CPSC-354-01 Report

Ayden Best
Chapman University

December 29, 2021

Abstract

Short introduction to your report ...

Contents

1 Haskell

This section of the report contains an introductory tutorial to Haskell. Among other options for a Haskell section of this report, a tutorial interested me the most due to my positive experience of teaching Haskell to others in the first assignment for this class.

1.1 What is Haskell?

1.1.1 History of Haskell

In the 1930s, Alonzo Church spent his time developing Lambda Calculus at Princeton University. Lambda Calculus was a mathematical notation that described the application and abstraction of functions, which explained the core of computations. This notation would later be the premise for constructing early programming languages to use for research.

By the 1980s many researchers were implementing Lambda Calculus by creating their own functional programming compilers. This was mainly due to the absence of an open-source compiler. This explosion of different compilers meant incompatibilities between each platform. This brought a small group of researchers together to form a committee that released the first iteration of *Haskell Language Report* in 1990. Haskell was released, and there was finally an open-source functional programming language would be available to the public.

1.1.2 Comparing to Other Programming Languages

Imperative Programming is likely what occupies most developers' experiences in programming. Imperative Programming works similarly to how computer hardware works, which operates sequentially. That is, Imperative Programming uses a series of instructions to make a program reach a goal. Processing information in this way however has many opportunities for bugs, because the programmer is restricted to keeping proper steps in the right order. The source of these bugs is *Side Effects*, as the dependence on state means that some steps can work improperly if executed at the wrong time. An example of an Imperative statement would be "Print out variable 'x', which I mentioned earlier." This statement depends on the order being kept, and that the program actually had declared and stored variable 'x'. **Functional Programming** is different, in that it is not restricted by order. On the development end, programmers describe mathematical statements that can construct a desired output. One could think about this as a rule book. The computer will then build the sequence of processing steps itself to reach the goal that you set for it. This will consequently help avoid *Side Effects*. An example of a Functional statement would be "milk belongs in the fridge." This type of programming is all about building declarations that allow the computer to build a sequence of steps from the given information to the goal.

Functional Programming Example:

Turn a horse into giraffe

- Lengthen Neck
- Lengthen Legs
- Recolor Body
- Apply Spots
- Replace Tail

As long as all instructions are executed, the solution will be processed. The order that each step is executed does not matter.

Imperative Programming Example:

Park a car in the garage

1. Note the state of the garage door
2. Stop the car in the driveway
3. IF garage is closed THEN open garage
4. Pull car into garage
5. Close garage door

Clearly the sequence of this solution is important, because the state of the garage must be known before the state is evaluated

1.1.3 Uses in Industry

The strengths of Haskell are exemplified through it's ability to avoid Side Effects. This makes the language very reliable, which helps prove itself better in some areas than imperative languages. The main benefits to functional programming and specifically Haskell are:[1]

- **Where correctness matters**: When failure of a component is unacceptable. Code that causes Side Effects (state management) is separated from the code that only processes information.
- **For building domain-specific languages**: It can embed smaller, domain-specific languages easily. *(Ex: Paradise, which is a language that created financial models)*
- **For multi-core parallelism**: Data structures are all immutable, so the language is not restricted to race conditions.

1.2 Features of Haskell

1.2.1 Haskell is Declarative

We all say that Haskell is declarative, but what does that really mean?

In an imperative language, we work procedurally. To compute the sum of an array, we would write a function that looks like this:

```
int sumArray(int arr[]){
    int i, sum = 0;
    for (i=0 ; i < arr.length() ; ++i) {
        sum = sum + arr[i];
    }
    return sum;
}
```

In this C++ code, we can see that we work through separate steps in order. The 3 components of this function are the initialization, the for-loop, and the return statement. If these components were in any other order, the function would not work correctly. We are telling the computer what **steps** to take in which **order**

However in Haskell, we can compute the sum of an array with much cleaner code. We will instead use recursion and the baseline of an empty array by typing:

```
sumArray [] = 0
sumArray [x:xs] = x + sumArray xs
```

When a request for sumArray of a list is sent, Haskell itself searches for solutions to the request and will recursively evaluate nested sumArray's until the base case of an empty list is evaluated. From here, integer arithmetic is evaluated to join sum up each item of the array. The language simply tries to evaluate the declarations its been given to the right side of each declaration.

1.2.2 Laziness

We also say that Haskell is lazy. This means that declarations known by the compiler are not evaluated until a result requiring that declaration is requested. For this example, lets say that we have a function **funct(int x)** that will take 1 hour to compute regardless of parameter x and returns a boolean randomly.

[1] Alex Handy: Everyone's Talking About Haskell

Back in C++ code, we can see than imperative languages would need to first gather results before comparing them.

```
bool foobarBoolean(int x, int y, int z){
    bool a = funct(x);
    bool b = funct(y);
    bool c = funct(z);

    if (a) {
        return b;
    } else {
        return c;
    }
}
```

Here we first declare a, b, and c before evaluating them. Remember, each execution of **funct** will take 1 hour to compute. In this example, storing all variables first will make **foobarBoolean** ALWAYS take 3 hours. Lets compare this to Haskell.

```
a = funct x
b = funct y
c = funct z

ghci> if a then b else c
```

The Haskell code looks like it would perform the same, but it will actually be faster. a, b, and c are not calculated initially because they are not yet needed. Instead, Haskell simply remembers *how* to evaluate these variables without needing to commit to evaluating them. The if-then-else statement evaluates a, and then based on the result will evaluate either b or c, but not both. This will result in a 2 hour execution for this code, rather than 3 hours. This example demonstrates Haskell's nature of storing methods of evaluation while avoiding full evaluation until it is absolutely necessary.

1.3 Haskell Types

Haskell has what is known as a static type system. When a program is compiled, Haskell will know about every declaration and evaluation of types written. This means that any and all type-errors will be caught by the compiler. This is a safety feature that makes Haskell unique, because it is what prevents side-affects. See *Table 1: Default Types* for a list of common types provided in Haskell. Additionally, the Glasgow Haskell Compiler Interactable-interface(GHCi) allows the command **t: [expression]** to view the type of any value or function.

Haskell also has type-inference, meaning types do not need to be declared. The compiler will know the difference between `1` and `1.0` without declaring that either is an Integer or Float.

1.3.1 Typeclasses

A typeclass is much like an interface that is associated with each type in Haskell. `Integer` is part of the typeclass `Eq` which provides use of the operator `==`. If `Integer` was not part of the `Eq` typeclass, then the compiler would not accept an evaluation for equality of Integers. Visit this table to see the hierarchy of typeclasses in the Haskell Prelude (Predefined typeclasses).

