CPSC-354 Report

Your Name
Chapman University

October 17, 2021

Abstract

Short introduction to your report . . .

Contents

1 Introduction

Replace Section 1 with your own short introduction.

1.1 General Remarks

First you need to download and install LaTeX.[1] Alternatively, you can use an online editor such as Overleaf. I prefer to have my own installation, but to get started Overleaf may be easier.

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.

- 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

1.3 Plagiarism

To avoid plagiarism, make sure that in addition to [PL] you also cite all the external sources you use.

2 Haskell

This section will contain your own introduction to Haskell.

To typeset Haskell there are several possibilities. 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
```

This works well for short snippets 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).

2.1 A Brief Introduction

The vast majority of readers to this point are familiar, if not proficient, with the concept of object oriented programming. It is characterized by the representation of things as "objects", offering users the option to hide certain processes, pieces of code, from outsiders as well as to store those processes in their respective sections. Another key aspect of object oriented programming is the manipulation of data through variables.

Functional programming, on the other hand, is based on the evaluation of functions. The manipulation of variables, as seen in object oriented programming, is not possible—functional programming deals with immutable data. Loops are also not supported here. Instead, iterative data is managed by recursion.

This seemingly radical shift in paradigms sound daunting at first. And, to an extent, it is. However, all good things are difficult at the beginning—no one is born knowing how to program, yet you, reader, have made it this far. As such, rather than as an academic paper, I will treat this as a conversation.

2.2 Two Languages, One Calculator

We will first be analyzing the differences between object oriented and functional programming by building some functions for a simple calculator with a language from each paradigm—Python and Haskell, respectively. This will allow us to understand how functions, variables, and types behave for each language.

Let's begin with addition. For any two numbers, a, b, $a + b = c$, where c, is also a number. How about we code it in Python. We define a function and its parameters, in this case a and b, and implement the operation inside the body. It looks like this:

```
def addition(a, b):
    c = a + b
    return c
```

As insultingly trivial as this may seem at first, let's look a bit more in depth at this function. It receives two variables, creates a new one, and returns it. More so, it doesn't specify the required types for addition. It doesn't matter if the input is an integer, double, floating point, or even strings for concatenation, our humble function just does it.

Now, let's do it with Haskell. Haskell does not support type inference, so we must specify exactly what we wish to compute and with what type of input. Let's begin with natural numbers. The first step is to explain to our computer what is a natural number. It is comprised of all positive numbers as well as zero—many mathematicians dispute 0's membership in this number club with much fervour, however we have more important things to do, such as defining addition. Back to it.

Knowledge of discrete mathematics is important here, mainly because we must formally define our number system. Let us define a natural number as any number *NN*. This number is either 0, or a successor of it.

We define it within Haskell as seen below:

```
data NN = 0 | S 0
```

Now that we have defined the data type, let's move on to writing the function. Recall that Haskell is based on the evaluation of functions rather than variables inside functions. Our function adds two natural numbers: in other words, it takes in *NN*, adds it to another *NN*, and finally returns some *NN*, where *NN* could be any natural number. Elegantly put, $NN \to NN \to NN$.

The issue is that we have defined our natural numbers to be either 0 or its successors, therefore our addition will be dependent on this definition. As mentioned before, Haskell is based on recursion, so we will have to incorporate this as well. From Discrete Mathematics, we know that successors are commutative. That is, adding $1 + 1$ is equivalent to adding $S(0 + 1)$. Furthermore, we know that the addition of zero and any number x is equal to x. Consider this our base case for our oncoming recursive implementation. Basing off from before, adding S(a), b, is equivalent to the sum of S(a + b).

Now that we have defined our data type, how our function treats the data type, as well as how it behaves, we can write it with Haskell. See below.

```
addN :: NN -> NN -> NN
addN O m = m
addN (S n) m = S (addN n m)
```

There we go, our first recursive function in Haskell. Now, let's up the ante a bit. Let's look at lists, loops, and the effects of implementation for each language.

2.3 More Differences

We have gotten quite far, give yourself credit—high five. Now, let's write the Fibonacci code. In Python, we could write it like this. [GFG]

```
def fibonacci(n):
    a = 0
    b = 1

    # Check is n is less
    # than 0
    if n < 0:
        print("Incorrect input")

    # Check is n is equal
    # to 0
    elif n == 0:
        return 0

    # Check if n is equal to 1
    elif n == 1:
        return b
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b
```

Something interesting is happening here. We are creating a global variable *result*, giving it an initial value of zero, and modifying it through each iteration. More interestingly, we have the option to reference this variable and modify it at will through different levels of access, as we see that our function is doing—by accessing the global variable and mutating it.

Before we compare this to Haskell, we must quickly look at pattern matching. Recall piece-wise functions from math. These functions behave differently based on the value of the given input. See the example below.

$$fib(x) = \begin{cases} 1, \text{ if } x = 0 \\ 1, \text{ if } x = 1 \\ fib(x-1) + fib(x-2), \text{ if } x >= 2 \end{cases}$$

Haskell's pattern matching works in a similar way. We specify different cases for a function so that Haskell knows what to do in each case. When we call the function, Haskell compares the given arguments to the argument patterns we have written. As mentioned in *A Gentle Introduction to Haskell*, "Pattern matching provides a way to 'dispatch control' based on structural properties of a value."[GH] Now that we have a better understanding of pattern matching, let's write the Fibonacci code in Haskell.

3 Programming Languages Theory

In this section you will show what you learned about the theory of programming languages.

4 Project

In this section you will describe a short project. It can either be in Haskell or of a theoretical nature,

5 Conclusions

Short conclusion.

References

[PL] Programming Languages 2021, Chapman University, 2021.

[GH] Case Expressions and Pattern Matching, *A Gentle Introduction to Haskell*.

[GFG] Python Program for Fibonacci Numbers, Geeks for Geeks.