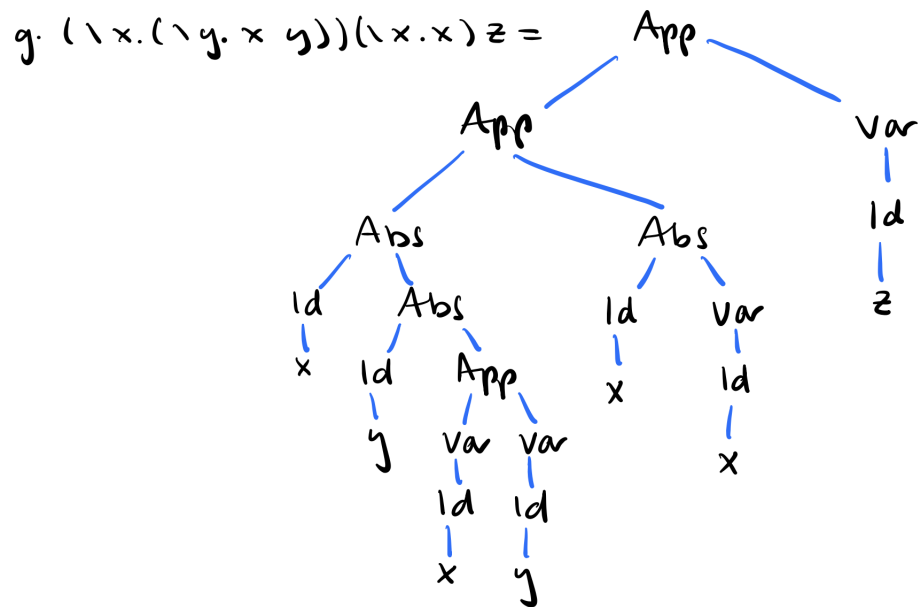
```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "\ x.x" | ./TestLambda
Parse Successful!
[Abstract Syntax]
Abs (Id "x") (Var (Id "x"))
[Linearized tree]
\ x . x
```

Figure 15: $\lambda x.x$



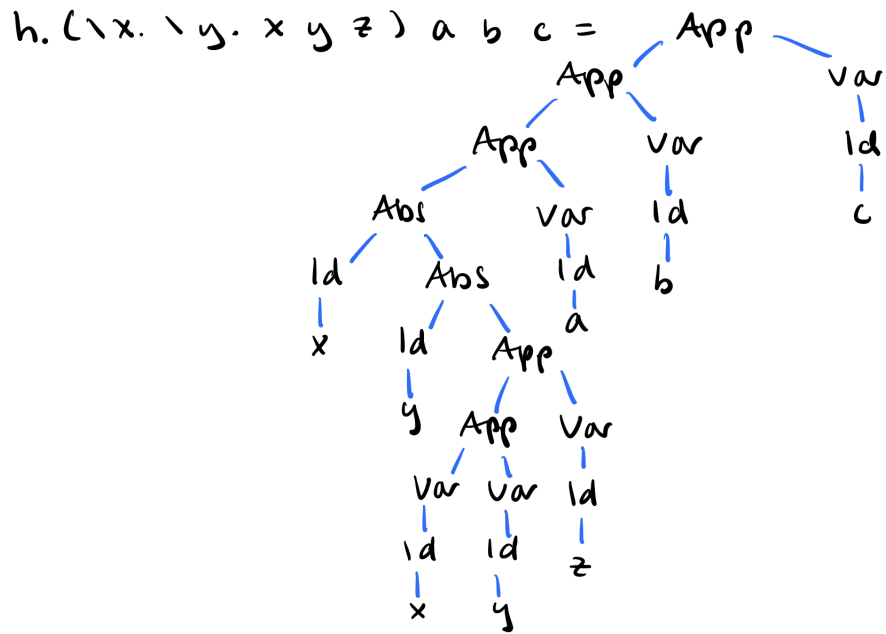
```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "\ x.x x" | ./TestLambda
Parse Successful!
[Abstract Syntax]
Abs (Id "x") (App (Var (Id "x")) (Var (Id "x")))
[Linearized tree]
\ x . x x
```