Table 1: Default Types

Type	Example	Usefulness
Bool	True	A single binary value
Int	14543	A natural number that maxes out at 2147483647
Integer	214748364700000	Like an Int, but has no value limit. This comes at a small cost to performance
Float	3.141592	A decimal number. Only allows for 23 points of precision after the decimal
Double	3.141592741012573242	A decimal number, but with 52 points of precision after the decimal
Char	'A'	A character
String	'Ayden'	A list of Char's (type is [Char])

1.3.2 Lists

Each datatype can be composed as a sequence of the same datatypes in a list. For example, a list of Integers could look like [1,2,3]. Another number can also be prepended to the list using `:` as so: `0:[1,2,3] = [0,1,2,3]`

As such, the list can also be represented as: 0 : 1 : 2 : 3 : []

Lists can also be initialized with a range between two sequenced values(Part of the `Seq` typeclass) using `..`. For example, `[1..10]` creates a list of Integers from 1 to 10, and `[1..]` creates a list that contains all Integers. But wait, how is it possible to create a data structure of infinite size? This is permissible because of the laziness feature of Haskell: The indices of the infinite list are not generated until they are needed.

1.4 Functions

As we have seen already, functions are a way to contain some functionality under a definition to recall somewhere else in our programs. As an example, we have the function in_range to return a True/False value based on whether the third parameter is between the first two parameters:

```
in_range floor ceiling x =
    x <= floor && x >= ceiling

in_range 0 3 2    -- True
in_range 0 3 4    -- False
```

We see here that function definitions in Haskell are also declarations of equality.

In functions we can also demand specific types for each parameter, as well as ensure the type of the returned value. As long as the programmer was sure that they called every functions with correct types, this feature would be useless. However the ability to force types allows us to catch more errors at compile time and avoid any mistakes in the development of a program. An example of setting types for a function is shown below.

```
in_range :: Integer -> Integer -> Integer -> Bool
in_range floor ceiling x =
    x <= floor && x >= ceiling

-- this allows us to make a function that forces the use of integers
in_range 0 3 2        -- True
in_range 0.0 3.0 2.0  -- Type Error
```

There is also an option to have typeclass restrictions rather than a strict single type for each parameter. This can be done by setting a typeclass to a name using the syntax: `(Eq a, Float b)=>a -> b -> Integer`, which requires an equitable parameter and a float parameter.

1.4.1 Pattern Matching

When defining functions, we can also define multiple function bodies based on the form of the inputs. Evaluation will descend the list of patterns until a match is made. This is done by deconstructing each input and comparing it to the structure of the pattern so find any similarities in composition.

This feature becomes extremely useful with the implementation of recursion. A function to evaluate any Fibonacci number is shown below:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

In the Fibonacci example, `n` was used as a placeholder for the parameter entered into the function. This was a base case for instanced where the patterns for 0 and 1 failed.

In the case that a parameter is not needed in the evaluation, a _can be used instead of a placeholder to improve performance.

Patterns can also refer to lists. Because pattern matching deconstructs a datatype to see if it matches each pattern, we can search for appearances of sequences in lists. The best way to understand this is by representing lists as nested appended items using the colon(:). Below is a few patterns for a list:

```
describe :: (Show i) => i -> String
describe [] = "Empty List"
describe x:[] = "Has 1 item"
describe x:y:[] = "Has 2 items"
describe x:xs = "List is long with head " ++ x ++ "and tail" ++ xs

describe [1,2,3]  -- List is long with head 1 and tail [2,3]
```

Pattern matching can also be performed with different syntax. This syntax is called a case-expression, and the above code translated into such syntax looks like this:

```
describe :: (Show i) => i -> String
describe x = case x of [] = "Empty List"
                       x:[] = "Has 1 item"
                       x:y:[] = "Has 2 items"
                       x:xs = "List is long with head " ++ x            ++ "and tail" ++
                           xs

describe [1,2,3]  -- List is long with head 1 and tail [2,3]
```

1.4.2 Guards

Guards also exist to handle different cases of the value of data, and can be used in conjunction with Pattern Matching. They are much like a switch-case statement, but provide much cleaner syntax like so:

```
allowed_to :: (Num a) => a -> String
allowed_to x
    | x >= 21 = "You can buy alcohol"
    | x >= 18 = "You can vote"
    | x >= 16 = "You can drive"
    | otherwise = "You can't do much..."
```

1.4.3 Functions as Parameters

When designing restrictions and parameters for functions, Haskell also allows the option to pass a function itself as a parameter to another function. This function can be used to apply to another parameter, or it can be used recursively to apply to all items in a list. This example specifically is called `map`:

```
map :: (a -> b) -> [a] -> b
map _ [] = []
map func x:xs = (func x) : (map func xs)
```

1.5 Syntax Mechanics

1.5.1 Substitution with Where vs. Let

Substitution is a feature that we can use in expressions to recycle code and improve organization. We can access this feature by using the keywords **let** and **where** to declare values in expressions. Though they perform similar operations, **let** and **where** serve slightly different purposes.

```
in_range floor ceiling x =
    let above_floor = x <= floor
        below_ceiling = x >= ceiling
    in above_floor && below_ceiling
-- OR
in_range floor ceiling x =
    above_floor && below_ceiling
    where  above_floor = x <= floor
           below_ceiling = x >= ceiling
```

let is used to substitute expressions into another expression with forward scope. The variables and values are defined first, and then used in the expression to follow. This makes it difficult to use in function definitions and guards, but is still possible with an overcomplicated workaround.

