CPSC-354 Report

Michael Lewson
Chapman University

December 21, 2021

Abstract

This report describes the theory and applications of the Haskell programming language. The first section is a tutorial and describes Haskell along with its applications today. The second section explores various aspects of programming language theory such as lambda calculus, abstract reduction systems, and string rewriting systems. The third section applies a string rewriting system in a project to implement arithmetic with Roman numerals.

Contents

1 Introduction

Haskell is a functional programming language with many applications in software, hardware, and technology companies. The first section of the report discusses Haskell as a functional language including a tutorial on Haskell going into depth on function structure, pattern matching, types, how to write programs in the language, as well as a brief look into Haskell's history and current uses today. The second section of this report delves deeper into programming languages theory with an emphasis on lambda calculus and exploring the concepts of confluence and termination with abstract reduction systems and string rewriting systems. The third section of the report further explores string rewriting with a project on Roman numeral arithmetic including addition, subtraction, and multiplication. The examples and project used in this report can be found here [ML].

2 Haskell

2.1 Haskell as a Functional Language

Haskell is a purely functional programming language. The functional approach to programming is similar to mathematical functions with expressions that cannot mutate any arguments but instead simply calculate a value and return the result. This varies greatly from imperative languages such as Python and Java with functions that are composed of sequential steps that are able to change a program's global state and use the states of global variables in their functions. In functional programming, there are no state changes which means that once a variable is set to a value it cannot be changed. Haskell cannot change the value of variables with functions. In addition, in functional languages, the order of execution of a function is not as important as functions can oftentimes be executed in different ways and still return the same result. However, this is not the case with imperative functions as the order of execution is critical.

Haskell's functional nature is important to create self-contained stateless functions. This is helpful to increase readability and overall maintainability of Haskell code as each function is created for a specific task and does not rely on external variables. This helps in terms of debugging functions as well as making the code easier to refactor and implement in other functions. These features combined help make Haskell easier to debug and test.

2.2 Learning Haskell

Haskell can be tricky to learn especially for newcomers who are trying to learn their first functional programming language. The best method I have found to learn Haskell is by doing various practical exercises. Each example reinforces the syntax and the users' understanding of the language as it is critical to learn by doing instead of just skimming examples. The following sections go into more depth about the various aspects of Haskell's syntax along with examples for newcomers to learn as they read along.

2.3 Haskell Types

In Haskell, every value has a type. In addition, Haskell is statically typed which means that variable types are explicitly declared. This allows the compiler to perform type checking at compile time [HA]. This is critical in Haskell as the compiler will check if the types of a function and variables match, preventing errors. Understanding type errors is the key to troubleshooting many problems in Haskell as oftentimes the compiler will give error messages of what types of variables it was expecting in its input. Oftentimes ensuring the types match will fix the problem in the output of the function entirely. In addition, Haskell has type inference which allows the language to infer the type of the variable being used in its functions.

2.4 Haskell Control Flow and Function Structure

As a functional language, the primary control flow of Haskell involves function calls and recursion. This differs from imperative languages such as Python and Java that can use loops in addition to function calls and recursion [MS]. Functions in Haskell do not use global variables, which can make debugging and testing easier as functions rely solely on their arguments. This allows for easier testing of any value including edge cases without influences of global states. Haskell functions are also easier to apply recursively due to how they are composed [MS].

Haskell's functional structure is unique and varies from imperative programming languages. To visualize this, a simple function to learn for Haskell will be the increment function that takes a single argument and returns the number incremented by one. In this case, it can be seen that Haskell functions are defined by their equations. The function name, increment, is followed by its argument x. The '=' defines what the function does which takes the argument x and returns the value of x + 1. Below is the implementation in Haskell followed by an implementation in Python.

```haskell
-- First Haskell Function
increment x = x + 1
```

```python
--Python equivalent of Increment
def increment(x):
   return x + 1
```

Both the increment functions from Haskell and Python define the function as increment and return the incremented argument. In Haskell, the argument does not need to be enclosed in parenthesis while it is a requirement in Python and Java. This detail while seemingly insignificant, is critical to be aware of as more complex functions can have multiple arguments. As a result of multiple arguments, parenthesis placement becomes critical as as an incorrect parenthesis can change the intended input value. In addition, it is important to note that Haskell is able to infer through type inference that the input value is supposed to be a number. This is helpful for saving time and allows the code to be more general and work on any numbers. However, this type inference has its limits as the function will not execute properly if a non-number input such as a string is entered as its argument.