Figure 16: $\lambda x.xx$



```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "(λ x . (λ y . x y)) (λ x.x) z" | ./TestLambda
Parse Successful!
[Abstract Syntax]
App (App (Abs (Id "x") (Abs (Id "y") (App (Var (Id "x")) (Var (Id "y")))))) (Abs (Id "x") (Var (Id "x")))) (Var (Id "z"))
[Linearized tree]
λ x . λ y . x y (λ x . x) z
```

Figure 17: $(\lambda x. (\lambda y. xy))(\lambda x. x)z$



```
(base) evelynlawrie@Evelyns-MacBook-Pro-2 grammar % echo "(λ x . λ y . x y z) a b c" | ./TestLambda
Parse Successful!

[Abstract Syntax]

App (App (App (Abs (Id "x")) (Abs (Id "y")) (App (App (Var (Id "x")) (Var (Id "y")))) (Var (Id "z"))))) (Var (Id "a"))) (Var (Id "b"))) (Var (
Id "c"))

[Linearized tree]

λ x . λ y . x y z a b c
```

Figure 18: $(\lambda x. \lambda y. x y z) a b c$

5.3 Semantics

The Lambda calculus semantics homework entails finishing the workout that we started in class. Below is an image of my typed out computation:

```

84  -- ORIGINAL
85  (\f.\x.f(f(x))) (\f.\x.(f(f(f x)))) =
86
87  -- STEP 1
88  \x. (\f.\x.(f(f(f x)))) ((\f.\x.(f(f(f x))))(x)) =
89
90  -- STEP 2
91  \x. (\f.\x.(f(f(f x)))) ((\f.\x0.(f(f(f x0))))(x)) =
92
93  -- STEP 3
94  \x. (\f.\x.(f(f(f x)))) (\x0.(x(x x0))) =
95
96  -- STEP 4
97  \x. (\f.\x1.(f(f(f x1)))) (\x0.(x(x x0))) =
98
99  -- STEP 5
100 \x. (\x1.((\x0.(x(x x0)))( (\x0.(x(x x0)))) ((\x0.(x(x x0)))x1) ))) =
101
102 -- STEP 6
103 \x. \x1. ((\x0. (x(x(x x0))))((\x0. (x(x(x x0))))(x(x(x x1)))))) =
104
105 -- STEP 7
106 \x. \x1. ((\x0. (x(x(x x1))))(x(x(x(x(x x1))))))) =
107
108 -- STEP 8
109 \x. \x1. (x(x(x(x(x(x(x(x x1))))))))

```

Figure 19: Workout Solution

6 Week 5

6.1 Variables, Binding, Scope, and Substitution

Substitution with function composition is discussed in this lesson, as the topics of free and bound variables come up when trying to implement function composition for functions that share variable names. Below is the correct formula for $f \circ g$ with $f(x)$ and $g(y)$ restated:

$$f(x) = x + z \quad (18)$$

$$g(y) = 2 * y \quad (19)$$

$$f \circ g = f(g(y)) \quad (20)$$

$$= 2 * y + z \quad (21)$$

Because the variable y in $f(x)$ is bound by f , it is a different variable than the one passed in by substituting $g(y)$ into f . To implement capture avoiding substitution, it is necessary to rename the variable y in $f(x)$ to a different name (I chose z in this case), making it a fresh variable. Therefore, this equation avoids ambiguity by separating variables that do not hold the same meaning.

Below is a general formula for $f \circ g$ written in Lambda calculus:

$$f \circ g = \lambda x. f (\lambda y. g y) \quad (22)$$

This formula works for any arbitrary functions f and g where the function g gets applied to the argument y in $g y$ and becomes the argument for the function f . The result of $f (\lambda y. g y)$ represents the composition of f and g .