Alternatively, substitutions can be written into guards much easier with **where**. This is because while **let** is expressive, **where** is instead declarative and scopes backwards. This means that we can bind **where** clauses to parts of a function that are not expressions on their own, such as guards. The above code can be reworked to look much cleaner like so:

```
-- Using WHERE
tall_enough :: Int -> Int -> String
tall_enough feet inches
       | height > 5.8 = "You can get on the ride"
       | otherwise = "You are not tall enough to
           ride..."
        where height = feet + (inches/12)
```

```
-- Using LET
tall_enough :: Int -> Int -> String
tall_enough feet inches =
    let height = feet + (inches/12)
    in case () of
    _  | height > 5.8 = "You can get on the ride"
       | otherwise = "You are not tall enough to
         ride..."
```

The downside to **where** comes in because it is not expressive and must apply to the right side of of a declaration (for example, guards after a pattern match). It cannot be applied to expressions, and this code will produce a parsing error: `print (1 + (2 * i + 1 where i = 10))`

1.5.2 Function Currying

Currying refers to the way in which functions are formed, and how parameters are accepted into the function. When writing functions that take 1 parameter, we simply put `func :: a -> b`. Similarly, when writing functions that take more than 1 parameter, we write `func :: a -> b -> c`. Intuition tells us that this is one function that takes 2 parameters, but Haskell works a bit differently. A function like `sum :: x -> y -> z` is actually a chain of functions that take 1 parameter like so: `sum :: x -> (y -> z)`. In this example, the function `sum` takes 1 parameter `x`, and returns a function `(y -> z)`. The resulting function `(y -> z)` is constructed to incorporate whatever `x` was in order to manipulate `y` into `z`. We can see here that Haskell sees multi-parameter functions

By understanding the nature of functions with multiple parameters, we should also recognize the applications of a function accepting the parameter in front to return yet another function. Currying allows us to partially fill functions to receive another function. With the above example we could define another function *let SumWithTwo = sum 2*, and call that function as *SumWithTwo 3 4* to find the sum of $2, 3and4$.

1.5.3 Lambda Functions

Alternative syntax for functions also exists. With the understanding of currying, we now know that there are only ever functions that take a single parameter and return a single value. These functions can be written using **Lambdas**. Below is a Lambda replication of a function we call plus_two:

```
plus_two :: (Num a) => a -> a
plus_two x = x + 2
plus_two 5        -- 7
    -- OR
(\x -> x + 2) 5   -- 7
```

Through the syntax, we can see that Lambda Functions are structured as so:

```
(\varName -> function(varName))input
```

1.5.4 Function Application

So far, all the code we have written evaluates from the left. To force evaluation of a block of code first, we have been using parentheses. There is an operator $ that we can use instead to evaluate everything right of the operator first. Looking at the function definition, we can understand how this works in Haskell:

```
() :: (a− > b)− > a− > bf x = f x
```

This operator seems useless at first, but it becomes very useful in the sense of writing clean code. Based on the pattern matching, everything to the right of the $ is evaluated into a single target to apply to the function left of the operator. This works identically to enclosing an evaluation in parentheses.

1.6 Structuring Data

We talked about predefined types in Haskell previously. Haskell gives us the ability to create our own data structures and types. Here are a few examples of structures that exist in the Haskell Prelude:

```
    data Bool  =  False | True   deriving (Read, Show, Eq, Ord, Enum, Bounded)
    data [a]   =  [] | a : [a]   deriving (Eq, Ord)
    type String = [Char]
--  data/type DataTypeName = Obj1 Obj2 | Obj 3 Obj4 | Obj5 deriving (class1, class2,
     class3)
```

Here we have the definitions for Bools, Lists and Strings. We will go into each component of these definitions in detail to understand how to construct new data structures.

New Data Structure: The declaration 'data' prompts the compiler to begin defining a new datatype

Datatype Name: The names 'Bool', '[a]' and 'String' are examples of names. When we call functions, this is the name that we will be calling for parameters and return values.

Constructors: To the right of the equals sign, we will define different forms that our data will be composed of. Data types can actually be composed of multiple objects, so we can think of them more like classes in many imperative languages. Similarly again, each set of objects will be a constructor for our new datatype. Data may also be defined to come in different forms, which is separated by a |. A great example of an application of this is seen in the definition for lists. A list can either be an empty list, or a variable with another list, separated by a colon.

Utilizing Typeclasses: At the end of the data declaration we have a 'deriving' call, followed by a list of Typeclasses in parentheses. This allows us to include functionality from different typeclasses. So far, we are limited to including typeclasses that subcomponents of our new datatypes are supported.

Keyword 'type': In the example of the 'String' definition, the keyword type is used instead of data. This keyword is used instead for type synonyms, meaning we are declaring that a String is identical to [Char]. If we had instead declared `data String = [Char]`, then String would be seen as entirely different from [Char]. This would mean that defining a function that accepts a String as a parameter would break when fed a [Char].

With a better understanding of each component of defining a new data structure, we can begin creating our own structures. Here is an example of defining a structure for roman numerals using only 'I':

```haskell
data Roman = Zero | Roman I
-- A user may or may not have a name, but will always have a height in feet and
    inches
count :: Roman -> Int
count Zero = 0
count (pred I) = 1 + count pred -- pred is codename for predecessor
```

By only using 'I', our new Roman Numeral system works much like a list that we can only find the length of. This is simply a virtual counting system that we can easily convert to an Integer.

1.6.1 Extending Typeclasses

Say we want to be able to compare our Roman Numerals. We could easily define a function that converts two Roman Numerals to Integers and then compares the outputs, because we have a definition for comparing Integers. If this were however a more abstract data structure that we could not represent as a prelude type, that option would not be present. In order to use == and /=, we need to look at how the Eq typeclass is defined.

```haskell
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
   x == y = not (x /= y)
   x /= y = not (x == y)
```

Here we have a template for equality operators that different types can implement. We should prepare to define the operator == that takes two of the same type and returns a Bool. Notice that == and /= are defined in terms of each other. This allows us to only have to redefine one of these operators for each type in order to make the other one work as well. Carrying on now, we can define the == operator specifically for our Roman Numerals by creating an instance of this class:

```
instance Eq Roman where
    Zero == Zero = True                  -- If both empty at the same time, they are
        equal
    (pred I) == (pred' I) = pred == pred' -- If both have at least 1 'I', remove one
        from each
    _ == _ = False
    -- The last pattern is reached if only 1 Roman has at least 1 'I' and the other
        is already empty
```

With this definition for Eq on Roman, we can now use the equality operators on all of our Roman Numerals without having to convert them. This also allows us to use Roman Numerals in functions that require its parameters to be in the Eq typeclass as so: `includes :: (Eq a)=>a -> [a] -> Bool`

1.7 Input/Output

Finally we reach I/O in Haskell. The two main I/O operations we will look at are `putStrLn` and `getLine`.

1.7.1 The IO Type

These IO operations have the following types:

```
putStrLn :: String -> IO ()
getLine :: IO String
```

We are now looking at an IO type that contains another component. For putStrLn, a String is sent into a function and an IO object is returned, while getLine simply returns an IO object containing a String with no parameters. This IO object is actually an action that hooks into the operating system. Outputting a String using putStrLn does not need a return value, so by default Haskell uses an empty tuple. getLine however calls a hook to the operating system to receive a String from the user, so Haskell must return a String value to bind to another object. This can be done with `obj <- getLine`.

