CPSC-354 Report

Everett Prussak
Chapman University

September 28, 2022

Abstract

Consisting of CPSC 354 material

Contents

1 Introduction

This Report contains mainly consists of the Homework for each week, and the main project. In section 1, you can find the stuff done by Professor Kurz. Section 2 consists of Homework for each week. Section 3 consists of the project.

1.1 General Remarks

First you need to download and install LaTeX.[1] For quick experimentation, you can use an online editor such as Overleaf. But to grade the report I will used the time-stamped pdf-files in your git repository.

LaTeX is a markup language (as is, for example, HTML). The source code is in a `.tex` file and needs to be compiled for viewing, usually to `.pdf`.

If you want to change the default layout, you need to type commands. For example, `\medskip` inserts a medium vertical space and `\noindent` starts a paragraph without indentation.

Mathematics is typeset between double dollars, for example

$$x + y = y + x.$$

1.2 LaTeX Resources

I start a new subsection, so that you can see how it appears in the table of contents.

1.2.1 Subsubsections

Sometimes it is good to have subsubsections.

[1]Links are typeset in blue, but you can change the layout and color of the links if you locate the `hypersetup` command.

1.2.2 Itemize and enumerate

- This is how you itemize in LaTeX.
- I think a good way to learn LaTeX is by starting from this template file and build it up step by step. Often stackoverflow will answer your questions. But here are a few resources:
  1. Learn LaTeX in 30 minutes
  2. LaTeX – A document preparation system

1.2.3 Typesetting Code

A typical project will involve code. For the example below I took the LaTeX code from stackoverflow and the Haskell code from my tutorial.

```haskell
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

Short snippets such as `run :: (State -> Char -> State) -> State -> [Char] -> State` can also be directly fitted into text. There are several ways of doing this, for example, `run :: (State -> Char -> State) -> State ->` is slightly different in terms of spaces and linebreaking (and can lead to layout that is better avoided), as is

`run :: (State -> Char -> State) -> State -> [Char] -> State`

For more on the topic see Code-Presentations Example.

Generally speaking, the methods for displaying code discussed above work well only for short listings of code. For entire programs, it is better to have external links to, for example, Github or Replit (click on the ”Run” button and/or the “Code” tab).

1.2.4 More Mathematics

We have already seen $x + y = y + x$ as an example of inline maths. We can also typeset mathematics in display mode, for example

$$\frac{x}{y} = \frac{xy}{y^2},$$

Here is an example of equational reasoning that spans several lines:

$$\begin{aligned}
\mathrm{fib}(3) &= \mathrm{fib}(1) + \mathrm{fib}(2) && \mathrm{fib}(n+2) = \mathrm{fib}(n) + \mathrm{fib}(n+1)\\
&= \mathrm{fib}(1) + \mathrm{fib}(0) + \mathrm{fib}(1) && \mathrm{fib}(n+2) = \mathrm{fib}(n) + \mathrm{fib}(n+1)\\
&= 1 + 0 + 1 && \mathrm{fib}(0) = 0, \mathrm{fib}(1) = 1\\
&= 2 && \text{arithmetic}
\end{aligned}$$

1.2.5 Definitons, Examples, Theorems, Etc

Definition 1.1. This is a definition.

Example 1.2. This is an example.

Proposition 1.3. *This is a proposition.*

Theorem 1.4. *This is a theorem.*

You can also create your own environment, eg if you want to have Question, Notation, Conjecture, etc.

1.3 Plagiarism

To avoid plagiarism, make sure that in addition to [**?**] you also cite all the external sources you use. Make sure you cite all your references in your text, not only at the end.

2 Homework

This section will contain your solutions to homework.

2.1 Week 1

Homework 1: Using Euclid's Elements Proposition 2 Algorithm on finding the Greatest Common Divisor amongst two numbers.

2.1.1 Python

In python, this algorithm can be written as:

