CPSC-354 Report

Connor Cowher
Chapman University

12/21/2021

Abstract

This report will dive into the aspects of learning Haskell and the functions to know when creating a project. I thought I would create a tutorial to learning Haskell the way I would have liked to learn. I found the language to be quite difficult in some areas and a guide on getting started would have been super helpful in comparing this new language to ones I have already had the pleasure of working in. I will discuss how Haskell is much different from other programming languages like Java and C/C++ and requires a more extensive knowledge to perfect. This software has an extensive Prelude, similar to a library, that should be looked into in order to better utilize the functions for your program. Haskell is a great introductory language for new programmers since it is its own unique assembly type language. Although you do not need a masters degree in mathematics, it would be beneficial to have some exposure in advanced arithmetic. Haskell is both as difficult and simple as you make it. As long as you study, understand, and make good decisions while programming, Haskell will be a great platform for you to utilize. I will also be explaining how to make a project in this language, while showing some example code in Haskell. Coding in this language seems very basic and very few characters are required per line, for the most part. A very useful topic to know and have in your back pocket would be lambda calculus. This version of mathematics is very helpful in computer calculation logic, a major aspect of Haskell. While helping out in logic, another helpful aspect of lambda calculus is its ability to simulate on a Turing machine. Though limited to Linux systems, a virtual machine or virtual box application can simulate an apple OS while on a windows machine. At the cost of RAM for some, any computer can run Haskell and utilize the programming language for a business, personal use, financial applications, etc.

Contents

1 Introduction

Haskell is a programming language where it may take some time to understand all the features and nuances. One aspect where Haskell differs from the common universal languages is being ”lazy”. This means Haskell will not do any work unless it needs to be completed. It is as close to assembly language we have had in all of the courses Chapman offers. With learning Haskell, you do not necessarily have to have the most extensive programming experience or knowledge. Haskell is so vastly different from all other languages that it makes it the perfect starting point. Although learning something so different and unique has its advantages, being skilled in mathematics has its benefits to this language. Much of the beginning is understanding and interpreting basic mathematics in different formats. Something as simple as a basic calculator proves its challenges and will make you have an appreciation for everyday programs we use without an extra thought. Since most students taking this course are upperclassmen, that does imply a more extensive understanding than most individuals so we can assume this isn't our first time learning a new language. Assembly language is vastly different from interpreter languages as one knows, but understanding how the integers and and negative numbers work with different operations will take some time. Firstly, functions are unique in the way they are created. There are multiple classes that must work hand in hand to make a program function correctly. The first assignment for Programming Languages was to create, from a template, a calculator and make our own operations. With a partner we concluded to do our basic 4 mathematical operations and added a couple more, commonly found on a TI-84 calculator. While these other operations, square root, squared, etc. are simple in theory, creating a function that can handle any type of input proves much harder to accomplish. With that out of the way we can dive into the beginning of starting this language. To begin, you should start downloading Haskell by visiting this link.

After familiarizing yourself with the user interface and paths needed to have interacting classes, a beneficial task to complete would be to go back and commit to understand lambda calculus. A majority of functions will require some knowledge of lambda calculus and understanding it would fast track any issues you run into while coding any mathematical functions in Haskell. A section of this tutorial will introduce the topic and touch upon the basics but to fully utilize Haskell, a great recommendation would be to immerse yourself in the subject for a few hours until the basics do not trip you up.

If Haskell is a name that you do not recognize then you are not alone. Before taking the Programming Languages course at Chapman I had no idea there was a functional programming language that existed and had a firm hold on the industry for years. Unfortunately for Haskell people moved on to bigger and better things but a desire for those who know the language still exists in today's world. Imperative programming languages are more abundant and more programmers are coming out of college with only knowledge in languages like Java, C++, Python, JavaScript, etc. While it is true that these languages are more common place in most businesses today it would not be a bad thing to have a language like Haskell in your arsenal. Luckily Haskell has been around for quite some time so there are an abundant amount of tutorials, videos, and forums with extensive knowledge to further your skills at mastering this functional language. One of the best sources of information can actually be found at the Haskell home page. Along with videos the homepage has tabs where you can find courses or books, and get involved with the community via forums.

Imperative programming languages are more abundant and require more programmers to work on them in most businesses.

2 Haskell

2.1 Haskell Introduction

Learning a new programming language is almost always a steep learning curve. At first it is very difficult to understand and utilize all the nuances of the language. However, this is not the case with Haskell. Unlike every other interpreter language professors use here at Chapman, Haskell is vastly unique and takes a lot more time to fully understand how it works. For instance, Haskell is not an interpreter language, but actually a functional programming language. A bonus to being a functional programming language is the ability to change state whenever you desire. When you run a program in Java or C/C++ a variable must stay consistent when being used in a defined function. This is not the case with Haskell, because you could assign variable ‘x’ to equal 10 in one line then go and change it to say 1 in another line down the program. A topic that was discussed in lecture was the fact that Haskell is a “lazy” programming language. When executing a function you wrote an answer will not be displayed until specifically asked by the user. While typing less in a user created program will theoretically save you time, the adjustment period may prove challenging to experienced programmers. Some changes programmers would have to make when

transitioning from an interpreter language to Haskell is learning to unlearn the variable naming convention of most modern interpreters. When using Java, a programmer might type the expression ‘int x = 1+1’. In Haskell there is no need for the expression ‘int’. Haskell is smart enough to deduce a variable is supposed to have an integer value when an equation is opposite the equal sign.

2.1.1 Background of Haskell

After the creation of the programming language ”Miranda” the world wanted to start to venture more into lazy evaluation. The most critical problem with Miranda was the fact it was a proprietary software, not open source for the public to use. In order to utilize this groundbreaking programming software you would have to pay a nice fee to obtain the licence. Just a two year timeline is all it took for a committee to meet and come to the conclusion that an open sourced functional programming language was really needed. The FPCA committee, that met in Portland in 1987, came to the conclusion that a new language, that took into account problems discovered in other languages like Miranda, was to be created.[1] Most importantly they wanted this software to be free to the public so it was not monopolized by the one percent or big corporations with limitless wealth. Thus in a short time Haskell was born. The creation of Haskell was able to trump the short dynasty of Miranda by becoming an open standard functional programming language, allowing millions of users to suggest and reform properties.