1.7.2 Completing a Program

Using IO actions, we can now finish up Haskell evaluations that users can interact with. For this, we will use the Roman Numerals example. The first thing to do is fill in the gaps so that we can convert Roman Numerals to Strings. We will do this by adding an instance of Roman to the Show typeclass. This typeclass has a single function `show :: a -> String`. The final thing we have left to do is add a function that converts some Integer to Roman Numerals. We will do both of these steps using recursion:

```
data Roman = Zero | Roman I
instance Show Roman where
    show Zero = ""
    show (I pred) = "I" ++ show pred

intToRoman :: Integer -> Roman
intToRoman a
         | a <= 0 = Zero
         | otherwise = I (intToRoman (a-1))

main = intToRoman 3   -- output: III
```

This can be further extended to include V, X, L, etc. by adding additional cases for when the Integer parameter exceeds 5, 10, 50 and so on. We now have a program to load that allows users to convert Integers to Roman Numerals and view the output String of that Numeral.

2 Programming Languages Theory

2.1 Lambda Calculus

In the Haskell Tutorial, we talked about the method of creating functions using Lambda syntax. This syntax will be the foundation for what we call Lambda-Calculus. This will give us a much stronger view of how functions substitute variables, as well as why currying works the way it does.

As it was previously explained, Lambda Functions have the syntax:

$(\lambda varName.\ f(varName)) < input >$

Or when applied,
$(\lambda x.\ x + 1)1$
$\rightarrow 1 + 1 = 3$

The Lambda Function is evaluated by substituting the leftmost parameter(1) for the leftmost variable (x) in the program declaration(x+1). This receiving the result, 2. We can also use currying to accept more than one parameter into a program:

$(\lambda x.\lambda y.\ x + y)\ 1\ 2$
$\rightarrow (\lambda y.\ 1 + y)\ 2$
$\rightarrow 1 + 2 = 3$

2.1.1 Binding and Scope

The substitution of different variables however brings potential for evaluation errors. It is important to understand how binding works to avoid assumptions for combining two independent variables based on their name. Here's an example of accepting this false assumption:

$(\lambda x.\ x + y)\ y$
$\rightarrow y + y = 2y$

This evaluation is wrong, because it assumes that the internal y and external y are the same. We will call the variables within the function definition **bound** and the variables passed into the function **free**. The bound variables have different scopes than the free variables. The value of the free variable y is passed into the occurrence of x in the function body. Say we had a nested Lambda Function that accepted y:

$(\lambda x.\lambda y.\ x + y)\ y\ z$
$\rightarrow (\lambda y.\ y + y)\ z = 2z$

Following the same logic, we should now pass y into the occurrence of x and pass z into the occurrence of y. This follows the definition of the function, which we expect to result in the sum of two variables, ie. f(x, y) = {x + y}. By capturing the free y into the bound y, we have assumed that both y's are comparable. Because the function has an internal scope, all variables within the scope are separate from the outside environment, and we should be able to rename function variables without changing the functionality.

We need to develop a method for Capture Avoiding Substitution. A solution to avoiding this error would be by using Barendregt's Variable Convention. This method is conducted by renaming variables so that they are not confused. Re-trying the previous function with this method, we get:

$(\lambda x.\ \lambda y.\ x + y)\ y\ z$
$=> (\lambda x.\ \lambda n.\ x + n)\ y\ z$ – In the function: Rename y to n to avoid confusion
$=> (\lambda n.\ y + n)\ z$
$=> y + z$

Example: A Beautiful Day In The Neighborhood

Suppose your are moving into a new neighborhood and want to know who the friendly neighbors are. You find that all the neighbors immediately adjacent to your house are friendly, and you trust that anyone they like are friendly as well. We can describe these rules as:

```
:-  => "If"
,   => "And"; || => "|"
likes(A, B) :- nextTo(A, B) -- "A likes B" if "A is next to B"
likes(A, C) :- nextTo(A, B), likes(B, C) -- "A likes C" if "A is next to B and B likes C"
```

Lets say we are house D and have neighbors A-C (3 houses). We are next to neighbor C. We have also interviewed the entire neighborhood to see who everybody likes. Our starting points are:

```
-- order is: A B C D
nextTo(A, B), nextTo(B, C), nextTo(C, D) -- to the right of
nextTo(B, A), nextTo(C, B), nextTo(D, C) -- to the left of
```

As explained, we are house D and we want to find out everyone we can be friends with. We need to find all possible instances of likes(D, _)

Apply: `likes(A, B):- nextTo(A, B):`

`likes(D, C)`

We can convert the nextTo claim into a likes claim because of the first rule

Apply: `likes(A, C):- nextTo(A, B), likes(B, C):`

`likes(A, C)if nextTo(A, B), likes(B, C)`
`likes(A, C)if nextTo(A, B), likes(B, C)` Substitute A for D
`likes(D, X)if nextTo(D, B), likes(B, X)` Substitute B for C $\rightarrow$ Naming Error, rename bound C to X
`likes(D, X)if nextTo(D, C), likes(C, X)` Substitute B for C
`likes(D, B)if nextTo(D, C), likes(C, B)` Substitute B for X
`nextTo(C, B)` $\rightarrow$ `likes(C, B)`
`likes(D, B)is True`

Result: `likes(D, C), likes(D, B)`
If we are in house D, we like the families in houses B and C

2.1.2 Church Numerals

Church Encoding is a concept that will force us to use Capture Avoiding Substitution even more. Say we create some function that contains multiple layers of nested functions `f`:

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

Note that this is a function and does not contain any parameters. The function accepts another function as well as a value, and applies that function to the variable 3 times. Here are a few examples of what we can use this for:

Binary Number Limit: We can describe a function ($\lambda z.\ 1 + z + z$) that finds the maximum value a series of bits can represent. For example, 3 bits can represent values between 0 and 7 (111 = 1 + 2 + 4 = 7):

$(\lambda f.\ \lambda x.\ f\ (f\ (f\ x)\ )\ (\lambda\ z.\ 1 + z + z)\ 0 \rightarrow$
$(\lambda x.\ (1 + (1 + (1 + x + x) + (1 + x + x)) + (1 + (1 + x + x) + (1 + x + x)))\ )\ 0 \rightarrow$
$(1 + (1 + (1 + 0 + 0) + (1 + 0 + 0)) + (1 + (1 + 0 + 0) + (1 + 0 + 0))) \rightarrow 7$

Boolean Logic: We can also construct functions that represent True or False, as well as Boolean logic like the if-then-else operator. Here we define True and False:

$\texttt{true} = \lambda x.\ \lambda y.\ x$
$\texttt{false} = \lambda x.\ \lambda y.\ y$

As you can see, these are both functions. `true` returns the first parameter given, and `false` returns the second parameter given. We will however not be deciding what the return cases for True or False are, as that will be decided by the parameters. Perhaps in most cases when we actually want a True or False value based on a condition, we call `true True False` and `false True False`. Maybe in some other conditions where we want to print the result, we can call `"Result is: "++ true 'True' 'False'` and `"Result is: "++ false 'True' 'False'` to get Strings that tell us what the result was.

