CPSC-354 Report

Aidan Wall
Chapman University

December 21, 2021

Abstract

Short introduction to your report . . .

Contents

1 Introduction **2**

2 Haskell **2**
2.1 What is Haskell? . . . 2
2.1.1 Haskell vs Python, A Comparison . . . 3
2.2 More About Haskell . . . 6
2.2.1 Recursion . . . 6
2.2.2 Haskell for Beginners . . . 6
2.2.3 Higher Order Functions Python vs Haskell . . . 8
2.3 Summary . . . 10

3 Programming Languages Theory **11**
3.1 Brief Overview and History . . . 11
3.1.1 Pre Programming Languages . . . 11
3.1.2 Foundation of Programming Languages and Foundation of Haskell . . . 11
3.1.3 Haskell Today . . . 11
3.2 Lambda Calculus . . . 11
3.2.1 Intro to Lambda Calculus . . . 11
3.2.2 Comparing Lambda Calculus to C/C++ . . . 12
3.3 Intro To Parsing . . . 13
3.4 Ambiguous vs Non Ambiguous Grammar . . . 14
3.5 String Rewriting . . . 15
3.5.1 Confluence . . . 15
3.5.2 Terminating . . . 16
3.5.3 Unique Normal Forms . . . 16
3.5.4 Example . . . 17
3.6 Hoare Logic . . . 17
3.6.1 Introduction and History . . . 17
3.6.2 Implementation and Usage . . . 18

4 Project **19**
4.1 Rock-Paper-Scissors . . . 19
4.1.1 Defining a Rock-Paper-Scissors Data Type . . . 19
4.1.2 Creating User Input . . . 20
4.1.3 Generating Computer Move . . . 20
4.1.4 Creating and Playing the Game . . . 21
4.2 What Game Looks Like . . . 22

5 Conclusions **22**

1 Introduction

The goal of this paper is to learn a little more about Haskell and some of the theory behind programming languages. After this we will implement a project in Haskell to showcase some of our knowledge from the semester. We will begin learning about Haskell by comparing and contrasting functional programming to imperative programming and its uses then compare some features/syntax of Haskell to another language that almost everyone is familiar with, Python. We will then talk about some of the history and foundations of computer science, and some of its roots in math. After all of this, we will finish with an implementation of Rock-Paper-Scissors, a classic childhood game, in Haskell.

2 Haskell

2.1 What is Haskell?

Haskell is "an advanced, purely functional programming language"[HASKELL]. Haskell is statically typed, meaning that every expression that has a type is determined at compile time. Similar to other languages, type matching and declaration of types are very important, as the code will be rejected at compile time. This is similar to other programs such as C based languages or Java, where if the right types of values are not used it will not compile. Here are some examples of type declarations.

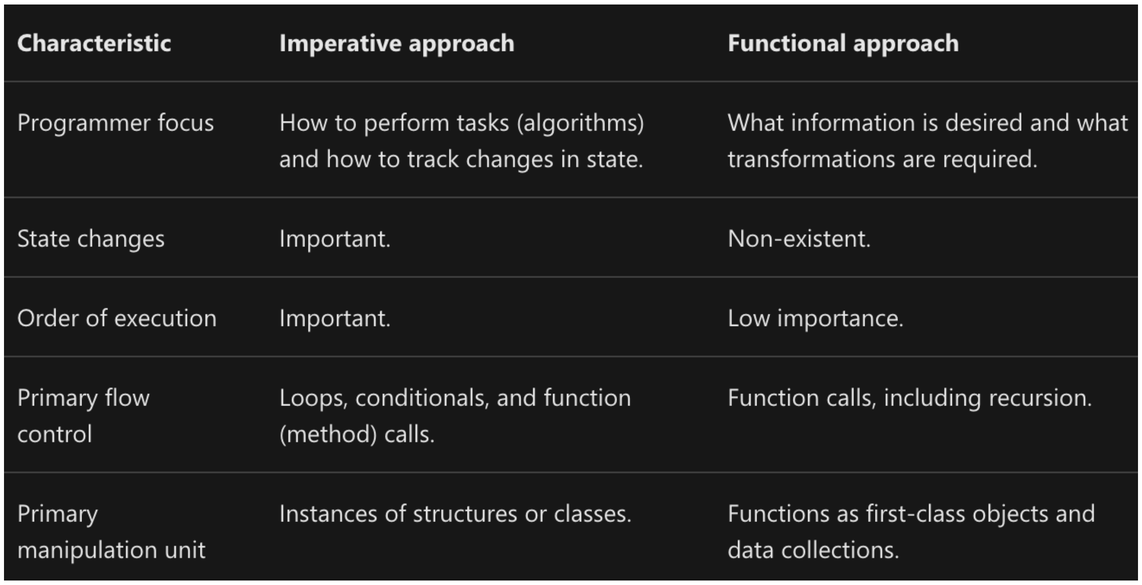
```
-- Basic Type Declarations
char = 'x'    :: Char
int = 22      :: Int
pair = ('a',1) :: (Char, Integer)
```

Haskell also recognizes stings as lists of Chars so to declare strings in Haskell so it must be declared as a list of Chars similar to Java or C/C++. Haskell also provides type synonyms which are used for commonly used types:

```
-- String Declarations
type String      =     [Char]
type Restaurant  =     (restaurantName, restaurantAddress)
type restName    =     String
type restAddress =     String
```

The type "Restaurant" is a type that has two characteristics/properties, restaurantName and restaurantAddress. This allows us to create custom objects or types.

Haskell is a "purely functional programming language". So before we get into too much depth about the advantages of Haskell, we first must understand what a functional programming language is and how it differs from a Imperative Programming Language. A functional programming language is a type of declarative programming that is high-level where the programmer must specify step by step what function the computer will perform. Here is a diagram showing some of the differences between functional and imperative programming styles:



Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

Basically, in a functional programming language, declare the problem and that is the solution.

”Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions” [HASKELL]. Haskell makes it very easy to create and use threads. Threads are useful for programmers when a program has many different tasks that are to be done separately from the others. This is useful if one task is blocked, the program can still complete other tasks using different threads and will not be blocked. Overall, threads and simple implementation is a huge benefit to programmers.

Haskell is considered a ”Lazy” programming languages, meaning ”functions don’t evaluate their arguments”. This means that it is very easy to write control constructs (such as if/else, AND, OR, NOT) and they can be expressed just by writing normal functions.

Because Haskell is an open-source language, it has many contributions and many people have created open-source features and packages that can perform a variety of tasks, and make a programmers life much easier.

2.1.1 Haskell vs Python, A Comparison

What are the main differences between Haskell and a language like Python?

