CPSC-354, Programming Languages Report

Arshia Sharma
Chapman University

December 21, 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. The paper starts off with an introduction to what topics that will be discusses, such as Haskell, its importance, discrete mathematics, recursion, Monads, type classes, Lambda-Calculus, Turing Complete, the Halting Problem, Abstart Recution System, and String Rewriting. The project will cover how to create a calculator using a Haskell GUI.

Contents

1 Introduction

What is the first thought you come up with when you hear the phrase programming languages? Maybe it is your favorite programming language, such as Python, Java, or C++. Or could it be straight forward thoughts, such as computers, AI, Machine Learning, school, your future, that one programming assignment from freshman year you put off till the last second that you still get nightmares about. Whatever it may be, have you ever thought about how you can make your own programming language? Or how languages such as Python, Linux, C++, or R Programming even came into existence?

Programming languages play an extremely vital role in our society, especially today. Various applications, such as Zoom, TikTok, Google, Angry Birds, and the calculator on your phone are all possible because of the programming language that was used to code those applications you use everyday. This report will cover important topics that relate to programming languages, such as the difference between functional and imperative languages, Haskell and its importance, Lambda Calculus, programming language core theories, such as Turing Complete, and a mini-project that goes over how to build your own Calculator using Haskell. Now take a seat, put your seat belt on, sit back, and relax as we begin our journey into the course of programming languages.

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 functions and loops in Python and C++ that returns the harmonic number of an integer.

Functions in 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++

```
#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 repeatedly 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.

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

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

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)

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

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

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

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

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

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 there 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 what an outcome is supposed to be and which 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 the 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].

```
class Monad m where
  (>>=) :: m a -> ( a -> m b) -> m b
  (>>)  :: m a -> m b        -> m b
 return ::  a                -> m a
```

Each Monad class has a similar structure, as seen in the code above. The structure is divided up into three chunks: a type constructor that assigns the type of the function, a function that takes one value and returns a calculation that returns a value, and a function that takes two calculations and executes them one at a time, allowing the result of the first calculation available to the second. The operator >>= binds the value returned from a computation of one function to another.The operator >> works similarly to >>=, except it ignores the value produced by the function [FL Monads].

The type class also has three rules, called the monadic rules: [TP Monads]:

1. Left Identity Law

```
        return x >>= f = f x
```

2. Right Identity Law

```
        m >>= return = m
```

3. Associativity

```
        (m >>= f) >>= y = m >>= (x -> f x >>= y)
```

The Left and Right Identity Laws have 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]. The laws work together to ensure that the transformations of imperative programs are correct and have a logical output [Monad Laws].

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 Functor with additional features [TP Functors]. The third rule is the ensure that Monads in Haskell follow the rules associated with Functors.

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

In this example of fmap, we utilize the map function, which applies a function to each element of a functor, such as a list, and returns an updated list of each individual element's output [TP Functors]. The map function is also an important function to be familiar with as it does come up in Assignment 2. Say that we are given a list of numbers, $[1, 4, 5, 9, 3]$ and we want to apply the function $squareElement$ to each number in the list. Since lists fall under the functor cateogory, we can use the function apply the $squareElement$:

```
main = do
   print(fmap (squareElement) [1, 4, 5, 9, 3])
   print(map (squareElement) [1, 4, 5, 9, 3])
   print (map (squareElement) (Just 3))
   print (fmap (squareElemen) (Just 3))
```

```
Output:
   [1, 16, 25, 81, 9]
   [1, 16, 25, 81, 9]
   error
   Just 9
```

Here we can see our code from running our new fmap function using Functor, and the built in map function that Haskell provides. We get the same output for an input of a list, but with other data types, such as Nothing or Just, we get an error for *map* since the data type is not compatible [TP Functors]. Functors open us up to more data types to be used alongside functions. Although this is may be daunting and confusing at first, think of monads and functors as bridges 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.

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

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:

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

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:

```python
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. Now taking these concepts, we can dive deeper into the history of Lambda Calculus and its creator, Alonzo Church.

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]. Church, alongside many additional accomplishments in his life, helped create the foundations of theoretical computer science, which is why we are here today, with so many technological advancements. An interesting fact about Church is that he was a university professor for both mathematics and philosophy which can be another reference to how math and philosophy can be connected, similar to Monads [NAS].

To represent numbers, Church denotes numbers as [CN]:

$$
\begin{gathered}
0 = \lambda f.\lambda\ x.x\\
1 = \lambda f.\lambda\ x.(f\ x)\\
2 = \lambda f.\lambda\ x.(f\ (f\ x))\\
3 = \lambda f.\lambda\ x.(f\ (f\ (f\ x)))\\
4 = \lambda f.\lambda\ x.(f\ (f\ (f\ (f\ x))))
\end{gathered}
$$

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]. There are addition and multiplication which can be represented in Church numerals as:

$$Addition = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$$
$$Multiplication = \lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x$$

which uses a similar structure as +1, but includes a new variable, m, to represent a number besides 1, where the placement of the parenthesis between the variables is important in syntax to ensure the correct operation. 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. More Lambda Calculus topics and concepts will be discussed in section 3 of this report as well, where we discuss the theory of programming languages, so keep in mind that Lambda Calculus is used in all aspects of programming languages; this is merely scratching the surface.

2.5 Introduction into the Haskell Project

For my Haskell Project, I would like to explore an integration between Haskell and a GUI interface for Assignment 1. For Assignment 1 of this course, we started with implementing a calculator in Haskell by using an interpreter and a parser. The calculator was tested on our local machines using the $echo"(1+2)^2*3--"|./Calculator$ command on our terminal where after $echo$, we would put the equation we want to evaluate. For this project, I will be modifying this assignment by creating a Calculator in Haskell using the Threepenny-Gui: GUI framework that will create a calculator interface on a web browser for users to interact with by clicking buttons to perform calculations [Threepenny]. The calculator would be able to perform addition, subtraction, multiplication, division, modulo, and exponential operations and handle integers and floats. It will also have error checking integrated where users would see an error message if they perform calculations such as division by 0. More description of the web interface and the Threepenny-Gui will be expanded on in Section 4.

3 Programming Languages Theory

So now that we have gained a brief overview of Haskell and all the ways it can be used in our code, let us switch over to the theory of programming languages. Haskell required a skill set that was beyond coding in another programming languages, such as Python, Java, or C++, as it also required users to know Discrete Mathematics, recursion in place of for or while loops, sets of numbers, and an introduction into Lambda Calculus to use the language to its maximum ability.