With the idea of an open sourced programming language taking form, another ground breaking addition had to introduced. Flashback to around 1925 at Princeton University, Alonzo Church was attending the school. He was an incredible student and graduated in four years with a degree in Mathematics. After finishing his undergrad work Alonzo decided to stay at Princeton and work tirelessly on his post-graduate work. Eventually in around 1935 Alonzo published his work on the topic of Lambda Calculus. The basis for the paper was centered around ’Entscheidungsproblem’ a theoretically impossible mathematical and computer science problem. This problem asks for an input that will universally be yes or no. At the time of conception there was no clear answer and stumped researchers for years. While no true, correct answer to the problem exists, Church’s findings lead to his development of creating Lambda Calculus.

Enough time passed and the creation of computers started to take the world by storm. Early programming languages were effective but not up to par with mathematicians standards. Eventually programmers got the idea to utilize Church’s work in Lambda Calculus and implement it into a functional programming language. With many different languages existing throughout the world, issues revolving around cross platform inefficacy started to arise.

2.1.2 Haskell’s Primary Usage

In the year 2021, it is nearly impossible to find someone who has worked with Haskell and even more rare to find a college student who knows what the functional programming language is. Companies in the private sector have actually used Haskell in their past in some way or another, including Google, Facebook, Microsoft, and Intel, but not for a long period of time[2]. As technology progressed new software was developed and left Haskell in the past. However before its demise, the language was able to prove helpful to a lot more

[1] Denis Oleynikov, Gints Dreimanis: Haskell. History of a Community-Powered Language
[2] Fighting spam with Haskell

companies than just the ones listed above. Some overseas companies in the Netherlands and Germany used Haskell to handle all the accounting and financial aspects of business. Other companies were able to utilize the language for different uses as well. For instance The New York Times, around 2013, brought in a programmer named Erik to talk to an audience about the use of Haskell within the company.[3] Erik Hinton talked about the initial backlash he received when trying to implement Haskell in the company and his eventual accomplishment of developing projects with the language at The New York Times company.

Some might question what is the point of learning such an out dated and somewhat abandoned programming language in this day and age. While not many Haskell jobs exist in the world today the need for Haskell programmers is in high demand. Also having the extensive knowledge of a functional programming language will set you apart from other newly graduated computer scientists. Obviously knowing how to use imperative languages in today's world is helpful, having to understand Haskell can grant you a bigger advantage with life lessons and job experience before you enter the private sector.

2.2 Types in Haskell

2.2.1 Static Typing

As stated earlier, Haskell is a static type system. This means the code will run safer and errors are caught at the instance of compiling, not during execution of actual program. The static typing does not allow a program to run with an error such as an int divided by a string. Furthermore there is no need for variable declaration in Haskell. Comparatively to imperative programming languages, Haskell will require a lot less testing resulting in a decrease of trial and error. This is accomplished due to Haskell's ability to find problems at the source and not wait until compile time to warn the user. However some drawbacks do exist within a functional programming language like Haskell.

Disadvantages in Haskell

1. Mastering takes upwards of months of learning
2. More redundant lines of code
3. Writing and testing code can take longer than Imperative languages
4. Strictly available on computers; no phones can run Haskell
5. Initially difficult to navigate user interface

Every software or programming language will have its pros and cons so understanding its short comings is vital to overcoming the challenges presented. Haskell is not an easy step for most people, especially since it varies from most modern mainstream IDEs like Java and C++.

2.2.2 Typeclasses

A new feature to programmers who have familiarized themselves with object oriented programming is typeclasses. Languages like Java or Python have classes that act as objects, with their own properties These interfaces act as, well an interface, not so much like classes as Java users are comfortable with. These typeclasses act as a definition of a specified behavior and not objects. Haskell is equipped with default types but allows users to create their own if they so desire, more on that later. Some examples of common typeclasses are listed below with an example of each.

[3] Erik Hinton: Haskell in the Newsroom

Haskell Types by Default

- Bool - True/False
- Char - 'x'
- String - 'Haskell'
- Int - 500
- Double - 5.25864753201772342354689721
- Integer - 400250500000
- Float - 5.75

There are some important things to note. First, the Bool type represents a single bit of data True or False which can be broken down to the binary equivalent of 1/0. Bool is the smallest bit value possible inside Haskell. Second would be the difference between an Int and an Integer. Ints are integers but have a maximum input of 2147483647. Any natural numbers greater than that will have to be labeled as 'Integer' because they have no maximum. The ability to have no max does impact performance but not enough to cause failures while compiling. The char typeclass consists of any singular character. Meanwhile a string is just a sequence of chars. Strings will obviously take more performance than a char but a string is actually in the typeclass char. The last two differences to discuss would be floats and doubles. They interact in a similar way to the dynamic of Ints and Integers. Both types are decimal numbers but floats can have a maximum of 23 spaces after the decimal point. On the other hand doubles can hold over 23 places passed the decimal but no more than 52, the maximum amount of places a decimal value can be in Haskell PL. One more type not listed, but important to note, would be the Eq typeclass. This class is responsible for equality checking in the code. The way this class is represented is with double equal signs '==' or to check if something is not equals, you would type '/='. Additionally all these types must be used with the same type. Meaning if you wanted to add an Integer and a float then it would be ill typed and would not compute the solution. A workaround for this issue is to assign the integer as a float but just add '.0' to end of the number so the types can go through the function.

2.2.3 Type Inference

The interpreter has type inference as well. The auto detection is sophisticated enough to deduce which type is being used in what instance. Most experienced programmers have to tell the IDE what type a variable is when it is declared. One way of looking at it is in imperative programming languages a programmer is giving the computer a set of commands to execute in a specified order of operations. On the other hand functional programming languages only have expressions that will change state when evaluated by the interpreter. Some functions and print statements in modern languages serve as steps to get to the next point of the program. Haskell demands that every line that has a double colon or an equals sign will have a certain type. If you use the Haskell compiler GHC you will never have to worry about mistyping a variable because of the built in type inference and Haskell's educated guessing.