Python offers programmers many different styles, including procedural, functional, and object-oriented programming styles [PvsH] but it is tricky to use and implement functional programming in Python. Haskell is a functional programming language, but can support procedural programming through the use of monads, see more below. Python is a interpreted language (it does not have a compiler). Haskell typically uses a compiler (the one most commonly used that we will learn about is GHC). Haskell and Python are both considered high-level languages, so they compromise on speed compared to C or C++. Through Haskell’s type system and declaration Haskell does provide more information at compile time than when running Python, meaning that Haskell does have a speed advantage over Python. ”Haskell and Python have strong (not weak) typing, meaning instances of a type cannot be cast into another type” [PvsH]. The difference between the two is that Haskell has static typing, and Python has dynamic typing. In statically typed languages, all variables and their types are known when compiled. On the other hand, in dynamically typed languages, the types of variables is not known until run time and ”objects can pretend to be different types by providing the correct functions" [PvsH]. Some other differences include Haskell is considered to have a much larger learning curve for those who are not accustomed to functional programming. To many, lazy evaluation is hard to grasp for someone who has exclusively coded in imperative programming languages.

- Significant Whitespace:
  - Python is a unique language in how it deals with whitespace. In other languages, syntax is delimited by curly braces, or will use begin or end keywords, and will end lines with semicolons. In python, blocks are delimited by white spaces. In Python, white space and indentation must be maintained otherwise errors will be thrown. Only when doing indentation, is whitespace significant. When skipping lines in between code whitespace is not significant.
    In Haskell, whitespace is also significant. Python's is very indentation sensitive, as an aligned block will always start on a new line. In Haskell, there are syntactic constructs that have aligned blocks that allow these blocks to start in the middle of an existing line. [HSYN]
- Types and Type classes in Haskell
  - Intro To Type classes A feature that makes Haskell really powerful is type classes. Type classes allow programmers to define interfaces that can provide the ability to apply a feature or function over different data types. Type classes are the basic functionality of most languages features such as equality testing(such as == and ≠) and numeric operators (such as greater than or lesser to).
  - Examples Of Type Classes
    - Equality Testing "Eq" Eq is to be used for types that support equality testing. for example Ints, Floats, Chars, Strings(arrays of Chars). Eq implements == and $\bar{o}$r equal to and not equal to and will return a boolean value. Shown in the GCI compiler:

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 10 == 8
False
ghci> "Hello World" == "Hello World"
True
```

    - Ord is a type class for types that have an ordering. It is used for all comparing functions such as <,≤,>,≥. It internally uses the 'compare' function, which takes to Ord values of the same type and returns an ordering for the two members. Ordering is to be used on types that can be compared (<,≤,>,≥). It is also possible to use the compare to compare two variables and it will return GT(greater than),LT(less than), or EQ(equal).

```
ghci> 5 <= 6
True
ghci> 5 >= 6
False
ghci> 2 'compare' 1
GT
ghci> "Hello World" 'compare' "Hello World"
EQ
```

    - Show is used to represent variables as a string.

```
ghci> show 5
```

```
"5"
ghci> show 5.12345
"5.12345"
ghci> show False
"False"
```

∗ Read is basically the opposite of the Show type class. It takes a string and returns a type that is a member of Read.

```
ghci> read "5" + 3
8
ghci> read "5.1" + 3
8.1
ghci> read "[1,3,5,7]" ++ [9]
[1,3,5,7,9]
```

However, this works because we are giving Haskell the type of output we expect and it is doing type matching for us. If we were to give it something like this:

```
ghci> read "5"
```

It would throw an error because the GHCI compile is unable to infer what type of result is desired from the Read class. A solution to this is to use type annotations or declarations. Here are some examples of these type annotations[TT]:

```
ghci> read "5" :: Int
5
ghci> read "5.1" :: Float
5.0
ghci> read "[1,3,5,7]" :: [Int]
[1,3,5,7]
ghci> read "('z',26)" :: (Char, Int)
('z',26)
```

In order for Haskell to determine what the type is of ”read ’5’” it would have to evaluate it. But because Haskell is statically typed, it needs to know all types at compile time so this is not possible.

∗ Enum can be used for sequentially ordered types that can be enumerated. It can be used to fill in ranges in list types that have already defined successors and predecesors that can be determined using the ”succ” and ”pred” functions.

```
ghci> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10]
ghci> ['a' .. 'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> succ 'a'
'b'
ghci> pred 'Z'
'Y'
```

- Learning Curve:
  - Haskell is known for having a very steep learning curve, as compared to other languages, including Python. One main reason for this learning curve is that most programmers nowadays are used to

writing in imperative programming languages, like C/C++. This jump into a purely functional programming language can be conceptually hard to grasp as it relies more on mathematical concepts than imperative languages do.

2.2 More About Haskell

2.2.1 Recursion

Recursion is a very important part of programming in Haskell. Recursion is a very important part of all functional programming languages. A common example of how recursion works is computation of a factorial. A factorial is a product of an integer and all the integers below it. Factorial 5 is the equivalent of 5*4*3*2*1 which is equal to 120. Write down the mathematical equation and then the computation is done.

```
factorial 0 = 1
factorial 1 = 1
factorial n = n * factorial (n-1)
```

These are the mathematical rules for our equation. The two base cases are defined for the factorial function with factorial of 0 = 1 and factorial of 1 = 1. Then is the recursive part, where factorial of a number n is equal to n times factorial of n-1. Recursion is an extremely important part of Haskell so here is an example, step by step, of how factorial of 4 is computed recursively:

```
factorial 4 = 4 * factorial (3)
           = 4 * 3 * factorial(2)
           = 4 * 3 * 2 * factorial(1)
           = 4 * 3 * 2 * 1 * factorial(0)
           = 4 * 3 * 2 * 1 * 1
           = 24
```

The equation starts with factorial of 4, and calls factorial of itself until it reduces to 4 * 3 * 2 * 1 * 1, giving a final answer of 24. [FACT]

2.2.2 Haskell for Beginners

- Use recursion instead of loops
  - To calculate the length of a list in Haskell using recursion

    ```
    len[0] = 0
    len(x:xs) = 1 + len(xs) -- where x is the head of the list and xs is the tail
    ```

- Assignment
  - x = 0

    The = operator is not assigning the value 0 to x as it would in C/C++, but rather saying that they are truly equal and both instances can be replaced.[HFA]
  - Write it down as a mathematical equation stating a fact
  - To use assignment in Haskell, this can be accomplished in one of two ways:
    - implicitly via pattern matching
    - explicitly using key words where or let [HasBasics]
- The Keyword Let
  - The let keyword can be used in 3 ways in Haskell.

  – let-expression

    let variable = expression in expression

```
    ghci> let (x = 5 in x*3)
    15
```

    This can be used whenever an expression is allowed.

  – let-statement

    This is only used inside of do-notation, it also does not use in.

