CPSC-354, Programming Languages Report

Arshia Sharma
Chapman University

November 8, 2021

Abstract

This report will discuss three key topics in Programming Languages: What is Haskell, its benefits and uses, the theory involved with Programming Languages in regards to Discrete Mathematics and other methodologies, and a mini-project. ...

Contents

1 Introduction

Replace Section 1 with your own short introduction.

2 Haskell

As Maria von Trapp once sang, let us start at the very beginning, a very good place to start. Before starting this course, I had several questions as to what programming languages even was. I had heard many things from my friends who have taken this course but had no idea what they meant by Lambda Calculus, functional programming, or parsing. Some questions beginners as myself, starting their journey into programming language, specifically relating to Haskell, may have included:

1. What is Haskell?
2. What is the difference between Haskell and other programming languages we have covered in other classes so far?
3. Where to start with Haskell?
4. What are type classes and monads?
5. What more can I do in Haskell that I cannot do as easily with other languages?

In this section of the report, we will be answering these questions, as well as key concepts and reminders of programming techniques that are commonly used in the Haskell language.

2.1 Let us Start at the Beginning

2.1.1 What is Haskell?

So what exactly is Haskell and why should programmers learn how to use it? Haskell is not the typical or "common" imperative programming language you may have used before such as Python, Java, or C++. Instead, Haskell is a purely functional programming language developed in the late 1980s by scholars to better communicate their theories and ideas [UPenn]. What is a functional program you may ask? Well, if you are familiar with Excel or SQL, the main idea is the same! Functional programming focuses on a single expression where our main focus is on what we are solving rather than on how to solve the issue at hand [Haskell.org]. We will dive into more examples between the difference of functional programming languages such as Haskell and imperative programming languages such as Python further in this report after we discuss one other main key aspect of Haskell: what is Haskell used for and why is it important for us as programmers?

2.1.2 Why is Haskell Important?

Haskell is used in various projects and applications at companies such as Facebook, Target, and NASA to name a few [serokell.io]. One of the most popular and useful applications of Haskell is Sigma, one of Facebook's software program which catches malware and spam on Facebook's platform and removes it to protect users from attacks. [Facebook Engineering]. Alongside protecting users on social media, Haskell also opens doors to a new framework of thinking and problem solving for programmers.

Haskell re-enforces the idea of the importance of understanding data types, how important discrete mathematics is to computer science to demonstrate how simple mathematical formulas, such as addition and multiplication, are programmed, and brings a philosophic mindset to a field that typically does not go into difficult questions such as "What is language but a string of characters". This new abstract way of thinking and an unfamiliar language may be daunting at first, but with the tools already in a programmers kit, such as familiarity with functions, data types, and recursion, let us dive into some examples of Haskell to apply what we know and what more we can add with Haskell.

2.2 Haskell Review of Concepts and Tutorial

To have the best grasp of Haskell, let us review three key concepts from our computer science journey so far and how they compare in Haskell:

1. Functions and Loops in Python and C++
2. Recursion, recursion, recursion
3. Discrete Mathematics and its importance in Haskell

2.2.1 Functions and Loops

Functions are a fairly common tool when it comes to programming in Python and other imperative languages. They are extremely helpful to the run-time, efficiency, and the organization or flow of code. With the use of parameters and best coding practices, functions at its core, reduces repetitions of code. Loops such as for, do-while, and while loops also allows programmers to iterate through arrays, lists, and sequences of numbers to perform calculations or return information to the user. Let's look at an example of function and loops in Python and C++ that returns the harmonic number of an integer.