To add types to a function, a type signature declaration is used to map the type to the inputs and output of a function.

```haskell
-- First Haskell Function Version 2.0
increment :: Int -> Int
increment x = x + 1
```

The :: can be interpreted as "has type" followed by the input type of integer with an arrow pointing to the right designating an integer output.[HAV]. Having a designated type signature helps for debugging as the compiler can check for type errors.

To designate more than one input, additional arrows are used. To showcase this, a function to add two integers and return an output was made called addTwoInts. Below is the function in Haskell and Java showing their respective type signatures.

```haskell
addTwoInts :: Int -> Int -> Int
addTwoInts x y = x + y
```

```
--Java Implementation
public static int addTwoInts(int x, int y){
   return x + y;
 }
```

The Haskell implementation utilizes two arrows with the first two Int's acting as the input type and the last arrow designating the output type. Both the Java and Haskell implementation now share the same type signature.

In this example, inputting a float to the Haskell function instead of an integer will now produce a type error. For example, the input "addTwoInts 5.0 2" will result in the error, "No instance for (Fractional Int) arising from the literal '5.0'". While the nature of the error is rather straightforward, a fractional int instead of an integer, when more abstract data types are used, it is important to note the type error and what type of argument was entered into the function.

Haskell arguments can also be enclosed in parentheses "(x,y)" vs. "x y". This detail is important as the difference between two inputs or a tuple of inputs has to be acknowledged when using the function.

```
addTwoInts (x,y) = x + y
```

The above function is known as a curried function as it has a tuple as an input [HAF]. This distinction is important to ensure the function has the appropriate input. In addition, precise parenthesis placement is critical for functions that have inputs of other functions. Below is a brief example of using a function with inputs of other functions using the addTwoInts function.

```
addTwoInts (addTwoInts 1 2) (addTwoInts 3 4)
10
```

The function adds the two values obtained from "(addTwoInts 1 2)" and "(addTwoInts 3 4)" yielding 10. Correct parenthesis placement ensures that the first addTwoInts function is getting two input arguments that are integers. Without the correct parenthesis placement, the program would not compile as the inputs of the function would not properly match.

2.5 Pattern Matching

In addition, Haskell functions have pattern matching, the ability to match patterns on the left-hand side of the function equation to the parameters. If the pattern matches the parameter, the right-hand side is then evaluated and returned as the result. If it does not match, the function proceeds to the next parameter and attempts to match it. Below is a simple example of pattern matching with integers yielding a string output.

```
--Haskell Pattern Matching
patternMatch :: Int -> String
patternMatch 1 = "Option 1"
patternMatch 2 = "Option 2"
patternMatch x = "Every other option"
```

In the case of the example above, it checks if the input integer matches the left parameter. If the input is 1, the output is "Option 1" and if the input is 2, "Option 2" is returned. The x is used as a catch-all if none of the options match. This would result in the output, "Every other option". If there was no option x, Haskell would throw an error for non-exhaustive patterns as seen below.

```
*** Exception: test.hs:(33,1)-(34,27): Non-exhaustive patterns in function patternMatch
```

Below is the equivalent program written in Java.

```
--Java Implementation
public static String patternMatch(int x){
  switch (x) {
    case 1:
      return "Option 1";
    case 2:
      return "Option 2";
    default:
      return "Every other option";
    }
}
```

In the Java implementation, a switch statement is used for each input. While this example is simple, Java will tend to have more lines of code for each case option, Compared to Java, Haskell is more concise with its pattern matching without the need of several if-else statements or case statements.