```
    do someAction
    let variable = expression
    someAction
```

  – This is similar to the last one, but it is used inside of list comprehensions

```
    ghci> [a,b] | a < [1..3], let b = x*x]
    [(1,1),(2,4),(3,6)]
```

  This function makes the second attribute of the list, b, the square of the first attribute in the list, a. [LET]

- Function Declaration
  - When defining a function, initialize it in a similar way to how it will be called. First, write the function name followed by the parameters, separated by spaces.
  - The types of the variables must also be declared or will receive errors.
  - For example, if there is a function addI that adds integers, it must be declared as:

```
  addI :: Integer -> Integer -> Integer
```

  - This means that in the function addI, The first Integer is being taken, it is then added to the second Integer, and then returns the third value which is also of type Integer.[PL]

- Computation is equational reasoning. This is demonstrated with a length function of a list example:

```
len(x:xs)= 1 + len xs
len[4,-2,3] = 1+len[-2,3]
len[-2,3] = 1 + (1 + len[3])
len[3] = 1+(1+(1 + len[]))
```

  It recursively calls the len() function, adds one and removes the head of the list.

- Using types
  - Declaring return types of functions

```
len:: [x] -> int
```

  - The return type of type list of type int is int

```
Func:: a->b->c->d
```

- In the function ”Func” declared above, (a b and c are input types and last, d is output type). The last variable declared is always the output variable, and the rest are input variables.
- In a C/C++ an addition function of two numbers would look something like this:

```
int add(int x, int y){
return x + y;
}
```

- In comparison, in Haskell, a function to add two numbers would look something like this

```
add:: Int->Int->Int
add x y = x + y
```

- It is/should not declared as add(Int, Int) → Int to specify two Int input variables and returns and Int. It is possible to do that syntax and it won’t fail but there are many benefits to using arrow notation.
- For example, if we want to implement a function that takes an Int and adds 3 to it. We can use the already declared add function and it would look something like this.

```
plusThree = add 3 :: Int -> Int
```

- This function takes one Int input, then calls add and passes 3 to that the addition function as long and returns an Int

• Make use of error messages

- In C or C++ one would call the function add() and pass the parameters 2 and 3 as: add(2,3) to pass arguments and get the answer 5.
- In Haskell to call it the syntax would be add 2 3. Haskell is very strict on syntax so trying anything else would fail. If you tried add(2 3) it would fail and the error message would be ”maybe you have not applied the function to enough arguments”. Everything inside the parenthesis is interpreted as one argument so the function is only registering one argument.
- Even if you try add(2,3) It will say ”Couldn’t match expected type Int with actual type (Int,Int)” This is because the expected type is Int → Int → Int, but the actual type of the first argument is (Int, Int). This is Haskell doing doing type inference.
  - ∗ 2::Int 3::Int
  - ∗ (2,3) :: (Int,Int)

2.2.3 Higher Order Functions Python vs Haskell

An important tool in programming is the implementation and use of higher order functions. Higher order functions are functions that either take other functions as parameters or return functions as parameters. Python and Haskell both support higher order functions, and they help us do many things.

2.2.3.1 Passing Function as Argument in Python In Python, a simple example of a higher order function is one that will take a string that has both capitals and lower cases, and then taking a function as a parameter, either make the sentence capital or lowercase.

```python
sentence = "This is My BAsic SentEnCE WIth Both CapitAls and lowER CAsE"

def makeUpper(words):
    return words.upper() #make words upper case

def makeLower(words):
    return words.lower() #make words lower case

def speak(myFunction): #passing a function into speak

    greeting = myFunction(sentence) #call the function passed through that prints
    print(greeting)

speak(makeUpper) #call speak function and pass makeUpper to make it upper case
speak(makeLower) #call speak function and pass makeUpper to make it lower case
```

This gives an output of:

```
THIS IS MY BASIC SENTENCE WITH BOTH CAPITALS AND LOWER CASE
this is my basic sentence with both capitals and lower case
```

We are defining three functions, the fist one, makeUpper, takes a string as an input and converts the entire string to capital letters. The second function, makeLower, also takes a string as an input and converts it all to lower case. A predefined string is at the top of the example, it contains both capitals and lower cases to better show how the function works. The third function defined, "speak", takes a function as an input, then creates an instance of this function, "greeting", with the sentence argument passed as a parameter. It then prints the instance of the function, which for all purposes will print out the given string in either upper case or lower case, whichever function is passed as an argument later. We then call speak() and pass makeUpper and makeLower as parameters, which will perform their respective operations on the strings within the speak() function.

2.2.3.2 Returning Function as Argument Functions are objects, so it is also possible to return functions from another function. In this example bellow, two functions are declared and defined and one function will return the other. Our function names are one and two.

```python
# define two methods

# second function will be returned by first function
def two():
    print("Hello from Inside function 2!")

# first method that return second method
def one():
    print("Hello from Inside function 1!") #print statement will be done first.

    # return second method
    return two

# make an object of first method to return the second function that will call second function
     inside
make = one()

# call second method by first method
make()
```

In the example above, the first function declared is function "two" which takes no parameters and prints a sentence saying where it is, that it is inside function two. The next function, "one" takes no parameters and prints its location, that it is in function one, then returns function 2. So when function one is being called, this is also calling function two in the return statement. We then create an object called "make" which is the return value of function one, which is function two. When make is called, it will call function one, print out the location of function one, then call function 2. This gives us our output in the expected order, and demonstrates how to return a function within another function. [PHOF] To compare and contrast the same example in both Python and Haskell, we will implement the map function, which is used to apply a function or argument over a list of strings.

```
    integers = range(0, 10) #create list of integers 1-10
    list(map(lambda x: x + 2, integers)) #use map function to apply argument to list
```

2.2.3.3 Higher Order Functions in Haskell.

Higher order functions are a staple of Haskell, and are unavoidable(but why would you ever want to avoid them?). This is because, in Haskell, computations are not being defined by what something is(often through recursion), rather than continuously looping through something to change the value.
The example we will be looking at in Haskell is the map function called mapList below, that takes a list and a function and applies the function to every element in the list. We also have a function called plusTwo, which takes an Integer x, and adds 2 it, and returns an Integer.