Functions in Python

```python
#Python function to calculate Harmonic Numbers
def harmonic(x) :
    if(x < 1):
        return 0
    harmonic = 1
    for i in range(2, x+1) :
        harmonic += 1 / i
    return harmonic

#main
harmonic(5)
harmonic(8)
```

Functions in C++

```cpp
#include <iostream>
using namespace std;

// C++ function to calculate Harmonic Numbers
double harmonic(int x)
{
    if(x < 1){
        return 0;
    }
    double harmonic = 1.00;
    for (int i = 2; i <= x; i++) {
        harmonic += (double)1 / i;
    }

    return harmonic;
}
// main
int main()
```

```
{
    cout<<harmonic(5);
    cout<<harmonic(8);
}
```

These two functions both calculate the Harmonic Number of a given value, in our cases 5 and 8, by using a for loop to iterate through the given range of numbers. The formula of to find Harmonic Numbers is relatively simple [Harmonic Numbers]:

$$\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \cdots + \frac{1}{n}$$

If you are already familiar with loops and functions, this is also fairly simple, the functions exactly follows the rules of the formula: it checks if the value in main is greater than 1, if it is less, it returns zero, else, it continuously adds each fraction together until it reaches the inputted value in a for loop. Excluding empty lines and curly brackets, the function is only 7 lines of code! Surprisingly, in Haskell, it can be even shorter. Here we have our first lesson when comparing Haskell to imperative languages - there are no loops in Haskell. Functions themselves are treated as arguments/variables within the code and are used in place of loops. Let's look at how Harmonic Numbers are determined in Haskell to see this concept firsthand:

Functions in Haskell

```
harmonic :: Fractional a => a -> a
harmonic 1 = 1
harmonic i = 1/i + harmonic (i-1)
```

Now in Haskell, the same function can be repeated used using our good friend recursion, which we review in 2.2.2, in three relatively short lines of code. To familiarize yourself a bit with type casting, we will discuss that topic further in 2.3.2 to better understand the first line of code [Haskell.org Fractional]. As we can see, functions in Haskell are used in place of loops programmers use with imperative languages! In that case, let us dive back into a review of what recursion is and another common application of it.

2.2.2 Recursion, Recursion, Recursion

Almost every computer science or engineering student learned about recursion the same way, with the popular Fibonacci Sequence: 1, 1, 2, 3, 5, 8, 13, 21, etc. where we repeatedly add each number together in a pattern. 1+1 = 2, 1+2 = 3, 2+3=5, and so on. The mathematical formula is as follows:

$$fib(0) = 0 \tag{1}$$
$$fib(1) = 1 \tag{2}$$
$$fib(n+2) = fib(n) + fib(n+1) \tag{3}$$

As noted in the third line of the formula above, $fib(n+2)$, fib is called within itself as long as the conditions are true. For example, if $n = 0$, the formula would be: $fib(2) = fib(0) + fib(1) = 1$, where we are calling the fib function within itself. This formula can also be easily replicated in Python as well.

```
def fib(i):
   if i <= 1:
       return i
   else:
       return(fib(i-1) + fib(i-2))

num = int(input("Enter the number of outputs for the Fib sequence: "))
```

```
print("Fibonacci sequence:")

for i in range(num):
       print(fib(i))
```

