CPSC-354 Report

Natalie Huante
Chapman University

September 30, 2023

Abstract

Short summary of purpose and content.

Contents

1 Introduction

2 Homework

This section contains solutions to homework.

2.1 Week 1

The homework for Week 1 is dedicated to allow myself the opportunity to get familiar with LaTeX as well as review the model of equational reasoning. In this lesson, we used the Fibonacci Sequence as an example of this but we will use the function of Greatest Common Divisor for the assignment. In terms of familiarity with LaTeX, I do have experience through the Algorithm Analysis report from the previous semester, however, this serves as a way to remind myself of the language.

For context, the definition the GCD function is as follows:

```
gcd(a,b):
Input: Two whole numbers (integers) called a and b, both greater than 0.
(1) if a>b then replace a by a-b and go to (1).
(2) if b>a then replace b by b-a and go to (1).
Output: a
```

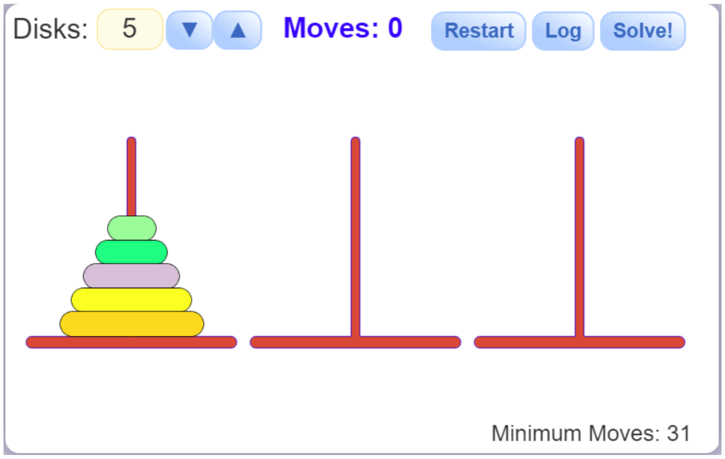
Now, I will write out the full computation of gcd(9,33) below:

$$\begin{aligned} gcd(9,33) &= gcd(9,24) \\ &= gcd(9,15) \\ &= gcd(9,6) \\ &= gcd(3,6) \\ &= gcd(3,3) \\ &= 3 \end{aligned}$$

As you can see above, the basis of the function is to iteratively decrement a and b until you reach a point at which you can no longer decrement them by the definition given.

2.2 Week 2

For this week's homework we wil take a look and practice based on our discussion of the Towers of Hanoi problem. To solve the Towers of Hanoi one must move all the rings from the left-most tower to the right-most tower. However, discs can only be placed on other discs larger than themselves. Therefore, we have to logically think about the order in which we move them as well as how we can most efficiently do so (the least amount of moves). I have included a visual representation of what this problem looks like:



Disks: 5

Moves: 0

Minimum Moves: 31

In our class discussion, we established a set of rules in order to then discuss how to represent our functions in code. Below I include both rules as well as repeat some notes on notation in order to allow for some context on the second half of this assignment.

```
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
```

- read `hanoi n x y` as ”move tower of n disks from x to y”
- think about how to move a tower of n+1 disks assuming we already know how to move a tower of n disks

- `hanoi n x y` is a function that takes three arguments: a number n (the number of disks), a number x (encoding the place where the tower is, a number y (encoding the place where the tower should go))
- `move x y` is afunction that moves one disk (the topmost disk) from x to y
- `other x y` denotes the third place which is neither x nor y

Now having laid down some rules for our discussion, we observed that the nature of the Towers of Hanoi problem is recursive given that in order to solve a tower of 4 disks we must first solve it with 3 disks and then that requires we solve it for 2 disks and so on. In class we observed the 5 disk tower solution of `hanoi 5 0 2`. For this assignment, I will write out the implementation of this function to cement what we learned about its recursion and logic.

```
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
```