```
mapList :: (a -> b) -> [a] -> [b]
mapList f [] = []
mapList f (x:xs) = f x : mapList f xs

plusTwo :: Int -> Int
plusTwo x = x + 2
```

So in order to make this a higher order function, we will need to call plusTwo within mapList, to sure plusTwo is done for every element in the list. This would look something like this:

```
ghci> mapList(plusTwo) [1..10] --type class enum to fill all numbers between 1 and 10
[3,4,5,6,7,8,9,10,11,12] --each number has 2 added to it
ghci> mapList(plusTwo) [10,20,30]
[12,22,32]
```

We implement higher order functions by calling plusTwo inside of our mapList function, so that plusTwo will be done for every value of the list given. In the first example, 2 is added to every number from 1-10 so our range is no longer 1-10 but 3-12. [HHOF]

2.2.3.4 Summary of Higher Order Functions

There are many different benefits of using higher order functions, as they allow programmers to simplify and shorten their code, while also making it more robust and reusable. Higher order functions allow programmers the ability to create new versions of something without necessarily writing all the code again.

2.3 Summary

To someone who has never used functional programming language, Haskell can be pretty daunting. After the course of a semester, and doing this project, did it finally get easier. There are many great uses for Haskell and a lot of advantages to a programmer. If coming from an imperative language, don't give up! If you are still struggling to get it, try reviewing Lambda Calculus, and recursion, these are very important parts of Haskell.

3 Programming Languages Theory

3.1 Brief Overview and History

3.1.1 Pre Programming Languages

A lot of the theory and foundations for theoretical computer science is attributed to one man, Alonzo Church. He was an American mathematician who developed the theory of lambda calculus, which will be talked more about below. He also worked with Alan Turing, who is also considered to be "a founder of computer science" when Turing was a PhD student under Church at Princeton University. Together, they developed and wrote the Church-Turing thesis, which sought to prove that a mathematical process was effective if presented as a list to be followed instruction by instruction. This was the foundation for the first algorithm. The idea was that any calculation done by hand was also possible to do on a computer, with sufficient resources. Turing also created the idea of Turing machines, which is the theoretical foundation and created the principles for which modern computers are founded as well as the concept of stored programs. [C-T]

3.1.2 Foundation of Programming Languages and Foundation of Haskell

The early development of programming languages occurred in the 1950's. The first ever commercially available programming language was FORTRAN (FORmula TRANslation) which was developed in 1956. Another early high-level programming language developed in the 1950's was LISP (LISt Processing) which had a functional style. It allowed for user functions to be defined, and passed around as values [HIS]. Throughout the 1980's there were a variety of different functional programming languages being developed and released. The research and resources were split across all the different programming languages and many of them were not open source so there was not much collaboration and discussion among academics and other people who were using these languages. A group of academics saw this as a problem, and teamed up to "implement a new language, which would be used as a vehicle for research as well as for teaching functional programming"[HIS]. This group of academics published the "Haskell Language Report" which they released in 1990. This was very important because it established a functional language that could be used by the entire research community. This allowed for quicker and faster development of the language as well as functional programming languages as a whole.

3.1.3 Haskell Today

Since Haskell's release in 1990, it has grown greatly in popularity. This is in thanks to the development of the creation of the Glasgow Haskell Compiler that has an interpreter ghci which also has a compiler, ghc. This makes it an easy, open source tool that is used in a variety of fields, not limited to academic research, teaching, and industry. We are taught about it here at Chapman University, in our Programming Languages(CPSC 354) class, as well as in Compiler Construction(CPSC402). Many companies also use Haskell today, for example AT&T, who uses Haskell in their network security division to automate the processing of internet abuse complaints. Facebook also uses Haskell to manipulate a PHP code base. Intel has developed a Haskell compiler to aid in their research on multicore parallelism at scale [HASKTOD]. Microsoft has also been a "key sponsor" in the research and development of Haskell since the late 1990's.

3.2 Lambda Calculus

3.2.1 Intro to Lambda Calculus

Lambda calculus is a mathematical concept that forms the fundamental bases for all functional programming languages. It is a type of notation for functions and applications. It is the basis for applying a function to different arguments. Lambda Calculus has three programming constructs, variables, abstraction, and application. We will look at how these three constructs are applied if want to write a function in lambda calculus to square a number. The function in lambda calculus would look like.

$$(\lambda x.x * x)$$

This function designated by the lambda character takes an input x. Then to the right of the period, it says that the function will compute x * x, which is x squared. [PL]

- Variables:
  The basic functions are just the variables. In this example our variable is x.

- Abstraction:
  Abstraction is the ability to make functions. It is a concept in to show that that an expression does not depend on the specific variables. This is shown in most basic terms as:

  $$\lambda x.e$$

  The function, which has as a formal parameter x. Which will be applied in some computation e. This computation e is not specific to just our squaring function, but can be replaced with any expression.

  $$\lambda x.e$$

  does not depend on x anymore, x is ”abstracted” away.[PL]

- Application: There is no symbol for application. This is how a function is applied to its argument(s). Put the two lambda expressions next to each other:

  $$e1e2$$

  Where the function e1 is applied to the function e2.

Lambda calculus provides a much simpler framework and notation for functions that are applied in all different aspects of computer science and are the foundation for programming languages.

3.2.2 Comparing Lambda Calculus to C/C++

In C/C++ to write a simple program to add two to a number, it would look something like this:

```
int plusTwo(int x) {
return x+2;
}
```

This basic program is of type int, takes an int as an argument x, then returns x+2. There are a couple differences when implementing in C vs in Haskell.
In Haskell nothing is being returned, and types are declared beforehand, so there is no need for a return statement.
With functions having no names, this would be written as plusTwo(x) x+2, implemented in lambda calculus as:

$$\lambda x.x + 2$$

The function takes an input x, and has an action done on it (x+2). There is no return value, but this function still has a parameter. To call the lambda calculus and C/C++ function:

```
plusTwo(5); -- how to call function in C/C++

plusTwo 5 -- how to call in Lambda Calculus
(\x. x+2) 5 --what is going on behind the hood
```

3.3 Intro To Parsing

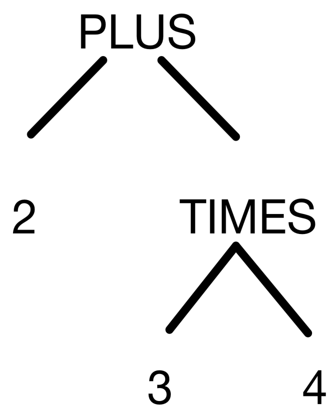
In computer science and programming parsing is the process of analyzing and interpreting characters or strings of symbols into a grammar that the programming language will recognize. Simply, it is the process of translating at type of data, into another type of data. To better understand parsing and context-free grammar, we will first look to understand the difference between concrete and abstract free syntax. We will show this by looking at the implementation of a calculator. [PL]
For example consider the equation:

```
2 + 3 * 4
```

Knowing what we know about mathematics and the grammatical rules of PEMDAS (Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction) that we all learned in elementary school, we know that we will do the multiplication first. But to a computer, it does not know any of this, and we have to define that syntax in the interpreter for the computer. Parsing is about telling the computer where to put the parenthesis, and what equations to do in what order, so it is turned into:

```
2+(3*4)
```

We know that we will compute (3*4) first which evaluates to 12. Then add 2 to it giving us a final answer of 14. In the eyes of a parser in a computer, it will be evaluated in what is called an Abstract Syntax Tree which is drawn as shown:



This abstract syntax tree will then be evaluated to a single value using "context-free grammar"[PL], which is "a set of rules" to define a language and set the rules for what will be interpreted so that it can be parsed into an abstract syntax tree. This is the basis of all programming languages, so that it can interpret strings that the programmer writes and make the right computation based off of it.
We can see examples of this context-free grammar as shown to us in our class notes[PL]:

```
Exp -> Exp '+' Exp1
Exp -> Exp1
Exp1 -> Exp1 '*' Exp2
Exp1 -> Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
```

The "strings" that we want to evaluate in this case is more specifically to define arithmetic expressions.

A string is valid in in the language if it can be derived from the ”start symbol” which in this example is Exp. We need the basic version of our interpreter to recognize addition(+), multiplication(*) and parenthesis(’(’ ’)’) in order to perform the designated operations. The expressions ’Exp’, ’Exp 1’, and ’Exp 2’ control which strings can be derived [PL].
Right now our interpreter cannot do subtraction, or any operation that does not involve Integers. If we were to update the grammar syntax for subtraction we would add something like this.