6.2 Interpreting Lambda Calculus

Using the file Interpreter.hs, below is the line by line evaluation of the expression $(\lambda x. \lambda y. x)y$:

```
-- interpreter steps:
-- 1. evaluate original statement
eval (App (Abs (Id "x") (Abs (Id "y") (Var (Id "x")))) (Var (Id "y")))
--      ( i ) (      e3      ) (      e2      )
-- 2. find App in eval function
-- 3. apply line 10
eval (subst (Id "x") eval (Var (Id "y")) (Abs (Id "y") (Var (Id "x"))))
--      ( i )      (      e2      ) (      e3      )
-- 4. eval e2 just equals e2 since it's a variable
-- 5. apply line 13
eval (subst (Id "x") (Var (Id "y")) (Abs (Id "y") (Var (Id "x"))))
--      ( id ) (      s      ) (Abs id1      e1      )
--      (      e      )
-- 6. apply line 28
-- line 29
let f = fresh (Abs (Id "y") (Var (Id "x")))
--      ( id1 ) (      e1      )
-- 7. compute fresh with the freshAux function
--   compute line 21
fresh = Id . freshAux
--   find freshAux function that matches
--   apply line 19
freshAux (Abs (Id "y") (Var (Id "x"))) = ("y") ++ freshAux (Var (Id "x"))
--      ( Id i ) (      e      ) ( i )      (      e      )
--   evaluate freshAux (Var (Id "x"))
--   apply line 17
freshAux (Var (Id "x")) = "x" ++ "0"
--      ( i ) ( i )
--   f = Id "yx0"
-- line 30
e2 = subst (Id "y") f (Var (Id "x"))
--      ( id1 ) (      e1/e      )
-- 8. evaluate e2
--   apply line 25 (this equation is true because id != id1)
subst (Id "y") f (Var (Id "x")) = (Var (Id "x"))
--      ( id ) (s) ( id1 )      ( id1 )
--   e2 = (Var (Id "x"))
-- line 31
-- 9. compute this line
Abs (Id "yx0") (subst (Id "x") (Var (Id "y")) (Var (Id "x")))
--      ( id ) (      s      ) (      e2      )
--   compute substitution --> apply line 25 (this equation is true because id = id1)
subst (Id "x") (Var (Id "y")) (Var (Id "x")) = (Var (Id "y"))
--      ( id ) (      s      ) ( id1 ) (      s      )
```

```

-- evaluate with line 12
Abs (Id "yx0") (Var (Id "y"))
eval Abs (Id "yx0") (Var (Id "y")) = Abs (Id "yx0") (eval (Var (Id "y")))
--      ( i ) ( e )      ( i )      ( e )
-- evaluate eval (e) with line 13 (this equation is true because variables evaluate to
-- themselves)
eval ((Var (Id "y"))) = (Var (Id "y"))
--      ( x ) ( x )
-- final output of the evaluation = \yx0. y =
Abs (Id "yx0") (Var (Id "y"))

```

As for the binders and scope, for Lambda calculus substitutions that follow the form $(\lambda id.e)s$, this equation gets simplified in the code to *subst id s e*. Therefore, for all calculations, *λid* is the binder and the scope of the binder is *e*. The scope of *s* and the scope of *λid* are both *e*. The assignments of scope and binder are based on the pattern matching performed above, where each term assigned to *λid* and *e* are the equivalents in each specific function call of *subst*. Below are the solutions for the binders and scope of the specific variables for each call to the function *subst* in the above calculation:

```

-- Step 5 subst calculation:
subst (Id "x") (Var (Id "y")) (Abs (Id "y")(Var (Id "x")))
-- binder = (Id "x"), scope of binder (bound variable) = (Abs (Id "y")(Var (Id "x"))), free
-- variable = (Var (Id "y"))

-- Step 7 line 30 subst calculation:
subst (Id "y") f (Var (Id "x"))
-- binder = (Id "y"), scope of binder (bound variable) = (Var (Id "x")), free variable = f

-- Step 9 subst calculation:
subst (Id "x") (Var (Id "y")) (Var (Id "x"))
-- binder = (Id "x"), scope of binder (bound variable) = (Var (Id "x")), free variable = (Var (Id
-- "y"))

