CPSC-354 Report

Eleas Vrahnos
Chapman University

September 11, 2022

Abstract

To be written at a later date.

Contents

1 Introduction **1**
1.1 General Remarks . . . 1
1.2 LaTeX Resources . . . 2
1.2.1 Subsubsections . . . 2
1.2.2 Itemize and enumerate . . . 2
1.2.3 Typesetting Code . . . 2
1.2.4 More Mathematics . . . 2
1.2.5 Definitons, Examples, Theorems, Etc . . . 3
1.3 Plagiarism . . . 3

2 Homework **3**
2.1 Week 1 . . . 3
2.2 Week 2 . . . 5

3 Project **8**

4 Conclusions **8**

1 Introduction

Replace this entire Section 1 with your own short introduction.

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

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`

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:

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

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 my solutions to the weekly homework assignments.

2.1 Week 1

The following is a Python implementation of the Euclidean algorithm:

```
def gcd(a,b):
    while a != b:
        if a > b:
            a = a-b
        else:
            b = b-a
    return a
```

We can test this code by going through the function with a sample input `gcd(9, 33)`, step by step.

1. `gcd(9, 33)`
   - The function is called, assigning 9 to variable `a` and 33 to variable `b`.
2. `while a != b:`
   - The while loop condition returns True, so the loop starts.
3. `else:`
   - `a > b` ($9 > 33$) returns False, so the else block executes.
4. `b = b-a`
   - `b` is now assigned to $33 - 9$, which is 24.
5. `while a != b:`
   - The while loop condition returns True, so the loop starts.
6. `else:`
   - `a > b` ($9 > 24$) returns False, so the else block executes.
7. `b = b-a`
   - `b` is now assigned to $24 - 9$, which is 15.
8. `while a != b:`
   - The while loop condition returns True, so the loop starts.
9. `else:`
   - `a > b` ($9 > 15$) returns False, so the else block executes.
10. `b = b-a`
    - `b` is now assigned to $15 - 9$, which is 6.
11. `while a != b:`
    - The while loop condition returns True, so the loop starts.
12. `if a > b:`
    - `a > b` ($9 > 6$) returns True, so the first block executes.
13. `a = a-b`
    - `a` is now assigned to $9 - 6$, which is 3.
14. `while a != b:`
    - The while loop condition returns True, so the loop starts.
15. `else:`
    - `a > b` ($3 > 6$) returns False, so the else block executes.
16. `b = b-a`
    - `b` is now assigned to $6 - 3$, which is 3.
17. `while a != b:`
    - The while loop condition returns False (`3 == 3`), so the loop ends.
18. `return a`
    - `a` is returned from the function, giving the correct greatest common divisor of **3**.

2.2 Week 2

The following are various implementations of functions in Haskell.

```
-- select_evens - lists the even-indexed elements of a given list
select_evens [] = [] -- in the case of a list with even number elements
select_evens (x:[]) = [] -- in the case of a list with odd number elements
select_evens (x:y:xs) = y : select_evens (xs)
```

We can test this code by going through the function with a sample input `select_evens ["a","b","c","d","e"]`, step by step.

1. `select_evens ["a","b","c","d","e"]`
   `select_evens ("a":"b":["c","d","e"]) = "b" :  select_evens (["c","d","e"])`
   - The input is pattern matched to the third case, starting the list with `"b"`.
2. `select_evens (["c","d","e"])`
   `"b" :  select_evens ("c":"d":["e"]) = "b" :  "d" :  select_evens (["e"])`
   - The next call to `select_evens` is pattern matched to the third case, adding to the list `"d"`.
3. `select_evens (["e"])`
   `"b" :  "d" :  select_evens ("e":[]) = "b" :  "d" :  []`
   - The next call to `select_evens` is pattern matched to the second case, adding to the list `[]`. A list with an even number of elements would end with using the first case instead.
4. `"b" :  "d" :  []`
   `["b","d"]`
   - A final list can now be formed with the colon operators.

```
-- select_odds - lists the odd-indexed elements of a given list
select_odds [] = [] -- in the case of a list with even number elements
select_odds (x:[]) = [x] -- in the case of a list with odd number elements
select_odds (x:y:xs) = x : select_odds (xs)
```

We can test this code by going through the function with a sample input `select_odds ["a","b","c","d","e"]`, step by step.