2.6 Recursive Functions

Pattern matching allows for clean and concise recursion in Haskell. One of the most basic examples of recursion is the factorial function. The factorial function takes a positive integer input and multiplies the number by all the whole numbers that are smaller than it. Below is the implementation in Haskell followed by the implementation in Java.

```
factorial :: Int -> Int
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

```
--Java implementation of factorial
public static int factorial(int x){
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial(x-1);
  }
}
```

The function calculates the factorial of any positive integer by recursively multiplying itself by the decremented input each time the function is called. The Haskell version of the function consists of three lines written concisely with pattern matching. The second line is the base case of 0 returning 1 and the third case is when the factorial recursively loops and decrements x by 1. The Java implementation is similar with a base case of 0 in an if else statement. It is important to always have a base case so that recursive functions end. Between the two functions, it can be seen that the Haskell function is more concise and easier to read than the Java. As functions become more complex, Haskell will yield more concise and compact functions through pattern matching.

2.7 Haskell as a Lazy Language

To better understand Haskell as a lazy coding language it is important to understand what a strict coding language is. A strict coding language such as Python or Java evaluates the arguments before they are passed into a function [SOH]. If a function were to have an argument with a value that took a long time to calculate, the function would calculate it first before passing it to the function to use even if the function did not use the value. This can result in time wasted as the argument may not have needed to be calculated in the first place. By contrast, Haskell is a lazy programming language which does the opposite; functions do not evaluate arguments until they absolutely need to return a result. This is important as oftentimes in recursive situations in Haskell, you may have an argument that is an expression that may not need to be evaluated immediately. This is important in regards to efficiency as Haskell calculates a value when its needed. Haskell's lazy nature overall adds more efficiency but it can also add a small degree of confusion of when statements are evaluated.

An example of Haskell's lazy nature can be seen below in the example, lazyEvaluation. This function takes two arguments as inputs and returns the first argument and does not use the second argument. The second argument in this case is the value (4/0) which would normally return an error.

```
lazyEvaluation :: Double -> Double -> Double
lazyEvaluation x y = x

print $ lazyEvaluation 4 (4/0)

4.0
```

As seen above, Haskell's lazy nature only evaluates the argument that is absolutely needed. As y in this situation is not needed, the value is never calculated. Below is the implementation in Python where the error would be thrown.

```
def strictEvaluation(x, y):
    return x

print(strictEvaluation(4.0, 4/0))

ZeroDivisionError: division by zero
```

In this case it is clear that even though the second argument is never used, Python evaluates it and throws an error.

Lazy programming is important in regards to pattern matching and recursion. It also allows some interesting uses with infinite data structures such as an infinite list.

2.8 A Brief Introduction To Monads in Haskell

Monads are an abstract way to structure a program to introduce a degree of imperative programming with state changes to Haskell. They are a complex topic but can be thought of as abstract data types in this context. Monads are based on category theory in mathematics which provides the names for classes and operations. They consist of three parts with the first part being a type constructor that returns a value of the designated type [IMT]. The second part is a function that takes a value and returns a computation that produces the value. The third part takes two computations and uses the result of the first computation in the second one.

The Haskell wiki describes the Monad class below [HW].

```
class Monad m where
  (>>=)  :: m a -> ( a -> m b) -> m b
  (>>)   :: m a -> m b        -> m b
  return ::  a                -> m a
```

Monads implement the class functions as well as follow the three monad laws, the left identity, the right identity, and the associativity law.

$$\text{Left identity: return a >>= f} \equiv \text{f a}$$

The left identity is equivalent to f(x). The ">>=" is equivalent to applying a value to a function. In total, a value is returned and applied into a function [LYH].

$$\text{Right identity: m >>= return} \equiv \text{m}$$

The right identity takes a monadic value that is returned with the ">>=" notation. This yields the value "m" which is the original monadic value. To simplify, both the left and right identities designate how values are returned for monads.

$$\text{Associativity: (m >>= f) >>= g} \equiv \text{m >>= (}\backslash x -> \text{f x >>= g)}$$

Lastly, associativity involves a chain of monadic functions that are nested. This law means that you can essentially rearrange how the functions are nested and still yield the same result. [LYH].

```
"(m >>= f) >>= g" is equivalent to " m >>= (\x -> f x >>= g)"
```

In this specific example m is a value that is inputted into the function f that yields the value and gives it to g. In the second expression, m is given to a function that gives the result of f x to g [LYH].
These laws are used to define how all monads should function.

One example of a simple monad is the Maybe monad. Below is the definition of the Maybe type.

```
data Maybe a = Nothing | Just a
```

The Maybe type is interesting as it could be a value of "a" which is represented as "Just a", or it could be empty represented by "Nothing". The maybe monad is helpful as an error monad as errors can be represented as "Nothing" instead of just crashing. Below is an example of division using Maybe.