As we can see this is a very lengthy program that relies on its recursive nature to solve the hanoi problem. Now, if we extract from this execution the moves that solve the puzzle (in their right order), we will see that there are 31 moves for the 5-disk tower that will solve the problem most efficiently. I say that these are the steps that actually solve the problem since they are the ones that will prompt moving a disk from one tower to another. I will rewrite the steps again below:

```
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
move 0 1
move 2 1
move 2 0
move 1 0
move 2 1
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
move 1 0
move 2 1
move 2 0
move 1 0
move 1 2
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
```

Now that we have written out the exection, we can also see that the word `hanoi` appear many times in the computation. I observed the number of occurrences and recorded them in the below table. By looking at the table, we can notice that the number of occurrences can actually be represented as a formula, for they increase in the same intervals exponentially. Therefore, we can say that with n disks, the number of times the word `hanoi` appears in the computation is $2^n - 1$.

n disks	num of hanoi
1	1
2	3
3	7
4	15
5	31

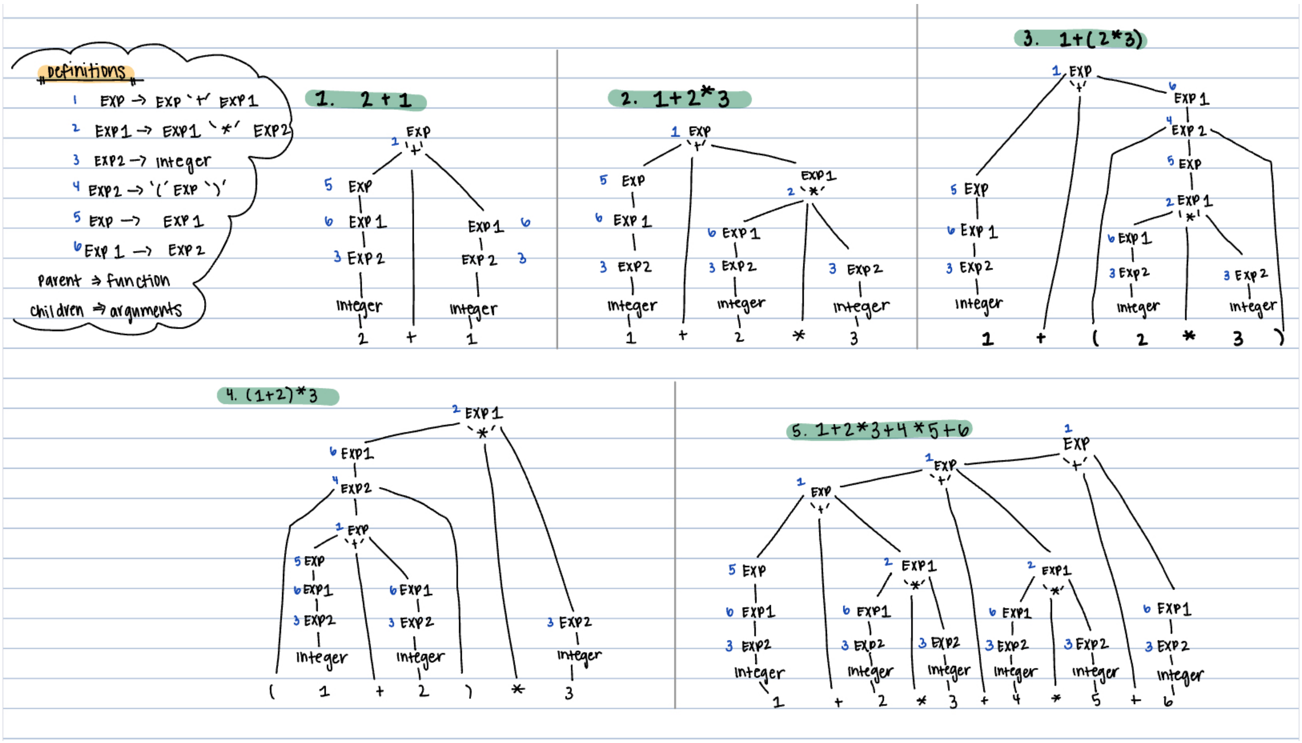
2.3 Week 3

In this week's homework, we are reviewing parsing and context-free grammars. The idea of parsing is immportant to understand as we need it to translate concrete syntax (what we see as more human-readable) into abstract syntax (what a computer will see as more readble). In class, we focused on processing abstract syntax in a 2-dimensional view, or what resembles a tree. A context-free grammar is a set of rules that defines a language. One of the examples we used, and the one we will use for the following problems, is a context-free grammar that defines arithmetic expressions. In other words, a string is considered to be part of the language defined if it can be dervied from the rules below.

