CPSC-354 Report

Natalie Huante
Chapman University

October 15, 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
```

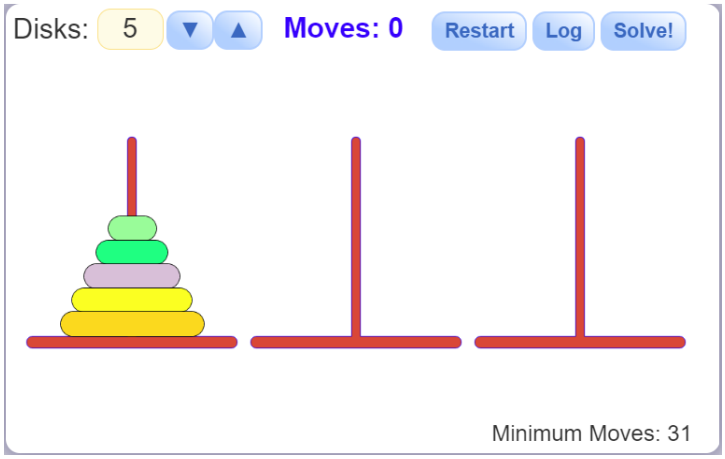
Now, I will write out the full computation of gcd(9,33) below:

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

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:



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

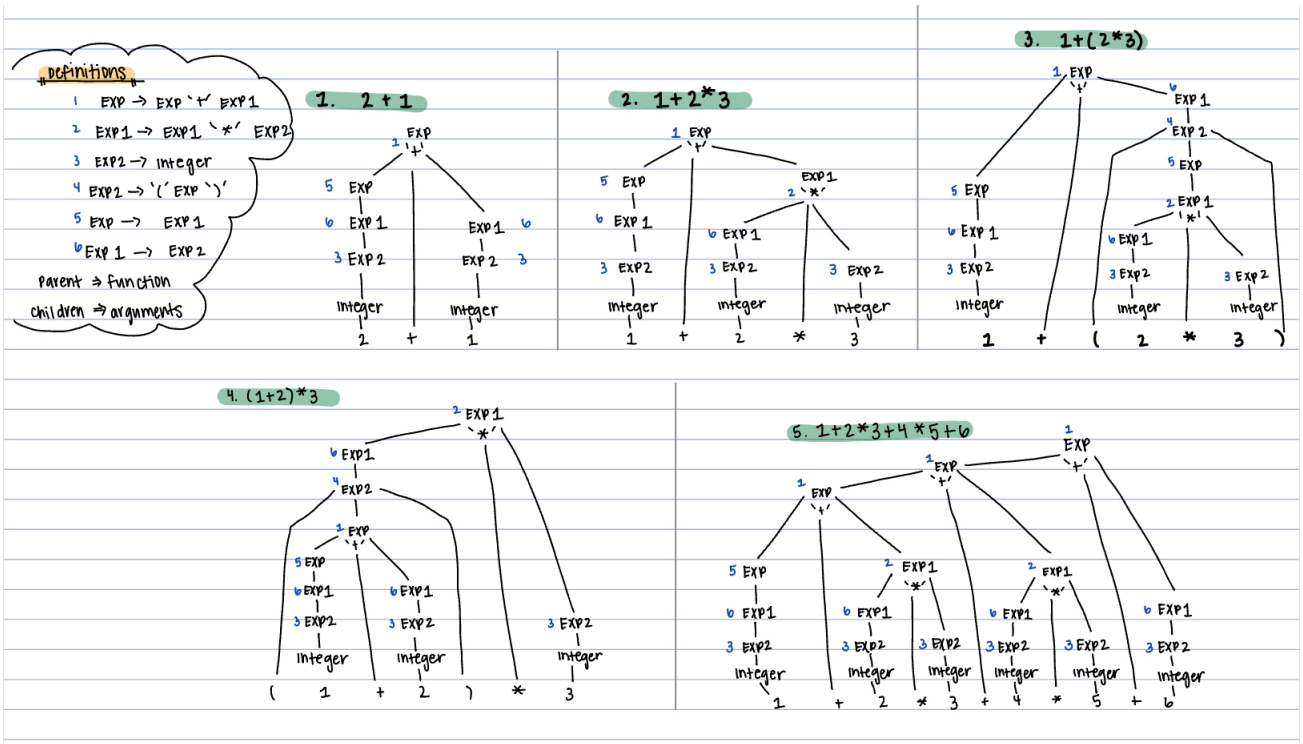
2.3 Week 3

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
```

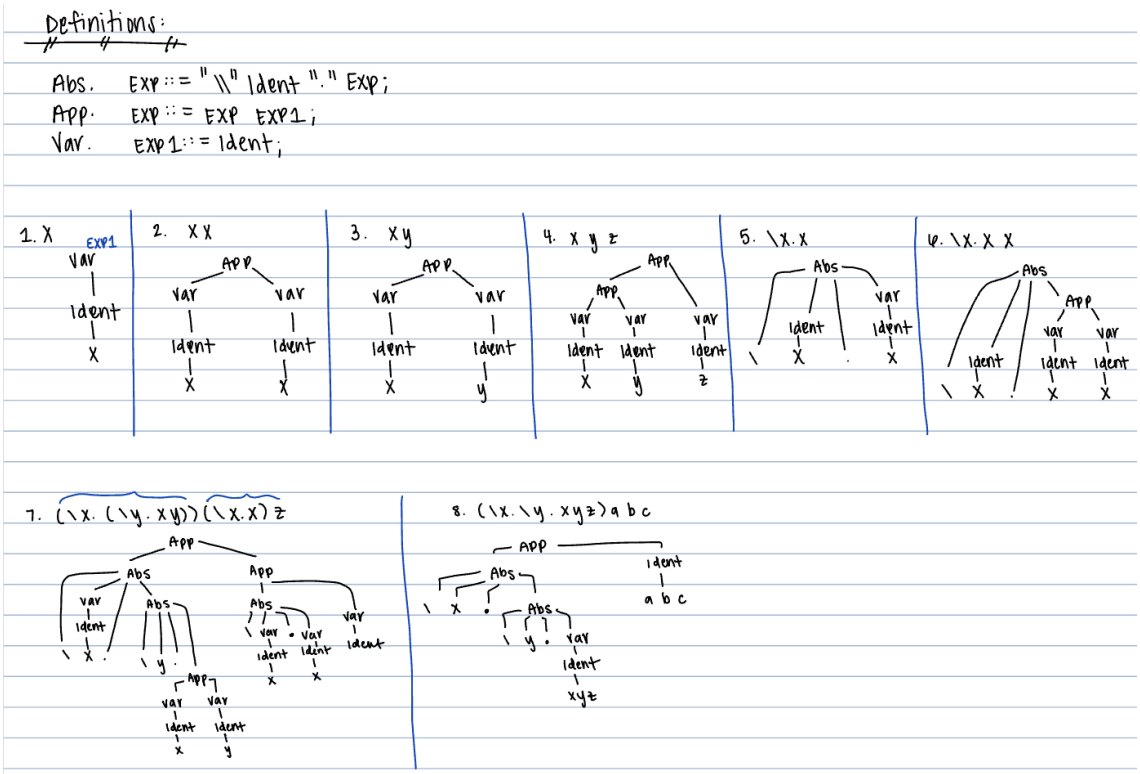
