

CPSC-354 Report

Joan Karstrom
Chapman University

December 22, 2021

Abstract

In CPSC 354, our class objective is to “have a look under the hood of programming languages.” Unlike other computer science courses, we are not focusing on learning a specific language, building huge repositories, and handling data. We are essentially comprehending *what* is a programming language and what do developers think of when they first are thinking of when designing a language.

Most of the time, we believe that the just syntax looks different, but there are so many more things to consider when choosing a language to program in. What hustles and benefits does the language have to offer you. In my report, I dive into Haskell, a purely functional programming language that is great for breaking down the blocks and simply showing the foundations of a language. Instead of creating programs with methods telling the computer what to do, Haskell allows you to say to the computer what *is*. Some theories of programming languages are also gone over in this report as we expand more of the grammar of languages.

Contents

1	Introduction	2
1.1	Haskell	2
1.2	Programming Languages Theory	2
1.3	Project	2
2	Haskell	3
2.1	History of Haskell	3
2.2	First Understanding	3
2.3	First Downloading Haskell	4
2.4	Coding in Haskell	4
2.5	Lazy vs. Non-Strict Programming	7
2.6	Recursive	9
2.7	Python V. Haskell	10
3	Programming Languages Theory	11
3.1	Parsing	11
3.2	Grammar	13
3.3	Lambda Calculus	14
3.4	Church encoding	19
3.5	In-Class Application	20

4	Project	20
4.1	Correcting Assignment 1	20
4.2	Creating an Interface	21
4.3	Issues and Problems	22
4.4	Links to Repositories	22
5	Conclusions	23
5.1	Take Away	23

1 Introduction

This report is broken up into sections to better understand the material and better map how these topics are related. Overleaf was used on this report to format everything together, all cite hyperlinks should be live and working.

1.1 Haskell

Haskell is a unique language and is very influential in understanding programming languages and the grammar of these languages that makes them so different.

As a student learning Haskell for the first time, a lot can be said about how one learns Haskell and how one can be successful in it.

In the first section of this report names "Haskell", the basis of Haskell are covered, including a tutorial on what is needed to run Haskell and how to code in the language.

It also goes over a description of Haskell and what makes it different than other programming languages. And taking a look at a deeper understanding of why Haskell is formatted the way it is.

1.2 Programming Languages Theory

In this section of the report, more is discussed about the building blocks of a programming language. Haskell is a great language to learn more about this concept by creating programs with custom grammar.

Creating a programming language's grammar can limit and also expand a language.

Parsing is an idea that is introduced when speaking about the grammar of a language which is explained further. In this class we created our own language based on Lambda Calculus.

1.3 Project

The last section of this report is the project. For my project, I looked at the Calculator I created as Assignment 1. I attempted to make an interface for the said application and also correcting past errors from the assignment.

I go more in depth in how I fixed the errors that I had with Assignment 1 and what is needed to further expand on the project.

I talk about the difficulties that I ran into for this assignment but also what I learned from the online interface creator for Haskell applications called IHP. This created a real time web page for the application.

2 Haskell

Haskell is the base language that we were taught in this course. For a purely functional programming languages there are a lot of benefits.

Haskell is completely free and can run on windows, macOS, and Linux.

It has is very readable this is because of being Statically typed. I would compare it to python in some sense which I will expand on later in my report.

It is also called a "purely functional" programming language which means it the functions in Haskell are all functional in the mathematical sense.[\[HHP\]](#).

Haskell's code is concise, and its syntax makes it easy to understand what you are coding, which means fewer mistakes.

2.1 History of Haskell

Haskell is a functional programming language that is used to learn and understand programming languages as it is used widely for research.

Before Haskell there was LISP which stood for List Processing which was one of the first high-level programming languages ever created.

The development of this functional language started in 1987 by a 25 person committee formed by John Hughes, Philip Walder, Kevin Hammond, John Launchbury, and Simon Peyton Jones of people who attended the FPCA '87 conference. The "Haskell Report" was published in 1991 and since then has grown into a widely respected language [\[HP\]](#).

Glasgow Haskell Compiler is the most common Haskell compiler, it is named after the logician Haskell Curry.

Haskell was created with the intent of research and teaching functional programming. It was a commonly used 'open sourced' language so that the community could be more together when conducting research. [\[BHH\]](#)

There are only two official versions of Haskell, its most current version is Haskell 2011 but as of 2016 a group is working on building Haskell 2020.

2.2 First Understanding

Haskell was tough for me to understand at first. When first learning coding you are taught to use python, Java or CPP which are all syntactically similar. Haskell is a bit different from that and uses many recursions and theories that use discrete mathematics.

When first learning Haskell I had to look up many tutorials to help me understand. Some tutorials that really helped me at first to start coding in Haskell was the website/blog "Learn You a Haskell" [\[LH\]](#).

When looking up outside resources to try and understand, I found that there were a lot of blogs teaching Haskell as it is not a very popular programming language. Reading people's own experience with the language helped ease my frustration with not understanding it at first, which seemed to be a common theme when first learning Haskell because of how different the language is.

I believe that this is more helpful than a book in the ways that it is explained in a way that I could better understand. It is less formal than a textbook, and sometimes these blogs would talk about how they first saw things and how they learned differently.

2.3 First Downloading Haskell

Just like Python or Java, Haskell is saved and compiled as files.

Haskell files are saved as a .hs file and in order to run these you first must download a few things.