Now let's shift our mindset from learning a functional programming language to now developing our own language. When developing a programming languages, functional or imperative, every programmer must first be familiar with the theory behind every programming language such as:

1. More knowledge of Lambda Calculus and Church's contributions to the theory of computer science
    a. Church Numerals in Python and Haskell
2. What is Turing Complete?
    a. What are Turing Machines?
3. What is the Halting Problem?
    a. Why is it undecidable?
    b. What are other undecidable problem?
4. Abstract Reduction Systems and String Rewriting
    a. Abstract Reduction Systems and Trees
    b. String Rewriting
        - Confluence
        - Normal Forms
        - Termination

Picking up from Lambda Calculus, we will first expand our skill set with more theorems and concepts that are vital to the theory of programming languages and how Alonzo Church and Alan Turing (who's story was told in one of my favorite films, The Imitation Game), both contributed to the field we are in today with Church Numerals, Lambda Calculus, Turing Complete, and Turing Machines. As Church was a close mentor and advisor to Turing, the two also created a concept called Church-Turing Thesis which is interesting to note the relationship and combination of brainpower between two important figures in this field. From there, we will touch upon the Halting Problem, which discusses if a program will end or continue forever depending on the program and its inputs. The Halting Problem relates directly to Turing Machines and is an interesting concept in computer science to discuss. String Rewriting will follow, focusing on confluence, termination, and normal form, where the theory portion of this paper will then conclude as we take a deeper dive into the theoretical side of this course and computer science itself.

3.1 More In-Depth Information of Lambda Calculus and its Importance in Computer Science

As discussed in the previous section, Lambda Calculus plays an important role in utilizing the core components of Haskell. Overall, Lambda Calculus plays a larger role than being a key feature in Haskell, it is a part of every functional programming language. It can be and is often considered to be the simplest or smallest programming language there is, as it represents computation of inputs and outputs based upon variable binding and substitution [JRebel]. In section 2.4, we discussed basic functions denoted in Lambda Calculus, Church Numerals and Booleans, and how addition and multiplication are represented in Lambda Calculus notation. Let's discuss the syntax of Lambda Calculus more in-depth and how it is a valid representation of the mathematical function:

$$3 = \lambda f.\lambda\ x.(f\ (f\ (f\ x)))$$

Starting with Church Numerals, let us break down this formula to get a better grasp of the syntax. In Section 2.4.1, when we discussed Church Numerals, we used two inputs, $\lambda f.\lambda x.$. The first two items of every Church Numeral always started with $\lambda f.\lambda x.$, where λf represents a function and λx represents a value. The λ symbol represents an input to the Lambda Calculus syntax, the function and the value, in that order. Church's interpretation of his Numerals consists of only these two parts where λf is used repeatedly to indicate the action of applying any given function a certain number of times to a value, λx. It is assumed by our understanding looking at Church Numerals that λx represents 0 and λf represents a successor function

(+1), which would give us a natural number output of 3. However, keep in mind that Church Numerals simply means applying a function to a value for a certain number of times. In our case, Church Numeral 3 means applying a function 3 times to a value [CE].

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

Now shifting over to Addition, we can now see we have four inputs, two more than we saw in Church Numerals. Let's dissect this formula in parts to understand how this is an appropriate representation of Addition. We can see that our four inputs are λm, λn, λf, and λx, where we see that our last two inputs, λf, and λx, are the same inputs we used for Church Numerals, a function and a value. Now the natural question would be what is λm and λn? Yet again, referencing back to Section 2.4.1, we can see that the general formula for Church Numerals where the number of times a function is applied to a value is now defined as:

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

In this example, λn represents the number of times λf, our function is applied to the value λx. In our addition example, as there are two values compared to one, where m is called before the f, indicated a $m+n$ function [CE]. $\lambda n.\lambda f.\lambda x.f\ ((n\ f)\ x)$ can be read as $m+n$ where λf is +. With a more in-depth explanation of the syntax of lambda calculus, you will now be able to define multiplication, exponents, and more!

3.1.1 Church Numerals in Python and Haskell

Now as we know from the symbol of Haskell, Lambda Calculus is a very important part of the language and is vital for us to know. Therefore, one may assume that there must be a way to implement Church Numerals in Haskell as well. Let's first start with implementing Lambda Calculus in a language that we are more familiar with, such as Python. Luckily for us, Python has a built in function to its language called lambda that can take a number of arguments, and results in one expression [LCP]. For example:

```python
def lambdaEx(n):
  return lambda x : x + n

plusTwo = lambdaEx(2)
plusTen = lambdaEx(10)

print(plusTwo(4))
print(plusTen(4))
```

will result in the number 6 and 14, respectfully. Although the variables plusTwo and plusTen both reference the lambdaEx function, both of their functionalities of adding 2 and 10 to the inputted number occur successfully. Notice how $print(plusTwo(4))$ and $print(plusTen(4))$ only take in one argument, the number to add two and ten (4), and not $plusTwo(4,2)$ and $plusTen(4,10)$. With the definition of $plusTwo = lambdaEx(2)$ and $plusTen = lambdaEx(10)$, the program already knew which number to add to the input. This may seem a tad bit obvious, but this is not a common syntax that we are used to. Remember that plusTwo and plusTen are variables and yet we are using them as functions with an argument! Now let us take a look at how Lambda in Python can also implement Church Numerals [LCP]:

```python
def num(function):
    def count(x):
        return x + 1
    return function(count)(0)

zero = lambda f: lambda x: x
one = lambda f: lambda x: f(x)
```

```
successor = lambda n: lambda f: lambda x: f(n(f)(x))
three = successor(successor(successor(zero)))

print(three)
print(num(three))
```

```
Output:
<function <lambda> at 0x1026909d0>
3
```

Here we can see a similar syntax as we saw for $n = \lambda n.\lambda f.\lambda x.f\ ((n\ f)\ x)$ for the variable successor in our code, which essentially represents +1. In our code, zero and one are also defined similarly as they would be as we can see on page 12, and we utilize our successor function three times to get an output of the number 3. Remember that Church Numerals represents the application of a function to a value for a certain number of times. In our case, the value zero had the successor function applied to its value three times, resulting in the number 3 as an output. The *num* function is used on our case to have that integer representation of the Church Numeral, opposed to the first print statement which is a memory address which does not tell us a whole lot about the Church Numeral.

Now in Haskell, the representation of Church Numerals Addition is as follows [Church Numerals in Haskell]:

```
type ChurchNumeral a = (a -> a) -> a -> a

zero :: ChurchNumeral a
zero = \f x -> x

two :: ChurchNumeral a
two = \f x -> f (f x)

churchN :: Integral a => ChurchNumeral a -> ChurchNumeral a
churchN n = \f x -> f (n f x)

churchAdd :: Integral a => ChurchNumeral a -> ChurchNumeral a -> ChurchNumeral a
churchAdd m n = \f x -> m f (n f x)

churchMulti:: Integral a => ChurchNumeral a -> ChurchNumeral a -> ChurchNumeral a
churchMulti m n = \f x -> m (n f) x

plus_one x = x + 1

churchToNum :: Integral a => ChurchNumeral a -> a
churchToNum n = n (plus_one) 0

main = do
  let three = churchN two
  putStrLn $ (show $ churchToNum three)
  putStrLn $ (show $ churchToNum zero)
  putStrLn $ (show $ churchToNum (churchAdd two three))
  putStrLn $ (show $ churchToNum (churchMulti two three))
```

```
Output:
3
0
```

```
5
6
```