As you can see, the abstract syntax outputted differs depending on where or if the paranthesis are placed in the expression. The first and last output do yield the same result, implying that if the parser is not given any paranthesis, it will prioritize executing from left to right. In this case, the 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:

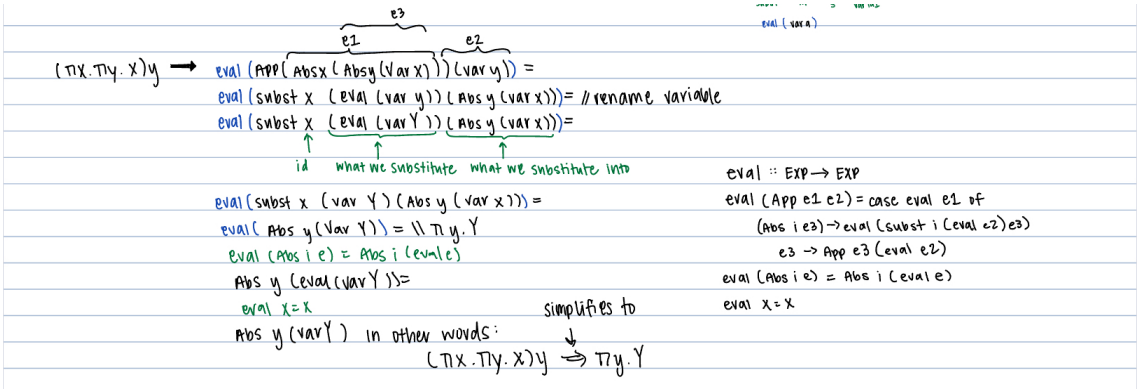
```
Abs.   Exp ::= "\\" Ident "." Exp ;
App.   Exp ::= Exp Exp1 ;
Var.   Exp1 ::= Ident ;

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.