```
a = 9
b = 33

while(a!=b):
    if(a>b):
         a = a-b
     else:
         b = b-a

print(a)
```

Using the sample given, 9 and 33, the Greatest Common Divisor is found in a simple way. Using the Euclid's Elements Proposition 2 Algorithm, we have two variables: **a** and **b**. To start, our samples are manually entered from the start of the program, and will begin the while loop. The while loop will continue until the **a** is the same value as **b**. The first line of code in the loop is an **if-statement**. This will compare **a** and **b** values, and will continue inside the **if-statement** if **a** is a large value than **b**. The Euclid's Elements Algorithm says: If

$$a > b$$

then replace **a** by

$$a - b$$

This is what happens in this first **if-statement**, as the variable **a** is replaced with a - b

If the **if-statement** is not executed, then the else statement will be preformed. Since our while loop tells us that it will continue until **a** is the same value as **b**, then we know that this else statement is:

$$b > a$$

This is the second part of the Euclid's Elements Proposition 2 Algorithm. It says that when:

$$b > a$$

to replace **b** with

$$b - a$$

This is what happens in this line of python code. The variable **b** is clearly replaced with b - a.

This while loop will continue until the values of **a** and **b** are the same. Once they are, the program will print the value of **a**. In this particular example, the value **3** would be printed.

2.1.2 C++

In C++, the algorithm can be written as:

```
#include <iostream>

using namespace std;

int main(int charc, char** argv){
    int a = 9;
    int b = 33;

    while(a!=b){
        if(a>b){
            a = a-b;
        }
        else{
            b = b-a;
        }
    }
    cout << a << endl;
}
```

Very similar code to the code in 2.1.1. The same process is being used, with two variables being created before the while loop. The while loop will continue until the values of the two variables are the same. Then the two conditions of

$$a > b$$

and

$$b > a$$

, are evaluated using an **if-statement** and **else-statement**. Once the correct condition is identified, the corresponding calculation done of each variable takes place. This will continue until the Greatest Common Divisor is found, and is printed to the screen. In this place, 3 is again printed.

2.2 Week 2

Homework 2: Create six Functions using Haskell: Select Evens, Select Odds, Member, Append, Revert, and Less Equal.

2.2.1 Select Evens and Select Odds

The task of these two functions could be used with each other. For Select Evens, the user would write the function name, then a list. The output would be the even element indices (Starting with 1 not 0). Here is the code for both.

```
select_evens [] = []
select_evens (x:xs) = select_odds xs

select_odds (x:xs) = x : select_evens xs
select_odds [] = []
```

These two functions are connected. Without one, the other will not work.

Going through the program, we will start with Select Odds. Using recursion, the head element will be split off first. Select Evens will be called with the other elements left in the list. Select Evens will then split the head and other elements again. This process will continue until the tail is an empty list. These methods will allow for only the odd indexed elements to be printed, or only the even indexed elements.

Here are a few outputs from the terminal:

```
ghci> select_evens ["a","b","c","d","e"]
["b","d"]
ghci> select_odds ["a","b","c"]
["a","c"]
ghci> select_odds ["a","b","c","d","e"]
["a","c","e"]
ghci> select_odds [1,2,3,4,5]
[1,3,5]
ghci> select_evens[432,34,543,2334,23]
[34,2334]
```

Here is the Task 2 Equational Reasoning for the Select Evens and Odds Function:

```
select_evens ["a","b","c","d","e"] =

select_evens ("a" : ["b","c","d","e"]]) = select_odds ["b","c","d","e"]

select_odds ("b" : ["c","d","e"]) = "b" : select_evens["c", "d", "e"]

select_evens ("c" : ["d", "e"]) = select_odds["d", "e"]

select_odds ("d" : ["e"]) = "b" : ("d" : select_evens ["e"])

select_evens ("e" : []) = select_odds []

select_odds [] =  "b" : ("d" : ([]))

["b", "d"]
```

Note that if Select Odds was called, the same procedure would occur, but would start with Select Odds first and would be the opposite of this output.

2.2.2 Member

In the member function, the user would ask for a ”True” or ”False” about if a list consisted of a particular element. If the list consisted of the element, then True would be returned. Otherwise, False is returned.