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

Now that we have the rules above, I will write out the derivation trees (aka parse trees or conccrete syntax trees) for the following strings. Note that I numbered the rules above and wrote, in blue, the number of the rule used in each step. The integer definition is implied here so there is no number next to those translations.

- 2+1
- 1+2*3
- 1+(2*3)
- (1+2)*3
- 1+2*3+4*5+6

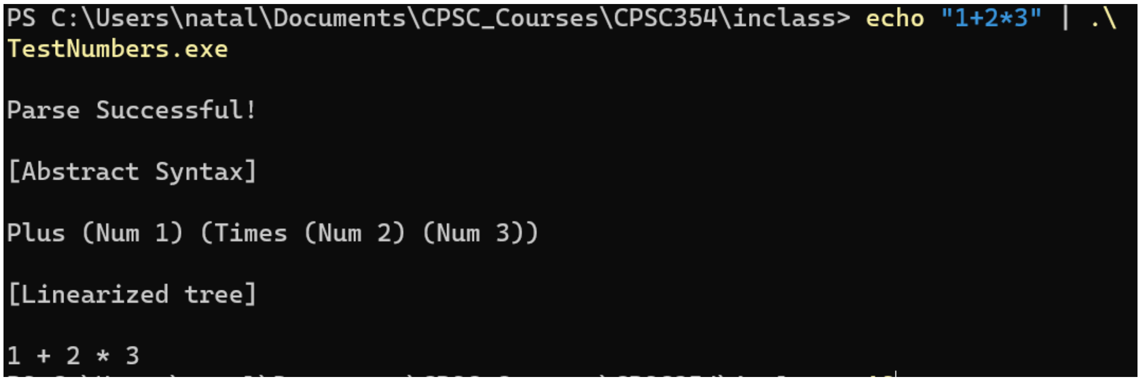


Now that we have covered what the parsing process looks like, we can think bigger. In other words, we can think of a parser that will automatically translate concrete syntax into abstract syntax. Looking at out lesson, we do not attempt to create this type of parser ourselves, but rather used a parser generator. In this case, we will input a context-free grammer (like the one above) and the generator will output a parser.

First, I installed the parser generator BNFC, which proved to be a feat of its own. After multiple error messages and researching quests, I was able to successfully download the generator as well as the necessary libraries (alex, happy, etc.). The first commands I used are those described in the lecture:

```
bnfc -m -haskell numbers.cf
make
echo "1+2*3" | ./TestNumbers
```

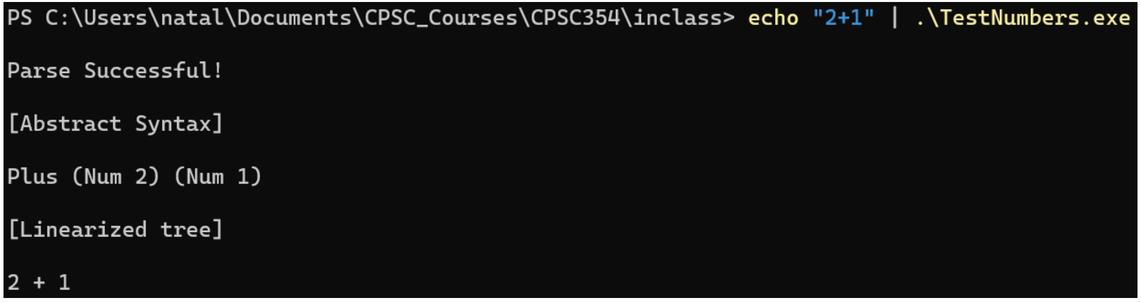
This generated some files and then gave me the following output:



```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+2*3" | .\
TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Num 1) (Times (Num 2) (Num 3))

[Linearized tree]

1 + 2 * 3
```