I will be explaining my experience as using Haskell on a MacBook Pro running macOS Monterey. First, you must download the Haskell platform by simply copying and pasting the following into your terminal.

```
curl -sSL https://get.haskellstack.org/ | sh
```

It will prompt you with a few questions on download options but will tell you when the download is complete. Then to open the Haskell Language Shell you would type

```
stack exec ghci
```

This will allow you to run simple .hs files as GHCi is an interactive console of the Glasgow Haskell Compiler which was used in our Programming Language Course.[\[PLIN\]](#)

Below is how it looks like on my terminal when I start the ghci shell.



```
joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

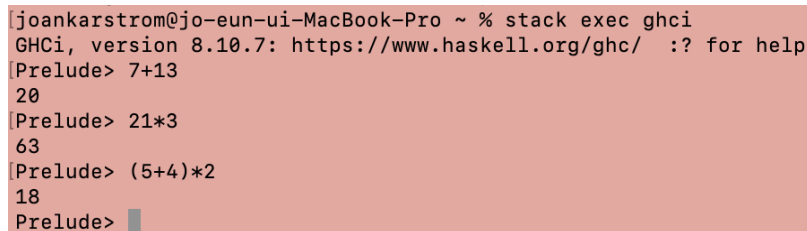
Figure 1: Starting GHCi

In this course, we also downloaded and installed BNF Converter, a compiler construction tool that generates a front-end compiler. It allowed us to closely study the grammar of a language and also assist us in creating our own language, which will be explained later in this report.

2.4 Coding in Haskell

Haskell is a highly intelligent language; if you were to open the ghci in your terminal, there are a lot of things you could do without even coding an entire .hs file. One can simply type simple math equations into the terminal's command line it would be able to compute.

Below is what my terminal looks like when I am testing this.



```
joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> 7+13
20
Prelude> 21*3
63
Prelude> (5+4)*2
18
Prelude>
```

Figure 2: Testing Addition on Terminal using GHCi

Similar to numbers, you can type strings into ghci. Strings are put in double quotes like "this" and can also be added together with ++ to create a larger string.

```
joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> "Hello"
"Hello"
Prelude> "Hello" ++ "World"
"HelloWorld"
Prelude> 
```

Figure 3: Testing Strings on Terminal using GHCI

One more thing one can do in the ghci terminal is to call functions and write arguments into them. To call a function one write the argument right after the wanted function.

This how it would look in the terminal.

```
joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> round 7.43
7
Prelude> sqrt 81
9.0
Prelude> 
```

Figure 4: Testing Functions on Terminal using GHCI

Lets move on to coding a Haskell file. The first thing a programmer is taught to code is Hello World. In an .hs file, the first line would look like

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

For Haskell to run, it needs a main function. Building on that, we have the function putStrLn, which takes a string and displays it on the screen instead of print, which calls show on its input first. This takes out "Hello World" and outputs it to the terminal.

Below is what the output looks like using an online Haskell Compiler.

```
$ghc -O2 --make *.hs -o main -threaded -rtsopts
[1 of 1] Compiling Main                ( main.hs, main.o )
Linking main ...
$main
Hello World
```

Figure 5: Hello World output

If you were to run this through terminal, you would first compile it in the command line. For this example, let's assume that the name of our haskell file is test.hs.

We would write in the terminal to compile

```
ghc test.hs
```

Once it has been compiled, you may run it in the terminal by writing

```
./test
```

With this, it should create the same output in the terminal as the online compiler as stated above. It is more steps than if you wanted to write something straight into the ghci, but once one creates more significant projects, it is necessary.

As stated earlier, Haskell is modular, meaning it is a series of functions. So how it tackles functions is a lot different than other languages. To define a function, you must also specify an input and output, which is to be expected, but the way it is written can be confusing.

For example let's write a function that takes in two integers and adds them together.

```
add :: Int -> Int -> Int
```

The name of our function, which is what we are trying to have it do is added. The first two Int represents the two arguments that the function is taken. The last Int represents what the function is going to return once it computes.

The following line would include the actions of the function which in this case would be adding the two inputs together.

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

This function was straightforward as it just took in single variables and returned them.

Lists are treated a bit differently in Haskell. When declaring a list as an input into a function you must declare the type of list.

For example this is what the declaration of a function addListLength which takes two lists and brings back an int that is the length of both the lists.

```
addListLength :: [Int] -> [Int] -> Int
```

There are a lot of built in functions that are available but you must remember that Haskell uses recursion in functions.

Two fast operations used in most functions that parse through a list are "head" and "tail". The reason that these are commonly used is because of their fast run times.

"Head" means getting the first element of the list and "tail" removing the first element in the list.

Just like earlier how it was discussed how simple functions can be used in the ghci terminal the same is for lists.

The keyword "let" can be used to name lists and variables alike in the terminal. You can also call them back in the same terminal session.

This is what it looks like in the terminal if one named and defined a list called test list and called it.

```
[joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
[Prelude> let testList = [2,4,5,8,9,12]
[Prelude> testList
[2,4,5,8,9,12]
Prelude> ]
```

Figure 6: Naming Lists on Terminal using GHCI

You can also add lists together just like integers and strings. The list being added will be added behind the original list.

```
[joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
[Prelude> [2,3,5,7] ++ [4,6,9,14]
[2,3,5,7,4,6,9,14]
[Prelude> ['H','E','L','L','O'] ++ ['W','O','R','L','D']
"HELLOWORLD"
Prelude> ]
```

Figure 7: Adding Lists on Terminal using GHCI

This "++" adds the two lists together although it does not sort them, and it just adds the second list to the first one.

The second input in the command line shows a list of char's being added together to say "HelloWorld."

When a programmer thinks of a list they would default thinking of a list to look like this.

```
[1,2,3,4]
```

In Haskell they are treated and look like this.