Now that type inference is understood, its time to see an example. Let's take a look at how Haskell varies from other languages, Java will be the contrasting language in this case.

```
--Setting Types in Java--
int x;
public timesTwo(){
    x = x + x;
    return x;
}
```

Java needs the user to tell it what variable types are before manipulating them. The programmer must declare 'x' an integer or else the compiler will have issues trying to figure out whether x is an integer, float, string, etc. This is not the problem with Haskell's type inference.

```haskell
--Setting Types in Haskell--
addN :: NN -> NN -> NN
addN O m = m
addN (S n) m = S (addN n m)
```

This example was taken directly from PL Assignment 1. Here our group was tasked with implementing a calculator application within Haskell. We were given a template arithmetic file we changed to have operations. The first line does not require a declaration for 'addN' on the left side of the colons because the other side infers the operation is adding natural numbers. We added this first line at the top level for added security that our add function will be taking in two sets of natural numbers and returning one natural number. This should only be done when working with functions, like in the example above, or one line expressions where no reference was made previously. The final line the code is indicating the computer takes the successor of the declared variable 'n'. This system of programming is different than most modern languages but is, in some ways, more efficient. As you can see, Haskell can require a lot less to complete the same sort of declaration.

2.3 Functions

2.3.1 Creating Functions

The process of creating a function is not much different than in C/C++ or Java. The way you start is by naming the function you are creating. I will name the function 'timesTwo' and the function will double the inputted number. In two short lines you can create a mathematical equation like the one listed below.

```haskell
timesTwo :: Integer -> Integer
timesTwo x = x + x
```

The first line of this function is the declaration. The function is called 'timesTwo' and it will take an Integer parameter and outputs an Integer as well. The next line is where we actually define the function. This lines purpose is to illustrate what is happening when we eventually call this function. Here timesTwo will take an integer parameter, add it to itself and return the value of the equation.

As you can see it is pretty simple. One thing to make note of is the use of if-else statements. In Java/C++ you could get away with using an if statement and not concluding it with an if-else/else clause. That is not the case with Haskell. If you utilize an if statement, then an else statement on the same level is required to catch all other exceptions. Now creating functions and understanding all the intricacies it has to offer are two completely different things. This introduction is not meant to overwhelm you but slowly establish a foundation to work off of.

2.3.2 Curried Functions

When creating functions in Haskell, the user created function requires only one parameter, no more and no less. That is the case with most functions but not all. In Haskell we have access to a unique trick called Curried Functions. These functions behave differently than regular define functions because they can take more than one parameter. In reality the functions are only taking in one parameter but the function is nested with other function calls. Each variable is in a specific sequence to complete the desired task. Below is an example of how to code a Curried Function and the command line inputs to execute it.

```
--Haskell Code--
maxNum :: (Ord x) => x -> x -> x

--Command Line Input--
maxNum 5 10 || (maxNum 5) 10
10
```

Initially the code is doing a very simple job. The arrows tell the machine to add an x into a function and then added again. Meanwhile in the command line there are a couple different ways to execute this function of finding the max number. The Parenthesis can almost be ignored in these types of functions because the white space and parenthesis act as an operation in itself.

2.3.3 Lists

Something you have created for a trip to the grocery store is present in Haskell. Much like modern languages have implemented, Haskell was ahead of its time when it utilized the list structure. This data structure is homogeneous and can hold a sequence of the same data type. For instance a list can be made up entirely of chars or entirely of Ints. The only restriction is you cannot have two different data types in one list. Lists are defined by square brackets '[]' and each value is separated by a comma.

```
let listEx = [1,2,3,4,5]
listEx
[1,2,3,4,5]
```

Here the 'let' phrase is in charge of naming the list that is being declared. Creating a list is as easy as naming it and inputting what values you desire into the brackets. As long as all the values are the same data type creating a list is quick and efficient way to organize and manipulate data. Another usefulness to list is the ability to combine with like data types. If you have two separate lists that are both Ints or both only contain chars, then you can combine the lists into one using the '++' operation. When combining chars or strings a space will be needed if constructing a phrase. In order to complete this you will need to add double quotes with a white space in between.

```
[1,2,3,4,5] ++ [6,7,8,9,10]
Output: [1,2,3,4,5,6,7,8,9,10]

[P,r,o,g,r,a,m,m,i,n,g] ++ " " ++ [L,a,n,g,u,a,g,e,s]
Output: Programming Languages
```

Lists inside Haskell are simple to create, manipulate and can be very useful to a programmer if utilized to the best of their ability. A fun key feature with lists that any programmer can use is indexing. Indexing can be done with a string but mainly inside a list to find a specific value. The way to index a value is very simple; First you enter the list, string, integer, etc. followed by double exclamation points, lastly followed by what index you want to search. An example of how to accomplish this is below.

```
let sampleList = [1,3,5,7,9]
sampleList !! 1
Output: 3
```

Indexing is a valuable skill to learn whether it be for finding certain data values or replacing/adding values into a list. As important as indexing is, there are quite a few more commands that can manipulate lists in Haskell. Here are some more to familiarize yourself with.

Haskell List Commands

- head [...] - returns the first element within a list
- last [...] - returns the last element in a list
- length [...] - outputs the length of a list based on how many elements are inside
- drop [...] - staring from index 0, start of list, drops the number of elements inputted from list
- maximum [...] - outputs the largest value within a list
- minimum [...] - outputs the smallest value within a list
- sum [...] - takes all integers in a list and returns the summation, cannot work with strings

Obviously this is not the full extent of all the commands possible but these are a great starting point if you are learning Haskell for the first time. One important aspect of lists to remember is it is impossible to have different types within the same list. If you start a list with a float then you must continue with floats and not use integers or strings. This is true with every predefined list created by Haskell. Where it differentiates is when you create your own custom list and can fill it with whatever attributes you need for that program, whether that be strings and integers, or floats, doubles, and booleans.