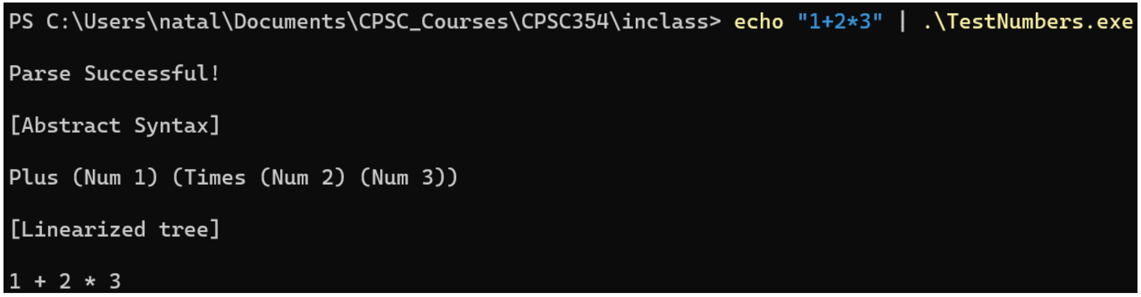
Having run the example successfully, I will now show the outputs for the following given strings: **You will note that these are the same exercises I translated by hand above**

$2 + 1$



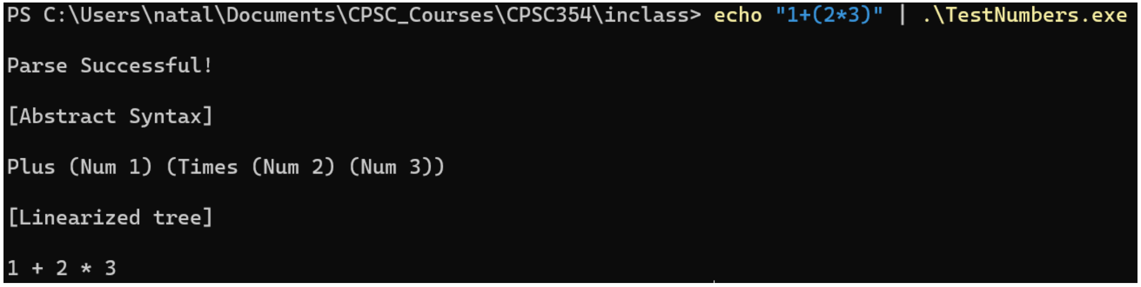
```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "2+1" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Num 2) (Num 1)

[Linearized tree]

2 + 1
```

$1 + 2 * 3$



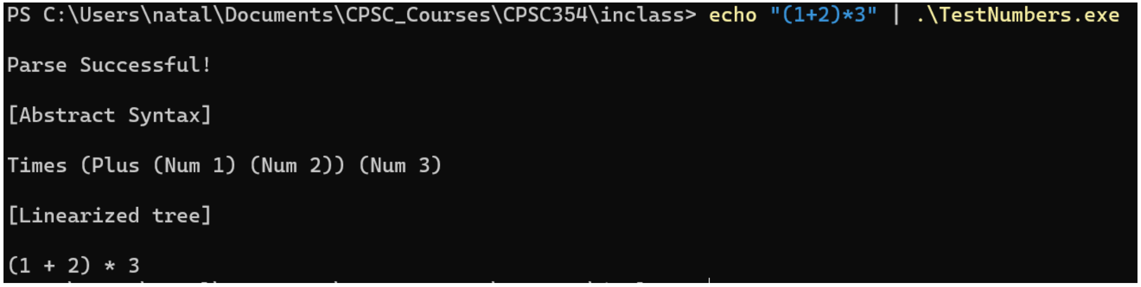
```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+2*3" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Num 1) (Times (Num 2) (Num 3))

[Linearized tree]

1 + 2 * 3
```