```
1:2:3:4: []
```

More things that are shown above is how variables can be added to lists using ':'. Using '!!' the variable at a certain spot. Lastly, lists can also hold lists, and they don't have to be the same size to be in the same list.

2.5 Lazy vs. Non-Strict Programming

One of the biggest things That Haskell is often described as is "Lazy," but it is not necessarily "lazy". Technically Haskell is non-strict. Haskell is one of the very few functional programming languages that default to a "lazy" way of evaluating, while other languages have this option but are not widely used.

```

[joankarstrom@jo-eun-ui-MacBook-Pro ~ % stack exec ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
[Prelude> 'H' : "ello World"
"Hello World"
[Prelude> "Hello World" !! 8
'r'
[Prelude> let listTest = [[1,0],[2,2],[3,3,3],[4,4,5,4]]
[Prelude> listTest
[[1,0],[2,2],[3,3,3],[4,4,5,4]]

```

Figure 8: Appending Lists on Terminal using GHCi

Lazy is a charming way of saying that the compiler waits and procrastinates to evaluate the expression [LP]. This means that specific values or projects are not considered until they are called upon or used.

Strict evaluation is used in almost every other language, which means that it is computed even if the expression is not needed.

Non-Strict in an algebraic sense, when you are assessing an argument, it works from the outside in, not the other way around. [NS]

In math, we are taught to work our way out as it is how to evaluate expressions using order of operations or “P. (parentheses) E. (exponents) M. (multiplication) D. (division) A. (addition) S. (subtraction).”

Lets take the equation

$$(a + (b - c))$$

First, we are going to evaluate this as we would in math class. Parentheses are first, which means we must complete what is in the innermost parentheses first so that we would calculate the expression

$$(b - c)$$

. For the sake of this example, we will say “b - c = d.” So the equation would now look like

$$(a + (d))$$

Since we have completed everything in the parenthesis, we can now add d to a, which would give us our final answer.

How lets compare this to what we would see in Haskell.

$$(a + (b - c))$$

In the non-strict schematic, an expression can have value even if some of its subexpressions do not.

In Haskell, you would first reduce the “+” and then you would reduce the inner

$$(b - c)$$

Now, as you can see, it works almost the complete opposite.

The reason that Haskell is “non-strict” is because of how it is used to evaluate expressions. Let’s say you have an expression that evaluates to “bottom”; has an infinite loop or error; if you were a programming language that worked in to out, you would always find that “bottom” but because Haskell or a “Non-Strict” language works out to in some of these are not evaluated. [LNS]

Though some people use it interchangeably, “Lazy” and “Non-Strict” do not mean the same thing as similar as they are.

For example, Haskell has some cases where it is not lazy at all, for instance, pattern matching. You can make this lazy but usually not in practice, since you are actively looking for matches.

2.6 Recursive

Recursion is most commonly used when talking about mathematics and means "the repeated application of a recursive procedure or definition." [\[RR\]](#)

In programming case its definition is "a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first", basically it is used to describe a function that calls upon its self.

Haskell is a perfect language to write recursive expressions and is sometimes the only way to write them. This is because there is no such thing as while loops or for loops in this language, which seems weird because we use these things all the time when programming.

The most common example of recursion is factorial.

In Haskell, this is what that function would look.

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
```

To show this, we are going to write a minimum function [\[MF\]](#) in Haskell. We will have it take in a list of things assuming that it can be ordered (using the Ord class) and return the lowest value.

First, we are going to start by importing the list.

```
minimum' :: (Ord a) => [a] -> a
```

In this next part, we must take in account if an empty list is passed, so we take the empty value `minimum' []` and we return the error "minimum of empty list".

The next thing we write is telling the program if there is only one value in a list, return that value since that would be the minimum number in that list!

```
minimum' :: (Ord a) => [a] -> a
minimum' [] = error "minimum of empty list"
minimum' [x] = x
```

Now its time to implement the function that minimum will execute. So in this function, if the value that is passed through is less than the minimum, then x will be the new minimum; if not, then "cur" will be the minimum value.

```
minimum' (x:xs)
  | x < min = x
  | cur = min
  where min = minimum' xs
```

Putting this all together, we would have the function: [\[MF\]](#)

```
-- run the transition function on a word and a state
minimum' :: (Ord a) => [a] -> a
minimum' [] = error "minimum of empty list"
minimum' [x] = x
minimum' (x:xs)
    | x < min = x
    | cur = min
    where min = minimum' xs
```

Now, this was a great example of how to use recursion to "loop" through information whereas in python or Java, one would just use a while for if then else function.

2.7 Python V. Haskell

At Chapman, the first computer science class that you take is taught in Python. In my opinion, I believe Python is one of the easiest programming languages to learn and is a great way to start introducing new learners of coding to logic.

Python, visually, is straightforward to read, for example unlike Java which requires methods and functions to be enclosed with " " and almost every line requires a ";" at the end.

For example, let's take a look at the bubble sorting algorithm in Python and in Haskell. Before we dive into the algorithm, let's explain what the bubble sorting algorithm is. It is a sorting algorithm that swaps adjacent numbers if they are in the wrong order [BSS].

First, let's take a look at the python version of the bubble sorting algorithm.

```
def bubbleSort(arr):
    n = len(arr)

    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

# code to test above
arr = [32, 54, 44, 23, 29, 49, 63]

bubbleSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i]),
```

[BSP]

We can break this down line for line. It starts off with defining the method and getting the length of the array setting it to n. They then have 2 for loops that loop through the array and then slowly sort through.

While it loops through the array it looks at the current spot compared to the number to the right and if that number is bigger or smaller, the number is moved respectfully.