Yet again, we see a very similar, if not exact same, syntax in defining Church Numeral 0, 3, and n. This is not very surprising since as we have discussed, Haskell heavily relies on Lambda Calculus for its functionality. Looking at the code above may be a bit confusing at first, so lets break it down to understand how and why it works. We begin with defining the type *ChurchNumeral*, which is referenced throughout the code as the typeclass of the Church Numerals we will be using. We define the Church Numerals *zero* and *two* as well, the same as noted on page 12 and in the simplest functional programming language that was created for Assignment 2. *churchN* serves as the successor function, which is similar to $n = \lambda n.\lambda f.\lambda x.f\ ((n\ f)\ x)$, where we have seen throughout this paper and in the Python script above.

churchToNum is a handy tool we use in this script to convert the Church Numeral we define to an Integral (or whole number) representation for an output. This was the trickiest part of the script, where $churchToNumn = n(plusOne)0$ is a short but critical component of the code. This line is essentially an if-then-else statement: if n is greater than 0, then apply the *plusOne* function to n for the numerical representation of the Church Numeral. Else, return 0. Without any part of this line, the script would error out, since the value, a function, and a base case is needed. Say we change the statement to be $churchToNumn = n(plusOne)1$, then the output of *churchToNum three* would increase by +1, to 4, and *churchToNum zero* to 1, which is incorrect, as the last part of the syntax is the base case, or the lowest number possible, which must be 0. We then apply the successor function *churchN* to *two*, which results in the program return a Church Numeral 3 as the output after converting from ChurchNumeral type to a whole number, and the same is done to zero with no successor applied.

Church Addition and Church Multiplication is also included in the Haskell code which also looks identical to the Addition and Multiplication on page 12, which gives us an output of 5 from *churchAdd two three* and an output of 6 from *churchMulti two three*. Church Addition is similar to Church Successor (*churchN*) with an addition input m which matches the function: $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$. Church Multiplication is also the same to its lambda calculus function of $\lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x$. Both outputs work as expected, showing us that Church Addition and Multiplication perform accurately and verify its accuracy. Church Numerals and Lambda Calculus in general, can be defined and created in multiple languages, such as Python and Haskell, and are an important part of programming languages theory as it is the starting blocks as to what we know the field computer science to be today.

3.2 What is Turing Complete

A phrase you may hear during one's beginning steps in Lambda Calculus is that lambda calculus is Turing Complete. Now if you are anything like me, a bit a film buff, my first reaction to this phrase was thinking about Alan Turing, who was portrayed in the film, The Imitation Game (2014). This film shows Turing's contributions in creating the foundations of important computer science theories today, such as Artificial Intelligence and Machine Learning, with his paper [Computing Machinery and Intelligence], proposing the first question of if machines can think (I would highly recommend this movie, it was very well done and one of my favorites). Alongside Turing's contributions to AI and Machine Learning, he also added his fair share to the advancement of programming languages and its theory as well with Turing Complete and Turing Machines.

Turing Completeness is a concept which essentially evaluates the capabilities of a programming language: is the language equipped to handle any and all sorts of algorithms? For example, HTML and JSON are not Turing Complete, as they cannot perform all sorts of algorithms. HTML is mainly used for website design and interface, whereas JSON is used to store structured data, making it easier for developers to pull data/information from. Below is a simple example of a HTML script that sets up the interface for addition between m and n while JavaScript is used to perform the operation itself:

```
<input type="text" id="m"><br/>
```

```
<input type="text" id="n"><br/>
<input type="text" id="+"><br/>
<input type="button" value="+" onclick="additionCalc()"><br/>

<script>
    <!-- End of HTML portion, Javascript used here alongside HTML-->
    function additionCalc(){
        let m = document.getElementById("m").value;
        let n = document.getElementById("n").value;
        let add = Number(m) + Number(n);
    document.getElementById("+").value = add;
}
</script>
```

Here we can see that without JavaScript (which is another programming language that can be integrated with HTML), the raw HTML code cannot perform an addition operation. Using the script functionality, it is able to utilize JavaScript, but there is no defined way for HTML on its own to perform any type of algorithm/operation. Although HTML and JSON are important to learn and provide a vital service, they do not have the capabilities to run algorithms such as mathematical operations or while loops. Without the functionality to perform computations, these languages cannot be considering Turing Complete [TC]. However, languages such as Python, Java, C++, and even Lambda Calculus are. It can be said that Lambda Calculus is the shortest and simplest programming language there is, but what exactly makes Lambda Calculus Turing Complete you may ask?

Even though Lambda Calculus is more abstract than other languages that you may be familiar with, such as Python or Java, it does pass this evaluation since it is a notation that represents functions and computations, with an input and an output. Since this is an algorithm, which takes an input to perform logic upon to produce an output, Lambda Calculus is Turing Complete. You may be wondering how exactly do you prove that Lambda Calculus is Turing Complete? Well, it is possible to write an interpreter in Lambda Calculus that can execute any Turing machine. Lambda Calculus is essentially rewriting expressions using beta reduction theory [PL]. Now thinking about three concepts, Turing Machines, Lambda Calculus, and String Rewriting, how do all these concepts relate? Well, let us cover Turing Machines and string rewriting and revisit this question at the end of the section [Turing Completeness].

3.2.1 What is a Turing Machine

A Turing Machine is considered to be an abstract mathematical theorem that breaks down the definition of what programmers interpret computations to be [Turing Completeness]. Next to Lambda Calculus, Turing Machines are the next simplest programming language there is which was invented to define what an algorithm is. It is also known as the simplest imperative language, which is a direct contrast to Lambda Calculus which is the simplest functional language. Now although it is misleading using the word machine in a concept that is said to be abstract, but let us learn a bit more about this concept to see how it can be considered to be a machine. Turing Machines are broken down into two aspects: a computational head and an extremely (infinitely) long tape [Turing Completeness]. A key takeaway in this definition is that the tape is infinite. If finite, the tape would then, in theory, end, and the machine would stop performing calculations. Today, the tape can be interpreted as random access memory (RAM) or external memory such as a disk that every computer has and the head is a read and write interpreter that reads the information that is passed, records it, and decides which direction to move the tape.

As we can see, although Turing Machines are described as abstract, it is abstract due to the time frame the concept was originated. Turing coined the idea back in the 1930s, where there were no computers and machinery that people today use in their everyday tasks. Alan Turing left an important mark in the computer science field, as his idea of a Turing Machine is the basic groundwork of every imperative programming language we use, as well as the laying the groundwork of the basic foundation of computers and laptops. However, since Turing Machines were invented before computers and the programming languages we use today, where we can clearly see an error in our code, there were many follow-up questions to pick at Turing's brain. As mentioned previously, a way to determine if a programming language is Turing Complete is if the language can replicate a Turing Machine. If one can successfully mimic a Turing Machine, it is said that it can compute any and all functions [Turing Completeness]. However, this definition assumes that all functions that are created are computable, and would also produce an output. Now, this definition does not account for functions that are incomputable or create infinite loops (does the function ever complete execution?). This question posed by Martin Davis is also known as The Halting Problem, which will be discussed further in the next section, Section 3.3 [Turing Completeness].

