CPSC-354 Report

Jesse Ruhl
Chapman University

December 21, 2021

Abstract

In this report I will be going over information pertaining to the programming language known as Haskell. I will be describing the background of Haskell, the history of it, and why one would choose to use it (or learn it) over a traditional programming language such as Python. In order to help the reader understand the importance of Haskell, I will be walking them through a simple tutorial of Haskell to show how powerful of a tool it is, while also being easy to understand once one learns the basics. Once the tutorial has been completed, I will go more in depth about some of the unique properties that are included in Haskell, and how these properties could be seen in the tutorial that was completed earlier.

Contents

1 Haskell

Within recent years, there has been an increase in people pursuing programming and now is a great time as any due to the vast number of available programming languages to learn. Of the vast number of programming languages, some of the most popular languages include Python, Java, JavaScript, and C. These languages are all considered to be based on imperative programming, meaning that the language uses statements that can and will change a program's state, as well as it focuses on describing how a program operates. This aspect of imperative programming brings some challenges to those who use languages based on such a design. One potential problem with imperative programming is that the code can quickly grow for problems that are more complex and need to be solved. This is where another type of programming, called functional programming, may arise to be a better alternative to imperative programming. Functional programming is a type of programming that is based on the use of mathematical functions and it is designed to handle both symbolic computations and list processing applications. One of the most popular languages that is based on this functional programming, is the language known as Haskell.

1.1 History of Haskell

Haskell was first introduced in 1987 with the Haskell 1.0 release. The language was created by multiple contributors including Philip Wadler, Stephen Blott, Simon Peyton Jones, and Warren Burton. In 1997, an updated version of Haskell (Haskell 98) was released in order to "specify a stable, minimal, portable version of the language and an accompanying standard library for teaching, and as a base for future extensions" [1]. The language was later revised and improved in the 2010 version, Haskell 2010. As the program language has grown, there has been an increase in implementations of it, with the main implementation being the Glasgow Haskell Compiler. The Glasgow Haskell Compiler has two parts to it, that being the interpreter (ghci) and the compiler (ghc). It will be shown how one is able to access and use the interpreter to run Haskell files in the tutorial section.

1.2 Haskell as a Functional Language

As stated earlier, Haskell is a purely functional language, which means that all of the functions within Haskell (both included and user made) are functions in the mathematical sense [2] [Haskell.org]. What this entails is that within Haskell there are not any statements or instructions to be run, there are only expressions which cannot mutate variables nor access state like time or random numbers[2] [Haskell.org]. To put this in terms of programming, it means that once a variable is given to a value, that value cannot be changed until the program is run again when the user changes that value. This is different from languages (such as C, Python, Java, etc.) that are not purely functional and their variables are mutable. In these programming languages, the value assigned to the variables can be changed within the function while the program is being executed. This way of handling variables can lead to problems however, such as a variable with a string value later having a float value (as seen in Python) or a problem with the memory of the variable. This is where the functional program aspect has advantages over non-functional programs, in which the stated problems are less likely to occur and the functions within Haskell will have no side-effects. Haskell being a purely functional program language is not the only main aspect of what makes it stand out from other programming languages, another aspect of Haskell is how the language is statically typed.

1.3 Haskell as a Statically Typed Language

Again, like imperative and functional programming, there is also a difference between programming languages being dynamically typed and statically typed. Dynamically typed languages are a certain type of language in which the type is associated with run-time values instead of variables or fields. What this means is that the programmer does not need to write (or specify) the type that the variable they are creating. This form of typing is most common in Perl, Python, JavaScript, and PHP. When compared to dynamically typed languages, statically typed languages require the user to state, or declare, the type of variable/value they want to create and work with. This means that the type of a variable is known at compile time. Furthermore, since the type is known at the time of compilation, all of the variables types that were put together by function application must match up in order to run the program run [2] [Haskell.org]. If the variable types do not match each other then the written program will be rejected by the compiler and not run, which in turn it means that a lot of errors that might occur on run will be found at the compile time.

This feature helps ensure that the variables and values do not get their type changed, which could lead to errors say for example a float turning into a string later on in a function where it is needed to be added to another float. Being both a statically typed and purely functional language helps to ensure that there are less errors on compile time in programs made through Haskell. However, this does not mean that Haskell is an easy language to learn. One could make the argument that it is too difficult and that it is not suited for a beginner at programming. To combat this argument, this report will go in detail of how to best learn Haskell through a simple tutorial, and how the steps there can be put together to create a small project. Another important idea to know, even before learning how to write programs in Haskell, is why one would choose to learn Haskell to begin with.

1.4 Why Learn Haskell?

As stated, Haskell can be a difficult language to learn, while also being less prone to errors and keeping functions more secure. So why would one want to learn Haskell? Like many other programming languages, there are many specified uses or scenarios in which it is an optimal choice of language. Currently, Haskell is a prime choice of language to teach someone who is interested in learning how to write programs in functional programming languages. Therefore it is currently a popular choice for those in research or academia who want to create a mathematical based program that is secure and stable to run. Because of this security and stability, Haskell is being used in companies and industries that need to have a secure program to run their platform or programs. Some of the general applications that Haskell has been applied to has been fields such as aerospace, defense, web startups, social media, and hardware design [3] [medium.com]. More in-depth examples of Haskell being used in industry include the following: ATT, where it was used in the Network Security division to automate processing of internet about complaints; Intel, where they have developed a Haskell compiler for part of their research on multi core parallelism; SQream, where they use Haskell for the compiler where it takes SQL statements which then turns those statements into low level instructions for high performance CUDA run times [4]. As seen from this short list of applications, Haskell is common in places where code must be secure to run important material and programs.

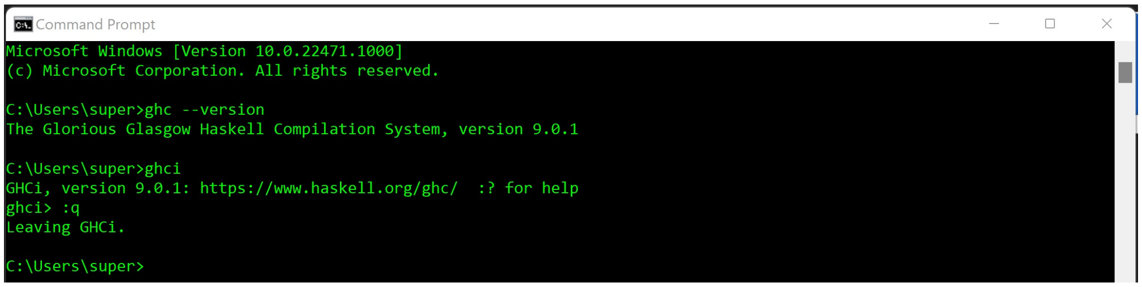
1.5 Haskell Tutorial

So far this report on Haskell has briefly explained what Haskell is, how it differs from other programming languages, and how it is used in a real-life application. It will now begin to explain how to use Haskell. However, before programming our first project in Haskell, users must first learn how to install it and run the files with the standard ghci compiler. To download and install Haskell for Windows, navigate to here to get to the website with more in depth details of the install process. Note, for Windows one will first need to install Chocolatey, which can be found here, before following the steps provided previously. Similarly, one can find the download and install instructions for Mac here. Once the Haskell platform is successfully installed on the system, navigate to either Power-shell or Command line. In the chosen environment, run in the command line the following

```
ghc -- version
```

and if Haskell was installed successfully, then the environment will print out the version number that is currently installed on the system as seen below.