2.5 Week 5

For this week's homework we will review two main topics. The first is our discussion of variables, bindings, scope, and substition and the second is our practice of understanding the interpretation of lambda calculus. Before I include the excercises for the section, I will include a review of the terminology and concepts we discussed. To start, there are three types of variables we introduced; they are as follows:

- fresh variable - a variables that has not been used before
- bound variable - a variables that can be renamed by a fresh variable without changing the meaning
- free variable - a variable that is not bound and receives its meaning from the context in which it appears

For example, in the expression `\x.x+y` the x is a bound variable and the y is a free variable. When we look at the scope of a variable, this simply means we have to take account the binding and paranthesis that apply to said variable. It's placement in the expression will affect the scope of the variable and therefore the meaning of it. This is a common topic in programming, especially as we turn to look at functions, classes, and their respective variables. As a simple example, take the following function.

```
  var f = function(a) {
    return a+3;
}
```

In the function defined above, the variable a only exists within the brackets of the function. Therefore, that is the scope of a. If a is referenced outside of its scope, an error will occur. Similarly, we can look at lambda calculus variables in the same way. The last term I will review is substitution. Substition is a common practice in mathematics, but it is important in lambda calculus as we begin to introduce the topic of capture avoiding substition. The reason we use CAS is to prevent combining variables that may appear to be the same, but are not necessary equal. Therefore, the substition we will perform will avoid the "capture" of these variables. Essentially, the idea is to look at the scope of the variables in the current expression and assess whether it is necessary to replace some of them with a fresh variable in order to dsitinguish the difference between itself and other variables in the following calculations. To see this in practice, I include the exercise provided for the homework below.

Definition: The function

$$f \circ g$$

(pronounced "f after g") is the function obtained from taking the output of g as the input of f. In the example, we want to replace the x in $f(x) = x + y$ by $g(y) = y * 2$. It is tempting to write

$$f \circ g(y) = f(g(y)) = 2 * y + y = 3y$$

but this gives the **wrong** forumlformula for $f \circ g$. Such mistakes arise from not taking proper care of the distinction of free and bound variables.

1. What is the correct formula for the function $f \circ g$ in the example above?
2. Use the notions of free and bound variables and score to explain your answer.
3. Using lambda calculus write down a general formula for $f \circ g$ that works for abritrary f and g.

Given the topic I have reviewed above, we can identify there is an issue in the above calculation regarding the variable y and its scope. Although the y variable in the functions f and g would be acceptable in many mathematical scenarios, this is not necessarily the case for us. Given that y is a free variable in the function f and a bound variable in the function g, we do not have enough information to ensure that they refer to the same value. It is possible that each y is referring to a unique value. In other words, the binding to y is not in the scope of the instance y in the function f. Therefore, we cannot assume that it is bound to the same value. So, it is incorrect to simplify the expresion to $3y$. In order to correct this formula, we must use capture avoiding substitution. We can assign the y in $f(x)$ a fresh variable to distinguish it from any other instances of y. The correct formula would therefore be:

$$f(x) = x + Y$$
$$g(y) = y * 2$$

$$\begin{aligned} f \circ g(y) &= f(g(y)) \\ &= 2 * y + Y \\ &= 2y + Y \end{aligned}$$

Since the Y is a free variable it will remain in the solution as a variable. Meanwhile the bound y will be substituted with a value if one is provided. As a general formula for $f \circ g$, we can represent it as:

$$f \circ g(y) = \texttt{\textbackslash y . f(g(y))}$$

As you can see, we do not include the x variable in this formula because it is being replaced by the output of $g(y)$. If we assign a value to $f \circ g$ we will only have to pass in one value, that of y, so we do not need it in the formula.

Now, for the second half of this homework, we will switch into the discussion of the lambda calculator interpreter we have been using and gaining a deeper understanding into how and when the rules apply. This will also give us a better understanding of the topics we already mentioned such as variables, bindings, scope, and substitution. We were given the following task:

To understand how the interpreter works do the following.