3.3 The Halting Problem

As stated previously in 3.2.1, a way we can determine if a programming language is Turing Complete is if the language can replicate a Turing Machine. This can also be translated to that all and every function will be able to successfully run in a Turing Machine. However, this implies that every function created is a computable function which will execute and terminate in a successful manner. As computer scientists, who have been coding for at least a few years I assume, we know this is far to be true. Most code that we write and run for the first time will error out due to either logic or syntactical issues, which in turn, makes them incomputable functions. In addition to, having infinite loops, which will continue on forever given a certain input poses the question if the program ever ends. Is it still a Turing Machine if it only runs one algorithm forever? This creates an issue in the Turing Machine theory, known as the Halting Problem [Turing Completeness].

```python
# example of an infinite loop problem in Python
def infinite_loop(x):
    if (x >= 2):
        print("The number is greater than 2")
    while (x < 2):
        print("The number is less than 2")

infinite_loop(4)
infinite_loop(1)
```

The main question the Halting Problem proposes is does the function/program terminate given a certain input. In the Python example above, we see an example of an infinite loop, where the program would first print out "The number is greater than 2" and then run forever, printing "The number is less than 2" since it is under a while loop that never is False. Since the condition is always True, and will never be set to False, it continues printing the same statement forever. Although we have a clear example of what an infinite loop looks like, it is difficult to predict what type of algorithm would be passed through every Turing Machine. If a function is incomputable, then there is no Turing machine that can compute an output [Turing Completeness]. Considering that Turing Complete is either a True/False or Yes/No question, where the language can be either Turing Complete or not, that makes the problem decidable. Since the Halting Problem deals with incomputable functions, that makes the problem undecidable.

Now three steps for computer scientists to follow when determining if a function/program is a decidable or undecidable problem:

1. There is a program and an input
2. The program passes the the Halting Problem test if it terminates given the input
3. The program fails the the Halting Problem test if it does not terminates given the input

Other examples of undecidable problems in abstract computer science theory is the Busy Beaver Problem and Rice's Theorem [Undecidable Problems]. Now when first being introduced to these concepts and theorems may be confusing, but remember that The Halting Problem simply asks the question does the program terminate given an input and if that is considered a Turing Machine. Having programs that run forever on a machine would not be cost efficient at all. Imagine having one machine that computes 1+1 all day, everyday, forever. This is why this question was and still is an important concept to be aware of when the engineering and computer science fields expanded and continues to grow. Ensuring the computers and future machines would have a way to handle these concerns was vital to the success of the field, since as we know today, we handle infinite loops by terminating the program ourselves from our terminal, using Control-C.

3.4 Abstract Reduction Systems and String Rewriting

Abstract Reduction Systems is another important concept in programming languages theory that relates to computer interpreters and how our machines understands the code that programmer execute. When programmers run their code, the interpreter of the language essentially rewrites the code in order for the code to execute successfully. The idea behind string rewriting may bring up several questions, such as how to implement the rules and if the order of the rules effect the outcome. The best rewriting systems would allow for any order of execution, which is ideal for programmers as we would be able to reduce the expression in the order that works best for our understanding and leads to faster runtimes. It is possible to find the fastest runtime since it would be possible to find the computation order that runs the program the fastest [PL]. Abstract Reduction Systems gives programmers the unique opportunity to inspect the question of order and runtime at the level of abstraction, which will be discussed further in the next section.

3.4.1 Abstract Reduction Systems and Trees

Abstract Reduction Systems (ARS) has two main components, a set and a relation. The set consists of the set of elements that is to be rewritten and the relation can be thought of as the rule or implementation for defining the one-step rewrites. In a mathematical sense, an abstract reduction system (ARS) is a set A together with a relation $\rightarrow \subseteq A \times A$. The element A represents all forms of data types such as strings, lists, integers, floats, and more. The notation of ARS is important to grasp to understand how concepts such as reduction and rewriting works, such as $\rightarrow$ and $\rightarrow^*$ [PL].

In an ARS, given two elements, such as a and b, we use the symbol $\rightarrow$ to represent reduction. In other words, ba $\rightarrow$ ab means that ba reduces to ab where $\rightarrow$ is a one-step computation that reduces (or simplifies) ba to ab. Given the string *abbabab*, to rewrite the string as *aaabbbb*, the system can compute a one-step computation for every first occurrence of *ba* to move to the first of the string:

1. *abbabab* $\rightarrow$ *ababbab*
2. *ababbab* $\rightarrow$ *aabbbab*
3. *aabbbab* $\rightarrow$ *aabbabb*
4. *aabbabb* $\rightarrow$ *aababbb*
5. *aababbb* $\rightarrow$ *aaabbbb*

This process can also be easily implemented Python as well, using a Bubble Sort algorithm or the sorted built-in function in Python:

```
#example of a rewrite in Python, using the concept of for every first occurrence of a to move to
    the front of the string using Bubble Sort
def rewrite(str):

   #string to list
   rewrite_str = []
   for i in str:
       rewrite_str.append(i)

   #bubble sort
   for i in range(len(str) - 1):
       for j in range(len(str) - i - 1):
           if rewrite_str[j] > rewrite_str[j + 1]:
               rewrite_str[j], rewrite_str[j + 1] = rewrite_str[j + 1], rewrite_str[j]

   #list to string
   new_str = ""
   for i in rewrite_str:
       new_str = new_str + i

   print("The rewritten string is:", new_str)


rewrite('abbabab')
rewrite('adbbcabadb')

#results:
aaabbbb
aaabbbbcdd
```

However, the ARS can follow all sort of rules to simplify for every last occurrence of *ba* to move to the back of the string:

1. $abbabab \rightarrow abbaabb$
2. $abbaabb \rightarrow ababab b$
3. $ababab b \rightarrow abaabbb$
4. $abaabbb \rightarrow aababbb$
5. $aababbb \rightarrow aaabbbb$

Another way to display this rewrite is with the use of a handy tool called an Abstract Syntax Tree (AST). Best compared to a binary search tree, an AST provides programmers a visual view of an ARS to find the all possible solutions, and see which one works best for the problem at hand. Keep in mind that Figure 1 below shows an extremely clean AST, following the examples of the ARS above. AST typically would branch out to all possible versions of an outcome to visually display all paths that may be followed, similar to Figure 2.