Carrying on, we can use these identity functions to construct Boolean logic. Here is an example implementing `and`:

$\text{and} = \lambda a.\ \lambda b.\ ((a\ b)\ \lambda x.\ \lambda y.\ y)$

And now we can implement that logic. Notice that after (a b) in the `and` definition, we have the definition for false(which takes 2 arguments). This means that we are passing `and` 4 values: 2 values to test for Boolean value, a result for returning True and a result for returning False. Note that when we pass `true` or `false` as a parameter, we will be passing the definitions that we created above.

```
       Ex: and resulting true (sub true and false for the definitions above)

(λa. λb. (a b) (λx. λy. y) ) true true "Res: T" "Res: F"
(λb. ((λx. λy. x) b) (λx. λy. y) ) true "Res: T" "Res: F"
((λx. λy. x) (λx. λy. x)) (λx. λy. y) "Res: T" "Res: F"
(λy. λx. λy. x) (λx. λy. y) "Res: T" "Res: F"
(λx. λy. x) "Res: T" "Res: F" -- definition of \lstinline{true}
"Res: T"

       Ex: and resulting false (sub true and false for the definitions above)

(λa. λb. (a b) (λx. λy. y) ) false true "Res: T" "Res: F"
(λb. ((λx. λy. y) b) (λx. λy. y) ) true "Res: T" "Res: F"
(λb. (λy. y) (λx. λy. y) ) true "Res: T" "Res: F"
((λy. y) (λx. λy. y) ) "Res: T" "Res: F"
(λx. λy. y ) "Res: T" "Res: F" -- definition of \lstinline{false}
"Res: F"
```

If we look closely at the examples on applying the `and` function, we can see that changing the first `true` to `false` caused some interesting activity in the bound variables. If a evaluates to the `true` definition, it "becomes" whatever b is. However when we make a evaluate to `false`, it "deletes" b and returns whatever the next parameter is. This is the definition of if-then-else. If "a", then "b", or else "c". We can see the similarity in the if-then-else definition below:

$\text{if-then-else} = \lambda a.\ \lambda b.\ ((a\ b)\ (\lambda c.\ c)\ )$

Numbers: Say we make the encoding function `f` take a single parameter and add 1 to it. We can then chain the function together to resemble a number system. Since we will have many free and unbound variables all named `f`, this application of Church Encodings will push our Capture Avoidance skills to the limit. All we need is a base-case and a successor function to define the entire natural number system(Thank you, MATH 250!).

zero: $(\lambda f.\ \lambda x.\ x)\ (+1)\ 0$

successor: $(\lambda a.\ \lambda f.\ \lambda x.\ (f\ (a\ f)\ x)\ )$

Examples:
1: $(\lambda f.\ \lambda x.\ f(x)\ )$
5: $(\lambda f.\ \lambda x.\ f(f(f(f(f(x)))))))))\ )$

We now have a number system working in Church Encoding. We can further apply this and discover functions for addition, multiplication and exponents like we covered in the course. Below is an example of how we would reduce one of these complex functions.

`Function given: mult 3 5`
$(\lambda multiply.\ \lambda three.\ \lambda five.\ multiply\ three\ five)$
$(\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x)$
$(\lambda f.\ \lambda x.\ f(f(fx))$
$(\lambda f.\ \lambda x.\ f(f(f(f(f(fx))))))$ `(+1) 0`

`Rename bound variables`
$(\lambda multiply.\ \lambda three.\ \lambda five.\ multiply\ three\ five)$
$(\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x)$
$(\lambda f'.\ \lambda x'.\ f'(f'(f'x'))$
$(\lambda f''.\ \lambda x''.\ f''(f''(f''(f''(f''(f''x''))))))$ `(+1) 0`

`Apply free variables to leading function`
$(\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x)\ \ (\lambda f'.\ \lambda x'.\ f'(f'(f'x')))\ \ (\lambda f''.\ \lambda x''.\ f''(f''(f''(f''(f''(f''x''))))))$ `(+1) 0`

`Apply Church(3) into m`
$(\lambda n.\ \lambda f.\ \lambda x.\ (\lambda f'.\ \lambda x'.\ f'(f'(f'x')))\ (n\ f)\ x)\ \ (\lambda f''.\ \lambda x''.\ f''(f''(f''(f''(f''(f''x''))))))$ `(+1) 0`

`Apply Church(5) into n`
$(\lambda f.\ \lambda x.\ (\lambda f'.\ \lambda x'.\ f'(f'(f'x')))\ ((\lambda f''.\ \lambda x''.\ f''(f''(f''(f''(f''(f''x''))))))\ f)\ x)$ `(+1) 0`

`Apply f into f’’`
$(\lambda f.\ \lambda x.\ (\lambda f'.\ \lambda x'.\ f'(f'(f'x')))\ (\lambda x''.\ f(f(f(f(f(fx''))))))\ x)$ `(+1) 0`

`Apply` $\lambda x''.$ `function into f’`
$(\lambda f.\ \lambda x.\ (\lambda x'.\ (\lambda x''.\ f(f(f(f(f(\lambda x''.\ f(f(f(f(f(\lambda x''.\ f(f(f(f(fx'))))))))))))))))\ \ x)$ `(+1) 0`

`Reduce nested` $\lambda x''.$ `functions into each preceding` $\lambda x''.$
$(\lambda f.\ \lambda x.\ (\lambda x'.\ (f(f(f(f(f(f(f(f(f(f(f(f(f(f(fx'))))))))))))))))\ \ x)$ `(+1) 0`

`Apply x into x’`
$(\lambda f.\ \lambda x.\ (f(f(f(f(f(f(f(f(f(f(f(f(f(f(fx))))))))))))))))$ `(+1) 0`

`Apply (+1) into f`
$(\lambda x.\ x+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1)$ `0`

`Reduce (+1) and apply 0 into x`
`0+15 -- 15 Church Numerals results in int 15`

2.1.3 Church Numerals in Haskell

Haskell itself already has syntax for Lambda Functions. Similarly to how we would write a Lambda Function for an identity property $(\lambda x.x)$, we can write a Lambda Function in Haskell as `(\x -> x)`.