- Use Interpreter.hs to evaluate in the style above (equational reasoning as in Example 1 and 2)

  ```
  (\x.\y.x)y
  ```

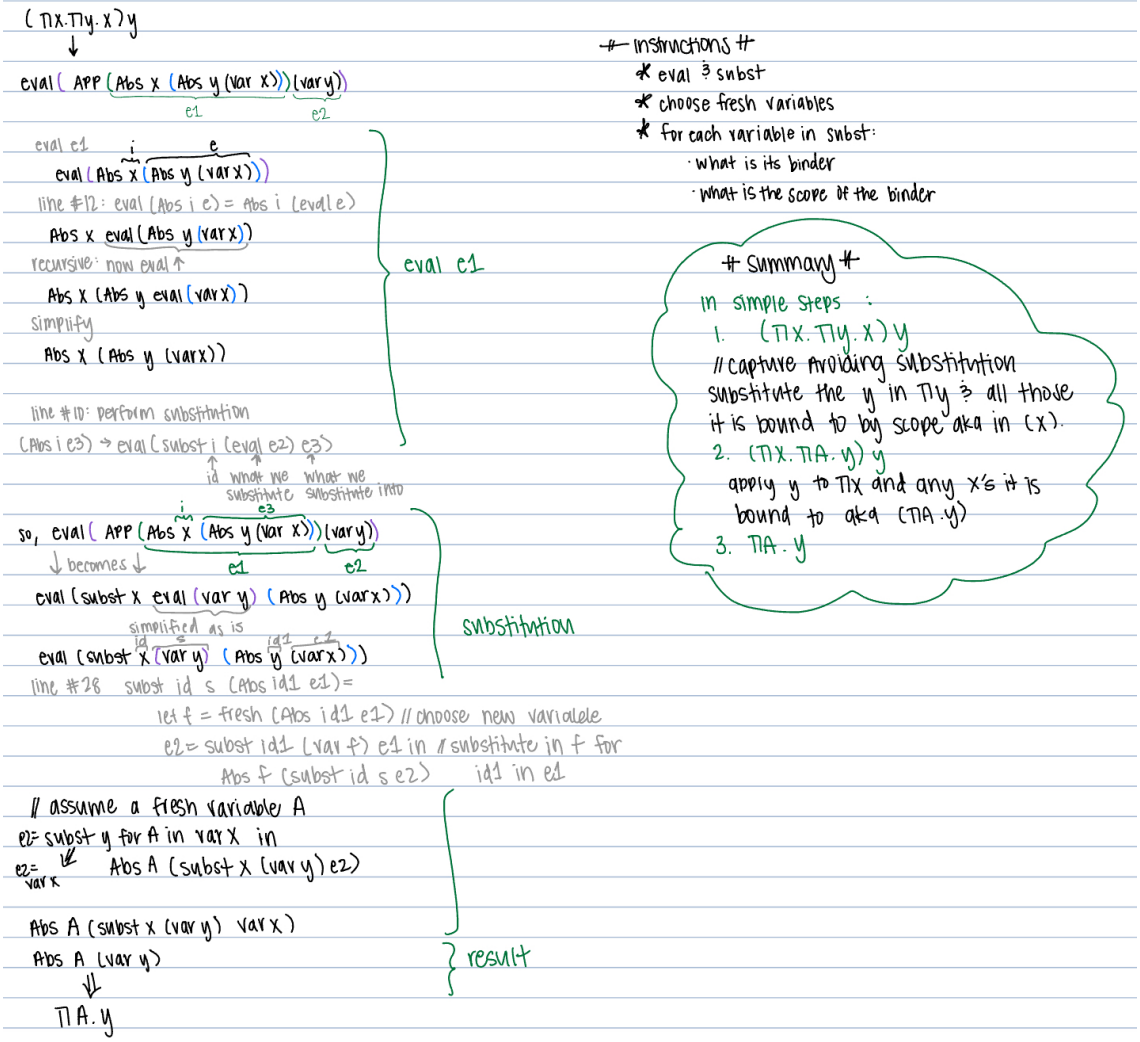
  Follow the implementation of eval and subst line by line, but choose your own fresh names to simplify the computation.

- For each variable appearing in subst say what its binder is and what the scope of the binder is.