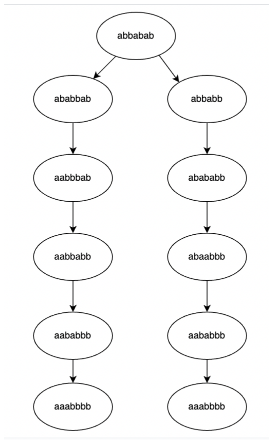

Figure 1: Abstract Syntax Tree of the two ARS examples above

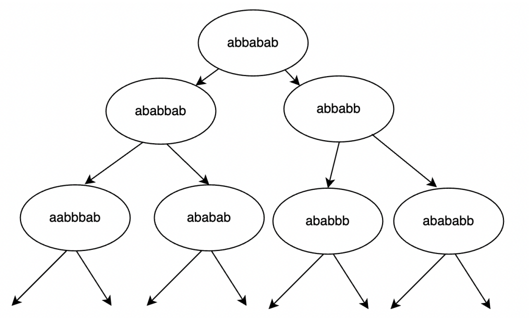

Figure 2: Typical AST layout with multiple branches and paths

There are many different ways that an ARS can reduce a string to be in an ideal form. Now let us shift over to review concepts learned in mathematics, such as reflexive, symmetric, and transitive properties:

1. Reflexivity revolves around the concept that for every real number (an element from set $\mathbb{R}$) x, $x = x$
2. Symmetricity revolves around the idea that for all real numbers (elements from set $\mathbb{R}$) x and y, if $x = y$ then $y = x$
3. Transitivity states that for all real numbers (elements from set $\mathbb{R}$) x, y, and z if $x = y$ and $y = z$, then $x = z$

For ARS notation, $\rightarrow^*$ represents reflexive transitive closure. Essentially, this means the reflexive and transitive closure of $\rightarrow$ and $\longleftrightarrow^*$ for the symmetric, reflexive and transitive closure, that is the smallest equivalence relation containing $\rightarrow^*$. $\longleftrightarrow^*$ is the reflexive, symmetric and transitive closure of $\rightarrow$ (can also be denoted as a $\equiv$ b) [PL]. These symbols will appear throughout programming language theory, especially when discussing ARS and is helpful to keep in mind going forward.

3.4.2 Confluence, Normal Forms, and Termination

Other terms and definitions that are key when discussing abstract reduction systems are confluence, normal forms, and termination [PL]:

1. Confluence relates to if x reduces to y and z, then y and z are join-able since y and z both reduce to the same element
2. Normal form is for given the element x, if there is no reduction computation to be performed ($x \rightarrow y$)
3. Termination occurs if there is no infinite chain or loop

Now this may be confusing at first without an application or visual to see how a set and relation is or is not confluent, terminating, or in normal form. Rather than explaining through words, Figures 3-6 show various applications visually that describe these concepts.

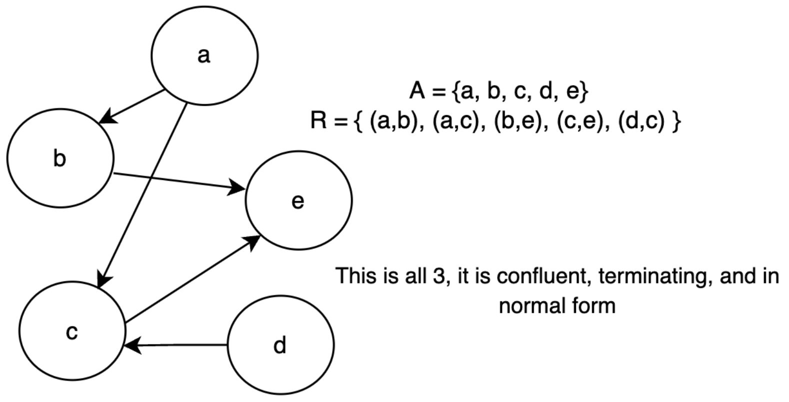
a
b
c
d
e
A = {a, b, c, d, e}
R = { (a,b), (a,c), (b,e), (c,e), (d,c) }
This is all 3, it is confluent, terminating, and in normal form

Figure 3: Graph that is confluent, terminating, and in normal form

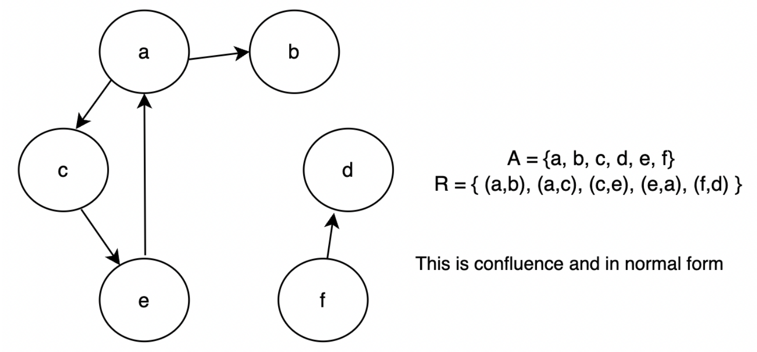
a
b
c
d
e
f
A = {a, b, c, d, e, f}
R = { (a,b), (a,c), (c,e), (e,a), (f,d) }
This is confluence and in normal form

Figure 4: Graph that is confluent and in normal form

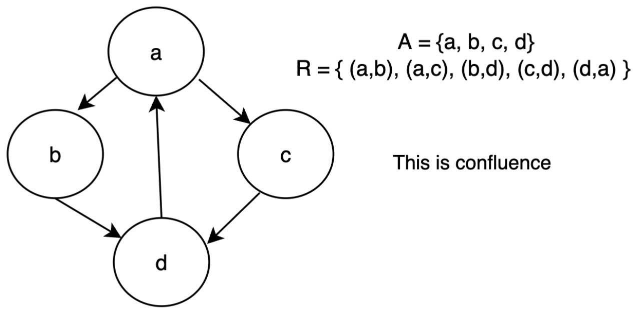

Figure 5: Graph that is only confluent

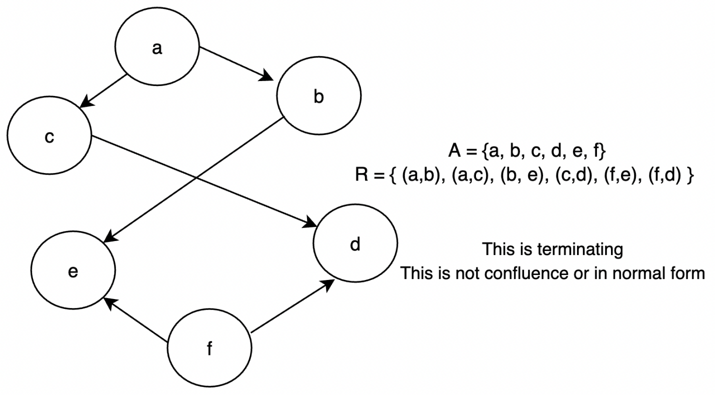
a
b
c
d
e
f
A = {a, b, c, d, e, f}
R = { (a,b), (a,c), (b, e), (c,d), (f,e), (f,d) }
This is terminating
This is not confluence or in normal form