```
maybeDivide :: Float -> Float -> Maybe Float
maybeDivide x 0 = Nothing
maybeDivide x y = Just (x/y)
```

Normally division would fail if a number were divided by 0. In this example, a "safe" division was created by using the Maybe type. When the denominator is 0, "Nothing" is returned. When the denominator is a non-zero number the value is properly returned with "Just (x/y)".

```
ghci> maybeDivide 4 2
Just 2.0
ghci> maybeDivide 4 0
Nothing
```

Above is a short example demonstrating the function. While this has been a brief dive into monads, monads have many uses in Haskell. The next section explores the use of the I/O monad for input and output in Haskell.

2.9 Input and Output in Haskell

Haskell uses monads to integrate I/O operations and make its I/O more similar to imperative programming languages[IOH]. The I/O monad allows a value such as a String to be read and printed to the screen as an I/O action [IOH].

One of the most basic outputs in all of programming is the famous, or infamous, "Hello, World". There are several functions that write to the output device or user terminal such as putChar, putStr, putStrLn, and print. In main, the function putStrLn will be used to output "Hello, World".

```
main = do
   putStrln "Hello, World"
```

Equivalents to this function can be seen in System.out.println() in Java or print() in Python. To get input, the function getLine can be used to take input from the terminal. This function utilizes a combination of basic input and output by prompting the user for a value and printing the value in a function.

```
main = do
   putStrLn "Please enter your name"
   name <- getLine
   putStrLn ("Hello, " ++ name)

Please enter your name
Michael
Hello, Michael
```

Overall, the simplicity of I/O in Haskell is due to the I/O monad.

2.10 Custom Data Types

To make a custom data type in Haskell, the data keyword is used. Data is used to define a new data type with the value after the = being known as the value constructors which are the different values that the new type can have. Below is a simple example of a new data type using boolean as a basis.

```
data Boolean2 = True | False
```

In the above example, a simple declaration of a new type is seen above where Boolean2 is declared as a type that is either True or False. The "|" is used an OR symbol.

A more complex data type could be for x,y coordinates.

```
data Coordinates = Coord Float Float
```

In this examples, Coord acts as a value constructor with two parameters for two floats. These could represent the x and y coordinates respectively.

```
ghci> :t Coord
Coord :: Float -> Float -> Coordinates
```

Checking the type using, :t, shows the type signature for the expression. An example below is made using the new type.

```
returnCoords :: Coordinates -> [Float]
returnCoords (Coord x y) = [x,y]

returnCoords $ Coord 1.0 2.0
[1.0,2.0]
```

This function takes a Coord as an input as returns the x and y values as values in a list. Another more complicated example uses two coordinates as inputs and returns the distance.

```
calculateDistance :: Coordinates -> Coordinates -> Float
calculateDistance (Coord x1 y1) (Coord x2 y2) = sqrt((x2-x1)**2+(y2-y1)**2)

ghci> calculateDistance (Coord 1.0 2.0) (Coord 3.0 4.0)
2.828427
```

This example extracts the x and y values from each coordinate and calculates the distance between them using the distance formula. It then returns the float value representing the distance.

There are additional features that can be added to a data type. Using deriving (Show), allows a type to be presented as a string which can be useful as the data type can now be printed to the terminal as a string.

```
data Coordinates = Coord Float Float deriving (Show)

ghci> Coord 1.0 2.0
Coord 1.0 2.0

coordExample = Coord 2.0 3.0
print $ coordExample
Coord 2.0 3.0
```

As seen above, the values in the coordinate can now be printed.

Without deriving (show), the following error will be shown.