```
member y (x:xs) = if y==x
    then True
    else if len(xs) < 1
        then False
        else member y xs
```

The element that the user wants to know is the y. It starts off by comparing y to x, which is the head of the current list. If y is the same as x, then the element that the user is looking for is indeed in the list and True is returned. However, if it is not, then we will use the len function to see the size of the rest of the list. If it is not above 1, then False is returned. False is returned because there is nothing left to compare to the user element. However if it is above 1, then member is called again with the user element and the rest of the elements. This simulates the element being compared to each element in the list.

Here are some outputs for this function:

```
ghci> member "a" ["a", "b"]
True
ghci> member "a" ["c", "d", "e"]
False
ghci> member 4 [2,3,5]
False
ghci> member 4 [2,3,3,4,5]
True
ghci> member 4 [4,1,6,2]
True
```

Here is the Task 2 Equational Reasoning for the Member Function:

```
member 4 [2,3,4] =
    4 == 2
    else if len([3,4]) < 1

    len[3,4] = 3

    else if 3 < 1
        then False
        else member 4 [3,4]

member 4 [3,4]
    4 == 3
    else if len([4]) < 1

    len[4] = 2

    else if 2 < 1
        then False
        else member 4 [4]

member 4 [4]
    4 == 4
    then True
```

2.2.3 Append

The append function takes two lists from the user, and appends all of the second lists elements to the back of the first list. This function was difficult for me at first, but I realized that the first list stays at the front of the list, and that only the second list needed elements to ”move”.

Below is my code for the Append Function:

```
append [] (y:ys) = if len(ys) > 0
    then
        y : append [] ys
    else
        [y]
append (x:xs) (y:ys) = x : append xs (y:ys)
```

The last line will be the first line that executes. The user will have the two lists, but the function will have to iterate the entire (x:xs) list first. This is because these elements will be in the front of the new list regardless. After this line is recursively called, the first line will be called with an empty list and all of the second list. The len function is also used in this function as well. The if statement looks at the size of the tail elements of the second list. If it is greater than 0, then it will recursively add the elements to the list that already contains the first list elements. Once the len(ys) is not greater than 0, the else statement will have [y] which will basically not recall any of the append functions, and will not append anything to the list. This will let the recursion end.

Here are some outputs of the append function:

```
ghci> append [1,2] [3,4,5]
[1,2,3,4,5]
ghci> append [1,2,3,4,5] [7,8,9]
[1,2,3,4,5,7,8,9]
```

Here is the Task 2 Equational Reasoning for the Append Function:

```
append [1,2,3] [7,8,9] =
    1 : (append [2,3] [7,8,9])
    1 : (2 : (append [3] [7,8,9]))
    1 : (2 : (3 : (append [] [7,8,9])))
    1 : (2 : (3 : (7 : (append [8,9]))))
    1 : (2 : (3 : (7 : (8 : (append [9])))))
    1 : (2 : (3 : (7 : (8 : (9 : (append[]))))))
    1 : (2 : (3 : (7 : (8 : (9 : [])))))
```

2.2.4 Revert

The Revert function will take a list from the user, and output the list with the elements in reversed order.

Below is the code to the Revert Function:

```
revert [] = []
revert (x:xs) = append (revert xs) [x]
```

This function uses the previously created function Append. Revert will call the append method recursively. It will append the first element of the list to the back, then continue with the rest of the elements. This was designed with some thought of a Stack.
Here are some outputs for the Revert Function:

```
ghci> revert [1,2,3]
[3,2,1]
ghci> revert [8,1,2,4]
[4,2,1,8]
```

Here is the Task 2 Equational Reasoning for the Revert Function:

```
revert [1,2,3] =
    append (revert [2,3]) [1]
    append (append (revert([3]) [2])) [1]
    append (append (append (revert []) [3]) [2]) [1]
    append (append (append [] [3]) [2]) [1]
    append (append [3] [2]) [1]
    append [3,2] [1]
    [3,2,1]
```

2.2.5 Less Equal

The last function that I created was Less Equal. This simply compared the elements from two lists to see if first list was less than or equal to the same indexed element of the other list. For example [1,2] vs [5,6] would be true because 1 is less than or equal to 5 and 2 is less than or equal to 6.

Below is the code to the Less equal Function

```
less_equal [] [] = True
less_equal (x:xs) (y:ys) = if x<=y
    then
        less_equal xs ys
    else
        False
```

The beginning of this function will start with the second line. If compares the head elements of list 1 and list 2. If x is less than or equal to, the Less Equal function will be called again, but this time with the tail elements. If at any point the list 1 element "x" is greater than list 2 element "y", False will be returned. If all of the elements are compared, then base case is called. This is because there are no elements left, so they are empty lists. Since nothing was flagged, then True will be returned, meaning that all of the list 1 elements are either less than or equal to the list 2 elements at the same index.

Here are some outputs for the Less Equal Function:

```
ghci> less_equal [1,2,3] [2,3,4]
True
ghci> less_equal [1,2,3] [2,3,2]
False
ghci> less_equal [1,2,3] [2,3,3]
True
```

Here is the Task 2 Equational Reasoning for the Less Equal Function:

```
less_equal [1,2,3] [2,3,4] =
    1 <= 2
    then
        less_equal [2,3] [3,4]

    2 <= 3
    then
        less_equal [3] [4]

    3 <= 4
    then
        less_equal [] []

    True
```

If the left value was ever larger, than False would be the value.

2.2.6 Len

I did not create this function. I took this from Hackmd.io, thanks to Professor Kurz. Below is the code used that my other functions also used.

```
len [] = 0
len (x:xs) = 1 + len xs
```

2.3 Week 3

Week 3 Homework was about the Hanoi Tower. The task was to complete the execution from the dots down. Here that is:

```
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
```

I followed this exact sequence with the Tower of Hanoi website. It took 31 turns, which is the minimum moves possible for a Hanoi Tower of size 5.

The word "Hanoi" appears 31 times. The word "move" also appears 31 times.
A simple equation can be expressed for the number of blocks in the Hanoi Tower. Assuming n is the number of blocks/rings in the tower, the minimum number of moves can be expressed as

$$2^n - 1$$

In this case, n was 5. Thus,

$$2^5 = 32$$

and then

$$32 - 1 = 31$$

Here are some other minimum moves required for other ring heights.

$$2^1 - 1 = 1$$

$$2^2 - 1 = 3$$

$$2^3 - 1 = 7$$

$$2^4 - 1 = 15$$

$$2^6 - 1 = 63$$

2.4 Week 4

Week 4 consisted of making Parse Tree and Abstract Syntax Tree of math 5 problems.

2.4.1 Parse Tree Diagrams

This is Part 1 of Week 4's Homework. The task was to write out the derivation trees for the following:

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

while using this Context-Free Grammar:

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

For

$$2 + 1$$

I got the following Parse Tree:

```
        Exp
       /    \
      Exp    Exp1
     /         \
    Exp1        Exp2
   /             \
  Exp2            1
 /
2
```

Here this tree starts with Exp. It expands for addition with Exp and Exp1. After this, I simply converted Exp to Exp1 to Exp2 then the number 2, and Exp1 to Exp2 then the number 1.

For

$$1 + 2 * 3$$

I got the following Parse Tree:

```
         Exp
        /    \
      Exp    Exp1
      /      /   \
    Exp1   Exp1  Exp2
    /        |     |
   Exp2     Exp2   3
    |        |
    1        2
```

This Tree starts similar to last time with Exp. Exp will expand to Exp and Exp1. Exp will expand directly until it is the integer 1. Exp1 will expand to Exp1 and Exp2 for multiplication. Exp1 and Exp2 from this will become 2 and 3. This will allow the tree to calculate 2 times 3 to become 6. Then the root will have the two nodes of 1 and 6 to add to become 7.

For

$$1 + (2 * 3)$$

I got the following Parse Tree:

```
        Exp
       /    \
     Exp    Exp1
    /          \
   Exp1        Exp2
   |             |
   Exp2         Exp
   |             |
   1            Exp1
               /    \
             Exp1   Exp2
              |       |
             Exp2     3
              |
              2
```

This is a little more complex than the prior example, but will end with the same answer of 7. Exp will expand to Exp and Exp1 for addition. Exp will expand until it becomes the integer of 1. Exp1 will become Exp2 then Exp because this had parenthesis. Exp will expand to Exp1, then Exp1 will expand to Exp1 and Exp2. These will become 2 and 3 so it will be multiplied together. Then this will finally become 7 if calculated.

For

$$(1 + 2) * 3$$

I got the following Parse Tree:

```
        Exp1
       /   \
     Exp1   Exp2
    /         |
   Exp2      Exp
  /         /   \
 3        Exp   Exp1
           |     |
          Exp1  Exp2
           |     |
          Exp2   2
           |
           1
```

This time the parenthesis is with the addition of 1 and 2. I started with multiplication (Exp1) because it is 3 being multiplied with the entire parenthesis. Exp1 will expand to Exp1 and Exp2. Exp1 will become the integer 3. Exp2 will first go back to Exp because this is a parenthesis and our rule allows to go back to Exp when Parenthesis is present. Exp will then expand to Exp and Exp1 for the addition. These will expand until the integers 1 and 2 are present. Then 1 added to 2 will become 3. 3 multiple with 3 will become 9. This answer is 9.

For

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

I got the following Parse Tree:

```
                ____Exp______
               /              \
            _Exp__           Exp1
          /        \             \
         Exp        Exp1          Exp2
       /     \       /   \         |
      Exp   Exp1   Exp1  Exp2      1
     /      /  \     |    |
    Exp1  Exp1 Exp2 Exp2  3
     |      |    |    |
    Exp2   Exp2  5    2
     |      |
     6      4
```

This was by far the most complicated of the problems. To start, I expanded with the addition of 1 with everything else. This means Exp would be the root with Exp and Exp1 being the child nodes. Exp1 will become Exp2 then the integer 1. Now I must account for the other 5 numbers in the equation. Starting with Exp again I decided for the sum of 2 multiple with 3 being added to entire sum of 4 multiplied with 5 and then added with 1. Exp will expand to Exp and Exp1 for this addition. Exp1 will become the 2 times 3 side of the equation. Exp1 will expand to Exp1 and Exp2. Exp1 will expand until integer 2 is reached, and Exp2 is expanded until integer 3 is reached. Now back to the Exp that is being added with 2 times 3. This Exp will be expanded to Exp and Exp1. Exp will be expanded until integer 6 is reached. Exp1 is the multiplication formula. Exp1 will expand to Exp1 and Exp2. These two nodes will then be converted until integer 4 and 5 are reached. Now the tree is complete. First 1 will be added with the rest of the equation. The rest of the equation will have 2 multiplied with 3 become 6. 4 is multipled by 5 to become 20 then added with 6 to become 26. This 26 is added with 6 from 2 times 3. Then our 1 is added to become 33.

2.4.2 Abstract Syntax Tree

The same 5 problems (Equations) that were used for the Parse Tree's are now used to be made into Abstract Syntax Trees.

These AST are to be made using this BNFC Grammar:

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

```
coercions Exp 2 ;
```

For

$$2 + 1$$

I got the following Abstract Syntax Tree:

```
  Plus
  /  \
 Num  Num
 |     |
 2     1
```

This Abstract Syntax Tree is easy to read. The only math being used is an addition sign. Simply Plus is used then expanded to its two terms of 2 and 1.

For

$$1 + 2 * 3$$

I got the following Abstract Syntax Tree:

```
   Plus
  /    \
 Num   Times
  |   /   \
  1  Num  Num
      |    |
      2    3
```

This Abstract Syntax Tree is bigger than the prior equation/problem. I once again started with Plus then expanded 1 and the product of 2 times 3. This works with our Grammar Rules.

For

$$1 + (2 * 3)$$

I got the following Abstract Syntax Tree:

```
   Plus
  /    \
 Num   Times
  |   /   \
  1  Num  Num
      |    |
      2    3
```

This is the same exact AST as the prior example. Since the parenthesis is around the multiplication, which would be first in this instance anyways, nothing changes. Other than the parenthesis being added, nothing changes, so the rest of the AST is the same

For

$$(1 + 2) * 3$$

I got the following Abstract Syntax Tree:

```
  Times
  /   \
 Plus  Num
 /  \    |
Num Num  3
|    |
1    2
```

This AST is different than the prior examples. Since the parenthesis are around the 1 plus 2, then this part must be done first in order to be multiplied with the three. I had Times be the root with the expansion of integer 3, and Plus. The Plus would then expand to 1 and 2. By this tree, 1 Plus 2 is 3 then 3 times 3 is 9.

For

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

I got the following Abstract Syntax Tree:

```
    Plus
   /     \
  Num     Plus
  /     /       \
 1    Plus      Times
    /    \     /    \
  Times  Num  Num    Num
 /    \   |   |       |
Num  Num  6   2       3
 |    |
 4    5
```

This AST is much more complex than the other examples. To begin, I had Plus become the root, as 1 Plus other side of equation will be the child nodes. The nodes will be integer 1 and Plus. Plus will expand to Plus and Times. The Times will expand to the first part of the other equation of 2 Times 3. The Plus will expand to Times and integer 6. The Times will become 4 and 5. Doing this AST from bottom up, will have 4 Times 5 become 20. Then 20 Plus 6 becoming 26. 2 Times 3 becoming 6. 6 plus 26 becoming 32. Then finally 32 will be added with 1 to have 33 as the grand total, the same as the Parse Tree example.

2.5 Week 5

For this week's homework, the task was to create Abstract Syntax Trees and evaluate lambda expressions.

2.5.1 Abstract Syntax Trees of Lambda Calculus

The following expressions were given to create 2-D Abstract Syntax Trees and Linearized Abstract Syntax Trees.

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

Here is the Linearized and 2-D AST for x:

```
EVar(Ident "x")

        Evar
         |
         x
```

Here is the Linearized and 2-D AST for x x:

```
EApp(EVar(Ident "x") EVar(Ident "x"))
         EApp
        /    \
       Evar  Evar
        |     |
        x     x
```

Here is the Linearized and 2-D AST for x y:

```
EApp(EVar(Ident "x") EVar(Ident "y"))
        EApp
       /    \
      Evar  Evar
       |     |
       x     y
```

Here is the Linearized and 2-D AST for x y z:

```
EApp (EApp (EVar (Ident "x")) (EVar (Ident "y"))) (EVar (Ident "z"))
       EApp
      /    \
    Evar   EApp
     |    /    \
     z   Evar  Evar
          |     |
          y     x
```

Here is the Linearized and 2-D AST for

$$\backslash x.x$$

```
EAbs (Ident "x") (EVar (Ident "x"))
            EAbs
          /     \
         x      EVar
                 |
                 x
```

Here is the Linearized and 2-D AST for

$$\backslash x.xx$$