2.4 Custom Structure

Haskell is equipped with customization if you can adequately structure your program. Maneuvering the intricacies of Haskell will take time to understand so making mistakes is encouraged, because failing is the best teacher. The functional programming language has the option to create your own data types if your code requires more than the default data structures given.

2.4.1 Custom Data Types

Creating your own data type or structure can be done several ways. One of the simpler ways is to use the 'data' keyword. To set up this function, you would type in the command line 'data' followed by the type you want to create. An equals sign separates the function in to two parts. On the second half, the constraints are to be determined. These constraints can vary from only two values to the limit Haskell can handle. The way to add more values is to list them on the right side of the equals sign '=' and separate them with the 'or' function (pipe key for Java/Python programmers)[4].

```
data suit = Hearts | Diamonds | Clubs | Spades
data value = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13
data Card = (suit, value)
```

Here we are creating a data type and calling it Card. Before we can make a card, however, we need to use specific parameters that make up a deck. If you are not familiar with playing cards there are two specifications that make each one unique. Out of the fifty-two cards in a deck there are four suits and each suit has thirteen values (1-10, Jack, Queen, King). In this example I started by creating the two constraints that will be going into the card data type, suit and value. The Ace is denoted by integer one, two through ten are as printed on each card, and numbers eleven through thirteen coordinate to Jack through King. Lastly the suit is a user created data type that will equal one of four strings. We could group the cards in a list called deck, with a collection of the fifty-two 'Cards' but for simplicity's sake, this would do the job it is intended for. Now you have the ability to create

[4]Making Our Own Types and Typeclasses

2.4.2 Custom Itemized List

Let's take a look at what it looks like to create a profile of a college student here at Chapman. We'll start by defining a new type 'Student' and it will take inputs of characteristics of a student.

```
data Student = Student String String String String Float

first :: Person -> String
first (Person first _ _ _ _ ) = first

lastN :: Person -> String
lastN (Person _ lastn _ _ _ ) = lastn

standing :: Person -> String
standing (Person _ _ standing _ _ ) = standing

major :: Person -> String
major (Person _ _ _ major _ ) = major

gpa :: Person -> Float
gpa (Person _ _ _ _ gpa ) = gpa

Here the underscores '_' are serving as a place holder for this list's values
```

The new Student data type will display a first name, last name, academic standing, major, and grade point average. To better illustrate how to actually input the details we will just follow the outline of the data type Student inputted at the beginning. To begin we'll use the 'let' command already implemented in Haskell

```
let me = Student "Connor" "Cowher" "Senior" "Computer Science" 3.50
```

Here we make a reference to the Student data type and name it 'me'. I input the items and their correct types that correspond to the Student type, String String String String Float. With that command a student is created and we can pick and choose what to see. For instance if I were to type in 'me' into the command line, all 4 Strings and singular Float would be outputted on the screen. If this list were to hold more than one student and you wanted to find the Connor's major, you could have Haskell only return that item. All you would need to do is type in 'major' followed by 'me' and the interpreter would return Computer Science since we inputted that earlier.

2.5 Haskell Input and Output

2.5.1 Understanding Functional Language

As a seasoned Java programmer one of the hardest changes to make in Haskell is understanding the difference between a functional programming language and an imperative language. The biggest difference is when a programmer is compiling the finished program for testing. In languages like C/C++ or Python a functioning program will look like a series of instructions that allows the computer to complete a specific task or function. A well structured imperative language program with functions has lines that will have some sort of impact on the computer's memory. Where Haskell is different is the fact that lines in the program define a state rather than complete a task. It sounds confusing but after seeing a few examples and struggling through your first program will teach you a lot.

Java Hello World

```
public static void main(String[] args){
     System.out.println("Hello World!");
}
```

Haskell Hello World

```
main = putStrLn "Hello World!"
```

Here some glaring differences are shown. The first contrast between the two is the lack of ending syntax for Haskell. Java requires a semicolon after every line, unless you are defining a function, to indicate when a line is done and the interpreter can jump to the next step in the code. Another thing to note is the Java example of Hello World is as simple as you can make it, the current form is not proper writing structure but does the job introducing the syntax to new programmers. A second difference between an imperative language and a functional language is Haskell can accomplish the same thing as Java can but do it in fewer lines. This is beneficial to programmers because shorter files with less lines of code usually equate to less bugs and problems when compiling. One last distinction between the two softwares is how much control Java/C gives you versus the safety net that Haskell gives the user. There are a great number of possibilities of programs a user can create with imperative languages due to the freedom those languages grant. Having the freedom is great and all until you run into compiling problems and a program riddled with bugs and warnings.

2.5.2 Making a Program

So far the examples in this tutorial have shown you the basics and how to start creating a file for execution. As shown above Haskell can take the input 'main' at the beginning of the program. Main acts as the starting point of the program. Java requires a much longer main line where it contains all the function calls and ordering of the program you are running, Haskell does not have that capability because order does not matter.

A new and fun way to test your code in Haskell that differs from Java is that the interpreter does not have a built in compiler button to quickly test your program. A simple way to work around this inconvenience is to have a terminal open in another window and cc-ing to the folder with the Haskell file located in it. After making your way to the file the next thing you want to do is type in "gch –make fileName ". After completing this input an executable file should have made its way into the folder you're currently in. This file is the one that will be executed. If we did "gch –make helloworld.h" in the previous step, the only thing left to do is run it and test the output. A simple ./helloworld will run the code that was created in the last sections code.

```
--make helloworld.h
...
./helloworld
Hello World!
```

That is all it takes to compile and run your code for testing.

2.5.3 Understanding Input Output

Having a properly functioning program is one thing, but what if you wanted an application to interact with the user and have outputs for the programmer set? This is where a couple of useful commands come in to play. Luckily for new Haskell users, these I/O operations come default when you download Haskell so there is no need to create your own.

```
putStrLn
getLine
```
putStrLn is an Input/Output operation that takes in a String and displays it to the screen. If we are programming an application that requires user input we would use the putStrLn to prompt the user for any information required for the process to work. Some programmers might have used the keyword print in the past or a variation of it. putStrLn is the Haskell equivalent to print and will return a an empty line, perfect for someone to input a String or Int, whatever the program calls for. getLine works in the opposite way of putStrLn in that it will return the input the user types. This command will stop and wait for an input and only works when an outside force presses return on the terminal.