$1 + (2 * 3)$



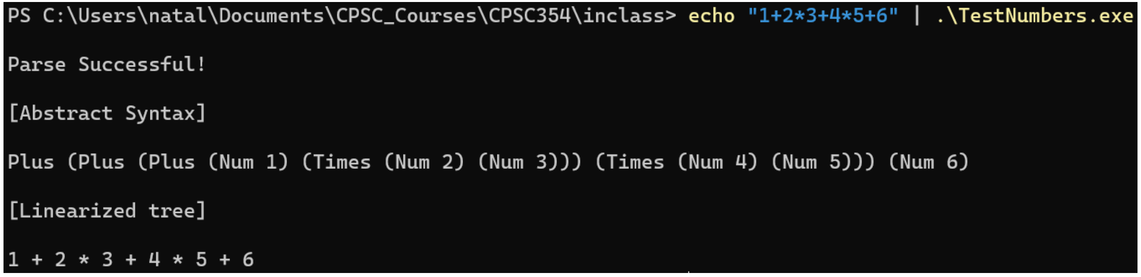
```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+(2*3)" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Num 1) (Times (Num 2) (Num 3))

[Linearized tree]

1 + 2 * 3
```

$(1 + 2) * 3$



```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "(1+2)*3" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Times (Plus (Num 1) (Num 2)) (Num 3)

[Linearized tree]

(1 + 2) * 3
```

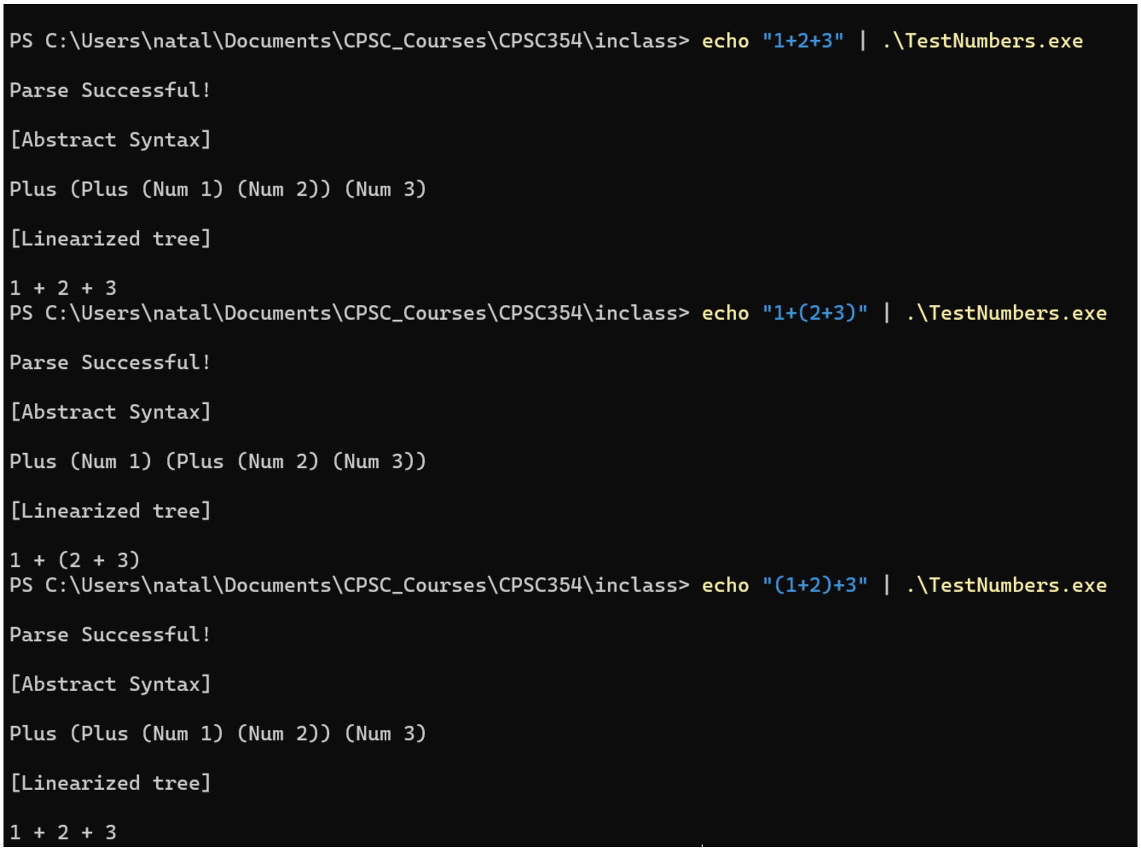
$1 + 2 * 3 + 4 * 5 + 6$



```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+2*3+4*5+6" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Plus (Plus (Num 1) (Times (Num 2) (Num 3))) (Times (Num 4) (Num 5))) (Num 6)

[Linearized tree]

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

We can also look at the how the use of paranthesis affects the output. Here we will compare the output between

- 1+2+3
- (1+2)+3
- 1+(2+3)