After setting up Haskell in the environment, and ensuring that it is installed correctly (via the ghc –version line), it is possible to begin to test what can be done in the ghci compiler. One can write their Haskell programs in either the interpreter using the ghci command, or alternatively one could use a text editor and terminal to load and run their Haskell programs. In order to load a Haskell file into a terminal the command ":l" (short for :load) can be typed, followed by just the name of the Haskell file. If the file was "Haskell.hs" for example, then the user would type just the "Haskell" portion of the file after typing the ":l" in the terminal.



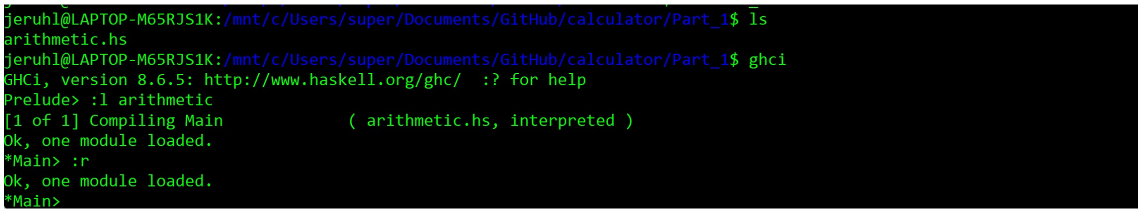
```
Microsoft Windows [Version 10.0.22471.1000]
(c) Microsoft Corporation. All rights reserved.

C:\Users\super>ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.0.1

C:\Users\super>ghci
GHCi, version 9.0.1: https://www.haskell.org/ghc/  :? for help
ghci> :q
Leaving GHCi.

C:\Users\super>
```

Figure 1: Haskell being run in the command line of a computer. Here the version control is shown in the prompt. Running the line ghci will access the interpreter of Haskell and allow the user to begin working with the Haskell Platform.



```
jeruhl@LAPTOP-M65RJS1K:/mnt/c/Users/super/Documents/GitHub/calculator/Part_1$ ls
arithmetic.hs
jeruhl@LAPTOP-M65RJS1K:/mnt/c/Users/super/Documents/GitHub/calculator/Part_1$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :l arithmetic
[1 of 1] Compiling Main             ( arithmetic.hs, interpreted )
Ok, one module loaded.
*Main> :r
Ok, one module loaded.
*Main>
```

Figure 2: Haskell file being loaded into a terminal in order to run the source code. :l is how one loads the file, followed by the file name that is to be loaded and run. :r is how one can reload the file after making changes to the file. Note for :r one does not have to put the file name afterwards (like with :l) if the user wants to reload the same file.

For the following tutorial, all of the code will be written into one .hs file (named HaskellTutorial.hs) and be used alongside a text editor and terminal to run the code. The first part of Haskell to go over is how to do simple arithmetic in the platform. Using the GHCi, simple arithmetic equations can be performed such as the following:

```
eq01 = 1 + 2
eq02 = 10 - 4
eq03 = 4 * 23
eq04 = 9.4 / 2.2
eq05 = 15 `mod` 2
eq06 = 4 ^ 13
```

In the above source code, it is shown how to write common arithmetic equations in Haskell. In order to run any one of these equations in the GHCi, one would load in the file that this is part of (or if writing directly in GHCi, we can skip that part) and type the name of the variable in the terminal. Moving onto learning how to create simple function, with no input or return type, the following is a sample of how writing that can be achieved:

```
tripleMe x = x + x + x
tripleUs x y = x * 3 + y * 3
tripleBothofUs x y = tripleMe x + tripleMe y
```

To restate what was said before, the above code is the creation of a few simple functions. With one being the addition of a number by itself three times, another being two numbers added together while both also being multiplied by three, and the last a combination of the previous two. However, one aspect to note about these functions is that they can accept either ints, integers, or floats in them at run time. The second function can even use both a float and an integer at the same time. This goes against what is stated earlier about how Haskell is a statically typed and function language where it is important to declare the types and they cannot change when we run our functions. So if this aspect of Haskell were to be utilized,the variables to be used must be declared before they are actually used (as seen with the following):

```
a :: Int
a = 3
b :: Float
b = 3.14
```

Try to apply type declaration to the variables in the three functions that was just made above. It is possible to take type declaration even further than applying it to just variables though. It can be done so by applying type declarations to functions as well when writing functions in Haskell. When creating functions and declaring them, it can be decided to choose to give our functions an explicit type declaration. The way declare types can be declared for a function goes as follows:

```
multFour :: Int -> Int -> Int -> Int -> Int
-- this function will take four integers and return one integer
multThree a b c d = a * b * c * d

-- another example with float type
circumference :: Float -> Float
circumference r = 2 * pi * r
-- this function takes in one float and returns one float
```

The two functions above show how to declare a type for a function. The important syntax part to understand here is the use of the two colons (::), which is understood as ”has type of”. The next important part is the use of (-¿), which is used to separate the parameters of the function. So when looking at the first function that is written, multFour, essentially it is being said that the function multFour has a type of Int. However, is the difference known between which of those five Ints are a parameter or a return type of the function? The way that it is read is by accepting that the last item in the declaration is the return type, and everything before is a parameter (or input). What this means is that the function multFour has four Ints as its parameter, with one Int return type, and for circumference there is one float parameter and one float return. These example have been discussing how to declare a function and what makes up parts of the function, however the discussion can be developed more by talking about adding pattern matching to the functions. To briefly explain pattern matching, it consists of specifying patterns to which some data should conform to and then checking to see if it does and deconstructing the data according to those patterns [5][learnyouahaskell.com]. As an example,a simple function is created that takes in a number from the user, and returns a certain string if the user input matches the pattern.

```
guessNumber :: (Integral a) => a -> String
guessNumber 25 = "You guess the correct number!"
guessNumber x = "You did not guess the correct number."
```

The above function is essentially checking to see if the user input matches any of the statements that were formed in our function guessNumber. In pattern matching, the patterns of a function will first be checked from the top of the function to the bottom of the function. Each time, the user input is compared against that pattern, and as long as the user input does not match the function body, it will move down to the next body. Once the user pattern is matched to a function body, then that body that has the same pattern as the input will be run. Of course, there is also a need for a case in the event that the input does not match any of the patterns. This is called a base case, and will cover any patterns that are not covered by the function. In the small function above, the input is first checked to see if it matches guessNumber 25. If it does, then the compiler will print out ”You guess the correct number”. If an input of any other pattern is submitted/checked, then it will print out the base case of guessNumber x. As stated before, it is possible to have multiple patterns to check against each with a different output. Here is an example of comparing a string input (which will be a state) against a function with other states in order to print out the capital of that state.

```haskell
cityName :: String -> String

-- here is the beginning of the different patterns to check our input against
cityName "Illinois" = "Springfield"
cityName "Alaska" = "Juneau"
cityName "Hawaii" = "Honolulu"
cityName "Iowa" = "Des Moines"
cityName "Massachusetts" = "Boston"
cityName name = "Unknown"
```

In this example of pattern matching it can be seen how there are even more function bodies to compare the pattern to and how it is possible to essentially put an infinite number of patterns to check against. When looking for more information on pattern matching, it is common to find other compare pattern matching to case statements in other languages. However, if one does not know what case statements are or one does not fully understand them, it is helpful to think of pattern matching as similar to if/else if/ and else statements. One can think of the base case (for our two functions they are cityName name = ”Unknown”; and guessNumber x = ”You did not guess the correct number”) as the else statement in an if/else statement. One can also think of the first body in the function (the top) as the first if statement, and all of the other bodies with a specific pattern to be else if statements. So far some of the most important basic capabilities of Haskell, and how to understand them, have been discussed and shown. The last and most important aspect of Haskell is its use of recursion. To briefly explain recursion, recursion is the act of a function to call itself one or more times until a certain condition is met. Recursion is not specific to Haskell, and can be seen in lots of other programming languages as well. For the function to test recursion with, the common mathematical equation of factorial will be used as the example to show how recursion works in Haskell. To take the factorial of a number, means to multiply the chosen number and multiply it by every number from it down to 1. A simple example of factorial in the common mathematical sense is as the following:

```
-note that factorial operation is represented as ! in math
example 1:
5! = 5 * 4 * 3 * 2 * 1
5! = 20 * 3 * 2 * 1
5! = 60 * 2 * 1
5! = 120 * 1
5! = 120
```