3 Programming Languages Theory

3.1 Lambda Calculus

Traveling back in history to around 1930 is where the theory of Lambda Calculus begins. As mentioned earlier and not for the last time is the person who helped develop the functional programming languages we utilize today, Alonzo Church. Church is the father of theory and has developed many tools for computing that are still in use to this day. One of the most notable theories he conceived is Lambda Calculus. In short, Lambda Calculus is a operations and rules that make function abstraction possible. Let's take a look at an example of an abstraction function written in Haskell

$\lambda a.a$

That is all there is to it. The first thing to break down about this simple input is the lambda symbol. Like in many other languages Haskell requires a symbol to denote the start of a function. We accomplish this by using lambda at the start of the line. After that you must define the parameters of the function and end the list with a '.'. The second a in the example is known as the body of the function. This part of the code will be returned when the function is called in the compiler. Some vocabulary to know is the variable in the example above is known as bound because the 'a' appears in both the parameters and the function body. If there were to only be one 'a' on one side of the '.' then the variable would be labeled as free. Bound and free are self explanatory but in short, a free variable indicates the function depends on the variable and a bound variable does not.

Haskell can utilize curried functions within lambda calculus when the function calls for more than one parameter. To illustrate lets take a look at a very basic addition function and use the parameters 2 and 4.

$$(\lambda a.\lambda b.\ a + b)2\ 4$$

Since 2 is the left most parameter, it will be substituted into the left most variable in the equation, the 'a' term.

$$(\lambda b.\ 2 + b)\ 4$$

That leaves us with 4. There is only one variable left in the equation so 4 will slide into where 'b' used to be.

$$(2 + 4)$$

After completing those two steps there are no more parameters left to distribute and the final step is to calculate the expression.

$$2 + 4 = 6$$

That is how to use lambda calculus with curried functions when two or more parameters exist outside the parenthesis. Not every mathematical equation can be done in four easy steps but visualizing how to complete it can help get programmers started on the right track

3.1.1 Applications of Lambda Calculus

Writing a lambda function is one thing to master and applying it to another function can be a difficult thing to learn. First of all, lambda functions are anonymous and do not have a name when being used. These functions differ from others because they do not take in variables. Instead they are nested by other functions. In every lambda function there exists at least 3 separate parts. They include the variable; logical value or string in the form of a parameter, application; applying function to an argument and lambda abstraction; definition of the function. In the example above the variable listed is the a after the '.'. The lambda abstraction is the whole entity. Lastly the application would be another $\lambda b.b$ following the other lambda abstraction.

3.1.2 Capture Avoidance Substitution

When the computer performs lambda calculus there are some cases when the variables inside the expression are required to change. When evaluating expressions with lambda calculus the usual route the computer takes is replacing variable names via recursion, more about that later. However in some cases there could be expressions that share the same variable name and would cause the evaluation to stay the same and not perform the operations. Obviously as a programmer you want to avoid this at all costs. To combat this situation, we will employ the help of Capture Avoidance Substitution. CAS is a very helpful strategy to utilize within Haskell to save your code when performing lambda calculus. The most important thing to remember is to switch up variable names when creating a function. For instance when I start a new expression, I name my first variable 'a' then 'b' all the way down the alphabet so no variable will repeat unless more than twenty six exist within that function. However I am not like every programmer and a large majority like to utilize variable names like 'x' and 'y'.

Incorrect Variable Naming

$$(\lambda a.\lambda b.a)\ b$$

$$(\lambda x.\lambda y.x)\ y$$

Here we have 2 identical lambda functions with varying variable names. Variable 'a' is the same as 'x' and 'b' is the same as 'y' in each example. Now if we were to try and evaluate these expressions something bad would occur. Both the 'y' and 'b' that exist outside of the parenthesis are free variables and completely different that the ones inside. Both sets of variables inside the parenthesis are bound to the lambda and therefore completely different than those that are free. If this expression were to be evaluated, the singular variable that exists at the end would end up being captured, and eventually bound to the lambda function. By simply renaming the free variables to something different we can avoid the headache of accidentally capturing the outside variable.

Correct Variable Naming

$$(\lambda a.\lambda b.a)\ c$$

$$(\lambda x.\lambda y.x)\ z$$

With one different keystroke the integrity of these lambda functions is saved and Capture Avoidance Substitution proves to be beneficial to know about. Obviously mistakes will happen and your code will fail on occasion but it is important to remember the strategies that help protect your code at different steps of assembly.

3.1.3 Church Numerals

Church numerals can be defined as the act of applying a function to a value, any number of times. The amount of times it is applied is added onto the end of 'church number n' where n denotes the number.

Furthermore they can also be described as the representation of natural numbers as functions. While church numerals serve as a representation of numbers in lambda calculus they can be used for all the major operations, addition, multiplication, subtraction, exponential but not division. Division of church numerals within Haskell do no perform the same way as the other operations so its best to stray away. A workaround would require utilizing an operation similar to division, but that would need to store the dividend and divisor along with all the possible quotients which would be less efficient due to the amount of memory it takes up. You would also have to figure out how to handle solutions like '1/2' which does not have a solution with the structure of natural numbers (floats would have to be used for regular and improper fractions)[5].Church numerals are the basis for computation with lambda calculus and deserve the proper time to learn and understand completely.

$$(\lambda fa.(f(f(f(fa)))))$$

This here is church number 4; meaning it is an action that applies a function, f, to a value, 'a', 4 times [6]. Why stop there? There is not a set limit for the amount of times you can apply a function to a value. One thing to take note of in the above example is the fact that the function does not take any parameter inputs. The overall construction of a church numeral is at least a function applied to a value. A specific use for church numerals could include utilizing lambda calculus to create a boolean logic.

$$(\lambda a.\ \lambda b.)\ a$$

$$(\lambda a.\ \lambda b.)\ b$$

Here we can see the representation of true and false. True will always return the 'a', first parameter input, and false will return the 'b' all other inputs that do not equal true.