```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+2+3" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Plus (Num 1) (Num 2)) (Num 3)

[Linearized tree]

1 + 2 + 3
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "1+(2+3)" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Num 1) (Plus (Num 2) (Num 3))

[Linearized tree]

1 + (2 + 3)
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\inclass> echo "(1+2)+3" | .\TestNumbers.exe

Parse Successful!

[Abstract Syntax]

Plus (Plus (Num 1) (Num 2)) (Num 3)

[Linearized tree]

1 + 2 + 3
```

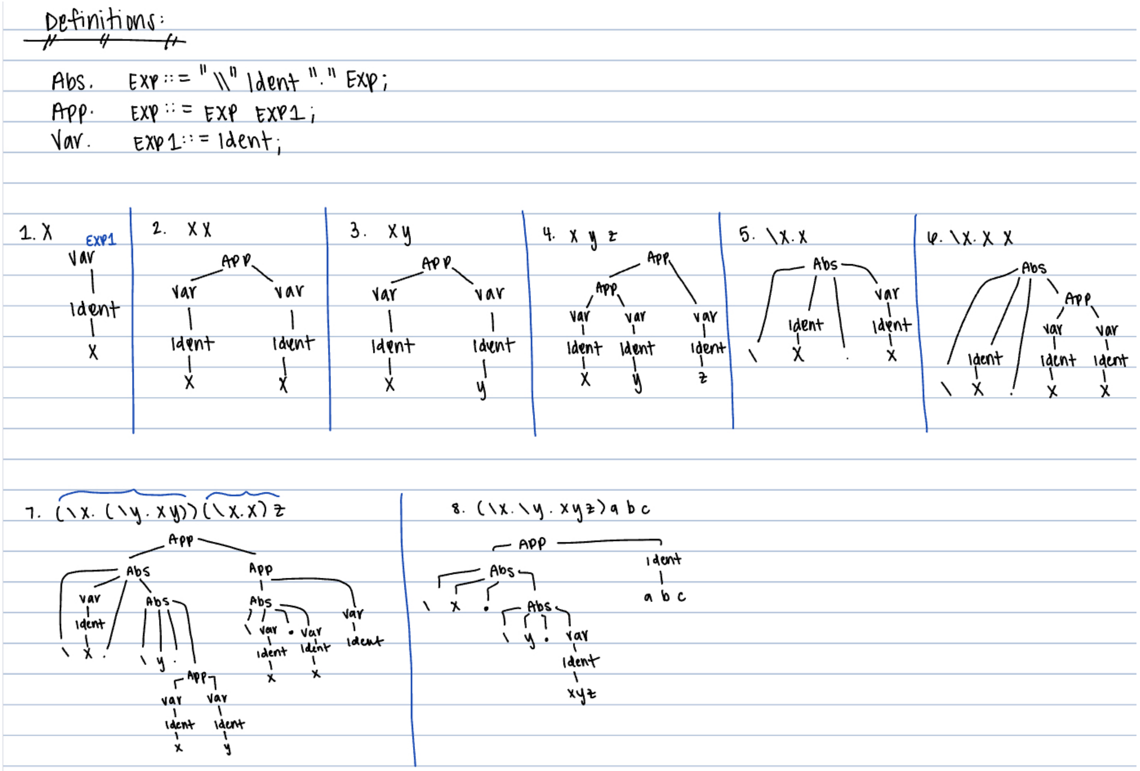
As you can see, the abstract syntax outputted differs depending on where or if the paranthesis are placed in the expression. The first and last output do yield the same result, implying that if the parser is not given any paranthesis, it will prioritize executing from left to right. In this case, the 1 and the 2 are added together first. Therefore, when the 2 and the 3 are placed in paranthesis, the abstract syntax changes as the order in which the addition functions are executed also changes. Overall, this homework focused on practicing with and establishing the BNFC environment as well as understanding how to convert concrete syntax into abstract syntax.

2.4 Week 4

This week's homework builds upon the last assignment and focuses on understanding lambda calculus. Using the layout of the previous interpreter for addition and multiplication, I first had to modify the definitions and interpreter to read lambda calculus. The first step of the homework was to "use bnfc and the grammar of lambda-calculus [given] to create a parser for lambda-calculus expressions." I include the definition provided below:

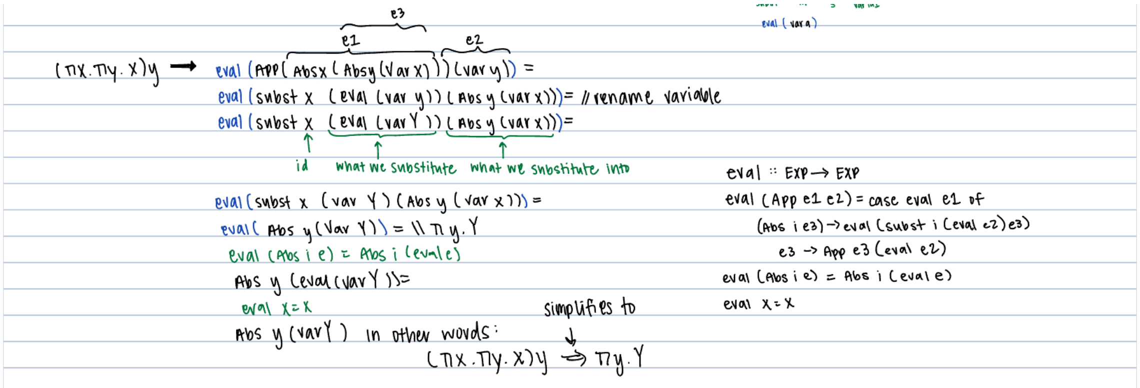
```
Abs.   Exp ::= "\\" Ident "." Exp ;
App.   Exp ::= Exp Exp1 ;
Var.   Exp1 ::= Ident ;