```
ReportHaskell.hs:58:5: error:
    * No instance for (Show Coordinates) arising from a use of ‘print’
    * In the first argument of ‘($)’, namely ‘print’
    In a stmt of a ’do’ block: print $ coordExample
    ...
```

Overall, custom data types are useful in Haskell as in other programming languages to streamline programming by organizing information in the type.

2.11 History Of Haskell

With a brief tutorial of Haskell finished, it is important to see how the language was created. In 1980s, there were several non-strict functional programming languages such as ML, Hope, and Miranda [HCPL]. As a result of several programming languages being present, there were few users and development in each of these was slow. In 1987, there was a conference on Functional Programming Languages and Computer Architecture to discuss the functional programming languages community. As a result of these discussions, it was proposed that a new functional programming language should be created that would be more practical and efficient [HHLC]. Three years later, the committee published the Haskell Report in April 1, 1990 discussing the motivation for the creation of the language and how it should be non-strict and purely functional. In addition, Haskell would be designed by the committee.

Over the years, the features of Haskell grew. In 1992, GHC was created which is the Haskell compiler as well as the first Haskell tutorial was created. In 1996, Haskell gained significant changes with the inclusion of the library report, monadic I/O, and the extension of algebraic data types[HCPL]. In 2005, there was a Haskell Workshop that sought to update and modernize the language with a series of smaller updates over time. The next significant change to the language happened with the release of Haskell 2010. This revision added a foreign function interface, hierarchical module names, pattern guards, and removal of the (n+k) pattern syntax. Plans for Haskell 2020 are in development but progress at the moment is currently stalled.

2.12 Haskell in the Industry Today

Currently, Haskell is used in a variety of commercial settings including aerospace and defense, finance, web startups, hardware design firms and even a lawnmower manufacturer. Some examples of companies that use Haskell are AT&T, Facebook, Google, and Amgen. The phone company, AT&T, uses Haskell for the automation of handling internet abuse complaints in their Network Security Division. Facebook uses Haskell for their tools and manipulation of a PHP code base. Google uses Haskell for internal projects, IT infrastructure, and the Ganeti project for managing virtual server clusters. In the biotechnology industry, the company Amgen is a human therapeutics company. Amgen uses Haskell to build software for mathematical models for their molecular biology and medicines, provide mathematical validation of their software, and give their developers new options to write software [HII]. Overall, Haskell has many applications in a variety of different commercial and industrial settings.

3 Programming Languages Theory

3.1 A Brief Introduction to Discrete Mathematics

To better understand some of the concepts of Haskell and Programming Languages theory, it is good to have an understanding of the mathematical concepts of collections and sets. A set is an unordered collection of elements. The notation for a set involves the elements enclosed in braces separated by commas such as a, b, c. These elements could be anything from numbers to letters to variables. To dictate whether an element is in a set, the notation

$$x \in A \tag{1}$$

is used to indicate that x is an element in set A.

Examples of sets can be seen as real numbers and natural numbers where $\mathbb{R}$ is the set of all real numbers and $\mathbb{N}$ is the set of all natural numbers. Real numbers can be defined as numbers that include whole numbers, rational numbers, and irrational numbers. Natural numbers are defined as positive integers starting at 1 and going to infinity.

A set is a a subset if every element of X is an element of set Y.

$$X \subset Y \tag{2}$$

An example of a subset could be 1,2,3 is a $\subset$ of $\mathbb{R}$

In addition, to properly understand some concepts in programming theory such as equivalence and unique normal forms, knowledge of some mathematical properties is also important. Three properties worth discussing are three different relations, reflexivity, symmetry, and transitivity.

Definition:

$$Assume\ R \subseteq \text{A x A with the relation R}$$

Following are the formal definitions of reflexive, symmetric and transitive relations.

Reflexive:

$$if\ \text{xRx}$$

$$\text{for all x} \in \text{A}$$

A relation is reflexive means for all x, x is going to relate to itself. For example, 1⊂A, then (1,1) will always be in $\mathbb{R}$.

Symmetric:

$$if\ xRy \Rightarrow \text{yRx}$$

$$\text{for all x,y} \in \text{A}$$

A relation that is symmetric means that for all elements x and all elements y, if x is related to y then y is related to x. An example of this can be seen below.

$$1{\neq}2 \rightarrow 2{\neq}1$$

$$\text{xRy} \rightarrow \text{yRx}$$

Inequality can be shown if 1 is not equal to 2, then 2 is not equal to 1. The next definition is for transitive.

Transitive:

$$if\ xRy\ \&\ yRz \Rightarrow \text{yRz}$$

$$\text{for all x,y,z} \in \text{A}$$

A relation is transitive if x is related to y and y is related to z, then x is related to z. A good example of transitivity is with the ≤ operator.

$$1 \leq 2 \text{ and } 2 \leq 3 \text{ which implies } 1 \leq 3.$$

While this is a very brief introduction to discrete mathematics, these concepts are useful for understanding theory later on in regards to computations in Haskell as well as general programming theory.

3.2 Turing Completeness

To understand Turing completeness, it is important to understand what a Turing machine is. A Turing machine is a concept created by Alan Turing in 1936 which is an abstract computational device that is used to determine the extent and limitations of what can be computed [TM]. Alan Turing created this in an attempt to solve the Entscheidungsproblem. The Entscheidungsproblem was whether or not an algorithm exists that could answer yes or no to all mathematical questions. The Turing machine was created for research into the foundation of mathematics with his idea at the time being similar to a human following a list of written instructions on a piece of paper tape. The human could read the instructions and write what's on the tape then move one place left or right on the tape which would be similar to a human following a computer program [TC]. This piece of tape could effectively be infinite and with an actual computer would imply infinite resources. In mathematics, this concept would be called finite but not unbounded as the Turing machine is not limited by resources [TC].

The definition of Turing complete is that a program can be used to simulate a Turing machine. In other words, for a program or machine to be Turing complete it would be able to solve any computational problem no matter the complexity. For this to work, it is also assumed that the machine or program is not bound by resources and memory to ensure that all problems that could eventually be solved would be computed.

3.3 A Brief Introduction To Lambda Calculus

Lambda calculus is a notation used to describe mathematical functions and programs. There are three major components to lambda calculus: variables, abstraction, application. The main concept of lambda calculus is its ability to apply a function to an argument and then form functions by abstraction [LCC]. This allows for the creation of functions as the rules of computations.

Abstraction is when a variable becomes bound in an expression. This is seen in the expression λx.e where x is the variable and e is the expression in which x is bound. This term is known as the identity function as it returns its argument, x, as its result. A value being bound to the expression is shown below.