3.2 Parsing Introduction

Parsing is a topic in Lambda Calculus that is similar to curried functions. The definition of a Parser is a component of an interpreter, which parses the source code of a programming language to create a form of internal representation. Parsing theory can be broken down into three separate stages to operate[7].

1. Lexical Analysis - Step 1; The analyzier is used to produce tokens from an input stream of string characters, then broken into small manageable components to form meaningful expressions. A token can be defined as the smallest unit in a programming language that still holds some meaning (such as operations like +, -, *, /; or a basic user created function)

2. Syntactic Analysis - Step 2; Analyzes the generated tokes to see if they create a meaningful expression. With assistance from context-free grammar, it utilizes the definition of algorithmic procedures for the small components. The components then work together to create an expression and establish a specific order in the way tokens must be inserted

3. Semantic Parsing - Step 3; After all that breaking down and ordering the semantic parsing sequence validates the expression that was created and determines the meaning and implications. Then the necessary actions are put into motion to complete the parsing sequence.

Furthermore if the data that is implemented inside the parser can be derived from the start of the grammar (rules) then we must differentiate between which way to derive the data.

[5] Martin Atelier: Church Numerals

[6] Programming Languages

[7] What is a Parser

1. Top-Down Parsing - Parsing starts at the start symbol, after it is transformed into an input symbol. All symbols are eventually transformed and a parse tree is built via the parameters of the input string. Then the Top-Down parser will search the the tree and look for the left-most derivations of the input string, through the method of top-down expansion. An alternative name to this type is referred to as recursive parsing.
2. Bottom-up Parsing - Alternatively to the previous method, bottom-up parsing goes in the reverse order by searching for the right-most derivation of an input string until the parse tree is fully complete. The input is rewritten back into the start symbol where this method will begin and the tree will finish at. Comparatively to its counterpart, bottom-up parsing is also known as shift-reduce parsing because a lot less movement is done when using this sequence.

However in order to fulling grasp this idea we need to learn about pattern matching and recursion.

3.2.1 Pattern matching

Pattern matching is syntax in Haskell that takes in a specific pattern and checks it with new data to see if it will be in accordance with the given pattern. The useful part is it can be utilized with any type of data Haskell comes default with. To make use of pattern matching you will have to define a function with the specific pattern. The purpose of pattern matching in Haskell is to help simplify your code and leave less room for errors. Creating a factorial function will help put this into a better perspective.

```
factorial :: Integer -> Integer

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

Here we have the function declaration again where the function factorial will input an Integer and return an Integer. Next we create a pattern for the function to follow. In a factorial a number is taken and multiplied by the number directly beneath it.That operation happens again and again until you reach 0. Obviously you do not want to do all those calculations just for your answer to be 0. So the first step in preventing that is to make sure when you get 0 as an input, its value is changed to one. Next you can get on to building the sequence. You would start by taking the parameter set by the user, 'a' in this example, and multiplying it by the value of 'a' minus one. That would be all we need to do if our application would only take in 2, 1, or zero. Obviously having a limit of three inputs would not be a productive use of code. To combat this, we can utilize recursion to save space and make our code run more efficiently. As touched upon in curried functions, the factorial function will call factorial within itself.

3.2.2 Recursion

Recursion is hopefully not a foreign topic to you as it is very useful within the inter workings of a functional programming language. The example above showed you the basics of recursion, keep doing an operation until a condition is met. As mentioned earlier recursion is the act of calling a function in the definition of that same function. When creating a recursive function you need to have an exit strategy or else the function will repeat forever and the computer must give up at some point. To prevent this from happening we utilize edge conditions to help the computer terminate the recursion when a specific condition is met. For instance lets take a look at the factorial function again.

```
factorial 0 = 1
--edge condition, cannot decrease less than 0

factorial a = a * factorial(a - 1)
--factorial is used to define its own definition
```

Here we place factorial 0 = 1 at the beginning of the function because when we use the factorial expression, you keep multiplying n - 1 until you reach 0. If you were to multiply the zero into the sequence then your function wouldn't work and zero would always be your answer. Let's take a look at a contrast factorial function in Java to see the differences.

Java Factorial example

```
int x = 5;
   public int factorial(x){
       if(x <= 0){
           return 1;
       }
       else{
           return (x * factorial(x-1));
       }
   }
```

Clearly Haskell can complete the same function in much less lines due to the fact it does not require if/else statements in this particular function. With the edge condition, all if/else statements are unnecessary because of pattern matching within Haskell. In addition there is no need to distinguish types in the Haskell definition because the interpreter can infer the function is an integer equation due to the variable 'a' and gradual decrease to a = 0.

A very under looked aspect of Haskell is the combination of recursion and pattern matching. With an imperative programming language like Java or C/C++ creating a factorial function requires a lot of lines and multiple if-else statements. Haskell does not have this issue because a simple pattern match tied together with a recursive function can quickly and efficiently create the factorial, or any other mathematical sequential function.

Functional programming languages like Haskell tend to perform recursion better than imperative languages. Haskell only allows pure functions, absent of side effects and state, resulting in more efficient and faster compiling times[8]. No state is required to use recursion so functional languages that allow pure code are best fit for this operation.

3.2.3 Parsing Continued

With a better understand of pattern matching and recursion we can dive deeper into the theory of parsing. Parsing is taking a concrete syntax and transforming it into an abstract syntax, meaning it will take in a input of characters and applies logic to conform to the software. A very important computer scientist who helped with multiple advancements including Parsing was Robert Floyd.

We now know a parser takes in an input and breaks it up into tokens for the computer to use for programmer. But how does one make a parser within the code. For starters, defining a parser can be done in a couple of ways in Haskell. Below is one way of creating a parser that will parse true values.

Parsing True Values

```
trueParser :: Parser bool
trueParser = Parser (\str ->
    if (isPrefixOf "True" str)
    then [(True, drop 4 str)]
    else []
    )
```

example source[9]

[8] Are functional languages better at recursion
[9] Haskell for All

If the parse is a success then we will know because the function will output one result, outputting true. If the parsing fails then an empty list will be returned and all hope is lost in finding the true value.

Implementing a parser can take a bit of time to understand how to do successfully. If you have worked in Java or Python then you may know about importing libraries and classes. Luckily Haskell has a similar import function you can utilize for parsing. This process is called staging a parser and is eerily similar to importing different objects in imperative languages[10]. In the main Haskell file you would type at the top the following...