coercions Exp 1 ;
```

Having generating a parser succesfully, I will now move on to the next section. Here, we will use the bnfc parser to write out the abstract syntax trees (in 2-dimensional notation) for 8 expressions. The first photo shows the 2-dimensional trees and the second code section shows the generated linearized abstract syntax tree I got from the bnfc-generated parser:



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

As the final part of the homework, I finish the workout we began in class. This workout is a pen and paper computation based on the same definitions we described above.



```
eval ( vara)

                             e3
                      e1            e2
(πx.πy. x)y ⟶ eval (App( Absx ( Absy (Var x) )) (var y)) =
               eval (subst x  (eval (var y)) (Abs y (var x)))= // rename variable
               eval (subst x  (eval (var Y )) (Abs y (var x)))=
                          ↑            ↑                  ↑
                         id    what we substitute   what we substitute into

               eval(subst x  (var Y ) (Abs y ( var x ))) =
               eval( Abs y (Var Y)) = \\ πy. Y
               eval (Abs i e) = Abs i (eval e)
               Abs y (eval (var Y ))=
               eval x = x                              simplifies to
               Abs y (var Y )  in other words:              ↓
                                         (πx.πy. x)y ⇒ πy. Y
```

```
eval :: EXP ⟶ EXP
eval (App e1 e2) = case eval e1 of
      (Abs i e3) -> eval (subst i (eval e2) e3)
        e3 -> App e3 (eval e2)
eval (Abs i e) = Abs i (eval e)
eval x = x
```

2.5 Week 5

. . .

3 Conclusions

(approx 400 words) A critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of software engineering? What did you find most interesting or useful? What improvements would you suggest?

References

[ALG] Algorithm Analysis, Chapman University, 2023.