```
(\ x . x) 1
Output: 1
```

In this example, the expression is simply x, and now x contains the value of 1. The next example shows the value 1 being bound and then applied to the expression.

```
(\x.x+1) 1
Output: 2
```

The value of 1 is bound to x and then increased by 1.

Application is when a function is applied to an argument. In this example, the two arguments are a function and a value which then applies the function to the argument.

```
(\f.\x.f x) (\y.y+1) 2
Output: 3
```

It can be seen above that the expression (λy.y+1) is bound to f then the value of 2 is bound to x. Then the expression f x is calculated with 2 being bound into y and incremented by 1 yielding 3.

In lambda calculus, variables are either declared or used. Where a variable is declared is known as the scope of the variable, where the variable is defined and usable. Inside the scope, the variable x is bound to some value [FBV]. If a variable is not bound, then it is free as there is no value bound to it within the scope of the function. A basic example of scope and a bound variable is seen below.

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

In this example, x is the bound variable as the input to the function is bound to x. The free variable in the function is y as there is no value bound to it within the scope of the function. The scope of the function is where x is defined in the braces. Calling function(1) would result in x being bound to 1 and then the value 1+y being returned.

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

In the lambda calculus equivalent of this function, x is bound to the first lambda. Once again, y is the free variable as it is not bound. The scope of x is defined within the parenthesis right of the lambda. If the value of 1 was given as an input it would yield the same output of 1+y.

Lambda calculus is Turing complete because it can be used to simulate a Turing machine which means lambda calculus is able to create expressions with states and functions that can apply to them [LCC]. This could also be used to recreate the original Turing machine by replicating the infinite tape. For example, one could have a lambda calculus expression that simulated the machine and tape. Assuming the tape is a list, one could encode each element in the list to be a tuple with multiple elements containing the information to move the tape and an operation to encode.

3.4 Fixed-Point Combinator

One method of encoding recursive functions in lambda calculus is with the fixed-point combinator. A fixed point of a function is an value that maps to itself by the function such as f(x) = x. A fixed-point combinator is a function that returned the fixed point of its argument function [FPC]. An example of a generic fixed point combinator is below.

```
fix f = f (fix f)
```

With additional repeated application, the function can expand forever to yield the following.

```
fix f = f(f(...f(fix f...))
```

One example that can showcase this would be the Fibonacci function as seen with the equation below.

```
Fib n = (if n == 0 then 0 else if n == 1 then 1 else fib(n-1) + fib(n-2) )
```

In lambda calculus, the Fibonacci function is shown below. The function has two lambdas, one for the function to bind to and another for the value.