```
import qualified Parser

...
```

3.3 Hoare Logic

Switching things up, a new computer scientist we have not discussed helped with programming theory around 1969. This scientist was named Tony Hoare and his contribution to computer programming is immortalized in a logical set of rules for configuring the correctness of a computer. To start, the foundation of Hoare logic begins with Hoare triple, as shown below.

```
{O} C {P}
```

In this Hoare triple, the execution changes the state of the computer. Both O and P are boolean expressions that indicate true, unless an unforeseen bug is in the code. The C denotes a command statement between the two expressions. In this example O is considered the precondition since it occurs before the command and P would represent the post condition. To further explain, when the O boolean condition is adhered to, then the command section will execute resulting in the typed P, post condition.

Empty Command Logic

```
{O} Skip {O}
```

This very similar structure of Hoare logic states that if no command separates the precondition from the post condition then the computers state stays the same. In short there is no post condition and everything true before the statement remains true after it as well.

3.4 Monads

A new structure being introduced is a monad. Monads are already implemented in Haskell with Input Output and lists. Although the term does not explicitly explain what they do, monads are designed to help make up strategies for problems that occur often so the workflow of your code is consistent and streamlined. At first this is a difficult subject of theory to understand considering there are countless websites and YouTube videos attempting to explain how they work. Haskell luckily has I/O monads, a subject already touched upon in this tutorial. With abilities to check if files are present on a machine to deleting files, I/O monads can prove to be very helpful to programmers who use them.

Default I/O Monad within Haskell

```
removeFile :: \Users\CCow\CPSCCourses\CPSC354\test.hs -> IO ()

--Here we do not care about an output because the removeFile function is running through the
    machine to delete the file via the given path
```

[10]Haskell in Haskell: Adding a new Stage

```
doesFileExist :: \Users\CCow\CPSCCourses\CPSC354\test.hs -> IO Bool
True
--Alternatively here we have a boolean that returns true or false given if a file is in
    existence via the file path location

--Additionally the file paths shown here are just examples of how to format the file path to
    check if a file exists and/or to remove a file, they will all be different on every machine
```

For instance if a programmer attaches an I/O value to a created function, the function will execute commands based solely on the files already present or inputs from the command line. After which a monad value is returned in the form of a system output, displayed on the screen. A way of creating your own monad is with the double greater than symbol, >> on keyboard. Typically when using I/O types in Haskell the programmer would name a value and have a pointer assigned to getLine or putStrLn. However we can now reverse the roles and make the I/O devices do more work. Let's see an example of welcoming a new user into a random database.

```
main :: IO
main = putStrLn "Welcome, please enter your username"
    getLine >>= username
    putStrLn "Hello " ++ username ++ ", the menu screen will load now"
    ...
```

Monad are a tricky structure to understand in Haskell. The comparison that helps draw the points together is when you program in an imperative language, you have to use semicolons to denote where a line is finished. Monads are in a way, a programmable semicolon for functional languages where the user does not have to put in the work for entering the combination manually. The monad will automatically determine how the combined computations translate into a new state of computation[11]. Monad are statically typed and work to free up time and energy for the user who can best utilize the structure.

As mentioned earlier there are numerous monads already loaded inside Haskell. Some of which are better suited in conjunction with a separate parser library. With monad classes existing in libraries, Monad and MonadPlus (super useful functions), there does not exist a short coming of usages for monads within Haskell[12]. Below is a short list of few standard monads found in these libraries and their usage to further your understanding.

Standard Monad Examples

1. Maybe - Computation that may or may not return a result
2. Reader - Computation that reads from a shared environment
3. Writer - Computation that writes data while evaluating values
4. IO - Computation that perform Input Output
5. State - Computation that maintain the current or specified state
6. Cont - Computation that can be interrupted and/or restarted

3.5 Theory Conclusion

Obviously these previous sections do not come close to covering all the theoretical aspects of Haskell but they act as a introduction to what Haskell is capable of.

[11] All about Monads
[12] All about Monads

4 Project

Projects in Haskell are much more intensive and require more time to accomplish the task you wish to complete. The biggest piece of advice I suggest would be it is okay to not get it immediately. Haskell is great for new open minded individuals but experienced programmers might struggle to switch gears into this new world. Baseball players struggle to hit golf balls because the swing is so different and the exact same concept can be applied here. Starting with a smaller project like Hello World! is always a great way to introduce a new language to a programmer. However after doing the same thing over and over and only changing a few syntactical aspects, the introduction application tends to leave a lot to the imagination. Instead of revisiting the age old Hello World! I would like to take a different route and show you something completely different and hopefully new. In Programming Languages at Chapman University we started using Haskell by tinkering with a calculator application. Today I will show you a mini project that is vastly different than applying mathematics to a grammar file. This application has no real world benefits except for the fact that it is fun to play.

4.1 Background Information

The act of gambling is such a thrill when you hit it big. That is why so many people invest in trips to Vegas for the chance of making a profit and heading back home with a couple more dollar bills in their pockets. However this is not the reality for a majority of people who actively gamble their money away. With some inspiration from GitHub[13] I was able to rework a blackjack simulator but without the betting aspect.

4.2 Project Code

```
--BlackJack game where you test your luck to obtain 21 with only two cards import
    System.Random

 --deck creation seperated into hand value and card face values
 data value = Ace | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten |
 Jack | Queen | King deriving(Show, Eq, Enum)
 data Card = (value)

 type Deck = [value]
 type Hand = [value]