As we know from our previous section 2.2.1, recursion is heavily used in Haskell in place of loops. The Python code from above replicates that mentality, which is extremely similar to the syntax used to define the Fibonacci Sequence in Haskell:

```
-- Fibonacci Sequence in Haskell
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Haskell relies on recursion throughout its functionality and is the first step into the world of Haskell. Becoming an official Haskeller starts with reminding yourself of how functions and loops are used as one in Haskell with the use of recursively calling the function in place of a loop. Once you master this train of thought, the language becomes more readable, less daunting, and you gain confidence in tackling the first lesson of Haskell.

2.2.3 Discrete Mathematics

Now shifting gears to a more abstract concept in programming, discrete mathematics is also the first steps into understand the fundamentals of the magic behind computer science and Haskell. Mathematics in the computer science field is not only applying functions, integrals, linear algebra or multi-variable calculus. Abstraction, understanding patterns, pattern-matching, and appreciating the behind-the-scene layers of the theoretical (possibly even philosophical) layout of mathematics is also vital in formulating solutions to solve engineering problems. As Haskell is commonly referred to as 'lazy' programming, as discussed further in 2.3.1, it heavily relies on the programmer having a fair grasp of key concepts of discrete mathematics such as:

1. Sets, Logic, and Mathematical Notation
2. Different Types of Numbers

Sets, Logic, and Mathematical Notation: Sets are one of the first step into discrete mathematics and is important to understand to gain a solid grasp of Haskell. Sets are simply an unordered collections of elements, such as a list in Python. The symbols that defines a set are curly brackets: {}. Empty curly brackets as displayed in the previous sentence represents an empty set, which means exactly that: there is nothing in an empty set. Having familiarity in mathematical notation would also aid your adventure into Haskell and discrete math so what does a set with elements look like you may ask?

$$n \in \mathbb{N},\ 4 \in \{4\},\ \ 4 \in \{1, 2, 3, 4, 5\}$$

The three examples above displays how an element is assigned to a set and if the element belongs to a set. $\in$ represents belonging. n belongs in the set of N, 4 belongs in the set of {4} and 4 belongs in the set of {1, 2, 3, 4, 5} as well. You can define sets, such as a set of all even or odd natural numbers, using set-comprehension notation.

$$Even \stackrel{\text{def}}{=} \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} \,.\, 2 \cdot m = n\}.$$

$$Odd \stackrel{\text{def}}{=} \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} \,.\, (2 \cdot m) + 1 = n\}.$$

The $\exists$ symbol represents existence, where if there exists an element m in the set of natural numbers $\mathbb{N}$, if true, 2 multiplied by $m = n$ and n is an even number or 2 multiplied by m plus $1 = n$ and n is an odd number.

Sets also incorporate basic mathematical operations such as and, or, and not, which are also referred to as intersection, union, and negation, respectfully. A table of these set operations, Boolean logic, and mathematical notation is displayed below for a refresh of these core theories.

Sets	Boolean
$\cap$ (intersection)	$\wedge$ (and)
$\cup$ (union)	$\vee$ (or)
$\oplus$ (exclusive or)	$\veebar$ (xor)
$\subseteq$ (belongs to)	$\implies$ (implication)
$\not\subset$ (not belongs to)	$\neg$ (negation)

$$Intersection \stackrel{\text{def}}{=} L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\} = \{w \in \Sigma^* \mid L_1(w) \wedge L_2(w) = 1\}.$$

$$Union \stackrel{\text{def}}{=} L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\} = \{w \in \Sigma^* \mid L_1(w) \vee L_2(w) = 1\}.$$

$$ExclusiveOr \stackrel{\text{def}}{=} L_1 \oplus L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ xor } w \in L_2\} = \{w \in \Sigma^* \mid L_1(w) \veebar L_2(w) = 1\}.$$

$$BelongsTo \stackrel{\text{def}}{=} L_1 \subseteq L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ implies } w \in L_2\} = \{w \in \Sigma^* \mid L_1(w) \implies L_2(w) = 1\}.$$