Now lets look at the bubble sort algorithm in Haskell.

```
bSort [] = []
bSort x = case new == x of
```

```

    True -> x
    False -> bSort new
where
    new = bSortInner x

    bSortInner [] = []
    bSortInner [x] = [x]
    bSortInner (x : y : xs)
        | y <= x = y : bSortInner (x : xs)
        | otherwise = x : bSortInner (y : xs)

```

[BSH]

We first start with our type signature in this case is `bSort` (bubble sort). It takes in a list and returns the list sorted (Assuming that the list is orderable). From there we take our base case which is an empty list and the next case is our exit condition.

`bSortInner` does the work in this sort algorithm, once we have simplified it down to one single case we can implement the sorting part.

We now have two inner cases.

```

y <= x
y > x

```

That is organized into the code along with the `otherwise` piece of the code.

Both of these are the bubble sorting algorithm but they are implemented in different ways due to the different programming languages. Not only that but because of how the languages are built.

In the next section of this report more will be said about the grammar of programs and how because they differ they also act differently.

3 Programming Languages Theory

In this course we have gone over multiple theories and how they pertain to programming languages.

Specifically we have spent a decent amount of time learning lambda calculus and how to evaluate functions with it.

Although I will probably never try and build a programming language from scratch, it is still very beneficial to learn the logic on how grammar is used in the language and helps bring another layer of understanding behind choosing a language to program in.

3.1 Parsing

The definition of parsing is "analyze (a string or text) into logical syntactic components, typically in order to test conform-ability to a logical grammar." [PAR] During the programming languages class we have talked about grammar as it being a way people literally create and build programming languages.

In English we use grammar to create sentences with meaning. We have nouns, verbs, adjectives ect. to better explain to others what we are trying to say. With better communication, the faster and more efficient work can be done.

It is the same in programming languages. Different languages have different syntax or ways to write similar things and parsing taking the grammar to read a language.

The most popular languages that programmers use when assigning a variable use "=" as opposed to ":=" in languages such as Ada or Maple, or even "def" in Postscript. Although this is more of a syntax opinion than a grammar.

```
minimum' :: (Ord a) => [a] -> a
```

The grammar is the building blocks of a language and can determine the limitations of that language or the weaknesses of it. What parsing does is it reads a grammar file looking for patterns.

It reads word for word looking at what is a variable and what is an expression and what the relationship between the two are.

Earlier in this report I compared Haskell to Python the programming language. Every programming language has a grammar to it but it is not necessary to parse it as the most popular languages have emulator such as docker to run the languages.

The grammar used in this class was not too difficult or big so it took a small amount of time to run and generate a parser but that is not the case with every language.

Below is a segment from Python's documented grammar.

```
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | '(' single_target ')'
      | single_subscript_attribute_target ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)
augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

global_stmt: 'global' ' ', '.NAME+'
nonlocal_stmt: 'nonlocal' ' ', '.NAME+'
```

[\[PYT\]](#)

As you can see above, this portion of the code is defining assignments. The actual grammar file is a lot bigger and can be seen at the sited website.

Python is not the only language that has this, as stated before, every programming language has some sort of grammar.

Listed below are some quick links pertaining to the grammar of some of the most popular programming language's grammar.

1. [Java's Grammar](#)
2. [Full Proof that C++ Grammar is Undecidable](#)
3. [Lexical structure of C sharp](#)

3.2 Grammar

As Grammar makes up the foundation of a language, there are two ways that one's grammar could be defined as "ambiguous" or "Non-ambiguous".

Ambiguous or "Context Free" grammar is grammar that when an input has multiple trees. When first learning about Haskell one learns how the language reads a statement and how it takes that apart. A tree is a visual way to break down the variables and arguments in the way that the program will read it.

In the example of ambiguous grammar when a tree is created, there are actually two trees that are created. One reading the input from left to right and the other one being read from right to left.

Below this is how the trees will look like if we were evaluating the expressions

$$2 + 3 + 4$$

In my tree I define the mathematics function being applied and the type of variable that is being affected.

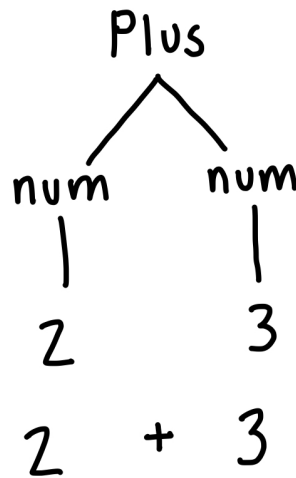


Figure 9: 2+3+4 Ambiguous Parsing Tree

On the bottom of each tree you can see that when you read down the tree, they are the same expression that both output a 9 but the trees look different. The part that makes this grammar ambiguous is that using the same grammar rules there were 2 different trees that were produced.

This is a very simple example but it could become a lot more complicated when you add multiple math expressions and different parenthesis. Below is an example of grammar that is "non-ambiguous". ** Not correct image right now

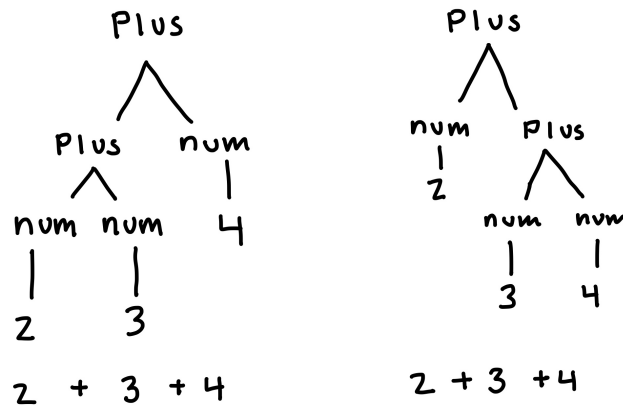


Figure 10: Non-Ambiguous Parsing Tree

I think it is very important that the trait of ambiguity does not have to do with the language, its all to do with the grammar of the language.