```
Exp1 -> Exp1 "-" Exp2
```

One issue is that we have not made any definitions for integers. We might want to do that by declaring:

```
Exp1 -> Integer
Exp2 -> Integer
```

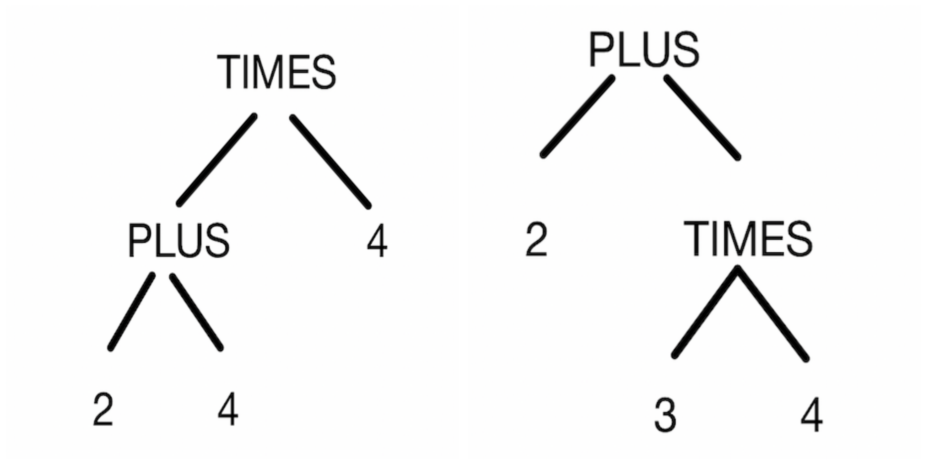
That means that this syntax will only be valid for Integers and not for floats or anything else like that. The cool thing about defining our own syntax is that we can choose any variable or character to represent an operation that we like, we just will implement it in interpreter differently. For example, the normal symbol for squaring or powering is ; but we can make that any symbol that we desire. So lets say we want to only recognize the dollar sign as our form of power, it would look something like this

```
Exp1 -> Exp1 "@" Exp2
```

Then we will define this in our interpreter to recognize it as to the power of. This is how all programming languages work, and allow the assignment of strings to the desired operations.
Now that we have defined our rules for how to interpret syntax, we can now look at how this concrete syntax will impact our abstract syntax tree.

3.4 Ambiguous vs Non Ambiguous Grammar

Ambiguous grammar is a type of parse tree that can be derived in many different ways and will give the same result. An example of ambiguous grammar is the first string we looked at 2+3*4. Here are two different abstract syntax trees for the different ways to parse this equation:



We know that 3 and 4 will be multiplied but it may be multiplied before or after being added 2. However, if we were add parenthesis to the string, and make it 2+(3*4), there becomes only one way to derive this string making it non-ambiguous. Non-ambiguous grammar is grammar/syntax that can only have a single way to draw a parse tree and arrive at the correct value. Adding parenthesis or defining grammar is how to make grammar non-ambiguous.

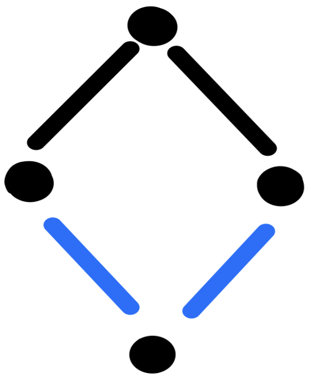
3.5 String Rewriting

String rewriting is a process in which "computation proceeds by rewriting expressions"[PL]. Rewriting works because there is predetermined set of rules that can be applied to a predetermined expression using pattern matching and string rewriting.

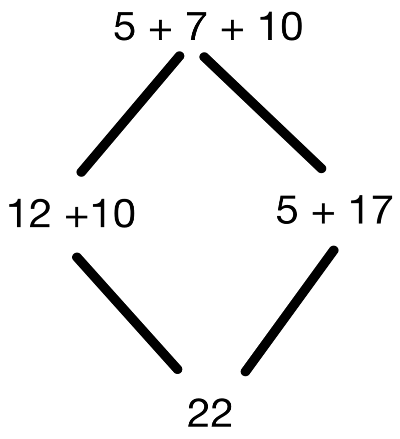
An "Abstract Reduction System" follows a certain set of rules. an ARS is the transformation of objects when certain rules are applied to them. When analyzing ARS's the two most important things to look for in an ARS is if it is confluent, if it is terminating and if it is in (unique) normal form.

3.5.1 Confluence

The first set of rule that we will be looking at is determining if an ARS is confluent or not. When determining if an ARS is confluent if for every peak there is is a valley, meaning that if they diverge they return to its original node. In the example bellow, the ARS is confluent if for every black "mountain" there is a blue "valley"



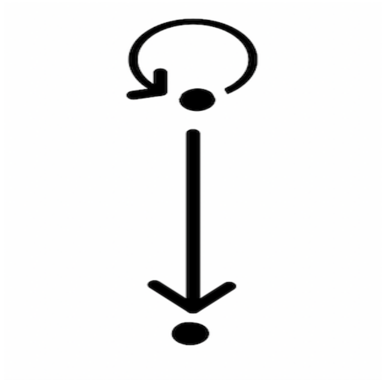
For example, arithmetic is confluent due to the commutative property of addition, which defines that when two or more numbers are added, their sum will be the same regardless of the order they are added in.



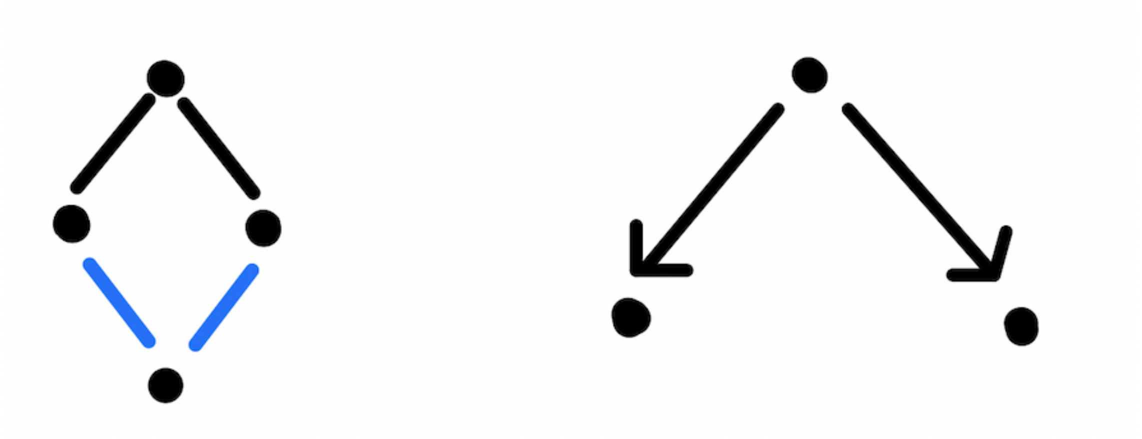
In this example, the arithmetic equation 5 + 7 + 10 is confluent because addition is commutative, and adding (5+7)+10 or 5+(7+10) will give the same answer of 22. There is one ”mountain” for every ”valley”.

3.5.2 Terminating

An ARS is considered terminating if it does not have an infinite computation. An example of an ARS that is non terminating and has an infinite computation:

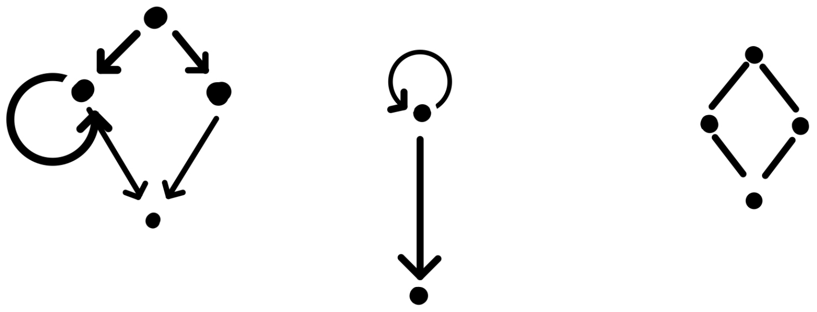


This ARS is non-terminating because the first node is in infinite computation. Some examples of ARS systems that are terminating are:



3.5.3 Unique Normal Forms

An ARS is considered normal form if the ARS ends at a singular node. An ARS is considered to have unique normal form if all nodes and elements has unique normal forms. Some examples of ARS that have unique normal forms include:



3.5.4 Example

To combine these different string rewriting techniques shown above we will do an example of an ARS with rules and determine how it normalizes (reduces further where no more rules can be applied). Here are the rules for our ARS:

```
ab->a
ba->ab
```

These rules state that ab reduces to a, and ba can be rewritten as ab. Now we will take our system babbaabbaaab and perform some steps to it. To help make the example a little more understandable, comments are included on the right side showing what action was taken in the step above to reduce to the current form in that line. For example, in line 2 of this example, the comments show what action was taken on the line above to arrive at the current state.

```
babbaabbaaab
babaabbaaab ab-> a (instance of ab at position 1 and 2, 0 based index)
baaabbaaab ab -> a (instance of ab at position 1 and 2)
abaabbaaab ba -> ab (instance of ba at position 0 and 1)
aaabbaaab ab -> ab (instance of ab at position 0 and 1)
aaababaab ba -> ab (instance of ba at position 2 and 3)
aaaabaab ab -> a (instance of ab at position 2 and 3)
aaaaaab ab -> a (instance of ab at position 3 and 4)
aaaaaa ab -> a (instance of ab at position 5 and 6)
aaaaaa Is the final answer in normal form.
```

The normal form, where no more rules can be applied, is aaaaaa or 6a. We would also say that this ARS is confluent because it does not have an infinite loop. It also follows all rules of confluence because it diverges if separated and returns to a . It is also terminating because there are no infinite loops because ab reduces to a.

3.6 Hoare Logic

3.6.1 Introduction and History

Hoare logic is a system that has different logical rules for determining and quantifying the correctness of computer programs. Hoare logic was created and developed in the 1960's predating the creation of Haskell, which was first released in 1990. Hoare logic is still used today both in academia and in real software systems. Per the class definition, Hoare logic seeks to "provide a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic and, therefore, the possibility of delegating such reasoning to a compiler or other verification software tools" [PL]. Hoare logic seeks to define and determine the correctness of different computer algorithms using math. This is with the intent to give these rules and calculations to a compiler or other other sorts of software verification tools. Used to formally test correctness of algorithms rather than just a bunch of test cases.

3.6.2 Implementation and Usage

Before when proving properties of functional programs, would do it via induction which is functions defined recursively. Hoare logic has only while loops, assignment, and conditionals. Each of these 3 logical rules will be talked about and demonstrated through examples.

Now doing it for imperative programming, which is a language like C/C++ with lines in order of instructions. While some stuff is true do some stuff. The goal is to also prove some properties of these imperative programs using logic, not a bunch of different test cases. To do this we use Hoare logic. The language/syntax of Hoare logic is called "Hoare triples" because it has 3 parts a precondition P, some code S, and a post condition Q where P and Q are boolean statements. [HOARE]

```
{P} S {Q}
```

If Hoare logic is valid, this means that if P is true, some code S will be executed and terminate in some final state. And when executed it guarantees that Q is true after. If it is not valid, S will be false.

3.6.2.1 While Loops

To see a basic example of this, we will look at a program that is decrementing itself by 1 until it gets to 0, where it will terminate and become false. [HOARE]. In an imperative programming language like C++, this while loop/ function would look like this:

```
while(x>=0){
x=x-1;
}
```

This while loops states that while x is greater than or equal to 0, decrement the value of X by 1, until the value of x is 0, when the condition will break. To use assignment of a variable in Hoare logic, assignment is designated by the operator ':='. So to set the variable x equal to five it would be written as x:=5. This is different but the same idea when viewing it as Hoare triple:

```
{X>=0} while (x!=0) do x:=x-1 {X=0}
```

Fitting the syntax P S Q, the precondition P is X $\geq$ 0. The snippet of code S is while (X!=0) do x:=x-1 (decrementing the value of x by 1. The post condition Q is X=0 meaning if the equation and the code is satisfied and correct, the value of X will be 0. If P is True, following code is executed and it halts, then Q is guaranteed after.

3.6.2.2 Conditionals

Another example to look at would be to to evaluate conditionals, such as basic if then else then logic but in a Hoares triple.