```

7 Week 6

7.1 Rewriting: Introduction

Week 6 introduces the topic of ARSs (Abstract Rewriting Systems) and string rewriting. An ARS is made up of a language (a set of words) *A* and rules *R* for rewriting strings created withing the language *A*. An ARS is confluent if every branch-off computation (where there can be multiple rules applied to the same string) result in the same computation in a further step. This is to say, every peak of the graph of the ARS evaluation has a valley. An ARS is terminating if it does not consist of infinite computations. Finally, an ARS has unique normal forms if all elements have a unique normal form. This is to say, it is convergent, which means it is confluent and terminating [ARS]. For an ARS to have unique normal forms, it must be confluent and normalizing, that is, every object in the ARS must have at least one normal form [ARS]. Below are a list of seven examples ARSs with pictures:

- Confluent = C
 Terminating = T
 Has Unique Normal Forms = UNF
- ① $A = \{\epsilon\}$, $\text{pic} = \square$ (no strings can be formed)
 \hookrightarrow C, T, UNF
 - ② $A = \{a\}$, $R = \{\}$, $\text{pic} = aaa$ (no rewrite rules available)
 \hookrightarrow C, T, UNF
 - ③ $A = \{a\}$, $R = a \rightarrow a$, $\text{pic} = \begin{array}{c} a \\ \downarrow \\ a \end{array}$
 \hookrightarrow C, not T, not UNF
 - ④ $A = \{a, b, c\}$, $R = \begin{array}{l} a \rightarrow b \\ a \rightarrow c \end{array}$, $\text{pic} = \begin{array}{c} a \\ \swarrow \searrow \\ b \quad c \end{array}$
 \hookrightarrow not C, T, not UNF
 - ⑤ $A = \{a, b\}$, $R = \begin{array}{l} a \rightarrow a \\ a \rightarrow b \end{array}$, $\text{pic} = \begin{array}{c} a \\ \downarrow \\ b \end{array}$
 \hookrightarrow not C, not T, not UNF
 - ⑥ $A = \{a, b, c\}$, $R = \begin{array}{l} a \rightarrow b \\ b \rightarrow b \\ a \rightarrow c \end{array}$, $\text{pic} = \begin{array}{c} a \\ \swarrow \searrow \\ b \quad c \end{array}$
 \hookrightarrow not C, not T, not UNF
 - ⑦ $A = \{a, b, c\}$, $R = \begin{array}{l} a \rightarrow b \\ b \rightarrow b \\ a \rightarrow c \\ c \rightarrow c \end{array}$, $\text{pic} = \begin{array}{c} a \\ \swarrow \searrow \\ b \quad c \\ \circ \quad \circ \end{array}$
 \hookrightarrow not C, not T, not UNF

Figure 20: Example ARSs

Below is the table of examples of the 8 possible combinations of ARSs with pictures:

Confluent	Terminating	Unique Normal Form	Example
True	True	True	<p>ARS #1/2</p> <p>(ex) \square (no strings can be formed)</p> <p>(1)</p> <p>aaa (no rewrite rules (2) available)</p>
True	True	False	N/A (not possible)
True	False	True	N/A (not possible)
True	False	False	<p>ARS #3</p> <p>(ex) aa</p> <p>aa aa</p> <p>aa</p> <p>(3)</p>
False	True	True	N/A (not possible)
False	True	False	<p>ARS #4</p> <p>(ex) aaa</p> <p>aba aac</p> <p>abc acc</p> <p>bba bcc</p> <p>(4)</p>
False	False	True	N/A (not possible)
False	False	False	<p>ARS #5/6/7</p> <p>(ex) aab</p> <p>aab abb</p> <p>aab bbb</p> <p>...</p> <p>(5)</p>
			<p>abbc</p> <p>cbbc bbbc</p> <p>cbbc bbcc</p> <p>cbbc ...</p> <p>...</p> <p>(6)</p> <p>abca</p> <p>baca abcc</p> <p>bbcc cbcc</p> <p>... cbcc</p> <p>...</p> <p>(7)</p>

Figure 21: ARS Table

7.2 String Rewriting Exercises

This section explores further information on the topic of string rewriting. Below are the answers for the example computation, finishing the work we began in class:

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

1. Why does the ARS terminate?

This ARS terminates because it always reduces. There are no cases in which this ARS results in an infinite computation. This ARS is set up to get reduced to either the letter a or b .

2. What are the normal forms?

The normal forms are a and b .