If this factorial is written as a function in Haskell pattern matching will be used to help with running the program.

```haskell
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

To begin the function, it must first be declared and then declare the type signature (for the example it will be Integral). The next step will be to use pattern matching for the function and first create the base case for it. In this function make factorial 0 = 1 the base case, with the reason for this being that the factorial should be stopped at 1. However, due to the program it is inevitable that ’n’ will reach 0 and it will be needed to set back to be one. The next line is utilizing pattern matching with recursion. Here is where it is possible to call the factorial on the number that is input in this function. As a simple example, the number three will be taken to show factorial 3:

```
--  first start with
factorial 3 = 3 * factorial(2)
--  then can find factorial(2)
factorial 2 = 2 * factorial(1)
--  then move to find factorial(1)
factorial 1 = 1 * factorial (0)
-- but here factorial 0 = 1
-- so now move back up and change
factorial 1 = 1 * 1 => 1
--  then move back up again with the factorial 1
factorial 2 = 2 * 1 => 2
-- again move up
factorial 3 = 3 * 2 => 6
-- with the final answer being 6
--  can also think of writing it as so
3 * ( 2 * ( 1 * 1))
```

Here it can see that as long as the factorial (and whatever number) matches the pattern, it will be called until we reach factorial 0. One factorial 0 is reached, then the recursion is stopped, and that is why factorial 0 is placed at the top instead of below factorial n. If factorial 0 was placed below factorial n, then there will exist a problem where the pattern matching is redundant. The reason for this is due to the fact that the pattern would catch all numbers, even 0 and the calculation would never terminate. Both using pattern matching and recursion allows programmers to create their own loops, which is useful since Haskell does not utilize loops in the language anyways. The reason why loops are created by the user is because of how programmers declare what something is instead of declaring how it is used. As long as the user knows what something is, they can apply it to other parts of the program and use it there. Through this tutorial, the important features of Haskell, and how to create/use them were discussed. There is a lot more to Haskell, with this report simply scratching the surface, if further knowledge is wanted, there are plenty of resources online dealing with more advanced topics of Haskell.

This paper will now talk about some of the features that were seen in the tutorial more in depth, and explain in more detail on what makes Haskell stand out from other common programming languages.

1.6 Haskell in Depth

So far this report, has gone over what Haskell is, some of the main features of Haskell, and a brief introduction of how to get started with programming in Haskell. The focus will now shift to exploring some of the more in-depth features of Haskell that are of more complexity than the functional and statically typed features that were explored earlier. The first new aspect of Haskell that will be explored is how the language is lazy. What being a lazy programming language means is that ”unless specifically told otherwise, Haskell won't execute functions and calculate things until it's really forced to show you a result”. This can be seen in some of the program examples earlier, specifically with the pattern matching portions. The function bodies will not be run unless the pattern that is being checked against reaches it, and if it doesn't then it moves onto the next body and just checks against it. It does not produce an output unless the pattern matches. What being lazy also means is that ”expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations”. Again, this shows how not only functions, but expressions as well are only used when they are called. This topic can be seen in the first example program that was written. In that example,empty variables are created and filled with simple mathematical expressions. However, when the program runs, it can be notice that nothing will be printed out (true, the program did not explicitly print them out). These equations will compile, but they will not run and run through the equation unless the program calls them from the command line or put them in another function such as one that will print them out, and we call that function. Having Haskell be a lazily typed language leads to some major benefits that make life easier to the programmer or user.

One large advantage about Haskell being lazy is that due to the functions and expressions not immediately running at start time, it allows the language run time to discard all of the sub-expressions and functions that are not directly dependent to the end result of the expression. Because of this discard, the run time of the program may decrease which further leads to the advantage that being lazy will reduce the complexity of an algorithm. As stated before, this is due to the discarding of the sub expressions, but also any temporary computations and conditionals. Besides the two reason stated, when would one see the use of this lazy type of programming in a real world application? Well if one is trying to use less resources and cut the run time of a program, this type of programming would be very well fit for loading data which will be infrequently accessed. However, while all of the previously stated advantages about lazy programming sound great, there are still some reasons as to why one may not want to use lazy programming or at least be more mindful of it. On occasion, Haskell being lazy will increase space complexity of an algorithm even though the time complexity is decreased. In addition to this, it may force the language run time to hold the evaluation of sub-expressions until it is required in the final result by created delayed objects. So if the sub-expression does end up being used, then its evaluation is held, and this is because they are still not quite needed yet and they will run last.

2 Programming Languages Theory

So far this paper has looked into the history of the programming language called Haskell. It has been shown how this language differs from other popular programming languages such as Python, Java, C, etc. For those interested in learning the Haskell programming language, there is also a simple tutorial that explains how to begin to program in Haskell, where the attributes discussed earlier can be seen. In addition to this, it is discussed where and how Haskell can be used in real world applications. However, there is more than just discussing the attributes of Haskell, there is a lot of theoretical work going on behind the scenes of this programming language that this next section will begin to discuss. One of the first theoretical points that will be looked at in Haskell is the theory of Lambda Calculus.

2.1 Lambda Calculus

Before learning how Lambda Calculus is theoretical to Haskell, it is important to first become familiar with what Lambda Calculus is. It was created by mathematician Alonzo Church during the time period of 1929 to 1932. Church argued that the main purpose of his newly formed calculus is that any function on the natural numbers (that can be effectively computed) can be computed using Lambda Calculus [6]. A formal definition of what Lambda Calculus is would be defined as, a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution (both of which will be looked at further later on) [7]. From the use of transforming function based abstraction and application computations with variable binding and substitution, Lambda Calculus is also considered to be a universal model of computation that can be used to simulate any Turing Machine. Where a Turing Machine is a mathematical model of computation that defines an abstract machine that manipulates symbols on a strip of tape according to a table of rules [8]. However, what specifically makes Lambda Calculus Turing Complete (defined as the ability to be a universal model of computation that can be used to simulate any Turing machine)? To answer this question, Turing completeness will be further looked at, and exactly how Lambda calculus works.

2.2 Turing Complete

Alan Turing, first described the Turing machine in 1936, in which these machines are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed. These Turing machines are useful when one wants to simulate any given computer algorithm's logic and also wants that algorithm logic to be constructed. This means that Turing machines, when setup correctly, are an optimal way to measure the resources (or efficiency) of a given algorithm or program. An even simpler definition of a Turing machine is that it takes a program, run the given program, and show some result from the given program. In essence, all programming languages theoretically could be considered a Turing machine due to their attribute of taking a program, running them, and giving a result back to the user of the program [9]. To take this theoretical concept even further, one could even say that anything that can be programmed at call can be programmed in the lambda calculus. With this logic, there could now be the reasoning that all programs are essentially Turing complete. However, this may not actually be the case. From this statement, again the question is proposed of how Lambda Calculus can be used to simulate any Turing machine (meaning that it is considered to be Turing Complete)? One of the first things to look at in order to help ensure that Lambda calculus is Turing complete, is the Church-Turing thesis. This thesis states that a function on the natural numbers can be calculated by an effective model if and only if it is computable by a Turing machine [6]. To understand whether or not something is a computable function, the formal definition must be provided. Where the definition of a computable function is a function that is the formalized analogue of the intuitive notion of algorithms, in the sense that a function is computable if there exists an algorithm that can do the job of the function. However, the formal definition of computability was not known until Kurt Godel, Alonzo Church, and Alan Turing proved that a function is lambda-computable if and only if it is Turing computable, and if and only if it is general recursive. This was formed when Godel formalized the definition of the class of general recursive functions, Church created the lambda calculus, and Turing created his machine [10]. One of the main takeaways from these formal definitions was with Church's lambda calculus and his Church numbers which help define whether or not a function is called lambda computable. This is done by if the corresponding function (from natural numbers) on the Church numerals can be represented by a term of the lambda calculus. This topic of Church numbers will be discussed later on and shown how they work.

2.3 Variable Binding

One of the main aspects of the Lambda Calculus is the ability to construct lambda terms and performing reduction operations on them. Like other programming languages, Lambda calculus has the ability for users to declare variables. It works similarly to other languages with variable declaration and use in functions. The following is a simple example of variable declaration and use in Python.