 --4 sets of suits so Ace-King repeats 4 times for full deck of cards
 deckOfCards :: Deck
 deckOfCards = [Ace,Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,Jack,Queen,King]++
              [Ace,Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,Jack,Queen,King]++
              [Ace,Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,Jack,Queen,King]++
              [Ace,Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,Jack,Queen,King]

 cardNum :: value -> [Int]
 cardNum Ace = [1, 11]
 cardNum Two = [2]
 cardNum Three = [3]
 cardNum Four = [4]
 cardNum Five = [5]
 cardNum Six = [6]
 cardNum Seven = [7]
 cardNum Eight = [8]
 cardNum Nine = [9]
```

[13] Wilfred Hughes: BlackJack

```haskell
cardNum Ten = [10]
cardNum Jack = [10]
cardNum Queen = [10]
cardNum King = [10]

--players receives 2 cards for hand and those cards are taken out of the deck
deal :: Int -> Hand -> deckOfCards
deal = (take 2 deckOfCards, drop 2 deckOfCards)

--checking if initial hand is black jack, (Ace + (10 | Jack | Queen | King))
blackJack :: Hand -> bool
blackJack [firstCard, secondCard] =
((firstCard == Ace) && (secondCard == 10 || Jack || Queen || King)) ||
((secondCard == Ace) && (firstCard == 10 || Jack || Queen || King))

blackJack _ = False

--check if deal granted BlackJack
winner :: Hand -> Int -> Str
winner hand
  | blackJack = "Winner you got BlackJack!"
  | otherwise = "Sorry Try again."

--Gameplay interface start
main :: IO
main = do
  putStrLn "Welcome to BlackJack, enter p to play"
  winner <- getLine >>= readIO
  putStrLn "Thanks for playing!"
```

4.3 Rational

As an avid small betting gambler I was inspired to pursue this project for a number of different reasons. First of all I am minoring in game development and have made numerous text based adventure games for school projects. The idea of creating a card based game came to me because professors and teachers in my past have always made similar projects and I wanted to explore something different. The game of black jack is very simple to learn and can have countless different games because a deck of cards has 52 factorial unique combinations. With a number so ridiculously high there is no possible way any two black jack games will be the same especially with a random shuffle implemented at the beginning of every game. Gambling can be fun but if you lose all your money, in real life and virtually, then you will not be enjoying yourself. That is why I made the conscious decision to not include betting in this project. Additionally I wanted anyone to be able to play, not just those who know how the game is played because only gamblers truly understand how to manage your money and possibly win against the house.

I chose to demonstrate this project because the Haskell blackjack application has a lot of the topics we discussed in this tutorial. From type inference to Input Output there are a bunch of different aspects to take away from this example. This type of application can also be altered to do way more things like adding in a betting feature where more risk is involved. Another aspect that could be reworked is adding a player 2 or AI to play against. Although it may seem daunting at first all that would be required is making a new data type. Here is how you would create these type of features for this specific project.

```haskell
--Creating the player 2 model and dealer for our program
   data opponent = player2 |
                   dealerBot
```

```
--optional bid typeclass that could add more risk to game
type bid = float
```

4.4 Project Conclusion

I really wanted to show you how easy it is to start your own Haskell project even if you have little idea what you want to do. There are endless possibilities of programs you can create with this functional programming language. Although creating applications that utilize mathematical functions is easier to work through, it is not a requirement to have lambda calculus or other theoretical aspects throughout your code. Although Haskell is definitely the language best suited to use if your application requires those types of aspects. Some of the most important takeaways from black jack is any application you make is limited to your imagination. As long as you can visualize a program you want to make, Haskell can accomplish this for you with enough time and effort. With an extensive default library all the functions and data types you need will be available for you to use. However if you do find yourself needing a different data type or typeclass, Haskell also has the ability to let you use creative freedom. Haskell is challenging and applications in this language are very unique to the programming world. Transferring the code into a different software would take less time to start over from scratch and use the Haskell code as a template.

5 Conclusions

So in conclusion we have gone over quite a bit of subjects regarding Haskell and its basics to understand completely before undertaking an assignment of your own. While having previous knowledge and experience is usually a benefit to your cause, Haskell might put that notion to the test. Something as simple as adding positive numbers can cause so many problems to programmers' complicated structured brains. To fully understand and master this functional programming language will take lots of hours and practice. Haskell is vastly different from most languages used throughout the industry today but can prove beneficial if you know how to use it. This tutorial has covered a large number of different aspects of Haskell and is completely expected that you do not fully understand all the material after completing this guide. However I hope you have a better understanding of how functional programming languages work and feel confident enough to give Haskell a shot for your next language to learn. While not many companies utilize Haskell as their primary software anymore it is definitely worth adding to your knowledge of languages. The need for programmers who understand an outdated language is always present and could be the deciding factor in whether or not you get a call back for a job. There has been a great deal of comparing and contrasting Haskell to other imperative languages like Java, Python, and C++. In the grand scheme of things it is almost irrelevant how they differ because Haskell is great for some aspects of programming and not so much in other ares. The language itself is not better than another, it all factors into what the programmer is trying to accomplish and which programming language is best suited to help accomplish this goal. Any programming language out there, JavaScript, HTML, R, and even Swift can all do the required job at hand. The decision to use which one is all based on how comfortable a programmer feels using that language and how well versed they are at navigating the intricacies. Variables within Haskell are safer than other imperative languages because of the naming syntax and Capture Avoidance Substitution. While you could forget about a program in Haskell for years upon years and jump back into it without any problem, the imperative languages do not have this luxury. With varying naming syntax-es in other softwares problems can occur when differing variable names are used to assign values to functions. Thankfully Haskell has the upper hand in some of these areas that separate it from its competitors.

References

[serokell.io] History of a Community-Powered Language [engineering.fb.com] Fighting spam with Haskell

[infoq.com] Haskell in the Newsroom

[learnyouahaskell.com] Making Our Own Types and Typeclasses

[mjoldfield.com] Martin Atelier: Church Numerals

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

[technopedia.com] What is a Parser

[softwareengineering.stackexchange.com] Are Functional Languages Better at Recursion

[haskellforall.com] Haskell for All

[cronokirby.com] Haskell in Haskell: Adding a new Stage

[]

[wiki.haskell.org] All About Monads

[github.com/Wilfred/Blackjack] Wilfred Hughes: BlackJack