$$NotBelongsTo \stackrel{\text{def}}{=} L_1 \not\subset L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ negates } w \in L_2\} = \{w \in \Sigma^* \mid \neg L_1(w) \wedge L_2(w) = 1\}.$$

Types of Numbers As referenced in the previous section, you may have noticed in our definition of even and odd numbers, we referenced the set of all natural numbers. This means that our odd and even number all belong, or are elements, in the set of natural numbers. So what are the natural numbers? Are there more sets of numbers? There are four set of numbers you may come across in Haskell: the set of all natural numbers ($\mathbb{N}$), positive natural numbers ($\mathbb{P}$), integers ($\mathbb{Z}$), and rational numbers ($\mathbb{Q}$).

The set of all natural numbers contains all whole values starting from 0: $\mathbb{N}$ = {0, 1, 2, 3, 4, 5, ...}. The set of all positive natural numbers contains all whole values starting from 1: $\mathbb{P}$ = {1, 2, 3, 4, 5, ...}. The set of integers contains all whole values, negative or positive: $\mathbb{Z}$ = {..., -2, -1, 0, 1, 2,...}. The set of all rational numbers are pairs of an integer and positive number ($\mathbb{Z}$, $\mathbb{P}$) where $\mathbb{Q}$ is equal to all pairs of an integer and positive number that can be divisible to produce a whole integer with no decimal points.

There are other sets of numbers such as irrational ($\mathbb{I}$), real ($\mathbb{R}$), and complex ($\mathbb{C}$) numbers as well. Although these sets have not been used in our learning of Haskell so far, it is still important to note that are sets that include all sorts of numbers. These sets also compare directly to different data types in programming languages such as C++, where the set of integers are signed integers, the set of of natural numbers are unsigned integers.

Connection between Haskell and Discrete Math The topics of sets, logic, mathematical notation, and types of numbers are important to consider while tackling Haskell and higher level computer science concepts going forward in your career. Haskell directly uses concepts such as implementing natural, positive, integers, and rational numbers in Haskell to prove arithmetic operations such as addition, subtraction, multiplication, and division as we do in the first assignment of this class, creating a simple Calculator. In fact, discrete mathematics first introduces the idea of 0 (O) and +1 (S), which is the backbone of our Calculator, which is further discussed in 2.3.2. Most of the mathematical concepts discussed in discrete mathematics is used as the background structure of programming languages where an abstract thought process is used to develop programming languages as a whole. Understanding sets and logical operations that can occur will aid your understanding in how to utilize sets of numbers. Learning mathematical notation will come in handy to grasp formulas you may come across in research, expand your skill set, and help to communicate your ideas to other computer scientists as well.

2.3 Key Concepts of Haskell

Haskell is comprised of many unique phrases and functionalities. It is normally referred to as a lazy programming language, which may confuse us programmers at first. How can a programming language at its base be lazy and yet be so useful? Why lazy? Haskell also has type classes and monads that play an important part in its language. Haskell is concise and prefers to be short and too the point. But why are these features and attributes important to its nature? Let us find out!

2.3.1 Haskell as a Lazy Programming Language

When you are in a deep discussion with a computer scientist, prepping for an interview, or diving deep into Haskell reddit, you will come across the phrase that Haskell is lazy. When I first heard of this, my immediate thought was that the language would require more work to compile and run, which would result in more code to write. To my surprise, this is not entirely the case. Haskell is very specific in the sense you have to specify exactly what you want the program to do. This may seem obvious since your computer cannot just do your code for you, but Haskell requires more guidance when trying to solve a problem.

First let us look at an example of the flow of programming as we are used too with this simple Python program that gives us the square root of all positive numbers:

```
#Python function to find the square root of x
def sqrt(x, val) :
    if (x < 1):
        return -1
    elif (x >= 1 and val > 0):
        return x ** 0.5
    return -1

#main
sqrt(4, 1)
sqrt(5, 3)
sqrt(5, -4)
```

Since types in Python are defaulted for us, the default return statement for our code above would result in a float. $sqrt(4, 1) = 2.0$ and $sqrt(5, 3)$ would be about 2.24. However, for $sqrt(5, -4)$, the function returns -1 since the inputs were not True for either expressions. You may be wondering why another Boolean expression was included in this script. Well, if we convert this this function in Python pseudo-code, it would look similar to:

```
#pseudo-code to find the square root of x
def sqrt(x, val) :
    if (x < 1):
        # return False
    if (x and val):
        # return expression as sqrt(x)
    else
        # return False
```

Imagine Haskell as a stubborn child who won't do anything unless they are explicitly told to. If they were told to eat all their carrots and broccoli, but they are no carrots on their plate, they simply would not listen since you told them to do something that does not exist, or is False. Haskell operates in a similar way, if a is not True, then the program will immediately stop since we already know one condition is False, there is no point to continue since we already know the statement is False no matter the Boolean condition of b.

This does, in hindsight, save computational power and resources, but with the program immediately ending, Haskell also does not tell you what went wrong. If you are working with a larger chuck of code, it may be troubling to find which portion of the code ran and where the issue arose. The trade off of saving resources and avoiding False conditionals that gives Haskell its lazy trait, means that programmers would have to pay close attention to what they are programming. Although the code is generally less than imperative languages, Haskell was built assuming that the coder will not need detailed error messages pointing them to the correct portion of code [LYH]. Do not assume that Haskell will respond to you to help you understand what it is saying, Haskell has to understand what you are saying. Just as you would do with a little one, you have to understand and work with them first before they work with you.

2.3.2 Haskell and Type Classes

As we have learned, Haskell is a purely functional programming language, and functional programming languages tend to have a static type system [LYH Types]. Being statically typed means that the Haskell compiler is aware of which type, such as strings or numbers, a variable is and an outcome is supposed to be and what operations are valid given two inputs. As mentioned in section 2.3.1, Haskell is a lazy programming language, so if there is an invalid operation, such as addition between a string and an integer, the compiler will catch that and the program will throw an error rather than crashing during execution [LYH Types].

Haskell's compiler can also deduce which type an object is, such as a string or Boolean value. Functions also have their own type in Haskell. As we discussed in section 2.2.1, functions and recursion are a vital tool in Haskell to create loops and an iterative process in Haskell. As Harmonic Number function, the function defined to be of a Fractional type that the Haskell compiler understood and recognized.

```
-- Haskell function that subtracts natural numbers
subtrN :: NN -> NN -> NN
subtrN m O = m -- flip of addN
subtrN (S m) (S n) = (subtrN m n)
```

Similarly in the example above, we are setting the subtrN function, which subtracts natural numbers ($\mathbb{N}$), to receive two natural numbers as an input and outputs one natural number as well declared its type on the first line of code. The $->$ symbol represents the order of the inputs and outputs of the type declaration: the first natural number represents the first input and the last represents the final output. Depending on the goal of the function and the number of outputs it may have, Haskell only cares for the order that the inputs and outputs are placed, using the arrows as a left to right process from step A to the final output.

You have the flexibility with Haskell to create your own types or utilize the built-in types Haskell has. As with the example above, type $\mathbb{N}$, or natural numbers, is defined as

```
-- Natural numbers
```

```
data NN = O | S NN
```

where O represents the number zero (0) and S represents +1 of type $\mathbb{N}$. The | bar represents flexibility in that there are two ways to represent a natural number. Haskell also has built-in type classes, such as Eq, Ord, Num, and Enum [LYH Types]. Eq (==, !=) is used for equality testing and results in a True/False Boolean value, depending on the outcome of the expression. Ord is used for ordering and utilizes >, <, >=, <= to compare the order of objects. Num can be used where the element given can act as any number if applicable, such as an Integer or Double. An interesting type class is Enum, where elements in a sequential order can be automatically enumerated by Haskell, given a certain range. For example, given the lists of elements:

```
ghci> ['b' ..'f']
ghci> succ 'S'
ghci> [4 .. 9]
```

The Haskell compiler would return 'bcdef', 'T', and [4, 5, 6, 7, 8, 9] as the outputs of the three prompts given. This is particularly interesting since the compiler can identify order of the alphabet and numbers.

2.3.3 Haskell and Monads

Monads is another important and interesting functionality that Haskell implements in its language. You have most likely already used monads before in other languages, as monads can be lists, Maybe type, and Inputs/Outputs [Haskell.org Monads]. When you google the definition of monad, you may come across two definitions, a programming and a philosophical one. As mentioned in our introduction to Haskell, programming languages expands a programmers mindset to focus on both the theoretical and philosophical side of computer science, so it is not surprising that the term is also used in programming and theory.

Monads lies under the metaphysics sect of philosophy where German philosopher Gottfried Wilhelm Leibniz coined the term as "non-composite, immaterial, soul-like entities" [IEOP]. The term can be further reduced to mean of which is one, has no parts and is therefore indivisible [IEOP]. In a computer science sense, the idea of being indivisible remains. Monads are constructed on top of a polymorphic type in Haskell and monadic classes are non-derivable [Haskell Monads]. The type class has three rules, called the monadic rules: [TP Monads]:

1. Left Identity Law
2. Right Identity Law
3. Associativity

The Left and Right Identity Laws has the same goal where the return statement would not change the value or anything in the Monad [TP Monads]. The goal of associativity is to ensure that Monads and Functors operate similarly [TP Monads]. Functors are another form of polymorphism implemented in Haskell where Lists, Maps, and Trees are all instances of Functors. Monads are a type of Applicative Functor, which is a Functors with additional features [TP Functors]. The third rule is the ensure that Monads in Haskell follow the rules associated with Functors. Although this is may be daunting and confusing at first, think of monads as the bridge between programming and philosophical thinking. Although the two concepts seem completely unrelated, a philosophical mindset was and is still critical in the creation of functions and types that allows for efficient coding standards.

2.4 Haskell and Lambda-Calculus

You may be looking at this header and think "Oh great, more derivatives and integrals". Well, never fear, we will not be discussing calculus the way you may be familiar with in your previous math courses. You

may have questions as to why Haskell's logo is a lambda sign: λ and not an H, similar to C++ logo or Java's, which is a coffee cup, a pun to Java coffee. Well, Haskell's logo is a lambda since Haskell heavily uses concepts in Lambda-Calculus, giving credit to an important aspect of the language [Haskell Logo].

So what exactly is Lambda-Calculus you may ask? At its core, Lambda Calculus focuses on simple and concise notations for functions [Lambda Calculus]. As covered so far in this report, Lambda Calculus is also used in many other different fields other than computer science, such as philosophy and linguistics [Lambda Calculus]. Let's look at some examples of Lambda Calculus to see what this interpretation would look like.

Using the formula $x^3 \cdot x - 4$, this is translated to Lambda Calculus syntax as: $\lambda x[x^3 \cdot x - 4]$ or $(\lambda x.x^3 \cdot x - 4)$ where λ represents abstraction over the variable x [Lambda Calculus]. Say in our example, we would want x = 3, that would translate to $3^3 \cdot 3 - 4$. In Lambda Calculus syntax, this would translate to $(\lambda x[x^3 \cdot x - 4])3$ where then the typical order of operations, PEMDAS, would kick in and evaluate the expression.

$$= (\lambda x[x^3 \cdot x - 4])3 \qquad \langle \text{Lambda Calculus} \rangle \tag{4}$$
$$= 3^3 \cdot 3 - 4 \qquad \langle \text{Substitute 3 for } x \rangle \tag{5}$$
$$= 27 \cdot 3 - 4 \qquad \langle \text{Exponent} \rangle \tag{6}$$
$$= 81 - 4 \qquad \langle \text{Multiplication} \rangle \tag{7}$$
$$= 77 \qquad \langle \text{Subtraction} \rangle \tag{8}$$

As you can see, this is simply another way to interpret inputs and outputs of functions and formulas. Now let us look at an example with more than one variable:

$$= (((\lambda x.\lambda y.\lambda z[x^z \cdot x - y])3)4)5 \qquad \langle \text{Lambda Calculus} \rangle \tag{9}$$
$$= ((3^z \cdot 3 - y)4)5 \qquad \langle \text{Substitute 3 for } x \rangle \tag{10}$$
$$= (3^z \cdot 3 - 4)5 \qquad \langle \text{Substitute 4 for } y \rangle \tag{11}$$
$$= 3^5 \cdot 3 - 4 \qquad \langle \text{Substitute 5 for } z \rangle \tag{12}$$
$$= 243 \cdot 3 - 4 \qquad \langle \text{Exponent} \rangle \tag{13}$$
$$= 729 - 4 \qquad \langle \text{Multiplication} \rangle \tag{14}$$
$$= 725 \qquad \langle \text{Subtraction} \rangle \tag{15}$$

The parameters are passed to its corresponding variable in the order that they are called in the beginning of the lambda calculus formula. If we changed the formula to be $(((\lambda x.\lambda z.\lambda y[x^z \cdot x - y])3)4)5)$, then z = 4 and y = 5. This formula can also be displayed in a simple Python script as well:

```
def formula(x, y, z) :
    return x ** z * x - y

#main
formula(3, 4, 5)
formula(3, 5, 4)
```

Here we can directly compare how the syntax between the two differ and how concise (possibly even lazy) Lambda Calculus is. Lambda Calculus uses dot notation to denote its input at the start and separates its inputs with the dot. It also takes in the inputs at the end, while the formula itself is in the middle.

2.4.1 Church Numerals and Booleans

In addition to having a concise syntax, there are also church numerals, named after the creator of Lambda Calculus, Alonzo Church [Lambda Calculus]. To represent numbers, Church denotes numbers as [CN]:

$$0 = \lambda f.\lambda\ x.x$$
$$1 = \lambda f.\lambda\ x.(f\ x)$$
$$2 = \lambda f.\lambda\ x.(f\ (f\ x))$$
$$3 = \lambda f.\lambda\ x.(f\ (f\ (f\ x)))$$
$$4 = \lambda f.\lambda\ x.(f\ (f\ (f\ (f\ x))))$$

You may notice a pattern fairly quickly in Church's pattern of +1 from zero. As in discrete math, successors (+1) is an important concept which is an introduction into the abstraction side of mathematics. Church Numerals displays the successor concept as well in Lambda Calculus to re-enforce its importance. The formula to show succession is also a pattern which can be simply displayed as:

$$n = \lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))$$

There are also Church Booleans, where Church denoted True and False values using the variables x and y:

$$\text{TRUE} = \lambda x.\lambda y.x$$
$$\text{FALSE} = \lambda x.\lambda y.y$$

Here, the formula takes in two inputs, x and y, where x (or the first inputted value) is True and y (the second inputted value) is False [CN]. As mentioned previously, Lambda Calculus is an important aspect in becoming an official Haskeller, the logo displays that clearly. Practicing its concise syntax and comparing it to syntax you are already familiar with, such as handwritten formulas or code, will boost your comfort with Haskell as well.

2.5 Introduction into the Haskell Project

For my Haskell Project, I would like to explore an integration between Haskell and Data Science. Currently, I am also taking a course in Machine Learning, where we use a package called TensorFlow to run neural networks and other deep learning models [TF]. While researching ideas of finding a way to use Haskell alongside my data science interest, I came across an article that mentions how to use Haskell and TensorFlow [HTF]. For my project, I'd like to explore this idea more and see if I could use Haskell to run machine learning models, such as Recurrent Neural Networks and Support Vector Machines. I'm interested in using a real dataset if possible, such as the popular IMDb dataset, but plan first on beginning with generated numbers and see if the connection between Haskell and TensorFlow could work.

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.

[Haskell.org] What is functional programming?, Haskell.org, 2021.

[UPenn] CIS 194: Introduction to Haskell, University of Pennsylvania, 2016.

[serokell.io] Software Written in Haskell: Stories of Success, serokell.io, 2019.

[Facebook Engineering] Fighting spam with Haskell, Facebook Engineering, 2015.

[Harmonic Numbers] Harmonic Number, Wolfram, 2021.

[Haskell.org Fractional] Numeric Coercions and Overloaded Literals, Haskell.org, 2021.

[LYH] Learn You Haskell, Introduction, learnyouahaskell.com, 2016.

[LYH Types] Learn You Haskell, Types and Type Classes, learnyouahaskell.com, 2016.

[Haskell.org Monads] All About Monads, wiki.haskell.org, 2021.

[IEOP] Gottfried Leibniz: Metaphysics, iep.utm.edu, 2021.

[Haskell Monads] About Monads, www.haskell.org, 2021.

[TP Monads] Haskell - Monads, www.tutorialspoint.com/, 2021.

[TP Functors] Haskell - Functors, www.tutorialspoint.com/, 2021.

[Haskell Logo] Haskell - Logo, wiki.haskell.org, 2021.

[Lambda Calculus] The Lambda Calculus, plato.stanford.edu, 2018.

[CN] Church Numerals and Booleans, opendsa-server.cs.vt.edu, 2018.

[TF] TensorFlow, www.tensorflow.org, 2021.

[HTF] Starting out with Haskell Tensor Flow, towardsdatascience.com, 2017.