Now that we have our first chunk of understanding for Haskell and Church Numerals, we can combine the two. As we know, Church Encoding is an application of a function and a value to return a new value. We will need to define a new datatype for `Church`, as well as write some conversions between `Church` and `int`.

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

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

success :: Church x -> Church x
success (Church n) = Church (\f x -> (f (n f x)))

intToChurch :: Int -> Church x
intToChurch 0 = zero
intToChurch n = success (intToChurch (n-1))

churchToInt :: Church Int -> Int
churchToInt (Church f) = (f (1+) 0)

churchToString :: Church String -> String
churchToString (Church f) = "(\\f.\\x. " ++ (f ("f("++) "x") ++ (f (")"++) "") ++ " )"

-- churchToString (churchToInt 4) => "(\\f.\\x. f(f(f(f(x)))) )"
```

There are a few key things to notice in this code. Let's first look at what we can do with `Church`. We define a `Church` structure that expects to replace a function and a variable, and then return another variable. Our structure if of type `Church :: ((x -> x)-> x -> x)-> Church x`, so as long as we follow the typing, all will go as planned. We first create `intToChurch` that expects `(Int -> Int)-> Int -> Int)`. Here we can see how the substitution is working with 3 Church Numerals:

```
(\f x -> f( f( f x))(f' -> 1 + f')0
(\x 1+( 1+( 1+ x))0
1+( 1+( 1+ 0)
3
```

We do this again for printing with Strings.

2.1.4 Advanced Church Numeral Functions: Division

2.2 String Rewriting

Moving the focus into the pattern matching of the Church Numerals in Haskell example, we can clearly see that the reduction of `zero` and `successor` terminates into a single `Church`. Using the rules `0 -> zero` and `n>0 -> success (intToChurch n-1)` will eventually result in the form of a chain of `success`es with a `zero` somewhere at the end of a chain. This is a simple example that was not easy to mess up, but what can happen in larger programs with many more patterns? Supposed a Haskell developer accidentally added a pattern for `1 = intToChurch(n+1)`. If that happened, then the patterns would infinitely loop. To evaluate the patterns that we have for a function, we can study String Rewriting to be sure that errors like this don't come up. This is much more applicable in larger applications such as an Operating System, where a massive program may work as unintended and follow a loop that slowly occupies more and more memory over time until an overflow crashes the program, or worse, the system.

Specifically, we will look at **Abstract Reduction Systems**, which are pairs presenting as *(A, A*$\rightarrow$*A)* for some set A. An ARS reflects the path that a recursive pattern matching series will follow. For the set `intToChurch`, the patterns were `(0`$\rightarrow$`zero)` and `(n`$\rightarrow$`success(intToChurch(n-1))`.

2.2.1 Confluence, Termination and Normal Forms

There are a few terms in String Rewriting that we should become familiar with. They will help shorthandedly describe the nature of an ARS.

Confluence: Confluence means that all reductions from one point can eventually reduce to a common point down the ARS. Say we have the reductions `(a→b)` and `(a→c)`, the ARS would be confluent if both `b` and `c` could both reduce to some 4th state `d`. Without any other states, this would require the reductions `(b→d)` and `(b→d)`. When an ARS in a system is confluent, we can be sure that the system will always yield the same result from a computation, regardless of which reductions it applies over other ones from each state.

Example: Addition is confluent. This is because `1+2+3` can either reduce to `3+3` or `1+5` for the first step, but from either state the next and final step will always lead to `6`.

Termination: Termination means that no state has reductions that can eventually lead back to itself. In the above example, the reduction `(c→a)` would break termination, because the computation could become `(a,b,d,a,...)` and continue on infinitely. Termination ensures that the ARS in a system will eventually complete, and not run indefinitely.

Example: Our Roman Numeral program is terminating. This is because `count ((Roman I)I)` reduces to `1 + count (Roman I)`, then `2 + count Zero`, and then finally `2`.

Normal Form: Normal Form is a state that is no longer reducible. If some state has reductions to a state that is in Normal Form, then we can call the initial state Normalising.

Example: In our Roman Numeral program, `count Zero` is a Normal Form because it cannot be further reduced by `count`.

Unique Normal Form: A state has a Unique Normal Form if it terminates at only one state.

Example: In our Roman Numeral program, `count Zero` is a Normal Form because it cannot be further reduced by `count`.

2.2.2 Proving Termination

The real problem here is dealing with a system that does not terminate. Of course we could always evaluate every possible reduction, but on larger systems the computational power required to visit every branch would be too expensive for whatever the job is that we're doing. Instead, we can create a Measure Function to prove that a computation is approaching termination. This is done by creating a function of a subset of the items in a state that computes a numerical value that is decreasing every computation. An example of a Measure Function evaluating is shown below:

```
State: (ℕ, ℕ)
Rules: $(i, j+1) \to (i, j)$, $(i+1, j) \to (i, i)$
Measure Function: $\phi(i, j) = i^2 + j$
$(4,3)$ $\to \phi(4,2) = 18$
                    $\to \phi(4,1) = 17$
                              $\to \phi(4,0) = 16$
                                        $\to \phi(3,3) = ...$
                              $\to \phi(3,3) = ...$
                    $\to \phi(3,3) = ...$
          $\to \phi(3,3) = 12$
                    $\to \phi(3,2) = 11$
                              $\to \phi(2,2) = ...$
                              $\to \phi(3,1) = 10$
                                        $\to \phi(3,0) = 9$
                                        $\to \phi(2,2) = ...$
                    $\to \phi(2,2) = 6$
                              $\to \phi(2,1) = 5$
                              $\to \phi(1,1) = 2$
```

$\rightarrow\ \phi(1,1) = 2$

$\quad\quad\rightarrow\ \phi(0,0) = Terminated$

$\quad\quad\rightarrow\ \phi(1,0) = 1$

$\quad\quad\quad\quad\rightarrow\ \phi(0,0) = Terminated$

2.2.3 String Rewriting to Check Types

We can apply these termination checks to Haskell through the type checking system. Say we try to write a Lambda Function `(\x. xx)(\x. xx)`. This Lambda Function has one reduction available to use at first sight: `(\x. xx)(\x. xx)`$\rightarrow$`(\x. xx) (\x. xx)`. And now we're right back where we started. This function's ARS is $(A, a \rightarrow a$, which does not get us anywhere. Looking back at the function again from a typing perspective, this function's type is `(A -> A)-> (A -> A)`, which is providing an infinite loop. We can see that this function does not have a Normal Form and does not terminate. So now we've learned that a function cannot call itself. This creates a constraint on the entire programming language.

3 Project

This project will involve creating a command line viewer for Multibrot zooms. The Mandelbrot set is "the set of complex numbers c for which the function $fc(z) = z^2 + c$ does not diverge to infinity when iterated from z=0".[2] This project took inspiration from multiple community threads connecting programmers from all around to create the Mandelbrot set in their own favorite language.[3] Many sites also provided design layouts for how exactly to calculate each state of the zoom recursively,[4] but most used external tools to do all the rendering work. I was not confident with working on a new GUI library, so I took on the task of rendering the fractal zoom in commandline using assigned Chars for the brightness of each rendered "pixel". This project puts much of the data structuring and recursive functions talked about before in this report.

3.0.1 New Type for Complex Numbers

The Mandelbrot zoom equation calls for using imaginary numbers, so we get to start out the project by implementing a new datatype to hold both a real and imaginary number. We will be implementing as well operations to create this new type from an `int` and perform basic operations between two complex numbers.