1. `select_odds ["a","b","c","d","e"]`
   `select_odds ("a":"b":["c","d","e"]) = "a" :  select_odds (["c","d","e"])`
   - The input is pattern matched to the third case, starting the list with `"a"`.
2. `select_odds (["c","d","e"])`
   `"a" :  select_odds ("c":"d":["e"]) = "a" :  "c" :  select_odds (["e"])`
   - The next call to `select_odds` is pattern matched to the third case, adding to the list `"c"`.
3. `select_odds (["e"])`
   `"a" :  "c" :  select_odds ("e":[]) = "a" :  "c" :  ["e"]`
   - The next call to `select_odds` is pattern matched to the second case, adding to the list `[]`. A list with an even number of elements would end with using the first case instead.
4. `"a" :  "c" :  ["e"]`
   `["a","c","e"]`
   - A final list can now be formed with the colon operators.

```
-- member - determines whether an element is part of a given list
member a [] = False
member a (x:xs)
    | a==x = True
    | otherwise = member a (xs)
```

We can test this code by going through the function with a sample input `member 2 [5,2,6]`, step by step.

1. `member 2 [5,2,6]`
   `member 2 (5:[2,6]) = member 2 ([2,6])`
   - The input is pattern matched to the second case, satisfying the `otherwise` conditional.
2. `member 2 [2,6]`
   `member 2 (2:[6]) = True`
   - The input is pattern matched to the second case, satisfying the `a==x` conditional. 2 is determined as a member of the given list.

```
-- append - appends a list to another list
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

We can test this code by going through the function with a sample input `append [1,2] [3,4,5]`, step by step.

1. `append [1,2] [3,4,5]`
   `append (1:[2]) [3,4,5] = 1 :  append [2] [3,4,5]`
   - The input is pattern matched to the second case.
2. `append [2] [3,4,5]`
   `1 :  append (2:[]) [3,4,5] = 1 :  2 :  append [] [3,4,5]`
   - The input is pattern matched to the second case.
3. `append [] [3,4,5]`
   `1 :  2 :  append [] [3,4,5] = 1 :  2 :  [3,4,5]`
   - The input is pattern matched to the first case.
4. `1 :  2 :  [3,4,5]`
   `[1,2,3,4,5]`
   - A final list can now be formed with the colon operators.

```
-- revert - reverses a list
revert [] = []
revert (x:xs) = revert (xs) ++ [x]
```

We can test this code by going through the function with a sample input `revert [1,2,3]`, step by step.

1. `revert [1,2,3]`
   `revert (1:[2,3]) = revert ([2,3]) ++ [1]`
   - The input is pattern matched to the second case.
2. `revert [2,3]`
   `revert (2:[3]) ++ [1] = revert ([3]) ++ [2] ++ [1]`
   - The input is pattern matched to the second case.
3. `revert [3]`
   `revert (3:[]) ++ [2] ++ [1] = revert ([]) ++ [3] ++ [2] ++ [1]`
   - The input is pattern matched to the second case.
4. `revert []`
   `revert [] ++ [3] ++ [2] ++ [1] = [] ++ [3] ++ [2] ++ [1]`
   - The input is pattern matched to the first case.
5. `[] ++ [3] ++ [2] ++ [1]`
   `[3,2,1]`
   - The lists can now be concatenated.

```
-- less_equal - checks if the element in a list is less than or equal to the same-indexed
    element in another list
less_equal [] [] = True
less_equal (x:xs) (y:ys)
    | x > y = False
    | otherwise = less_equal (xs) (ys)
```

We can test this code by going through the function with a sample input `less_equal [1,2,3] [2,3,2]`, step by step.

1. `less_equal [1,2,3] [2,3,2]`
   `less_equal (1:[2,3]) (2:[3,2]) = less_equal ([2,3]) ([3,2])`
   - The input is pattern matched to the second case, satisfying the `otherwise` conditional.
2. `less_equal [2,3] [3,2]`
   `less_equal (2:[3]) (3:[2]) = less_equal ([3]) ([2])`
   - The input is pattern matched to the second case, satisfying the `otherwise` conditional.
3. `less_equal [3] [2]`
   `less_equal (3:[]) (2:[]) = False`
   - The input is pattern matched to the second case, satisfying the `x > y` conditional since $3 > 2$.

3 Project

To be written at a later date.

4 Conclusions

To be written at a later date.