```
def function(x):
    return x + y
```

In this example, the x in-between the parenthesis is the declaration of the parameter x. The second use of x is when that variable is actually in use and an operation is being performed on it. In programming, each variable declaration defines a scope for that variable. In which a scope of a variable declaration is the collection of all parts of the program in which this variable is accessible via its name or identifier [11]. In the example above, the scope of the variable is the body of the function, and when x is used within the function (or within the scope of a declaration) the former is bound to the latter, and the latter is the binding occurrence of the variable. In the python example of variable binding, the x within the parentheses is the binding occurrence of this variable and the actual use of x (on the next line) is bound to the binding occurrence. In the case of a variable that is not bound in the expression, it is said to be free. Again in the example above, the variable y would be a free variable since it is not declared in the function before its use, and it is only available within the function. To achieve variable binding, Lambda calculus uses static binding, where each variable use is bound to the variable declaration by the same name in the smallest lambda abstraction that contains the variable use. In this next example, variable binding will be shown in Lambda Calculus.

```
\y. (\x. x (y x))
```

The first use of y (outside the parentheses) is the declaration of it within the function, and the second use of y (within the function) is the process of binding it to the variable y. The scope of this declaration is

```
(\x. x (y x))
```

in which it is implied that the rightmost occurrence of y is bound to the left most occurrence (which is the one outside of the parentheses). Now that the variable binding of y has been covered, the binding of x will be discussed. It is very similar to the binding of y, but notice how there is an extra use of x in the function. In Lambda calculus, only the first use of a variable after declaration is the act of binding a variable to that declaration. So in the case of this example the scope of the binding occurrence of x in

```
\x. x
```

is just the second x. The third x within the parentheses is a free variable due to it being a use of x that does not belong within the scope of any declarations of x. The following examples will be binding variables

```
\y. \x. (( y y) (y \z. x))
```

In this example, there are three declarations of variables, with them being x, y, and z. When first looking at the declaration and binding of y, the y variable has three binding occurrences. Here the scope is the portion of

```
((y y) (y \z.x))
```

Because the scope is large, the y shows up three times within the scope and each one is a binding occurrence to the declaration. Likewise when looking at x and its declaration/binding (again within the same scope), the x next to the declaration of lambda z is the binding occurrence of x. Finally, the declaration of z has no binding and is therefore left unbound. The next few examples will look at the free variables that may appear in lambda calculus variable binding.

```
(((z y) \x. u) \v. \u. v)
```

In this example, there are three occurrences of a free variable as well as a few declared (yet unused) variables. The free variables in this example include the z, y, and u. It is possible to see why this is due to the fact that the z and y variables do not have an accompanying declaration before their use. Similarly, the variable u does does not have a declaration of the variable before use. However, it can be noticed that there is a declaration of u after the first use, but also note that this is then out of scope of that declaration. The scope of the the later declaration consists of only being v. The scope that the use of u belongs to is

```
((z y) \x. u)
```

and nowhere in here is u, z, or y declared. In another example, a similar pattern can be seen where the scope of the variables is outside the declaration of itself, or that there is no declaration to begin with.

```
(((( u y) \v. v) \v. (v y)) \v. (y z))
```

Here, there are three free variables including the u, y, and z variables. Note here that y has three separate occurrences of being unbound and free. The first scope

```
((u y) \v. v)
```

has the declaration of v, and right after is the binding of the v variable to that declaration. With the scope of that being only (lambda v. v). Looking into the next scope of this function,

```
\v. (v y)
```

it can be seen that again, v is being declared and bound within this scope. However, it is also seen that the y variable is not being declared at all before the use of this. Likewise, with the final scope, it is seen that v is declared but never used, and both variables y and z are not declared before use.

```
\v. (y z)
```

It has been shown how variable declaration and binding, as well as scopes, work in the lambda calculus. It is similar to other programming languages such as Java and JavaScript. This ability, is what allows the reduction of longer functions to be reduced in the lambda calculus. However, it is possible to reduce even further by using Church numerals.

2.3.1 Church Numerals

As discussed earlier, Church numerals help in the definition of encoding natural numbers in lambda calculus, which helps prove how and why Lambda calculus is Turing complete. Here, Church numerals will be looked at in more detail and shown how they work. Church numerals are essentially a representation of the natural numbers using lambda notation [12]. This idea is similar to the functional representation of natural numbers as seen in the Haskell programming language. In which "succ" could be a function that returns the successor of the natural number that it was given. In Church numerals they are all considered to be functions with two parameters [13]. In this paper "a single backslash" will be used to represent the lambda symbol. To begin with, it is a good place to start with a simple outline of what a Church numeral function looks like, which is very similar to lambda calculus.

```
\f. \x. something
```

The above is a simple function where f is a parameter where it is the successor function that should be used. Next, x is another parameter that represents zero. So making this go from lambda calculus to Church numerals would now take on the form of the following:

```
C0 = \f. \x. x
```

Where the above is the Church numeral for zero [13]. When this is applied, it will return the value that represents zero, which means that it should return x. It can also apply the successor function to x, or whatever the value is that represents zero.

```
C1 = \f. \x. f x
```

With this knowledge now, it is possible to apply the successor function to x in a varied number of Church numerals, such as the following:

```
C2 = \f. \x. f (f x)
C3 = \f. \x. f (f (f x))
c4 = \f. \x. f (f (f (f x)))
C5 = \f. \x. f (f (f (f (f x))))
C6 = \f. \x. f (f (f (f (f (f x)))))
.
.
.
Cn = \f. \x. f^n x
```

What these examples have been showing is how to use minimal lambda calculus to create Church numerals [13]. However a problem arises, where we are unable to do a lot with these Church numerals when it comes to more complex math operations such as addition and multiplication. Here, we are simply making numbers by increasing the numbers values through the use of successor. If we create numbers to our own language, then it would be possible to convert Church numerals to decimal numbers, if S and Z were to be used.