Figure 6: Graph that is only terminating

3.4.3 Graph descriptions

Now for a bit of explaining for each of the graphs to better understand what their meaning is. For all graphs, A represents the set of elements that makes up the graphs and R are the multiple relations that are displayed with the arrows. Figure 3 displays a graph that meets all 3 definitions above, as it is confluent, terminates, and is in normal form. Figure 3 has 5 elements and 5 relations, have no infinite loops (which indicts termination), in normal form, and for every fork, there is a join (which indicts confluence). Figure 4 shows an example of a graph that is non-terminating. Looking specifically at the relations (a, c), (c, e), and (e, a), there is an infinite loop represent where it is possible to run $a \to c \to e \to a \to c$ ... forever, hence making this graph non-terminating. It does however, meet the definition of normal form and confluence where for every fork, there is a join. Figure 5 displays a graph which is both non-terminating and not in normal form. It is not terminating as there is an infinite loop constantly from either direction you may chose to go: $a \to b \to d \to a \to b$ ... or $a \to c \to d \to a \to c$ ... which continues forever. It is also not in normal form since there are outgoing arrows. Figure 6 only meets the termination definition, where it does end at one place and there is no infinite computation possible. It is not confluence since every fork does not have a join and not in normal form as well. These concepts may take a bit of time to get used to, but similar to Haskell, with a bit of practice and using visuals to draw out your understand, ARS and AST is a vital tool for programmers to be familiar with and use in the future.

3.4.4 Lambda Calculus and Turing Completeness

Let us now circle back to the question that we had in Section 3.2 about Lambda Calculus and if it is Turing Complete. We left off with asking how Turing Machines, Lambda Calculus, and String Rewriting relate to each other. Now that we have covered each topic, we can see some similarities. Turing Machines are equally expressive in Lambda Calculus since it can be reduced down to a Turing Machine. Lambda Calculus is equally expressive to Turing Machines as it can be reduced to Lambda Calculus. We know now that any Turing Machine can be represented as some form of a string rewriting system. The question now shifts over to whether all string rewriting systems can be implemented in Lambda Calculus. Recall in Assignment 2, we wrote an interpreter in Lambda Calculus (LambdaNat.cf), which served as the rules of string rewriting and used Haskell to bridge the gap by following the chain of: string rewriting $\to$ Haskell $\to$ Lambda Calculus. An interesting problem to tackle for the future (or next semesters report) would be to have a full mathematical proof to denote this verification, but for now, take a look at Assignment 2 and 3 for some interesting applications of this concept using Lambda Calculus and Haskell.

4 Project: Haskell Calculator GUI

As mentioned in Section 2.5, the project portion of this report will focus on creating a Calculator GUI interface in Haskell. Creating a calculator in Haskell is the beginning steps in this course to introduce key concepts of Haskell and programming language theory, such as the interpreter and parser. Haskell does not have many popular GUI interfaces that can be used, unlike Python, which has Flask, a popular micro web framework that integrates Python and HTML on the same file, but it does have a toolkit called [Threepenny] which provides a GUI framework that uses the web browser as a display, which was first published in 2013 and has been revised earlier this year, in May 2021. The library allows programmers to utilize the HTML DOM alongside JavaScript events with Haskell code.

4.1 Inspiration and Documentation

Inspiration to pursue this project came from the first assignment of the course, where we created a Calculator to run in our terminal using the $echo"(1+2)^2 * 3"|./Calculator$ command in Haskell. During user testing, it was difficult to rewrite the same line of code to test out different formulas, which led into the question if there can be a way to create a GUI interface to run a Calculator in Haskell. Figure 7 displays the final product of the calculator, which after compiling, will display on the link http://127.0.0.1:8023. Those who have contributed to the Threepenny library provide amazing documentation and examples of how to use the framework on Github to help guide the process of learning this tool [ThreepennyElectron] [ThreepennyCalc] [ThreepennyExample].

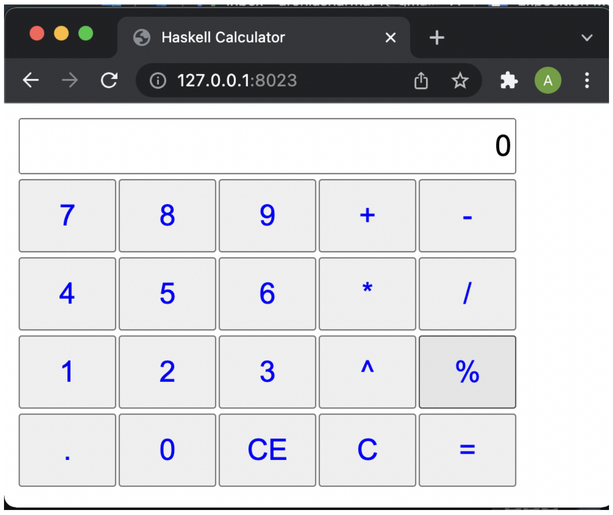

Figure 7: Final Product of the Haskell Calculator using a GUI interface

The final product allows for users to perform addition, subtraction, multiplication, division, modulo division, and exponents. The default data type for this calculator are floats, which allows for decimal calculations. CE is a functionality that clears the error message and C clears the current number and restarts to 0.

4.2 Code Setup

The set up this interface to run on your desktop is relatively simple, thanks to active Github user Aleksei Pirogov who provides documentation on how to run the Threepenny Haskell code using cabal [ThreepennyCalc]. To run, simply pull from the final project repository on Github [FinalProjectRepo] and run the two commands to interact with the application on http://127.0.0.1:8023:

```
cabal new-build
cabal new-run threep
```

The code is split up into two files, *Main.hs* and *Calculator.hs*, where *Calculator.hs* focuses on handling the computational side of the program, such as performing the calculation and handling errors and *Main.hs* focuses on the interface components and user interface, such as the buttons and the size of the output.

4.3 Calculator.hs Overview

Starting off with *Calculator.hs*, the file starts off with defining important function established in the program and defining vital data types:

```haskell
-- operations calculator can handle
data Operation = Add | Sub | Mul | Div | Mod | Exp deriving (Show, Eq)

-- data type of the numbers on the Calculator
data Digit = Zero | One | Two | Three | Four | Five | Six | Seven | Eight | Nine deriving (Show,
    Eq)

-- data type representing commands executed by calculator
data Command = Digit Digit      -- number
             | Dot              -- decimal point
             | Operation Operation -- operation
             | Equal            -- =
             | Clear            -- C
             | ClearError       -- CE
             deriving (Show, Eq)


data State = FormulaA   Formula                -- Formula register A
           | EnteredAandOp Double Operation       -- A, Op
           | FormulaB    Double Operation Formula -- A, Op, Formula register B
           | Calculated  Double Operation Double -- A, Op, B
           | Error       Double String            -- A, Error Message
           deriving (Show, Eq)
```