```
{True} if (x=y) then x:=x-1 else z:=0 {x!=y}
```

In this example we look at a conditional statement. If the precondition is True, if the variable x is equal to the variable y, then decrement x by one. If x and y are not equal to each other, set the variable z equal to 0. The post condition is x is not equal to y. If x is not equal to y before running the code, then it would never enter the code block, meaning that the post condition that x $\neq$ y is satisfied because they were never equal to each other even before the code block. If x is equal to y before the code block, then it will enter and x will be decremented by 1, while y remains unchanged. This will guarantee that X and Y are no longer equal to each other, satisfying the post condition that x $\neq$ y. [HLogic]

3.6.2.3 Assignment

Assignment works if the precondition has some expression or function that is True. If this function is true, then some code will be executed and the value of one variable be replaced with another. Assignment is just the replacement of different variables with

```
{F(n)} x:=n {F(x)}
{X+10 = 15} Y := X+10 {Y = 15}
```

For this example of assignment, we take a precondition X+10 = 15. This is quite simple algebra so when we subtract 10 from 15 to isolate x, we get that X = 5. In the code chunk the value 15 is replaced with Y which is assigned to the equation X+10. To satisfy the precondition, it is already known that X is 5, because it has already done that computation, so in the code chunk it assigns 5 to the X value and then the Y value is calculated to be 15. This then satisfies the post condition of Y = 15, giving a valid Hoare triple. [HEX]

This is how Hoare Logic is used to formally determine the correctness of a program. Test cases don't determine if a program is correct for all test cases, just the specific ones that were tested.

4 Project

The project I will be doing for this class is the simple game of rock-paper-scissors. This is a good project because it taught me to use Haskell in a different way from how we have learned it in class. My game allows a user to play rock-paper-scissors against the computer who will randomly generate rock paper or scissors then compare the results to user input and determine a winner. [RPS] [RPSHASK]

4.1 Rock-Paper-Scissors

I will use the System.Random package when generating a random move for the computer. In order to do this the random package needs to be imported:

```
import System.Random (randomRIO)
```

4.1.1 Defining a Rock-Paper-Scissors Data Type

To start this project, I first created a data type called "Move" that can be one of three options in the game, either Rock, Paper, or Scissors.

```haskell
data Move --create the different moves
  = Rock
  | Paper
  | Scissors
  deriving (Show, Eq)
```

Our data type move is one of the three possible moves in rock-paper-scissors using the — operator to designate or. This is followed by a deriving (Show, Eq) called the deriving clause. It lets us tell the compiler that we want to be able to automatically create instances of our "Move" data type [DER].

- Show is a type class that "takes any value of an appropriate type and returns its representation as a character string" [HTYP].
- Eq is a type class for equality that is used to compare and to show that it is a member of some type class.

This will allow us to create instances of Moves with ease later on in our game.

Because we now have a data type Move we can create a function that will return what move beats what to later use when comparing two moves, user move and computer generated move.

```haskell
beats :: Move -> Move -> Bool --define what beats what and return a bool
beats Paper Rock = True --Paper beats rock returns true
beats Scissors Paper = True --scissor beats paper
beats Rock Scissors = True --rock beats scissor
beats _ _ = False
```

This function takes in two moves, and returns a boolean. It returns True, meaning that the first move beats the second move if the first move is paper and the second is rock. Or if the first move is scissors and the second is paper, or finally if the first move is rock and the second is scissors. This is all the logical part of the game and will allow us to pass user move and computer moves into this function to help determine a winner.

4.1.2 Creating User Input

We create a function called getUserInput which is of type IO Move, which allows the user to input a string and if it matches one of the strings defined below, it will set it equal to one of our Move data types.

```haskell
getUserInput :: IO Move
getUserInput = rockpaperscissors <$> getLine --getLine function gets user input
  where
    rockpaperscissors "scissors" = Scissors --all inputs are case sensitive
    rockpaperscissors "rock" = Rock
    rockpaperscissors "paper" = Paper
    rockpaperscissors _ = error "invalid input please try again"
```

getLine reads a line from the standard input device. All of the inputs are case sensitive, so if it does not match one of those strings char for char, it will use the last statement, and throw an error and crash the program, with the error message ”invalid input please try again”. Storing the IO as a move will allow us to compare this value to the computer type Move when we randomly generate it next.

4.1.3 Generating Computer Move

We define a function, dorps, which takes 3 Ints as parameters then generates an IO Move. It uses the rand class to