```haskell
-- We need to implement a combination of real and imaginary numbers
-- Here we have a complex number consisting of 2 floats (real, imaginary)
newtype Complex = Complex (Float,Float) deriving (Show,Eq)

instance (Num) Complex where
    -- simple conversion from int to imaginary
    fromInteger n = Complex (fromIntegral n,0.0)

    -- add complex numbers: add reals and imaginaries
    Complex (x,y) + Complex (z,t) = Complex (x+z, y+t)

    -- subtract complex numbers: just like add but not really
    Complex (x,y) - Complex (z,t) = Complex (x-z, y-t)

    -- multiply complex numbers
    -- ex: (1+2i)*(2+3i) = (1*2+(2i*3i)) + (3i*1 + 2*2i)
    Complex (x,y) * Complex (z,t) = Complex (z*x - y*t, y*z + x*t)

    -- divide complex numbers
    -- This does not work! The Num typeclass doesn't have a div operator, apparently...
    --Complex (x,y) / Complex (z,t) = Complex (x/z , y/z)

    -- equation for absolute value of complex numbers
    abs (Complex (x,y)) = Complex (sqrt (x*x + y*y),0.0)

    -- ex: only using it for: (1+i) / (2+0i) => (1/2 + i/2)
complexdiv :: Complex -> Float -> Complex
complexdiv (Complex(x,y)) r = Complex(x/r, y/r)
```

fromInteger: Of course, we need a way to instantiate our Complex numbers. Simple enough, we can use `fromIntegral` to cast an Int into the floating point numbers that we're using, because we need to hold decimals.

Operators: We still have no context for our Complex numbers, so we had to create instances of the operators `+, -, *, and /`. We also notice that there is no `div` operator for the `Num` typeclass, so we have to define that

[2] Wikipedia: Mandelbrot set
[3] Generate a Mandelbrot Fractal
[4] A Mandelbrot Set

separately. Since we cannot divide imaginary numbers(sanely), we ignored the operation on the second floating point number.

3.0.2 Getting Math help from the Community

To get ourselves set up with the predetermined values that will cause the Mandelbrot equation to zoom into the right location for more detail, we looked to the community for our initial parameters.[5] These guidelines suggested a particular initial viewpoint and offset for each state change that would continuously zoom into diverging locations in the fractal. The initial conditions were declared as so:

```
   -- speed multiplier /100%
zoomSpeed = 100

   -- initial view-range, recommended for mandelbrot tests
planeStartBottom = Complex (-2.0,-1.0)
planeStartTop = Complex (1.0,1.0)

   -- offset from the center while zooming
   -- keeps the view on the edge of the set, found this by trial and error
startOffset = Complex (-1.713,-0.000)
```

planeStartBottom/Top: This zoom will be centered around having a viewport and rendering everything within it. These variables resemble the bottom left corner and the top right corner of the viewport.

startOffset: Each render, the center of the viewport will be shifted slightly. I'm not sure why the number set was required, but it seems that each progression of the function will have more and more exciting events happening when the center of the viewport shifts by this value each render

zoomSpeed: We are also not using any special graphical engines. The equation will progress as quickly as the computer running it is able to handle, and will consequently print out the next frame of the fractal. We should set a speed variable in case a slower computer needs to zoom in a little bit more each render. A faster computer would progress equation depth sooner, and may even need to reduce the zoom speed. As a base, the zoom is set at 100 to resemble 100 percent speed. This value can be changed to raise or lower the zoom speed

3.0.3 Applying the Zoom Initializers

To apply these initial values that we just set, we need to create a function that will take the current viewpoints, the offset, and the zoom speed to calculate how far into the fractal we will zoom in for each render. This is done with a messy but simplistic function here:

```
-- Initial frame bottom point -> Initial frame top point ->
-- Centering offset -> zoom multiplier*render loop number
-- takes the current corner points of the frame and adjusts them to be smaller based on the
    zoomSpeed and offset
zoom :: Complex -> Complex -> Complex -> Float -> (Complex,Complex)
zoom bot top offset zoom = (zoomAmount bot, zoomAmount top)
   where -- shifting the viewport, based on a function from typeclasses.com
      zoomAmount initialPoint = (complexdiv (offset*(Complex (zoom, 0.0)) + initialPoint ) (zoom
          + 1))
```

planeStartBottom/Top: As implemented in the hrefhttps://typeclasses.com/art/mandelbrottypeclass.com program, the viewport can be updated as a function of the offset, zoom, and initial viewpoint. The return

[5]Generate a Mandelbrot Fractal

value is a pair of complex numbers, which will be our new viewpoint(remember? bottom left corner, top right corner)

3.0.4 Rendering to CommandLine

Now it's time to tackle the annoying but fun part. Rendering to commandline was not as intuitive as I planned, but I found some more help from the community. We need to create a list of characters to resemble the diverging and safe location in our viewport, and then map those characters to a heatmap of values.