3. Is there a string s that reduces to both a and b ?

No, there are no strings in the ARS that evaluate to both a and b because the rewrite rules do not permit it. This is because the ARS is confluent: any combination of a and b evaluates to either a or b ; each string has unique normal forms.

4. Show that the ARS is confluent:

This ARS is confluent because every peak in the reduction of a given string in the ARS has a valley that makes every branch in the computation reduce to the same final step. Below is an example of an overlapping critical pair that shows confluence of this ARS:

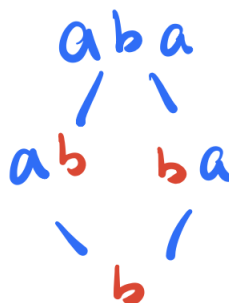


Figure 22: Demonstration of Confluence

5. Replacing \rightarrow by $=$, which words become equal?

Given the rewrite rules, $aa = bb$ and $ab = ba$. To generalize this statement, any word with an odd number of b 's $= b$ and any word with an even number of b 's $= a$.

6. What is the nicest way you can find to characterise which words are equal by one English sentence?

Any words with an even number of b 's evaluates to a and words with an odd number of b 's evaluates to b .

7. Repeat the last item using modular arithmetic:

For any string s in the ARS, f is the frequency of b 's in s . $f \equiv 0 \pmod{2}$ shows if the frequency of b 's

in s is even or odd. Therefore, this formula written in modular arithmetic is as follows:

$$s = \begin{cases} a, & \text{if } f \equiv 0 \pmod{2} \\ b, & \text{otherwise} \end{cases} \quad (23)$$

8. Which specification does the algorithm implement?

This ARS represents an XOR logic gate, since the truth table of the rewrite rules outputs the same values as the logic for this gate. It determines whether there is an odd or even number of 1's (b 's) in the original string. Below is an image of these truth tables to visualize their equivalence:

XOR Gate:			ARS:		
x	y	Output	a	b	Output
0	0	0	0	0	a
0	1	1	0	1	b
1	0	1	1	0	b
1	1	0	1	1	a

Figure 23: ARS and XOR Truth Table

8 Week 7

Given the following rewrite rules:

```
ab -> cc
ac -> bb
bc -> aa
```

and starting from 15 a's, 14 b's and 13 c's, is it possible to reach a configuration in which there are only a's or only b's or only c's?

Example calculations:

```
letters:                                counts (parities):
aaaaaaaaaaaaaaaaabbbbbbbbbbbcccccccccccccc 15,14,13
cccccccccccccccccccccccccccccccccccccccccc 1,0,41
bbcccccccccccccccccccccccccccccccccccccccccc 0,2,40
bacccccccccccccccccccccccccccccccccccccccccc 2,1,39
cccccccccccccccccccccccccccccccccccccccccccc 1,0,41
ccbcccccccccccccccccccccccccccccccccccccccccc 0,2,40
...
```

```
aabbcc 2,2,2
accbcc 1,1,4
bbcbcc 0,3,3
baabcc 2,2,2
ccabcc 1,1,4
cbbbcc 0,3,3
aaccbb 2,2,2
...
```

Invariant:

Two (or more) parities (counts) are always matching (even or odd) in each calculation step.

Based on the above calculations, reducing 15 a's, 14 b's, and 13 c's never allows for the possibility of reducing to a string with only a's, b's or c's. This is due to the invariant described above: two or more letters' parities are always either even or odd.

9 Lessons from the Project

Recursion is the basis for many of these weekly projects. Understanding in detail how recursion works behind the scenes aids in the implementation of the coded solutions in this report. Recursion is a problem-solving technique that utilizes the call stack to create functions that call other functions, or more simplified versions of themselves, within them. To visualize and fully understand the structure of recursive calls, it is best to draw out the problem as a tree data structure, with each recursive call being a node of the tree. Every time a base case is reached, that node becomes a leaf and then the branches are explored upwards from there, branching off when needed and computing calculations from the bottom up. This is describing depth-first search, in which one branch of the tree is explored all the way until reaching a base case. This process is repeated for the whole tree, usually starting at the left and moving right. Parsing, rewriting strings into trees, is a method of visualizing how a computer executes a certain computation and can be vital in the comprehension of the recursion problems explored in this report.