Below I will include three pictures. The first is a simple screenshot of the Interpreter.hs file in order to understand what lines I am referring to in my computation. The second picture is a screenshot of the terminal output when the interpreter file is run with the input of the function above. The third picture is my handwritten computation of the equation `(\x.\y.x)y`. In my computation, the lines in grey represent my explanations or references to what is prompting the next step of computation. I also used different color parenthesis to avoid any misunderstanding of scopes of variables. Overall, the computation is quite simple. We must first evaluate the expression e1, which results in a recursive evaluation of both Abs cases in the expression. Once we have evaluated this, we can go back to the original expression and replace e1 for its evaluated value. Then, we can look at the definition of how to evaluation an App expression. This leads us to a substition. The reason this substitution is important is due to the fact that both y's in the original expression are not necessarily equal in value. We must substitute one of them to avoid combining them. In this example, we substitute the `\y` for a reason. If we prioritize substitution this variable, we can easily replace the instances of y it is binding. In this case, there are no y's to replace in the body of the `\y.x` expression. Therefore, our result to this expression is `\A.y`. Since there are no A's in the body of this, the expression would simply evaluate to `y`.

```
1   module Interpreter where
2
3   import AbsLambda -- provides the interface with the grammar and is automatically generated by bnfc
4   import ErrM
5   import PrintLambda
6   import Control.Monad.State
7
8   eval :: Exp -> Exp
9   eval (App e1 e2) = case eval e1 of
10      (Abs i e3) -> eval (subst i (eval e2) e3)
11      e3 -> App e3 (eval e2)
12  eval (Abs i e) = Abs i (eval e)
13  eval x = x
14
15  -- a quick and dirty way of getting fresh names. Rather inefficient for big terms...
16  freshAux :: Exp -> String
17  freshAux (Var (Id i)) = i ++ "0"
18  freshAux (App e1 e2) = freshAux e1 ++ freshAux e2
19  freshAux (Abs (Id i) e) = i ++ freshAux e
20
21  fresh = Id . freshAux
22
23  -- (\id.e)s reduces to `subst id s e`
24  subst :: Id -> Exp -> Exp -> Exp
25  subst id s (Var id1) | id == id1 = s
26                       | otherwise = Var id1
27  subst id s (App e1 e2) = App (subst id s e1) (subst id s e2)
28  subst id s (Abs id1 e1) =
29      let f = fresh (Abs id1 e1)
30          e2 = subst id1 (Var f) e1 in
31          Abs f (subst id s e2)
32
```

```
PS C:\Users\natal\Documents\CPSC_Courses\CPSC354\LambdaCalculus>  echo "(\x.\y.x)y" | cabal exec Lambda

 Input:
(\ x . \ y . x) y

[Abstract Syntax]

App (Abs (Id "x") (Abs (Id "y") (Var (Id "x")))) (Var (Id "y"))

 Output:
\ yx0 . y
```



2.6 Week 6

The aim of this week's homework is to review what we learned in class about ARSs. An ARS, or an Abstraction Reduction System, provides you with rules that will allow you to reduce any expression composed of the letters in the given alphabet. I will first include the example we did in class and then include the excercises given to us as practice.

So, the in-class example is as follows. Given the rewrite rules below, answer these questions:

```
aa -> a
bb -> a
```

```
ab -> b
ba -> b
```

- **Why does the ARS terminate?**

  This ARS terminates when there is only one character left in the expression because this means there are no more rules that will shorten the string. In other words, none of the given rewrite rules can be applied. Following this logic, there are three final outcomes that an expression can be simplified to. These are 'a', 'b', and e. (Note: e stands for epsilon, which represents the empty string). The ARS cannot apply any of these rules to a single character or no character because each rule being with a pair of letters. Each string will simplify to a single character because the rules account for every possible combiantion of a and b.

- **What are the normal forms?**

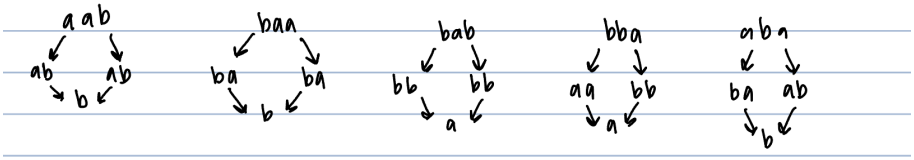
  The normal forms, or the strings that you cannot rewrite, are `a`, `b`, and `e`.

- **Is there a string `s` that reduces to both `a` and `b`?**

  There is not a string that reduces to both a and b.