There is not a set way to get rid of the ambiguity, instead you must go through the grammar expression by expression and double check there is only one way to do it according to the definitions that have been set.[\[AG\]](#)

Although it is troublesome to do, trees are a very simple way of checking this and it is easier to read then the actual coded grammar itself.

Lets make our first expression that we worked with non-ambiguous. This is a very simple expression and takes a very simple fix for it to only create one tree and that is parentheses.

I explained earlier in my paper about in math P.E.M.D.A.S. is very practical to helping solve logistic issues in computing as it has the same theory.

Below is the tree that would be if we changed $2+3+4$ to $(2+3)+4$

As you can see the tree looks very similar to the left tree in our first example as its reading left to right like in normal math. In this tree, the expression in the parenthesis is further defines lower in the tree and just like the earlier example the output is 9.

The right tree from the first example would not work as you must keep the parenthesis together.

3.3 Lambda Calculus

Lambda Calculus is " a simple notation for functions and application. The main ideas are applying a function to an argument and forming functions by abstraction."[\[LA\]](#) Earlier when the history of Haskell was discussed, LISP was mentioned. Lambda Calculus was the basis of this language.

A lot of what we did with Lambda Calculus was translating this mathematical language into code and computing it as it is not a common language to compute in.

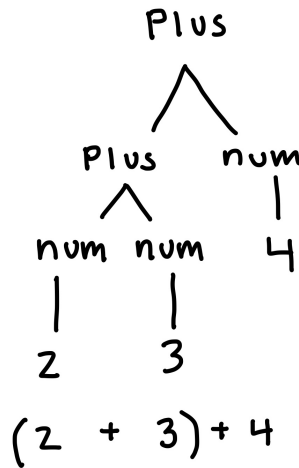


Figure 11: $2+3+4$ (Un)Ambiguous Parsing Tree

To better understand Lambda Calculus, I will discuss a few examples. First is pure lambda calculus.

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

This would be simplified down to just.

$$y$$

The next example is a plus one function, it is not pure lambda as it includes a plus sign.

$$\lambda x.x + 1$$

To break this down this function has one parameter which is 'x' and the body of the function is 'x+1'.

This is technically not a pure lambda function as it has a + sign.

We can expand on this example and use this function! Lets run 2 through this and to do that we would add it at the end of the expression. In lambda equal signs are rarely used so adding the 2 on the end is sufficient enough to understand you are plugging in 2 for this function.

$$(\lambda x.x + 1)2$$

Just looking at this would be a bit difficult so we can draw a parse tree to break down what exactly this function is trying to say.

Earlier we went over order of operations in parsing trees and you can see how in this example how high up lambda is on this tree. If we use the rule that we learned earlier we know that lambda is taking in expressions to evaluate so if we want to get the answer to this function we would need to simplify things down.

If we were to simplify this tree to get out it would look like.

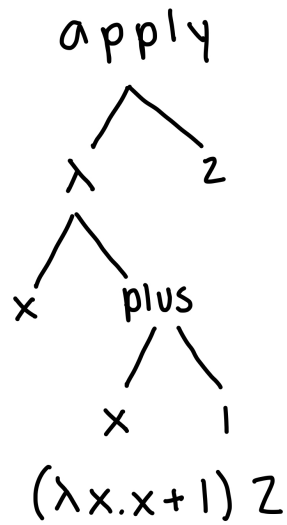


Figure 12: Lambda Example 1 Tree

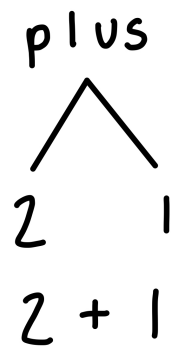


Figure 13: Lambda Example 1 Simplified Tree

Now this is exactly what output was wanted from this function, since it is a plus one function we would want whatever number was inputted to the function to be one higher which in this case would be $2+1=3$.

That was a super simple example but lets do something more complicated with two different functions.

$$(\lambda x.x + 4)(\lambda y.y + 3)2$$

Just like last time, lets create a tree that would better help us understand what the function is asking us to do.

[LFG]

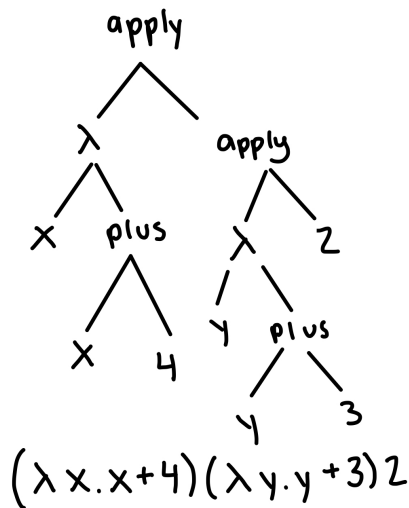


Figure 14: Lambda Example 2

The final equations that we would get from this would be

$$2 + 3 + 4$$

Below is what the parsing tree would look like once we start to simplify the process.

Lambda just a early base version of computing and with these examples we have gone over what it means to use functions to evaluate. Although it might seem like a lot of work for a simple $2+3+5$ but from a logical standpoint it is necessary especially when you start creating huge arguments and more demanding.

When expanding in Lambda Calculus you must learn about scoping. There are two types of variables when it comes to this. A bound variable which is a variable that is associated with a lambda while a free variable is one that is not associated with any lambda.