The first data type defined is the Operations that users can perform (addition, subtraction, multiplication, division, modulo, and exponential). The data type Digit is defined as well that represents the 10 digits (0-9) that appear on the application. The data type Command is used to define the buttons that users select on the calculator, such as if they select a Digit, the Equal sign (=), or the Clear symbol (C). State is used to handle the operation the users wants to perform, such as when they input the first Digit (A), when they input the first Digit (A) and an Operation (Op) and when they input the second Digit (B). The answer to the formula is shown in Calculated or an Error is displayed if the computation is not possible. In order to convert our data type from a Integer to a Float, we defined the data type Formula, which takes in a String of the inputted Digit and a Boolean, which is True if the . button is pressed. By default, the first variables, FormulaA is always set to 0 and False.

```haskell
-- the digit as a string & boolean if there's a float number (dot is pressed)
type Formula = (String, Bool)

-- default Formula
instance Default State where
  def = FormulaA ("0", False)
```

Next, the button labels are assigned to the Command, Digit, or Operation that they represent:

```
-- taking a string and parsing to a command type
parseInput :: String -> Command
parseInput x = case x of
  "0"  -> Digit Zero
  "1"  -> Digit One
  "2"  -> Digit Two
  "3"  -> Digit Three
  "4"  -> Digit Four
  "5"  -> Digit Five
  "6"  -> Digit Six
  "7"  -> Digit Seven
  "8"  -> Digit Eight
  "9"  -> Digit Nine
  "."  -> Dot
  "+"  -> Operation Add
  "-"  -> Operation Sub
  "*"  -> Operation Mul
  "/"  -> Operation Div
  "%"  -> Operation Mod
  "^"  -> Operation Exp
  "="  -> Equal
  "C"  -> Clear
  "CE" -> ClearError
  _    -> undefined
```

The same idea goes for the next portion of the code where the reverse takes place (Digit Zero → "0" and so on) for the code to assign the Command/Operation/Digit to the symbol, which is displayed in lines 88-109. There are also portions of code that handle type conversion (Formula to Double and Double to Formula). In order to process and execute the equation the user inputs, the code defines different operations that the users would be allowed to perform, such as selecting a Command such as Equal (applyCommand), inputting Digits (addDigit), inputting a float number (addDot), or selecting a certain operation (applyOp).

```
-- compute the calculation/operation
processCommand :: Command -> State -> State
processCommand cmd = case cmd of
  Digit x      -> addDigit x
  Dot          -> addDot
  Operation op -> applyOp op
  command      -> applyCommand command
```

The functions addDigit and addDot serve a vital role to the application since they define the type of number that is being computed and recognize and save the Digit that is selected by the user. addDigit is set up as a case statement, which handles when the user selects the first Digit, the second Digit, an Operation, and outputs the computed Digit at the end. addDot is called when the . button is selected by the user where the variable FormulaB is used to save the trailing digits after the decimal point (FormulaB = 324 and FormulaA=5 in the number 5.324). These two variables are then concatenated together to represent the float number.

```
addDigit :: Digit -> State -> State
addDigit x s =
  case s of
    (FormulaA a)       -> FormulaA (update a)
    (FormulaB a op b) -> FormulaB a op (update b)
    (EnteredAandOp a op) -> FormulaB a op (num x, False)
    Calculated {}      -> FormulaA (num x, False)
    _ -> s
  where
    update (a, False) = (ccc a (num x), False)
    update (a, True) = (ccc a (num x), True)
    num i = labels (Digit i)
    ccc "0" "0" = "0" -- avoid to create leading 0 digits
    ccc a b    = a ++ b

-- addDot function to handle floats
addDot :: State -> State
addDot s =
  case s of
    (FormulaA a)          -> FormulaA (dotted a)
    (FormulaB a op b)     -> FormulaB a op (dotted b)
    _                      -> s
  where
    dotted (a, False) = (a ++ ".", True)
    dotted (a, True) = (a, True)
```

The function performCalc serves as the core of the program. This is where the computation of the program takes place, where it will either generate an output or an error message. Below, it can be seen that performCalc takes in a Double, an Operation, and another Double data type and outputs either a String (as the error message) or a Double (as the output). If the user tries to perform division or modulo division by 0 (2/0), an error message will be printed instead to alert the user of the error. Otherwise, the mathematical operation will take place. Note that $Mod- > modu$ is the only function that does not point to a symbol. Instead it points to the function modu, that is displayed above performCalc. Since Double does not have the mod function but Integral does, the modu function changes the data type to an Integral to give us the expected result from modulo division. The output is then generated using $f\ a\ b$ where f represents the function, and a and b represent the inputs. Notice how this syntax is similar to Lambda Calculus!

```
-- adding a modulo function to the calculator
modu :: Double -> Double -> Double
modu a b = fromIntegral $ mod (round a) (round b)

-- setting up formula & getting output of an operation
performCalc :: Double -> Operation -> Double -- A & operation & B
         -> (String -> a) -- error
         -> (Double -> a) -- result
         -> a
performCalc _ Div b calcError _ | b == 0 = calcError "Division by Zero!"
performCalc _ Mod b calcError _ | b == 0 = calcError "Mod by Zero!"
performCalc a op b _ calcResult =
  let f = case op of
           Add -> (+)
           Sub -> (-)
           Mul -> (*)
           Div -> (/)
           Exp -> (**)
           Mod -> modu
  in calcResult $ f a b
```

The last portions of *Calculator.hs* focuses on handling the Operation or Command order that the user inputs to the program. If the user selects a Digit and hits multiple Operations, an error message will appear, which is handled by the code below. It ensures that the order of operations is syntactically correct and if it is not, it outputs an error for the user that it cannot perform an invalid operation. Without this code, the application would crash as there would be no proper way to keep track of order of operations and invalid commands.

```
-- operation
applyOp :: Operation -> State -> State
applyOp op s =
  case s of
    (FormulaA a) -> EnteredAandOp (fromFormula a) op
    (FormulaB a op' b) -> performCalc a op' (fromFormula b)
                                  (Error a)
                                  (`EnteredAandOp` op)
    (EnteredAandOp a _) -> Error a "Invalid Operation"
    (Calculated a _ _) -> EnteredAandOp a op
    _ -> s


-- unary command (C, CE, or =)
applyCommand :: Command -> State -> State
applyCommand cmd s =
  case (cmd, s) of
    (ClearError, Error a _)      -> FormulaA (asFormula a)
    (Clear,     _)               -> def
    (_,         Error _ _)       -> s
    (Equal,     FormulaA _)     -> s
    (Equal,     EnteredAandOp a _) -> Error a "Invalid Operation"
    (Equal,     FormulaB a op b) -> calc a op (fromFormula b)
    (Equal,     Calculated a op b) -> calc a op b
    _                            -> s
  where
    calc a op b = performCalc a op b
                 (Error a)
                 (\a' -> Calculated a' op b)
```