- **Show that the ARS is confluent.**

  An ARS is confluent if every "peak" contains a "valley". To test this, we will take the five cases of overlapping pairs, or the critical pairs, and compute whether these two expressions will converge. In the picture below you can see how I tested these cases. The reason we want these to converge is because we want to see if they will all possible routes will simplify to the same end result. So, we take each possible overlap and draw out the routes one could take to simplify it. In these cases, the left arrow points to the string that results from applying the appropriate rewrite rule to the left-most pair of letters in the original string and copying over the last letter. The right arrow points to the string that reults from copying over the first letter and applying the appropriate rewrite rule to the right-most pair of letters. We can see that in every case, the different routes of computation converge to the same end string. Therefore, we can say that this ARS is confluent.



- **Replacing `->` by `=`, which words become equal?**

  If we replace `->` by `=`, then all words that simply to `a` become equal and all words that simplify to `b` become equal. Logically, this makes sense because if two strings simplify to the same letter, then the rewrite rules can be used backwards to result in either string. As we discussed this question and as I tested many cases to understand the nature of the rules, we also discovered that if a string contains an odd number of `b`s its normal form is `b` and if a string contains an even number of `b`s then its normal form is `a`.

Now having reviewed the in-class example, we will look at the assigned exercises. First we are prompted to draw a picture for each of the following ARSs:

1. `A = {}`
2. `A = {a} , R = {}`

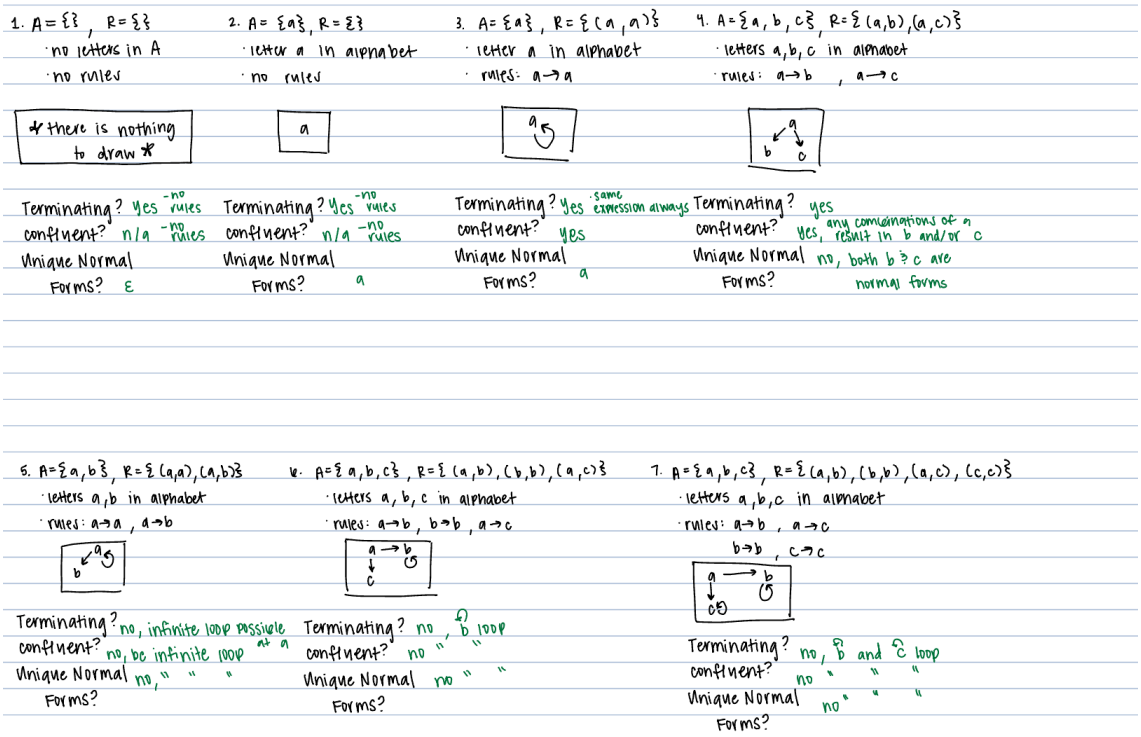
3. A = {a} , R = {(a,a)}

4. A = {a, b, c} , R = {(a, b), (a, c)}

5. A = {a, b} , R = {(a, a), (a, b)}

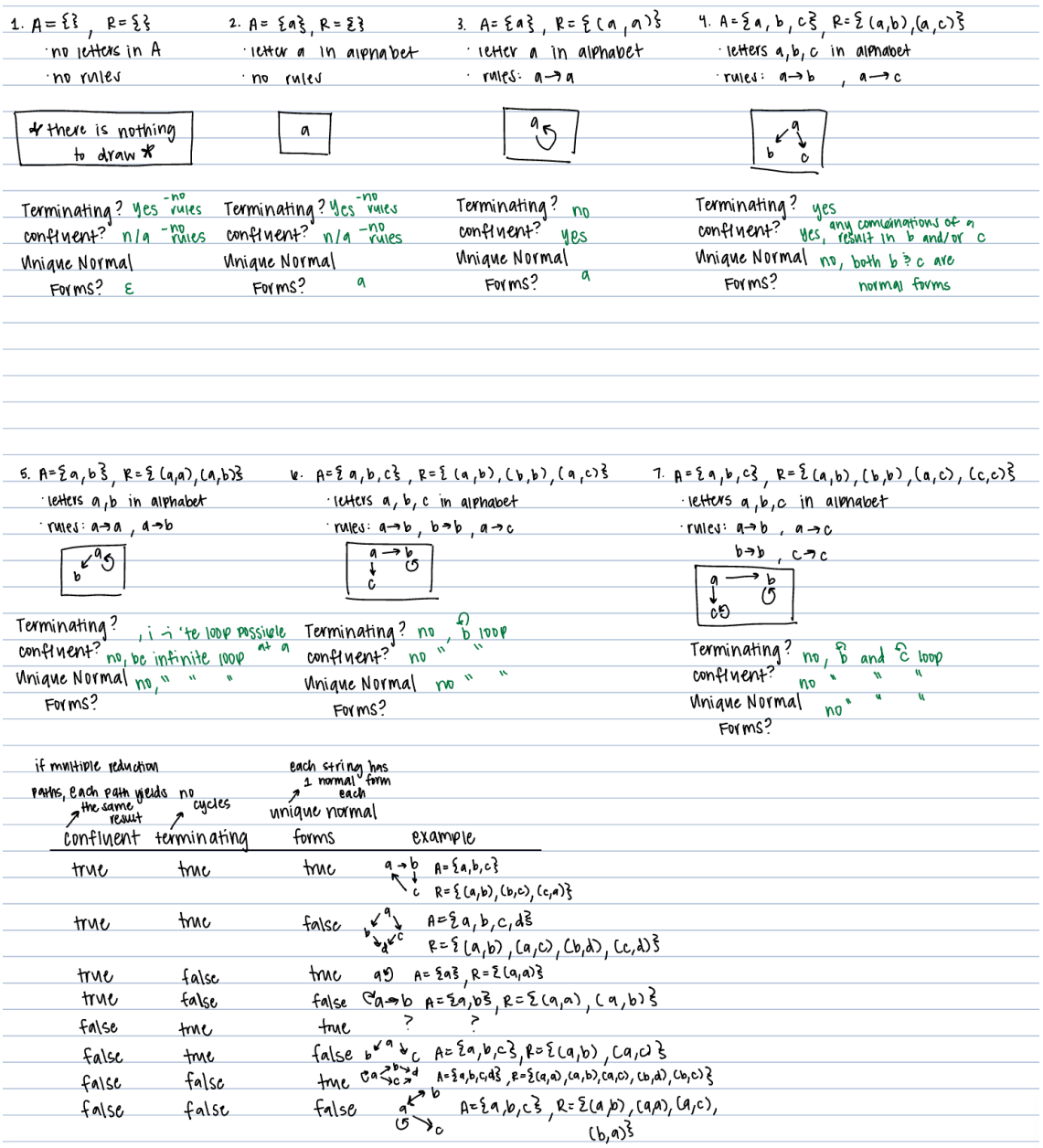
6. A = {a, b, c} , R = {(a, b), (b, b), (a, c)}

7. A = {a, b, c} , R = {(a, b), (b, b), (a, c), (c, c)}



Lastly, we will find an example ARS for each of the following combinations in the table. I include my solution as a screenshot from my digital handwritten notes.