```
S = \r. 1 + r()
Z = \r. 0

-- note that now C6SZ is equivalent to 1+1+1+1+1+1+0
```

Now using this new knowledge of how to create Church numerals, it is possible to use them in other minimal lambda calculus math operations such as addition [13]. Here, it is shown how to setup an addition function using Church numerals:

```
C3+4 = \f. \x. C3 f (C4 f x)
```

Here, the function has two parameters, where C3 is applied to f (the successor function), and the C4 f x portion is now the value that represents zero. Further more it is possible to show the work behind this function, which can be seen as the following:

```
-- remember that C3 is made from the following
C3 = \f. \x. f (f (f x))

-- also remember that C4 is made from the following
c4 = \f. \x. f (f (f (f x)))

-- the two can be combined as such
C3+4 = \f. \x. C3 f (C4 f x)

-- which can be expanded to
C3+4 = \f. \x. (\f3. \x3. f3 (f3 (f3 x3))) f (\f4. \x4. f4 (f4 (f4 (f4 x4))) f x)
   -- here the program writes (\f3. \x3. f3 (f3 (f3 x3))) into f (f (f x)), where (f x) is the f4
       portion
   = \f. \x. f (f (f \f4. \x4. f4 (f4 (f4 (f4 (f4 x4))) f x)))
   -- agin the program rewrites \f4. \x4. f4 (f4 (f4 (f4 (f4 x4))) f x))) into the C4 which is f
       (f (f (f x)))
   -- note that abstraction is also used to drop the parentheses
   = \f. \x. f (f (f (f (f (f (f x))))))
   = C7
```

If the user wants to create another function that will hold the Church numbers as arguments and then return the sum of them, then it is possible by simply creating a new function like so:

```
add =  \M. \N. \f. \x. N f (M f x)
addC4C7 = C7
```

Here, the M and N are similar to the S and Z from the previous example, where they hold the values of the Church numerals that were found earlier. Also note, that within this function there are four lambdas. Two from the Church numbers and two from the original C3+4 function. Now that is has been seen how to formulate Church numerals in traditional mathematical format, now the question can be posed of how can Church numerals be made in the Haskell programming language? First, before just trying to use lambda calculus in Haskell and converting into Church numerals through the use of variables (to potentially hold them), know that this approach might not work. The reason for this is due to there not being an expression of a type that doesn't have an instance "Show" defined, which is used to print things in Haskell by the GHCi when using the print function. The reason as to why the print function is unable to be used here is because in Haskell, it is not possible to print values. It is only allowed to print functions that have used defined values [14]. In order to help with this problem, it is suggested to create a new type just for Church numerals, which could look like the following:

```
data Church x = Church ( (x -> x) ->x -> x)

zero :: Church x
zero = Church (\f x -> x)

-- succ is already used in Haskell so use a similar variable name
succ_ :: Church x -> Church x
succ_ (Church n) = Church (\f x -> (f (n f x)))

instance (Num x) => Show (Church x) where
    show (Church f) = show $ f (1 +) 0
```

Now from here, it is possible to write some functions in Haskell that are using Church numerals [17]. Not only did this code create a type for the use of Church numbers, but it also added an instance declaration for the Show function. Looking back at how to create the Church numerals that were simple successor numbers, that can be achieve from the following:

```
-- for zero
c0 = \f x -> x

-- for one
c1 = \f x -> f x

-- for two
c2 = \f x -> f (f x)
```

Now, with the creation of these Church numbers from lambda calculus, the Church numerals can take the place of traditional lambda calculus. From here, there is a reduction to the use of lambda calculus, where now the Church numerals are implicitly calling/using the lambda calculus to create their own functions. This can be seen in the addition function with Church numerals, and this too could be replicated in the Haskell programming language. Here are some of the other following operations that can utilize Church's numbers combined with the lambda calculus.

```

-- successor
succ n f x = f ( n f x) -- function definition
\n. \f. \x. f (n f x) -- lambda expression

-- multiplication
mult m n f x = m f (n f x) -- function definition
\m. \n. \f. \x. m f (n f x) -- lambda expression
\m. \n. n succ m -- lambda expression

-- exponentiation
exp m n f x = (n m) f x -- function definition
\m. \n. \f. \x. (n m) f x -- lambda expression
\m. \n. \f. m (n f) --lambda expression
```

Here, it is seen how to use Church numbers to represent natural numbers, and how it is possible to do calculations with them. However, now the question may be asked is it possible to do subtraction with Church numbers? When these numbers were first created by Church, the subtraction implementation was not yet discovered. That was at least until Kleene discovered how to implement the subtraction function with Church numbers through the use of using a predecessor function.

2.4 Combinatory Logic

When one is both learning about and working with Lambda Calculus, another type of logical math will be mentioned. This other form of math is known as Combinatory Logic. It is similar to Lambda Calculus except for the fact that combinatory logic eliminates the need for quantified variables, therefore making this form of math variable free. This can be useful for where the status of variables are difficult or when implementing certain things [14]. Combinatory logic is based on combinators which are high order functions that uses only function application and earlier defined combinators to define a result from its argument [14]. In addition to this, combinatory logic helps to untangle to problem of substitution (that may be found in Lambda-Calculus), because formulas can be prepared for the elimination of bound variables by inserting combinators [15]. This means that an expression that has no bound variables represents the logical form of the original formula. Because of this, Combinatory Logic can emulate Lambda-abstraction even though Combinatory Logic has no variable binding operators. This in turn makes this form of programming a suitable language for functional language, which is how one is able to connect this back to Haskell. Like Lambda-Calculus, Combinatory Logic can be used and written in Haskell since Haskell is a functional programming language. To further talk about this, Combinatory Logic is also Turing Complete just like Lambda-Calculus. The reason for this is based on the definition that was just stated. Combinatory Logic can be used to recreate Lambda-Calculus in its own definition. Therefore, since Combinatory Logic is essentially Lambda-Calculus, without bound variables, then it means that it is Turing Complete since it works (essentially) the same as Lambda-Calculus. In fact, both of these are used for higher order logic and computing in order to underpin important logical systems and programming languages [16]. However, they are hidden from users most of the time due to their high level of abstraction.

The reason why Combinatory Logic is said to be equivalent to Lambda-Calculus is because it is a formal system that can express functions without the use of formal variables. In turn this means that, every term is a function and there is just one binary operation,application. Combinatory Logic can even be used as a simplified model of computation. It can be viewed as a variant of lambda calculus, it is easy to transform lambda expressions into combinator expressions (and combinator reduction is must simpler than lambda reduction). These two reasons are why it has been used to model some non-strict funtional programming languages and hardware [14].

To see how Combinatory Logic would be formed, here is short example displaying it. First a function is given that is being wanted to be translated into CL.

```
fgxy = (fg) xy = ((fg) x) y
```

First, note that all combinators are dervived from two standard combinators S and K. Where S and K are given the definition as:

```
S fgx = fx (gx)
K xy = x
```

Sometimes, if something is needed to be referenced, then a third standard combinator (I) can be used. Where I is defined as:

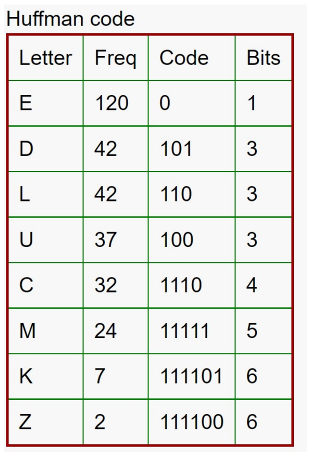
```
I x = SKKx = x
```

Where I is being defined in terms of both S and K, and because of this I can be known as pure combinatory logic. Taken these defined terms again, it is possible to convert them into Lambda Calculus where:

```
Sfgx = fx (gx) -> S = (\f(\g(\x(fx)(gx)))) -- turning S from CL to LC
-- and
Kxy = x -> K = (\x(\yx))
```

3 Project

It has been shown how Haskell is a unique programming language that has a lot of complicated but useful aspects to it. With these aspects, it is possible to create projects to see all the aspects that are available to use. For this report, a project will be made using Haskell to showcase the Huffman code. The Huffman code is a type of optimal prefix code (or a type of code system distinguished by its possession of the ”prefix property”, which requires that there is no whole code word in the system that is a prefix of any other code word in the system) that is commonly used for lossless data compression [19]. It assigns code to characters such that the length of the code depends on the relative frequency or weight of the corresponding character. It can also be visualized as its own type of tree which is known as, a Huffman coding tree. The Huffman tree is a full binary tree where each leaf of the tree corresponds to a letter in the given alphabet [19]. Below is a visual example of a simple Huffman tree table assigning values, as well as a tree plotting the leaves into the tree.



Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

Figure 3: A visual table assigning letters to frequency, code, and bits.

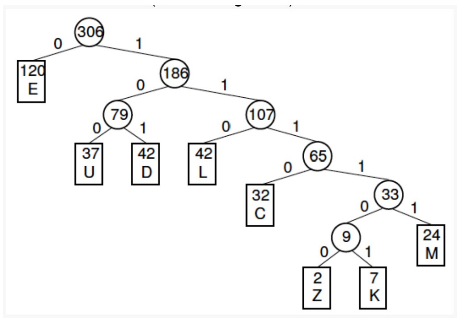


Figure 4: A visual tree plotting the leaves on the tree..

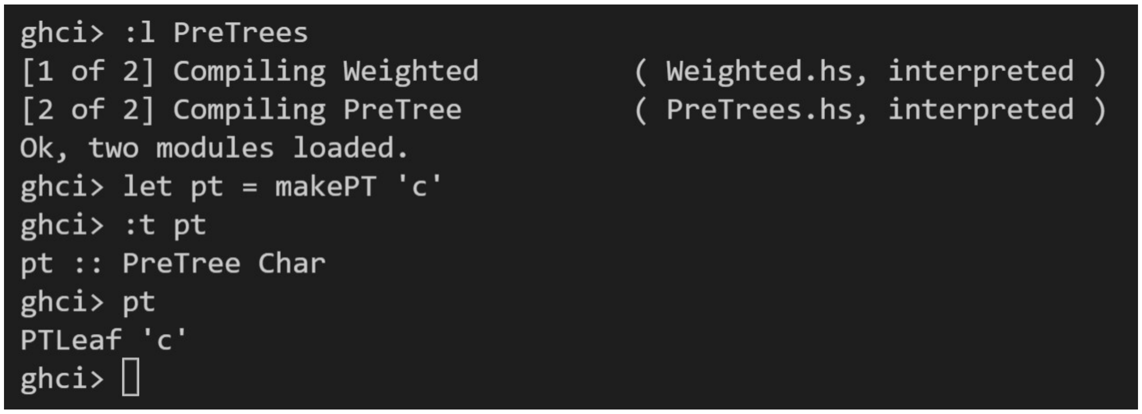
In this project, the goal will be to look at two types of binary trees, used to implement immutable and persistent priority queues and prefix trees. The first step in creating a Huffman tree is to use priority queues to build the tree itself from the ground up using the two lowest-frequency nodes that are popped from the queue, combining them into a new node, and finally placing them back in the queue [19]. Before this though, the tree itself must be created, and in order to achieve this prefix trees can be used since all data is stored in the leaves, and all internal nodes have two children. This is useful to hold the letter and frequency from the table that were seen earlier. In order to place the data (the letter and assigned number), into a node, which is then in turn placed into a leaf, the following code will be used (written in a haskell file called PreTrees.hs)

```
data PreTree a = PTLeaf a
    | PTNode (PreTree a) (PreTree a)
    deriving (Show, Eq, Generic)
    -- where a is parameterized so its type can be determined later
```

Of course, it may be wanted by the user to place something onto an empty tree. In this case, it means to create a leaf containing just that data. This can be achieved by writing

```
makePT :: a -> PreTree a
makePT = PTLeaf
-- here PTLeaf is a data constructor that calls and creates a PreTree when PTLeaf is used
-- also note that here, makePT is essentially the same as PTLeaf since PTLeaf is already a
    function that was defined earlier
```

When the code is run, it will be run as the following Now that an empty tree has been created, it is possible



```
ghci> :l PreTrees
[1 of 2] Compiling Weighted         ( Weighted.hs, interpreted )
[2 of 2] Compiling PreTree          ( PreTrees.hs, interpreted )
Ok, two modules loaded.
ghci> let pt = makePT 'c'
ghci> :t pt
pt :: PreTree Char
ghci> pt
PTLeaf 'c'
ghci>
```

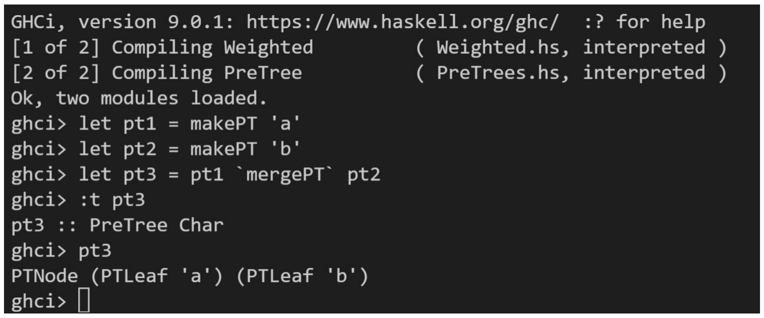
Figure 5: Here, a variable named 'pt' is being created in order to store the function that was made, makePT. Then the type of the function is shown as well as its current stored data. Lastly, when printing pt, the original input 'c' is shown.

to create multiple trees, and then combine them together. The reason as to why one may want to do this is because in order to build one tree (with lots of data), one must merge the trees (or leaves) until the data is combined and connected within the tree. In order to achieve this, it is possible to place two PreTrees into one function and store them into a new object (which will be a node). This can be accomplished with the following:

```
-- here two PreTrees are used as the input, and one PreTree (the combined one) is the output
mergePT :: PreTree a -> PreTree a -> PreTree a
mergePT = PTNode
```

Again, running the mergePT function will look like the following

Now that there are PreTrees being made (with data in them), now it must be made possible to compare the two trees. The way that comparison will work will be by comparing the weights or priorities of the two PreTrees when the tree is being created. In order to achieve this, a data type will be used to include both a PreTree and the weight of an integer. To make this data type comparison, the following code will be used (written in a new Haskell file named Weighted.hs)



```
GHCi, version 9.0.1: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Weighted         ( Weighted.hs, interpreted )
[2 of 2] Compiling PreTree          ( PreTrees.hs, interpreted )
Ok, two modules loaded.
ghci> let pt1 = makePT 'a'
ghci> let pt2 = makePT 'b'
ghci> let pt3 = pt1 `mergePT` pt2
ghci> :t pt3
pt3 :: PreTree Char
ghci> pt3
PTNode (PTLeaf 'a') (PTLeaf 'b')
ghci>
```

Figure 6: Description.

```
data Weighted a = WPair {
                     _Weight :: Int
                     , _WItem :: a
                     } deriving (Show, Functor)
```

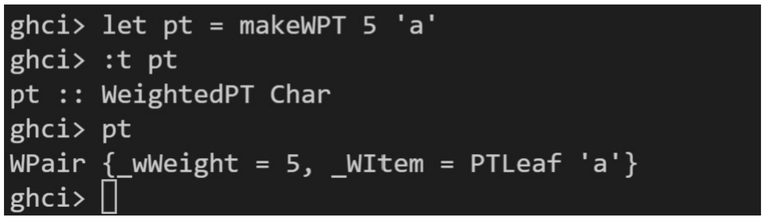
Here, Weighted a is an a that is associated with the weight of an integer. This will allow the user to create a PreTree containing any character, and assign an integer to weigh it. Since this function is useful to assign and see the weight of a character in a tree, it will be a good idea to turn this into a typedef. This action will be done in the PreTrees.hs file with the following code being:

```
type WeightedPT a = Weighted (PreTree a)
```

Again, it is possible to create the same make and merge functions for the WeightedPT as done before with the normal PreTrees. This is done in a similar fashion (again, in the same PreTrees.hs file)