Another key takeaway that improves upon the concept of recursion is memoization. This deals with the repeated nodes or function calls within a function's tree. With simple recursion, it is common for the same calculation to be computed multiple times, which decreases efficiency of the code. This creates exponential run times which can prove highly costly for large computations. The process of memoization stores values that the function has already computed in a hash table or similar data structure for easy lookup and avoidance of unnecessary repeated calculations. The recursive function call then implements a quick lookup in this data structure (constant runtime) and extracts the output if the current input value has already been calculated. If the current parameter has not been calculated, it performs the regular calculations and then stores the new value inside of the memo object. The process of memoization greatly decreases the time complexity of recursive functions, in the majority of cases bringing it down from an exponential to a polynomial or linear type runtime. This problem-solving technique stood out to me as a highly important lecture topic that serves as a way to optimize the algorithms presented in this report.

From Week 2's addition to this report, the syntax that displays the example computation is designed to illustrate how the stack grows and shrinks with recursive function calls. The stack grows from left to right until a base case is reached, shrinking from there going right to left. Working through this example thoughtfully can aide in the understanding of how recursive functions are interacted with on the stack. We are able to visualize through the Towers of Hanoi problem what the call stack looks like behind the scenes and why recursive functions work the way that they do. Recursion is the basis for many of the computations explored throughout this course, making a thorough understanding of its implementation crucial in fully absorbing its key takeaways.

Additionally, parsing is an important topic in this class, as it is an essential process that machines

employ when evaluating most computer code. Computers parse concrete syntax into abstract syntax, which is then evaluated by the virtual machine, using the model of computation. This process is something we will encounter daily as computer programmers. Better understanding how a computer parses and understands the information we give it can make us more efficient programmers. Being able to decipher parse trees given a context-free grammar and examples of its language allows us to better understand how computers process code. BNFC and other types of parser generators define rules that the computer follows when interpreting concrete syntax strings. Approaching computer programs in terms of language and grammar teaches us to interpret programs logically and think of parsing as something that follows strict rules and does not take semantics into account. The way humans read and interpret programs is very different from how computers parse them, and we as programmers have to keep in mind that what the computer sees is taken at face value and evaluated according to the rules we have defined. This teaches us to be explicit and straightforward with how we program, avoiding ambiguity and confusion.

Week 4 gives us an introduction to Lambda calculus and its applications. Lambda calculus is the simplest possible programming language. The syntax may look very different to most programming languages, but it contains all of the same concepts. This allows us to strip back a lot of the syntax and get to the root of the basis of programming languages. To be able to write programs using only variables, abstraction, and application is a good starting point to understand the logic of why computer programs work the way that they do. Using variable binding and substitution for Lambda calculus computations (using the Alpha and Beta rules) teaches us how to rename formal parameters and perform capture avoiding substitution, which are the building blocks of Lambda calculus. In capture avoiding substitution, we rename the formal parameters (bound variables) and then replace that parameter in the body of the function by the argument. Reducing Lambda calculus expressions according to these rules demonstrates how this logic is computed by the machine in its simplification. We are able to better understand how computers interpret code by working step-by-step through these computations and having a foundational understanding of Lambda calculus as a whole.

The lessons from Week 5 surround the topics of capture avoiding substitution, pattern matching, and implementing Lambda calculus in Haskell code. Capture avoiding substitution contains lessons of free and bound variables. These topics are pertinent in programming, when topics such as static variables and variable scopes becomes important when dealing with overlapping variable names and ambiguous variable updates. In section 6.1, the formula for function composition deals with the scope of variables and the need to rename variables to perform capture avoiding substitution. Some definitions of different types of variables are as follows. A bound variable is one that can be renamed by a fresh variable without resulting in any changes in its meaning. A fresh variable is defined simply as one that has not been used before. Lastly, a free variable is a variable that is not bound by any particular binder. Dealing with the topic of capture avoiding substitution allows us to better understand the meaning and values stored in our variables when we program. It reminds us to create unique variables names and work towards avoiding ambiguity in overlapping namespaces.