Lets take our example from earlier and replace the ints with variables.

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

This is how we would define these variables.

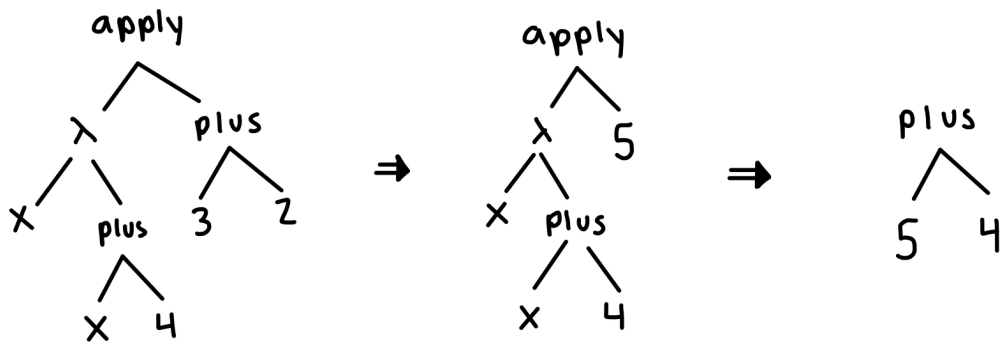


Figure 15: Lambda Example 2 Simplified

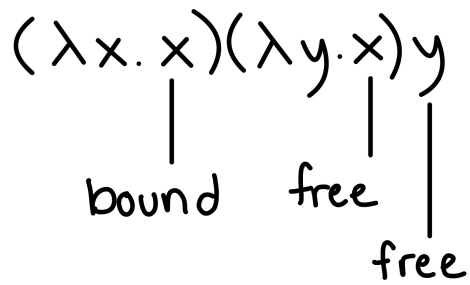


Figure 16: Free and Bound Variables

A variable can appear twice in the same expression which can be seen above but depending on the situation it is in it can be bound but also free.

This is a great way to understand what variables are dependent on especially with different functions.

3.4 Church encoding

So how one represents data in Lambda notation is called "Church encoding" and what is used to represent natural numbers is called "Church numerals." Lambda was actually created in the 1930's by a man named Church.

In lambda there are no ints or booleans, the only thing that is recognized is functions as this is not intended to have a practical implications. Its just to show that specific data types are not needed to show calculations. [CEN]

To transform these expressions in to actual data types you would need to add additional functions such as the $+$ sign as we talked about a plus function earlier. Pure Lambda Calculus does not have these such things as they were not originally intended to have.

When one is interpreting Lambda Calculus people look at it in a way called "Intensional equality". It basically means that it is a proposition, so that it can assume things using induction to prove.

We use lambda terms to represent functions, and to get to the normal form or finding the lambda term's value you preform reductions which can be seen in the parse trees earlier in the report.[CEN]

Under Church encoding there is a way to have natural numbers be represented in Lambda. This is called "Church Numbers."

How this works is that Church Numbers are basically functions that take in two parameters. In Lambda Calculus it is defined as such "0 not applying the function at all, proceed with 1 applying the function once, 2 applying the function twice, 3 applying the function three times, etc."

[CEN]

Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f. \lambda x. x$
1	$1\ f\ x = f\ x$	$1 = \lambda f. \lambda x. f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f. \lambda x. f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f. \lambda x. f\ (f\ (f\ x))$
\vdots	\vdots	\vdots
n	$n\ f\ x = f^n\ x$	$n = \lambda f. \lambda x. f^{\circ n}\ x$

Figure 17: Example of Church Numbers

If we wanted to actually calculate with these rules the following chart shows how each expression would be evaluated.

[CEN]

If we wanted to compute our plus one function from earlier it would look like.

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

Function	Algebra	Identity	Function definition	Lambda expressions	
Successor	$n + 1$	$f^{n+1} x = f(f^n x)$	$\text{succ } n f x = f (n f x)$	$\lambda n. \lambda f. \lambda x. f (n f x)$...
Addition	$m + n$	$f^{m+n} x = f^m (f^n x)$	$\text{plus } m n f x = m f (n f x)$	$\lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$	$\lambda m. \lambda n. n \text{ succ } m$
Multiplication	$m * n$	$f^{m*n} x = (f^m)^n x$	$\text{multiply } m n f x = m (n f) x$	$\lambda m. \lambda n. \lambda f. \lambda x. m (n f) x$	$\lambda m. \lambda n. \lambda f. m (n f)$
Exponentiation	m^n	$n m f = m^n f^{[1]}$	$\text{exp } m n f x = (n m) f x$	$\lambda m. \lambda n. \lambda f. \lambda x. (n m) f x$	$\lambda m. \lambda n. n m$
Predecessor*	$n - 1$	$\text{inc}^n \text{ con} = \text{val}(f^{n-1} x)$	$\text{if}(n == 0) 0 \text{ else } (n - 1)$	$\lambda n. \lambda f. \lambda x. n (\lambda g. \lambda h. h (g f)) (\lambda u. x) (\lambda u. u)$	
Subtraction*	$m - n$	$f^{m-n} x = (f^{-1})^n (f^m x)$	$\text{minus } m n = (n \text{ pred}) m$...	$\lambda m. \lambda n. n \text{ pred } m$

Figure 18: Calculation of Church Numbers

In the table above it is called succession but is the same as plus one.

Using this Church Encoding we can do more with lambda calculus as a functional language than just evaluating functions.

3.5 In-Class Application

We created our own parser in Assignment 2 for Lambda Calculus. In the folder grammar, we have LambdaNat which at first is just a simple languages with "Prog.", "EAbs.", "EApp.", and "EVar.". But as we went on we further developed the language with other mathematical expressions, all with our lambda language.