```
makeWPT :: Int -> a -> WeightedPT a
makeWPT w = WPair w . makePT
```

where first, use MakePT with the weight (w) and then add that results to a WPair w. Running this code will should result in a similar output such as the following:



```
ghci> let pt = makeWPT 5 'a'
ghci> :t pt
pt :: WeightedPT Char
ghci> pt
WPair {_wWeight = 5, _WItem = PTLeaf 'a'}
ghci>
```

Figure 7: Description.

Here, makePT is being called by the function makeWPT, where the character ’a’ is being put into the PTLeaf and then being placed into a WItem into the Weighted typeset. It is also possible to merge the two Weighted PreTrees the same way that the normal PreTrees are merged. This accomplished in the PreTrees.hs with a similar code of:

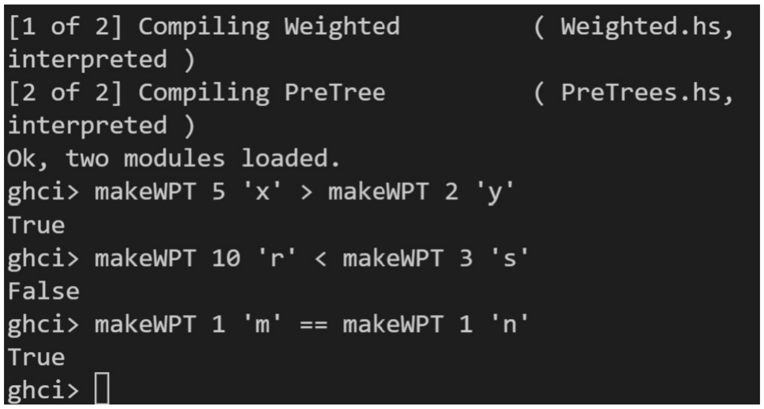
```
mergeWPT :: WeightedPT a -> WeightedPT a-> WeightedPT a
mergeWPT (WPair w1 pt1) (WPair w2 pt2)
    = WPair (w1 + w2) (mergePT pt1 pt2)
```

Again, there are two WeightedPreTrees being used as input in the function, with one WeightedPT being the result/output of the two previous trees merging. Here the total weight of the two PreTrees, being merged into one, would be the sum of weights of the two subtrees. Lastly, Haskell has a typeclass called Ord that abstracts comparing operations. This is useful for comparing items, which it what is needed in the program to compare the weighted items and order them in some way. This can comparing operation can be achieved with the following code in the Weighted.hs file:

```
instance Ord (Weighted a) where
    compare (WPair w1 _) (WPair w2 _) = compare w1 w2

instance Eq (Weighted a)where
    WPair w1 _ == WPair w2 _ = w1 == w2
```

Now, when loading the PreTrees.hs file in the ghci, it is now possible to compare the data within two weighted pretrees. This is done by using the makeWPT function and comparison operators to determine which is greater than, less than, or equal too. It can be seen in the following output where: With these two



```
[1 of 2] Compiling Weighted         ( Weighted.hs,
interpreted )
[2 of 2] Compiling PreTree          ( PreTrees.hs,
interpreted )
Ok, two modules loaded.
ghci> makeWPT 5 'x' > makeWPT 2 'y'
True
ghci> makeWPT 10 'r' < makeWPT 3 's'
False
ghci> makeWPT 1 'm' == makeWPT 1 'n'
True
ghci>
```

Figure 8: Description.

files being completed, it is now possible to make the single leaf nodes of the tree, give them a weight, and compare them. The completed PreTrees.hs code should resemble:

```
{-# LANGUAGE DeriveGeneric #-}

module PreTree where

import Control.Applicative      ((<$>), (<*>), (<|>))
import Data.Binary
import Data.List                (unfoldr)
import Data.Map.Strict          (Map)
import Data.Monoid              ((<>))
import GHC.Generics
import Weighted
import qualified Data.Map.Strict as M

data PreTree a = PTLeaf a
    | PTNode (PreTree a) (PreTree a)
    deriving (Show, Eq, Generic)

-- put something into empty tree, so create a leaf containing just that data
makePT :: a -> PreTree a
makePT = PTLeaf

-- merge two PreTree a's
mergePT :: PreTree a -> PreTree a -> PreTree a
mergePT = PTNode


type WeightedPT a = Weighted (PreTree a)

makeWPT :: Int -> a -> WeightedPT a
makeWPT w = WPair w . makePT

mergeWPT :: WeightedPT a -> WeightedPT a-> WeightedPT a
mergeWPT (WPair w1 pt1) (WPair w2 pt2)
    = WPair (w1 + w2) (mergePT pt1 pt2)
```

and the completed Weighted.hs code should resemble:

```
{-# LANGUAGE DeriveFunctor #-}

module Weighted
    ( Weighted(..)
    )where

data Weighted a = WPair { _wWeight :: Int
                        , _WItem :: a
                        } deriving (Show, Functor)

instance Ord (Weighted a) where
    compare (WPair w1 _) (WPair w2 _) = compare w1 w2

instance Eq (Weighted a)where
    WPair w1 _ == WPair w2 _ = w1 == w2
```

Now that the nodes can be created and compared, it is time to put the trees onto a priority queue organized by weight (or frequency). In order to fully understand the Huffman tree, this project will create its own priority queue (in a file called PriorityQueue.hs) using a skew heap. Where a skew heap is a type of heap that doesn't explicitly maintain its balance, but it maintains "heap ordering". So the first step in creating the priority queue, using skew heaps, is to define a new data type for the skew heap which can be done as the following:

```
data SkewHeap a = SEmpty
    | SNode a (SkewHeap a) (SkewHeap a)
    deriving (Show, Eq, Foldable)
```

From here, it is possible to create the three available operations of the skew heaps (making a new one, merging two skew heaps, and popping off the root). These three operations can be made with the following code:

```
-- create a skewheap
makeSH :: a -> SkewHeap a
makeSH x = SNode x SEmpty SEmpty

-- pop the root out of skew heap
popSH :: Ord a => SkewHeap a-> (Maybe a, SkewHeap a)
```

```haskell
popSH SEmpty = (Nothing, SEmpty)
popSH (SNode r h1 h2) = (Just r, mergeSH h1 h2)

-- merge two skew heaps
mergeSH :: Ord a => SkewHeap a-> SkewHeap a -> SkewHeap a
mergeSH SEmpty h = h
mergeSH h SEmpty = h
mergeSH hA@(SNode xA lA rA) hB@(SNode xB lB rB)
    | xA < xB = SNode xA (mergeSH rA hB) lA
    | otherwise = SNode xB (mergeSH rB hA) lB
    -- merging a skew heap with an empty heap is that same skew heap
    -- when merging two heaps, the new heap is an SNode with the smaller root, and its children
        are the merge of the smaller tree and the original children.
```

The first function of the skew heap (makeSH) creates a skew heap from a single item. The second function of the skew heap (popSH) pops the root (with the highest prioritized item) out of a skew heap. It then return 'Just' if the skew heap is not empty and 'nothing' if the skew heap does have items within it. It will also return the resulting skew heap that has been modified. Lastly, with mergeSH, that function merges two skew heaps and preserves heap ordering. Weight also comes into play here, where having a lower weight means having a higher priority, and having a higher weight will have a lower priority.

Now, while it is certainly possible to leave off the code just there, it is certainly a worth while idea to create an interface for the priority queue type. The reason for this is because so that the skew heap implementation is hidden from users who may accidentally access the underlying skew heap. Why this may want to be hidden is due to the abstraction that is brought with Haskell, and those not too familiar with it may not understand the skew heap, or may accidentally change it and ruin the program. So in order to hide the skew heap and create an interface, adding the following code (into PriorityQueue.hs) will work:

```haskell
newtype PQueue a = PQ (SkewHeap a) deriving Show

emptyPQ :: PQueue a
emptyPQ = PQ SEmpty

insertPQ :: Ord a => a -> PQueue a -> PQueue a
insertPQ x (PQ h) = PQ (mergeSH h (makeSH x))

popPQ :: Ord a => PQueue a -> (Maybe a, PQueue a)
popPQ (PQ h) = (res, PQ h')
    where
        (res, h') = popSH h

sizePQ :: PQueue a -> Int
sizePQ (PQ h) = length (toList h)
```

In this portion of code, Pqueue is made a newtype so that it can wrap around a skew heao so that it can keep the implementation not seen. The following functions are fairly straight forward. Beginning with the emptyPQ, it is used to create a new empty priority queue. The insertPQ function is created in order to insert an item into a priority queue and then return the new priority queue. The popPQ is a bit more involved since it tries to pop the highest-priority element of the queue that was made earlier (using emptyPQ). It will return 'nothing' if the queue is empty and 'just x', with the highest priority element if it is not empty. Lastly, the function sizePQ will return the size of the given priority queue.

Now that the necessary items have been made to populate the Huffman tree, it is time to begin creating the tree itself. The first step however, it to create a frequency table (similar to one in figure 3). In order to build the frequency table, the following will be used and created (in a new file called HuffmanTree.hs):

```haskell
import qualified Data.Map.Strict    as M

type FreqTable a = Map a Int

listFreq :: Ord a => [a] -> FreqTable a
listFreq = foldr f M.empty
  where
    f x m = M.insertWith (+) x 1 m
```

The first step in creating this table is to create a new type of data that is using a type of Map. This is what FreqTable does, by building a map of items to their frequency of occurrence. However, this can only be used if Data.Map.Strict is imported into the file which is the first line in the code above. To test if the FreqTable is working as intended, listFreq is created in order to generate a frequency table from a random string. It builds a table of 'a's from a list of a's.

Now that the frequency table has been built, it is now time to build a queue in which it will be comprised of Huffman leaves built from elements with associated weights. This can be accomplished by using M.foldWithKey which gives the folding function both the key and the value (e.g. 'a' 1).

```haskell
listQueue :: Ord a => [a] -> PQueue (Weighted a)
listQueue = M.foldrWithKey f emptyPQ . listFreq
  where
    f k v pq = insertPQ (WPair v k) pq
```

This function builds a priority queu of weighted 'a's from a list of a's, where lower weighted items have a higher priority.

Now it is time to actually build the tree with states. The build process of a tree works as first popping an item from the queue, then popping another item, and finally merging the two popped items and pushing them back to the queue. In order to build the actual tree, the following will be used:

```haskell
buildTree :: State (PQueue (WeightedPT a)) (Maybe (PreTree a))
buildTree = do
    t1' <- state popPQ
    case t1' of
      Nothing ->
        return Nothing
      Just t1 -> do
        t2' <- state popPQ
        case t2' of
          Nothing ->
            return (Just (_wItem t1))
          Just t2 -> do
            let combined = mergeWPT t1 t2
            modify (insertPQ combined)
            buildTree

runBuildTree :: Ord a => [a] -> (Maybe (PreTree a))
runBuildTree xs = evalState (listQueueState xs >> buildTree) emptyPQ
```

This code first starts off with a Maybe(PreTree a) in order to cover the case where a build would fail if the queue was empty. That is also covered further in the first case where if the queue was empty at the start, the build will fail and nothing will return. The next case is a success where there is only one item inside the queue, and a 'just' is returned to show that the build is a success. In addition to this, the return(just) will break out of the loop. Looking at the Just t2 -¿ do next, will perform the act of merging the popped items and pushing them back onto the queue. Lastly, a recursive call is ran in order to go back to step 1 of popping an item from the queue. All together the code of the file HuffmanTree.hs will resemble the following:

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
module Huffman where

import Control.Monad.Trans.State.Strict
import Control.Monad
import Data.Map.Strict              (Map)
import qualified Data.Map.Strict    as M

import PQueue
import PreTree
import Weighted

type FreqTable a = Map a Int

listFreq :: Ord a => [a] -> FreqTable a
listFreq = foldr f M.empty
  where
    f x m = M.insertWith (+) x 1 m

listQueue :: Ord a => [a] -> PQueue (Weighted a)
listQueue = M.foldrWithKey f emptyPQ . listFreq
  where
    f k v pq = insertPQ (WPair v k) pq


buildTree :: State (PQueue (WeightedPT a)) (Maybe (PreTree a))
buildTree = do
    t1' <- state popPQ
    case t1' of
      Nothing ->
        return Nothing
      Just t1 -> do
        t2' <- state popPQ
        case t2' of
          Nothing ->
            return (Just (_wItem t1))
          Just t2 -> do
            let combined = mergeWPT t1 t2
            modify (insertPQ combined)
            buildTree

runBuildTree :: Ord a => [a] -> (Maybe (PreTree a))
runBuildTree xs = evalState (listQueueState xs >> buildTree) emptyPQ
```

With these files, it is now possible to build a simple Huffman tree. This project teaches one how to create the data structure of Huffman code within the programming language, Haskell. It showcases how users are able to create their own data typesets to be used in various ways, as well as how to hide some of these typesets and wrap another type around them. Of course, recursion is found throughout this program many times in order to correctly make the nodes and leaves of the tree. This project is a good project for a more intermediate user of Haskell to attempt due to the level of abstraction that is present within it.

4 Conclusions

As it has been shown, Haskell is a programming language that offers a lot to the users although it it may be complicated and more abstract than other languages. This report began by discussing the history of Haskell and how it differs from other common programming languages. This was seen by how Haskell is a functional language as well as statically typed. In order to see these different aspects being used, a short tutorial was created to help beginners of Haskell used to the syntax and abilities available to them. While the tutorial explains some of the more common aspects of Haskell, section two delves into the theory of programming languages. Concepts here are lambda calculus (and how it it used in Haskell), Turing completeness, variable binding, church numerals, and combinatory logic. These topics were related back to Haskell, and programming languages in general, through the use of explanation and examples. To bring these concepts together, a mini project was created and reviewed to show how Haskell can have real world applications.The project looked at how to create a Huffman Tree in the Haskell programming language. This project can also be used to further the knowledge that was given earlier from the tutorial. All together, these topics in this report help readers understand both Haskell topics and theoretical topics about programming languages.

5 Bibliography

References

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

[1] Haskell (Programming Language), Wikipedia.

[2] HaskellWiki, Haskell.org.

[3] Why Haskell?, Greek Culture.

[4] Haskell in Industry, Greek Culture.

[5] Types and Typeclasses, Learn You a Haskell.

[6] Lambda-Calculus: The Other Turing Machine, Blelloch and Harper, 2015.

[7] The Lambda Calculus, Stanford.edu.

[8] Turing Completeness, Wikipedia.

[9] What Makes a Programming Language Turing Complete?, Gruhn Niklas, Dev.to, 2020.

[10] Church-Turing Thesis, Wikipedia.

[11] 3.3 Free and Bound Variables, OpenDSA.

[12] Church Encoding, Wikipedia.

[13] Comp 311 - Review 2

[14] Combinatory Logic, Wikipedia.

[15] Combinatory Logic, Stanford.edu.

[16] History of Lambda-Calculus and Combinatory Logic,Hindley, Roger J, 2006.

[17] Church Numerals in Haskell, StackOverflow.

[18] Huffman Coding — Greedy Algo-3,Geeks for Geeks, 2021.

[19] Huffman Coding, Indiana.edu.

Reference to code for the project:

[20] Huffman Encoding in Haskell

[21] Huffman.lhs, Github

[22] Code/Decode Huffman Tree Haskell, StackOverflow.