confluent	terminating	unique normal forms
true	true	true
true	true	false
true	false	true
true	false	false
false	true	true
false	true	false
false	false	true
false	false	false

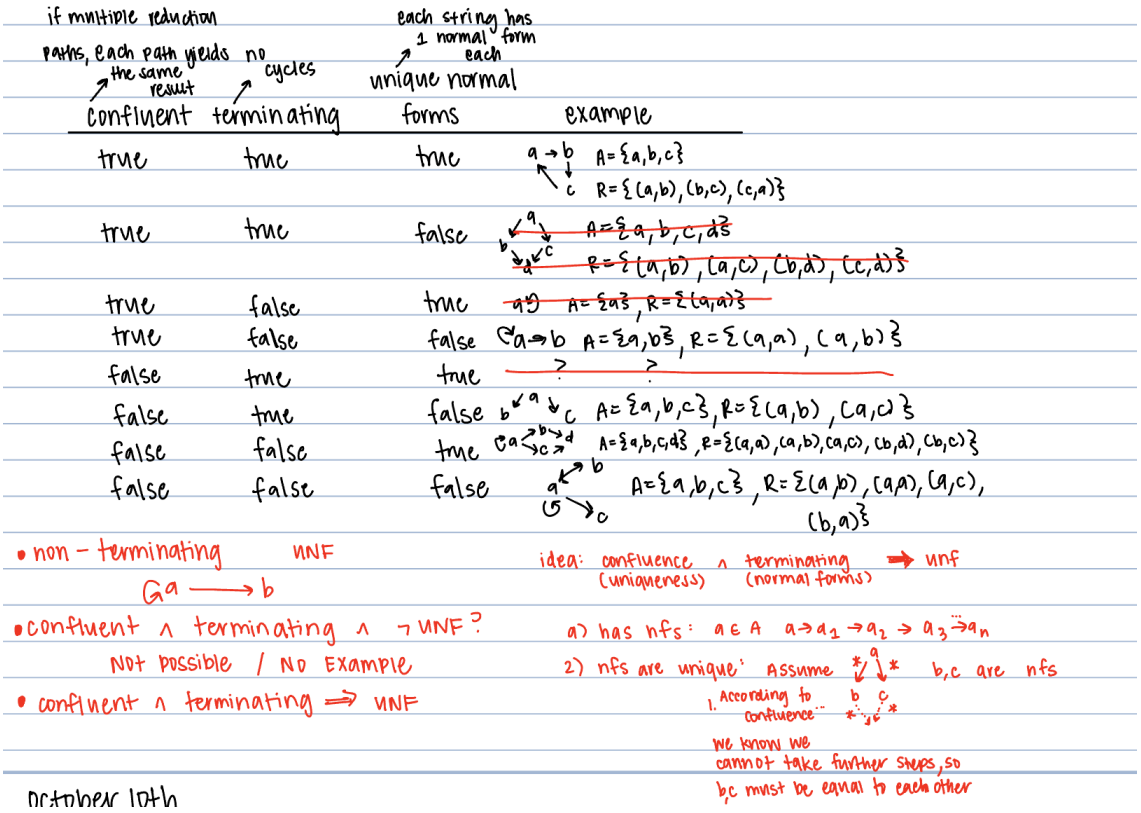


if multiple reduction paths, each path yields the same result

each string has 1 normal form each

confluent	terminating	unique normal forms	example
true	true	true	a → b, a → c; A = {a,b,c}, R = {(a,b),(b,c),(c,a)}
true	true	false	A = {a,b,c,d}, R = {(a,b),(a,c),(b,d),(c,d)}
true	false	true	A = {a}, R = {(a,a)}
true	false	false	a ⇄ b; A = {a,b}, R = {(a,a),(a,b)}
false	true	true	? ?
false	true	false	A = {a,b,c}, R = {(a,b),(a,c)}
false	false	true	A = {a,b,c,d}, R = {(a,a),(a,b),(a,c),(b,d),(c,c)}
false	false	false	A = {a,b,c}, R = {(a,b),(a,a),(a,c),(b,a)}

EDIT: I add this note after having gone over a few of the examples above during our lecture. One of the main ideas we discussed was whether we could say that any of the three properties (that is: confluence, termination, and unique normal forms[UNF]) implies the others. We were ultimately able to agree upon the idea that if an ARS is both confluent and terminating then this implies it has UNFs. The confluence gives the ARS its uniqueness given that all "peaks" have a "valley" that lead it to the same end result. The termination of the ARS means it has normal forms. So, together they imply the existence of unique normal forms for the ARS. Having this new understading of the relationships of these three properties, we can look back at my notes and note that some of my examples are incorrect. Specifically, the row that describes an ARS that is confluent and terminating but does not have UNFs. This is not possible since the first two imply the last property. From my understanding this would also mean that when either of the first two are false, the UNFs property must be false as well. I include an edited version of my notes below.



2.7 Week 7

...

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.