```
EAbs (Ident "x") (EApp (EVar (Ident "x")) (EVar (Ident "x")))
             EAbs
           /     \
        EApp     Evar
        /   \      |
    Evar    Evar   x
      |       |
      x       x
```

Here is the Linearized and 2-D AST for

$$(\backslash x.(\backslash y.xy))(\backslash x.x)z$$

```
EApp (EApp (EAbs (Ident "x") (EAbs (Ident "y") (EApp (EVar (Ident "x")) (EVar (Ident "y"))))) (EAbs (Ide


           EApp
        /        \
       EApp       Evar
    /      \        |
  EAbs      EAbs    z
  /  \      /  \
 x   EAbs   x   Evar
     /  \        |
    y   EApp     x
       /    \
      Evar  Evar
       |     |
       x     y
```

Here is the Linearized and 2-D AST for

$$(\backslash x.\backslash y.xyz)abc$$

```
EApp (EApp (EApp (EAbs (Ident "x") (EAbs (Ident "y") (EApp (EApp (EVar (Ident "x")) (EVar (Ident "y")))

                            EApp
                          /      \
                       EApp       EApp
                     /     \     /     \
                    EAbs   EApp Evar   EApp
                  /    \    |    |    /    \
                 x    EAbs  Evar c   Evar   Evar
                     /   \   |        |      |
                    y   EApp z        b      a
                        /  \
                      Evar  Evar
                       |     |
                       x     y
```

These expressions start easy with simple free variables such as x. The problems gradually got harder to include bound variables, and would need to be changed if we were evaluating these expressions.

2.5.2 Lambda Expression Evaluation

This section is to evaluate the following Lambda Expressions and solve:

```
1.  (\x.x) a
2.  \x.x a
3. (\x.\y.x) a b
4. (\x.\y.y) a b
5. (\x.\y.x) a b c
6. (\x.\y.y) a b c
7. (\x.\y.x) a (b c)
8. (\x.\y.y) a (b c)
9. (\x.\y.x) (a b) c
10. (\x.\y.y) (a b) c
11. (\x.\y.x) (a b c)
12. (\x.\y.y) (a b c)
```

Here is the evaluation for number 1:

```
(\x.x) a = a
```

Here is the evaluation for number 2:

```
\x.x a = a
```

Here is the evaluation for number 3:

```
(\x.\y.x) a b
(\y.a) b = a
```

Here is the evaluation for number 4:

```
(\x.\y.y) a b
(\y.y) b = b
```

Here is the evaluation for number 5:

```
(\x.\y.x) a b c
(\y.a) b c
(\y.a) c = a
```

Here is the evaluation for number 6:

```
(\x.\y.y) a b c
(\y.y) b c
(\y.b) c = b
```

Here is the evaluation for number 7:

```
(\x.\y.x) a (b c)
(\y.a) (b c) = a
```

Here is the evaluation for number 8:

```
(\x.\y.y) a (b c)
(\y.y) (b c) = b c
```

Here is the evaluation for number 9:

```
(\x.\y.x) (a b) c
(\y.a b) c = a b
```

Here is the evaluation for number 10:

```
(\x.\y.y) (a b) c
(\y.y) c = c
```

Here is the evaluation for number 11:

```
(\x.\y.x) (a b c)
(\y.a b c) = a b c
```

Here is the evaluation for number 12:

```
(\x.\y.y) (a b c)
(\y.y) = y
```

Using Interpreter.hs, I got

```
(\x.x)((\y.y)a)
(\x.x)(\y.a) = (\x.\y.a)
= a
```

3 Project

Introductory remarks ...

The following structure should be suitable for most practical projects.

3.1 Specification

3.2 Prototype

3.3 Documentation

3.4 Critical Appraisal

...

4 Conclusions

(approx 400 words)

In the conclusion, I want 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 programming languages and software engineering?

References

[PL] Programming Languages 2022, Chapman University, 2022.

Hackmd.io Professor Kurz, Chapman University 2022

Tower of Hanoi