CPSC-354 Report

Everett Prussak
Chapman University

September 13, 2022

Abstract

Short summary of purpose and content. Must Do soon.

Contents

1 Introduction

This report consists of Homework and the main project for CPSC 354. Section 1 consists of an introduction to LaTeX from Professor Kurz. Section 2 consists of homework for each week. Section 3 consists of the project. Section 4 consists of the sources, references, and websites cited.

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.

1.2.2 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.

```
-- 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`

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

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:

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

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 [PL] 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.

$$2^1 - 1 = 1$$

$$2^2 - 1 = 3$$

$$2^3 - 1 = 7$$

$$2^4 - 1 = 15$$

$$2^6 - 1 = 63$$

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