4.4 Main.hs Overview

Shifting over to the GUI portion of the code, Main.hs handles how the application looks and interacts on your browser. Using built-in packages, such as Graphics.UI.Threepenny and Graphics.UI.Threepenny.Core, Haskellers are able to display the code in a simple interface that looks like a calculator on your phone. To set up the GUI interface, every Main.hs file must start with creating an entry point:

```
-- entry point from main.js launch script
main :: IO ()
main = startGUI defaultConfig setup

-- setup window layout and import css file
setup :: Window -> UI ()
setup win = void $ do
  -- define page
  _ <- return win # set title "Haskell Calculator"
  UI.addStyleSheet win "layout.css"
```

After starting the GUI interface, the next step is to set up the window UI as well. In this example, we set the title of the browser window to ”Haskell Calculator” and added a css file to improve the UI of the calculator to be smoother and pleasing to look at. Next, the output box and button displays are initialized in the UI. The output box is defined as a regular textbox and the buttons are split into two parts: displaying a certain button and displaying all buttons in a table. First, we define the variable buttons to map to the buttonDefinitions, which consists of the layout of the buttons on the app. The next portion handles the HTML/CSS side of the button display which uses HTML elements to recreate a table grid.

```
-- UI controls
  outputBox <- UI.input
               # set (attr "readonly") "true"
               # set (attr "style") "text-align: right; min-width: 400px; min-height: 40px;
                   font-size:24px"
               # set (attr "align") "center"

  -- button grid
  buttons  <- mapM (mapM mkButton) buttonDefinitions

  -- define page DOM with html combinators
  _ <- getBody win # set (attr "style") "overflow: hidden; align: center" #+
    [ UI.div #. "ui raised very padded text container segment" #+
      [UI.table #+ [UI.row [UI.div #. "ui input" #+ [element outputBox]]] #+
                  map (UI.row . map element) buttons]
    ]
```

Next, we setup the connection of the buttons to its corresponding functionality in *Calculator.hs*. buttonMap maps the buttons to its function, clicks keeps track of what the user selects, commands calls the processCommand function in *Calculator.hs*, calcBehaviour keeps track of all the commands that the user is selecting, and outText displays the result or the error to the user.

```
let
     buttonMap = zip (concat buttons) (concatMap (map fst) buttonDefinitions)

     -- register mouse click events to all buttons
     clicks = buttonClicks buttonMap

     -- use processCommand function to build the equation
     commands = fmap processCommand clicks

  -- calculate behaviour by accumulating all commands
  calcBehaviour <- accumB def commands

  let outText = fmap display calcBehaviour

  -- output textbox
  element outputBox # sink value outText
```

Lastly, the end of *Main.hs* focuses on how the buttons and table is displayed and wrap up the connection of the two files and process the Commands that are selected to run. mkButton represents the individual buttons and how they are displayed. buttonDefinitions focuses on how the table looks like on the application, where the order that it is defined in the variable is the same order/structure you see on the application. buttonClicks processes the clicks that the user selects and sends the inputs back to *Calculator.hs*. Finally, Color is a data type that was defined as the background text of the buttons.

```
-- button UI
  where
```

```
    mkButton :: (Command, Color) -> UI Element
    mkButton (cmd, clr) =
      let btnLabel = labels cmd -- get the button text
      in  UI.button #. ("ui " ++ color clr ++ " button")
                   # set text btnLabel # set value btnLabel
                   # set (attr "type") "button"
                   # set (attr "style") "min-width: 80px; min-height: 60px; font-size: 24px"
                   # set style [("color","blue")]

    color :: Color -> String
    color = map toLower . show

    -- buttons on the app
    buttonDefinitions :: [[(Command, Color)]]
    buttonDefinitions =
      [ [(Digit Seven, Grey), (Digit Eight, Grey), (Digit Nine, Grey), (Operation Add, Grey),
          (Operation Sub, Grey)]
      , [(Digit Four, Grey), (Digit Five, Grey), (Digit Six, Grey), (Operation Mul, Grey),
          (Operation Div, Grey)]
      , [(Digit One, Grey), (Digit Two, Grey), (Digit Three, Grey), (Operation Exp, Grey),
          (Operation Mod, Grey)]
      , [(Dot, Grey),      (Digit Zero, Grey), (ClearError, Grey), (Clear,     Grey), (Equal,
          Grey)] ]

    -- function to handle when the button is clicked
    buttonClicks :: [(Element, Command)] -> Event Command
    buttonClicks = foldr1 (UI.unionWith const) . map makeClick
      where
        makeClick (elmnt, cmd) = UI.pure cmd <@ UI.click elmnt

-- button background color
data Color = Grey deriving (Show, Eq)
```

4.5 Future Steps and Summary

By researching and following multiple tutorials, creating a GUI interface for a Haskell Calculator is successful! This was an interesting project to explore as it involved improving my understanding of Haskell and combining it with a HTML/CSS interface in a HTML/Haskell hybrid. As we saw in Assignment 1, Part 2, setting up Interpreter.hs was easier than this approach as it had minimal lines of code, where we only wanted a terminal output. With this new approach, although more code is necessary to build this application, I believe that it would be very interesting for more students to conduct this sort of project and experiment with further improving the UI to see more fascinating results.

5 Conclusion

In conclusion, I greatly enjoyed this class and learning not only a different programming language, but a different style of thinking as well. Special shout out to Professor Kurz for an amazing semester and to Dan Haub for being a wonderful SI. My favorite part of this class was revisiting Discrete Mathematics and applying concepts we learned to actual code instead of paper (as it was in Discrete Mathematics) and the final project. Thank you for an amazing semester (and if anyone else reads this, I hope you are enjoying the class or just overall enjoying my paper) and greatly looking forward to next semester in Compiler Constructions to carry on the journey!

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.

[FL Monads] Introduction to Monads , www.futurelearn.com/, 2021.

[Monad Laws] Monad Laws, wiki.haskell.org, 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.

[NAS] Alonzo Church, www.nasonline.org, 2021.

[JRebel] What Is Lambda Calculus and Should You Care, www.jrebel.com, 2013.

[CE] Church encoding, www.wikiwand.com, 2021.

[Computing Machinery and Intelligence] Computing Machinery and Intelligence, academic.oup.com, 1950.

[LCP] LCP, akajuvonen.github.io, 2020

[Church Numerals in Haskell] Church Numerals in Haskell, github.com, 2018.

[TC] Turing Complete, dev.to, 2020.

[Turing Completeness] Turing Completeness, medium.com, 2017.

[Undecidable Problems] Undecidable Problem, en-academic.com, 2021.

[CTNF] SI Worksheet 10/10/21, hackmd.io, 2021.

[Threepenny] threepenny-gui: GUI framework that uses the web browser as a display., hackage.haskell.org, 2021.

[ThreepennyElectron] Writing Haskell native GUI Applications with Threepenny GUI and Electron., github.com, 2021.

[ThreepennyExample] Threepenny Example, github.com, 2021.

[ThreepennyCalc] threep, github.com, 2019.

[FinalProjectRepo] FinalProjectRepo, github.com, 2021.