In this grammar file, we were quite literally creating the grammar that is used to create the language, everything must be defined from how to create comments to what to write for equal.

From there we created a parse which allowed us to compile our own code.

4 Project

For my project I decided that I would attempt to create a working interface for our calculator app (Assignment 1).

4.1 Correcting Assignment 1

The first part of my project was to fix my first assignment 1. My original assignment one had Non-exhaustive patterns which prohibited my code from working correctly it also was missing one of the required operation.

I fixed my numbers.cf file, with some guidance from Curren Taber and Dan Hub, to have all the operations made sense and adding the missing ones as well. One thing that was not in my original assignment was parentheses.

My new file's code looked like this below.

```

Plus. Exp ::= Exp "+" Exp1 ;
Minus. Exp ::= Exp "-" Exp1 ;
Times. Exp1 ::= Exp1 "*" Exp2 ;
Divide. Exp1 ::= Exp1 "/" Exp2 ;
Mod. Exp1 ::= Exp1 "%" Exp2 ;
Power. Exp2 ::= Exp2 "^" Exp3 ;
Negate. Exp3 ::= "-" Exp4 ;

```

```
UMinus. Exp4 ::= "-" Exp3 ;
Parentheses. Exp4 ::= "(" Exp ")" ;
Abs. Exp4 ::= "|" Exp "|" ;
Max. Exp ::= Exp ">" Exp1 ;
Min. Exp ::= Exp "<" Exp1 ;
Num. Exp4 ::= Integer ;
coercions Exp 4 ;
```

Since I went and corrected this, I needed to go back to my interpreter to implement the new operations into my interpreter.

```
module Interpreter where

import AbsNumbers

eval :: Exp -> Integer
eval (Num n) = n
eval (Plus n m) = (eval n) + (eval m)
eval (Minus n m) = (eval n) - (eval m)
eval (Times n m) = (eval n) * (eval m)
eval (Divide n m) = (eval n) `div` (eval m)
eval (UMinus n) = - (eval n)
eval (Mod n m) = (eval n) `mod` (eval m)
eval (Power n m) = (eval n) ^ (eval m)
eval (Max n m) | (eval n) > (eval m) = (eval n) | (eval n) < (eval m) = (eval m)
eval (Min n m) | (eval n) > (eval m) = (eval m) | (eval n) < (eval m) = (eval n)
eval (Abs n) = abs (eval n)
```

I made sure to test and run the program and once I was confident that it worked I moved on to the interface portion of my project.

4.2 Creating an Interface

I did some research online before choosing what type of interface I wanted to create for this app or even where I wanted to have the application be able to be accessed from.

This class was very challenging for me as my specialty interest in computer science is UX/UI design so I thought trying to make this program into an application or at least have a visual aspect would be very fun.

At first I looked into making this project into an iOS app as it is something I enjoy doing but ran into a few difficulties as Xcode uses swift and would not be able to implement something in Haskell.

I came across this website called IHP [IHP]. It is has a web-based schema design and works with Haskell projects!

They had a few projects that used their platform to create full fledged apps using Haskell! I will link a some of these examples as they were very interesting to look at and were super informative.

CO2 Database is a project that uses this site to create an interface to look up C02 producers. The GitHub for the project is public and was very interesting to look at just because the topic is very interesting but now I have more of an understanding of Haskell and better read the program.

4.3 Issues and Problems

Unfortunately I was not able to get a working version of my code to work and create a successful interface. The code I worked on used GitPod to edit the code and unfortunately did not save when I pushed back to github. The more I looked into the issue I found that the site was down.

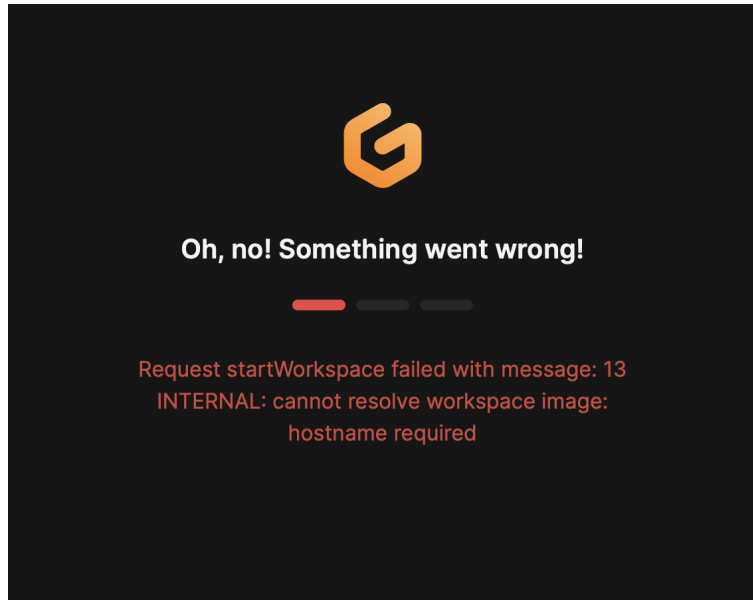


Figure 19: GitPod Error Message

This was okay as I did not get far into the code as I wanted to have user input as it could be a working application that could be used live but I can provide pictures of what the web client looked like.

[IHP]

Although I was not able to get my code to work, the website had a few examples of applications that were built in Haskell which were super informative!

When I do UI/UX design I usually use programs like Xcode or Unity and build off their build in functions, I learned that building a UI from scratch is VERY difficult ex. creating a website.

4.4 Links to Repositories

Here are my GitHub Repositories that were used for my project.