```haskell
dorps :: (Int, Int, Int) -> IO Move --how to get create computers move
dorps (r, p, s) = rps <$> rand --generate random input to return rock paper or scissors
  where
    rps x
      | x == s = Rock
      | x == s + r = Paper
      | otherwise = Scissors
    rand = randomRIO (1, r + p + s) :: IO Int
```

the randomRIO function takes in this case, is of the type IO Int, and will randomly generate a number between ghe first argument and the second argument within its parenthesis [RAND]. For example, if the function was rand = randomRIO (1,10) ::IO Int rand would be a random Integer between 1 and 10. In our game, the random value will be between 1 and the sum of r p and s (which we will all set to 1, so will be a random number between 1 and three). We will use r, p and s later when creating our game to be able to generate either rock paper or scissors. We will make the second parameter equal to 3 later in the game, so the values possible will be 1, 2 and 3. If x is equal to 1, it will be rock, if x is equal to 2 it will be Paper, if anything else (x is equal to 3) it will be scissors.

4.1.4 Creating and Playing the Game

Finally, we have our game. Our game is a function that accepts 3 Int inputs that are used to determine the computers move as explained above. It prints a line that will prompt the user: ”rock paper or scissors” then calls the getUserInput function to get the input. If the user does not enter ”rock”, ”paper”, or ”scissors” exactly in lower case, the program will exit. It then shows the user what they entered. After that the program generates the computer’s move, using the arguments passed into game to call the dorps function, which creates a move for the computer.

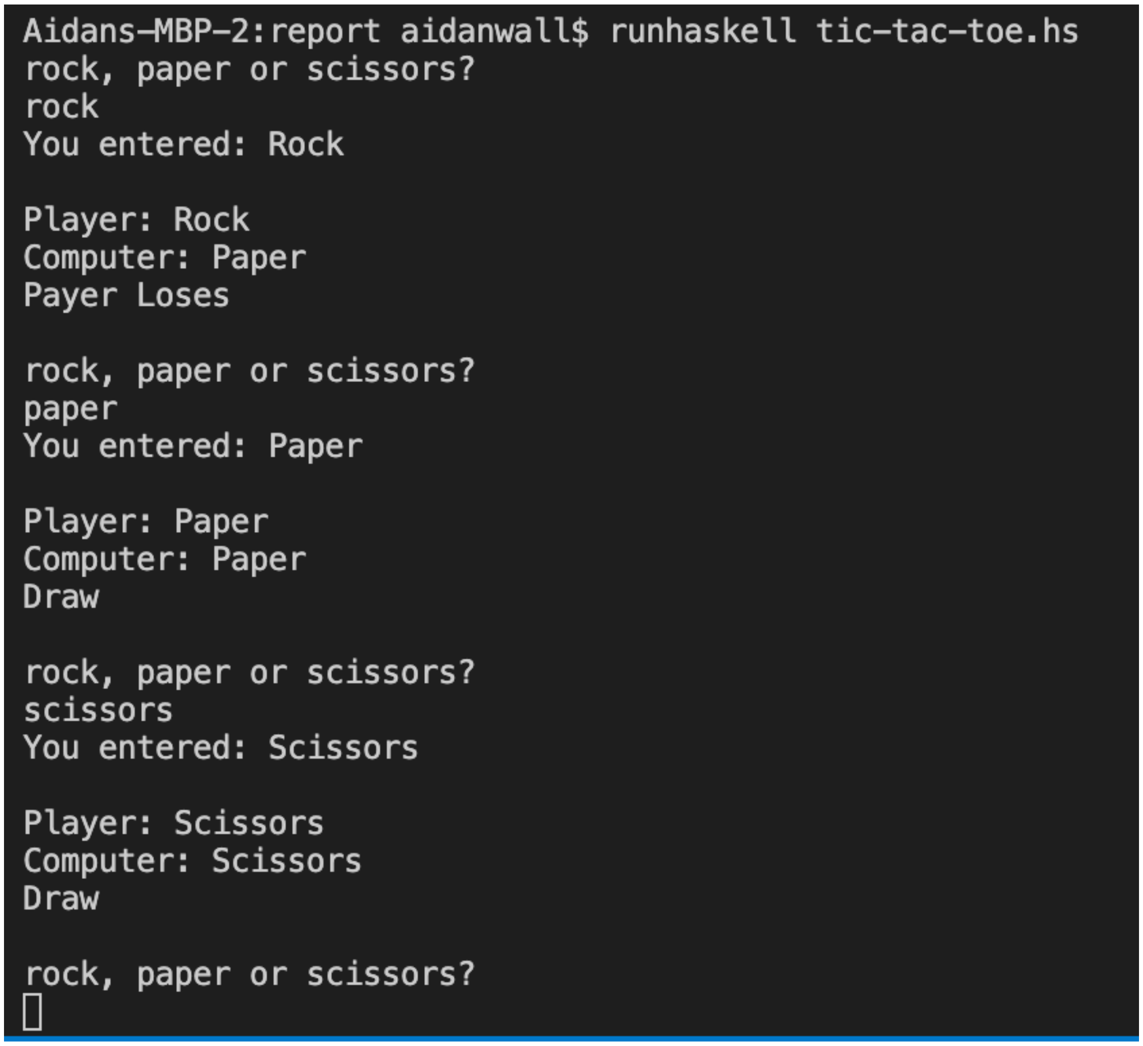
```
game :: (Int, Int, Int) -> IO a
game (rock, paper, scissor) = do
  putStrLn "rock, paper or scissors?"
  userInp <- getUserInput

  putStrLn ("You entered: " ++show userInp)
  compInp <- dorps (rock, paper, scissor)
  putStrLn ("")
  putStrLn ("Player: " ++ show userInp ++ "\nComputer: " ++ show compInp)
  putStrLn
```

The players move and the computers move are then printed to the terminal to show the user what moves were made. Then the two moves are compared using the beats function we previously defined. If user move beats the compute move, the program prints to the terminal ”Player Wins”. If the computer move beats the player move, it prints ”Player Loses”. If it is neither of these options, it means the player move and the computer move are the same, so they draw and ”Draw” is printed to the terminal. For the last part of our game, we define and design how our game function will be called and use the inputs over and over recursively so that we can play the game many times and the computer value will not stay the same and continuously update. If the user input is rock, it will increase the rock value will increase, if it is paper, then it will increase the value of paper, and the same for scissors. It will then call the game function with these updated values so we will get a new computer input, allowing the user to play as many times as they desire, without repeating the same computer input value.

```
    (if beats userInp compInp
       then do "Player Wins\n"
       else if beats compInp userInp
              then "Payer Loses\n"
              else "Draw\n")
  let rr =
        if userInp == Rock
          then rock + 1
          else rock
      pp =
        if userInp == Paper
          then paper + 1
          else paper
      ss =
        if userInp == Scissors
          then scissor + 1
          else scissor
  game (rr, pp, ss)
```

4.2 What Game Looks Like

This is what the game looks like when running it in terminal. To quit the game the user just has to give an invalid input move and the program will exit.



```
Aidans-MBP-2:report aidanwall$ runhaskell tic-tac-toe.hs
rock, paper or scissors?
rock
You entered: Rock

Player: Rock
Computer: Paper
Payer Loses

rock, paper or scissors?
paper
You entered: Paper

Player: Paper
Computer: Paper
Draw

rock, paper or scissors?
scissors
You entered: Scissors

Player: Scissors
Computer: Scissors
Draw

rock, paper or scissors?
```

Figure 1: Game Interface

We now have a functioning Rock-Paper-Scissors Game that allows a user to play against a randomly generated value from a computer. This project helped me to further my understanding of Haskell and functional programming languages. I learned about how to use IO input from the user, as well as different type classes and pattern matching. The first part of the project does not feel complete without actually doing some coding. It is hard to just talk about it without actually doing it. I'm glad I learned a lot about Haskell and successfully implemented Rock-Paper-Scissors in Haskell.

5 Conclusions

Overall, I have enjoyed taking this class this semester because I got to learn a new programming language, Haskell, but also begin to understand how other programming languages, and compilers/interpreters are created. I never understood how syntax was declared and how the foundations work to create functions and programs, and how much of this is based in math such as lambda calculus. I did not understand how programming languages are just extensions of different mathematical concepts and procedures but just expediting these procedures with the help of a computer.

References

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

[COMP] Functional vs Imperative Programming, 2021

[TT] Type Classes Haskell

[HASKELL] HASKELL Documentation, 2021

[HasBasics] Haskell: Basics

[HIS] Brief History of Haskell, 2021

[HTYP] Type Classes in Haskell

[FACT] What is a Factorial

[RPSHASK] RPS For Beginners in Haskell

[HFA] Haskell for All

[PvsH] Comparing Python and Haskell

[HLogic] Hoare Logic Video

[HEX] Hoare Logic Example

[HASKTOD] Haskell in Industry

[HHOF] Haskell Higher Order Functions

[LET] Using Let in Haskell

[Functor] Functors in Haskell Video

[HOARE] Hoare Logic

[C-T] Church-Turing Thesus

[IOMONAD] IO Monads

[EHandling] Error Handline in Haskell

[IOError] Handling IO Error

[DER] Understanding Deriving in Haskell

[PHOF] Higher Order Functions Python

[RPS] Rock-Paper-Scissors

[DB] Debugging IO Error

[RAND] Random In Haskell

[HSYN] Whitespace in Haskell