The Lambda calculus interpreter section of Week 5's homework details how the interpreter processes Lambda calculus computations implemented in Haskell. Working step by step through the given exercise taught us not only about Haskell syntax in more detail, but also how computers interpret code. We can better understand the interpretation step of computer program execution knowing how an expression gets decoded and which lines get evaluated. The exercise dealt with pattern matching to decipher what lines get executed when the given expression is evaluated by the interpreter. This is a logical exercise that strengthens our capacity to make sense of which lines of code are visited for any given expression. This can help with error prevention and debugging, as understanding the logic behind what the interpreter executes in a program can allow us to see any potential logic or syntax errors present more easily.

Week 6 taught us about Abstract Rewriting Systems. ARSs come with various classifications and elements. The topics of confluence, termination, and having unique normal forms were previously discussed. Evaluating ARSs for these qualities allows us to classify ARSs and better understand their behavior. Knowledge of ARSs can assist us in understanding Finite State Machines (FSMs) and automata, since ARSs are simply unlabeled state transition systems [ARS]. Since these are core topics in the fields of programming languages and algorithm analysis, they can assist with the comprehension of this field of computer science

and logic. Here are a few other definitions that relate to ARSs that are key takeaways from this lesson and further our understanding of the topic:

1. Equivalence:

a and b are equivalent if $a \equiv b$, where \equiv is the reflexive, symmetric, and transitive closure of the reduction arrow function \rightarrow .

2. Reduction:

a is reducible if there is a b such that $a \rightarrow b$.

3. Normal Forms:

a is a normal form if a is irreducible. Additionally, b is a normal form of a if a reduces to b and b is a normal form.

4. Joinability:

a and b are joinable if they reduce to the same element.

5. Local Confluence:

For all b and c that are the outputs of two different rewrite rules in one given reduction step, the ARS is locally confluent if b and c are joinable. As well, if the ARS is terminating, local confluence implies confluence.

These definitions allow us to evaluate ARSs and the logic associated with them. Simplifying expressions of this type brings us back to topics such as reduction and simplification that we learned in high school algebra. Although this lesson implements these themes in a more structured way, we as students have been performing simplification in mathematics for years. These lessons strengthen our abilities to use logic and mathematical vocabulary, which we can apply to projects in our classes, future careers, and personal lives.

Week 7 dealt with invariants, which is a common topic in dealing with ARSs. We define invariants by thinking about what doesn't change in every step of an ARS's calculations. This is formally defined as:

$P: A \rightarrow B$ is invariant of (A, \rightarrow) if $P(A) = P(B)$

where change is defined as:

$\text{all } a, b \text{ within } A \text{ where } a \rightarrow b.$

This can allow us to better understand the behavior of ARS rewrite rules and further how to evaluate strings based on their criteria. Invariants help us find patterns in ARS calculations that strengthen our logic processes. Invariants hold a profound significance as they provide a stable foundation upon which to understand and analyze dynamic processes in various domains, from computer science to mathematics and beyond. These invariants are essential because they encapsulate characteristics that remain unchanged throughout an ARS's evolution. In a broader context, the study of invariants teaches us an important lesson about the fundamental principles governing change and transformation. By identifying and preserving these invariants, we gain insight into the underlying structures and symmetries that persist amidst complexity. This, in turn, deepens our understanding of not only ARSs but also real-world phenomena, enabling us to make predictions, ensure correctness, and design more efficient algorithms and systems.

10 Conclusion

This report details various logical problems and their solutions implemented throughout this semester. As illustrated in this report, these algorithms can be approached in various different ways. Exploring different approaches to the same problem hones the skill set of novel solution creation, which allows programmers to complete tasks by thinking outside of the box in ways that seem less than straightforward. The thought processes behind the solutions to these problems can be applied to technical interviews, future computer science courses, and projects assigned in the workplace within the field of computer programming. The problem-solving techniques developed in this class and through this report serve as a basis of knowledge to assist students with our future endeavours.

References

- [LTX] [Cases and piecewise functions in LaTeX](#), LaTeX-Tutorial.com, 2023.
- [GCD] Alexander Kurz, [gcd and Euclid's algorithm](#), YouTube, 2020.
- [WIK] [Tower of Hanoi](#), Wikipedia, 2023.
- [ARS] [Abstract rewriting system](#), Wikipedia, 2023.