```
-- Render function
-- Takes a complex number, maps darker chars to pixels that diverged the least
mandel :: (Complex,Complex) -> String
mandel (bottomleft,topright) = concat maprenderToText renderValues bottomleft topright
   where
      renderToText (i,nextRow) = "
          .’‘^\",:;Il!i><~+_-?][]}{1)(|\\/tfjrxnuvczXYUJCLQOOZmwqpdbkhao*#MW&8%B@
```

We now have a function that will take some viewpoint and return a big long string that maps darker characters to higher values.

renderToText: We will create an indexing function for our characters. The 70 characters are taken from hrefhttp://mewbies.com/geek$_{f}un_{f}iles/ascii/ascii_{a}rt_{l}ight_{s}cale_{a}nd_{g}ray_{s}cale_{c}hart.htmlMewbiews.com, becausetheyjustsoh$

3.0.5 Creating the Heatmap

Perfect. Now we have a function that will map characters to values that we expect to be between 0 and 32 based on a point's divergence as z approaches 0. That means we need to create a heatmap that will convert diverging and non-diverging values into a scale between 0 and 32. This is where I got the most help from the community, as the paper A Parallel ASCII Mandelbrot Renderer in Haskell referenced the commandline rendering implementation. This is implemented below:

```
-- Somewhat unknown function?
-- Initially mapped values across a 2D plane and converted them to different values based on a
    scale
-- The scale we use is 32. For whatever reason, this number works out with the contrast for our
    shading chars
renderValues (Complex (a,b)) (Complex (x,y)) = map (\z -> (progress (Complex z) (Complex(0,0)) 32,
    (fst z > right - hstep/2 ))) [(x,y) | y <- [bottom,(bottom + vstep)..top], x<-[left,(left +
    hstep)..right]]
   where
      top = y
      bottom = b
      left = a
      right = x
      vstep=(top-bottom)/40
      hstep=(right-left)/80
      -- dividing the range of the vertical and horizontal step will scale the number of values
          per row and allow us to set the resolution of the output
```

The syntax of this function is extremely confusing to me, and I'm not sure that I would have been able to figure out how to do this myself. This function accepts pairs, and draws a heatmap of coordinates between the pairs. We manipulated this to include our Mandelbrot equation which will be able to scale extreme values up to 32. As you probably noticed, there is a function called `progress`. This is our implementation of the Mandelbrot function that we still need to create. Let's define that now.

3.0.6 Tying it Together

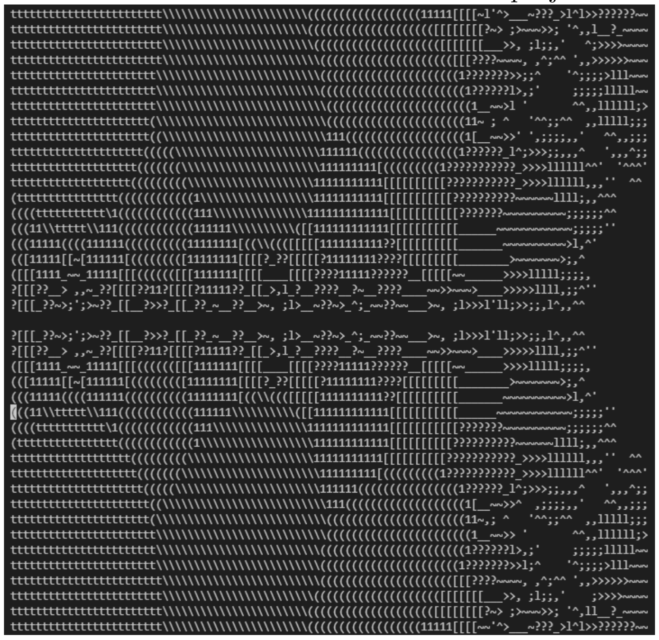
```
    -- apply the progressive function
    -- on a complex number value, keep squaring it and it may or may not diverge. Non-divergence
         will fall out of the shading sweetspot
    -- after 32 squares, we will assume it has diverged enough already and we will not print that
         pixel
progress :: Complex -> Complex -> Int -> Int
progress c z 0 = 0
progress c (Complex (x,y)) n = if (abs x > 2) then n else progress c (((Complex (x,y))*(Complex
     (x,y)))+c) (n-1)
```

Note that we will be looking at extremely small values and begin zooming in on them. By testing out the look of the graphical output, it seemed that values that ever reach 2 began to diverge quickly and flooded the view. Many of these numbers were trial-and-error to find out which combination of values provided the right sensitivity that offered a satisfying output. The last thing we need to do is implement a looping function.

```
    -- main function
    -- starts loop and redoes it
main = do
    putStrLn (loopNextFrame 0)
    where
        loopNextFrame n = mandel (screen n) ++ "\x1b[H\x1b[25A" ++ loopNextFrame (n+1)
        screen n = zoom planeStartBottom planeStartTop startOffset (.01+zoomSpeed*0.00005)
```

Like we covered before, this function recursively calls itself without any timeout. Renders will be executed as soon as the system running it has dedicated enough resources to do so. We are calling our string pairing function on the new viewpoint every frame, which in turn gathers a heatmap of values to pair to a shading character. The use of `"\x1b[H\x1b[25A"` was meant to clear the console whenever a new render was executed.

That's it! That was all there was to it for this project. But how does it look?



3.0.7 Project Conclusion

Excellent! The shading characters worked out perfectly. This project put use to creating data structures, pattern matching, designing Lambda Functions, and most importantly for the graphics, recursion!

A link to a Replit page showcasing the project can be accessed here.

4 Conclusions

Throughout the report, we covered many important concepts in Haskell. To bring it further, we looked at the theory of Lambda Calculus to better understand the low-level workings of the language. We covered the historic uses of Haskell and where it originated(and how it got its name). We hit the conceptual parts of the language to explain the usefulness of declarative programming and the seemingly endless benefits of being lazy. The applications of types and functions were focused and presented to demonstrate why the creators of Haskell implemented the features they had. The depth of how Haskell works in terms of better syntax with functions and currying was also mentioned, with many examples of how to improve and reuse code. The theories we covered on programming languages in general included Lambda Calculus and String Rewriting, with plentiful examples of how they relate back to our work with Haskell. The report was finished with a mathematical beauty, all written in ASCII characters. The final application of all the tools talked about in the report brought seriousness to all the features of Haskell. It has been a privilege to live in a time where we can enjoy the great breakthroughs that researchers and developers brought to the world.

References

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

[Stackoverflow] Why does x*x where x = 6 fail in Haskell?

[haskell.org] Predefined Types and Classes

[Dummies.com] Complex Number Operations

[typeclasses.com] Mandelbrot with GUI

[stackexchange.com] Generate a Mandelbrot Factal

[cs.columbia.net] CLI Rendering

[Mewbies] Shade Character String