[This is the link of my github repository](#) which has the template I had used to attempt to make an interface for this project. You can also use this link <https://github.com/jkarstrom/ProgramingLanguages>

[This is the link of my github repository](#) which has my corrected Assignment 1 in it. I corrected it on my old repository I had with my partner for the duration of the class. You can also use this link <https://github.com/bugster2010/Assignments354/tree/main/Assignment1>

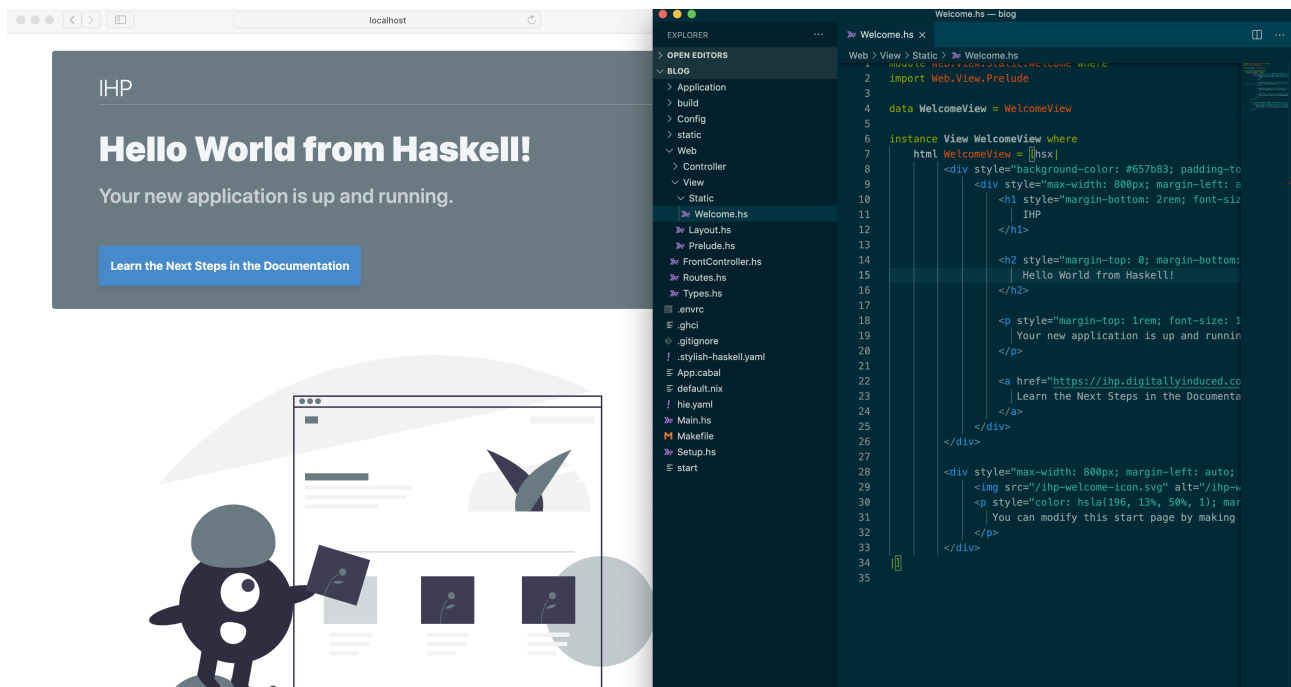


Figure 20: Web Client with Editor open

5 Conclusions

In this paper we have gone over the basis of the purely functional programming language Haskell and how it differs from other languages that programmers use today.

In order to create your own language you must first create the grammar of your language and decide how things will be evaluated, once you have that you create your own parser to compile and run files created in said language.

Although that is very simply said it is actually very difficult to implement and takes a lot of logistic planning.

5.1 Take Away

This course was extremely difficult not in an there was a lot of work or assignments but in a this is very difficult material type of way.

We learnt a whole new programming language that I had never used before and not to mention we tackled building projects that we have not done in other classes before. Although it was very difficult it was very rewarding learning about the basics of programming and how deeply connected the theories are to math and research.

The report in the end was a lot of work but I feel like I have a better grasp of what we learned. Doing more research on the subject and not keeping to what was in the talked about in the class and instead looking at things that interested me more I definitely had a more productive learning curve of the material.

This is a very important class that I believe all programmers should take no matter if they will work with the grammar of languages, the logic behind the languages is something that has definitely opened my eyes about the languages that I use day to day.

References

- [PL] [Programming Languages 2021](#), Chapman University, 2021.
- [IHP] [IHP](#)
- [HHP] [Haskell Remarks](#)
- [PLIN] [Programming Languages 2021](#), Chapman University, 2021.
- [LP] [What is Lazy Evaluation?](#) Sept. 28, 2018
- [BHH] [Brief History of Haskell](#)
- [PYT] [Python Grammar](#)
- [MF] [Recursion](#)
- [LNS] [Lazy vs. non-strict](#) 5 April 2021
- [RR] [Recursion](#)
- [LFG] [Lambda Calculus](#)
- [HP] [Haskell History](#) August 23rd, 2019
- [CEN] [Church encoding](#)
- [PAR] [Haskell. History of a Community-Powered Language](#) August 23rd, 2019
- [NS] [Lazy vs. non-strict](#)
- [BSP] [Bubble Sort](#) 24 Sep, 2021
- [LH] [Learn You a Haskell](#)
- [BSS] [Bubble Sort](#) 24 Sep, 2021
- [LA] [The Lambda Calculus](#) 24 Sep, 2021
- [BSH] [Building algorithms with Haskell](#)
- [AG] [Ambiguous Grammar](#)