```
(\ fib. \ n. if n == 0 then 0 else if n == 1 then 1 else fib(n-1) + fib(n-2) )
```

An example of a Fibonacci calculation is below with fib 0.

```
fib 0 = (\n. if n == 0 then 0 else if n == 1 then 1 else fib(n-1) + fib(n-2) ) 0
Beta reduction binds 0 to n yielding the expression
if 0 == 0 then 0 else if 0 == 1 then 1 else fib(0-1) + fib(0-2)
As 0 == 0, it returns 0.
```

Fib 0 is relatively straightforward as beta reduction binds 0 into the factorial equation where ”0 == 0” returns 0. Another example with Fib 2 is shown below.

```
fib 2 = (\n. if 2 == 0 then 0 else if 2 == 1 then 1 else fib(2-1) + fib(2-2) ) 2
Beta reduction binds 2 to n yielding the expression
if 2 = 0 then 0 else if 2 == 1 then 1 else fib(2-1) + fib(2-2)
fib 1 + fib 0 = (\n. if n == 0 then 0 else if n == 1 then 1 else fib(n-1) + fib(n-2) ) 1 + fib 0
if 1 == 0 then 0 else if 1 == 1 then 1 else fib(n-1) + fib(n-2) + fib 0
1 + fib(0)
1 + 0
1
```

In this example, fib 2 requires several computations and reductions. In the first step, 2 is bound into the equation where the else statement is triggered and the value of fib(1) + fib(0) is now called. The reduction for 1 follows the same logic as 0 where 1 is bound into the expression and returns 1 which is added to fib 0 yielding 1. Being able to apply recursion in lambda calculus is important as its a powerful tool for computations.

3.5 De Bruijn Indices

De Bruijn indices are tools for representing terms of lambda calculus without naming bound variables [DBI]. Essentially, de Bruijn indices replaces variables with numbers that represent the occurrence of the variable in the lambda term. These numbers also denote the number of binders in the scope between the occurrence and its corresponding binder[DBI]. The exact number of whether or not it starts with 0 or 1 varies with the definition but the indices function the same regardless. For this paper, 0 will be used as the variable numbers will be based on stack offset with the last variable being accessed with 0, the second to last with

1, and repeating following that pattern. To use de Bruijn notation, parentheses (or braces) are used for grouping and numbers are bound into the free variables with beta reduction.

Lambda calculus: λx. x

De Bruijn Indices: λ. 0

Reduced Form: [0]

In this case, with only one variable present, the value 0 is used and then brackets are used to group around the value.

An example with two variables is shown below.

Lambda calculus: λxy. x

De Bruijn Indices: $\lambda\lambda$. 1

Reduced Form: [1]

In this case, it can be seen that the value is one as x would be the second to last variable while y being the last variable would be assigned 0.

Another example that shows how de Bruijn indices help simplify lambda calculus can be seen with the y combinator seen below [MRLC].

Y = λf. (λx. f (x x)) λx. f (x x)

Y = [[1 (0 0)] [1 (0 0)]]

This example reduces the equation to 1's and 0's and shows the repetitive nature of the function with [1 (0 0)] repeating. Overall, de Bruijn indices are useful to simplify lambda calculus terms and can make them more easily readable.

3.6 Halting and Undecidable Problems

The halting problem is a decision problem in computability theory which is whether an input given to a program will result in the program terminating or running forever. It is undecidable because there's no general procedure to determine if a self-contained computer program will eventually terminate [THU]. Alan Turing proved in 1936 that there was no Turing machine that could fit this specification.

The halting problem can be seen as a decision problem H with two arguments consisting of a program, p, and an input, x, and two outputs, true for loop and false for terminate. In other words,

```
H(P,x) = P terminates on x.
```

To prove the halting problem does not exist, we prove by contradiction. We assume H does exist then prove that it cannot. One way to do this is to use P as both the program and input to H. If H loops, it means that P with input P does halt but if H loops it means that P doesn't actually halt [PL, HP]. Likewise, if H halts, it means the program doesn't halt on itself but if H does halt it means that P has to halt. These two statements are paradoxes with a true = false situation that is used to derive the contradiction. As the halting machine cannot exist, the Turing machine cannot exist. Therefore, there are no general solutions to the halting problem.

The halting problem can be used to prove other problems are undecidable. A real world example could be turning our halting machine into a halting check application [HP]. For example, now our halting machine is now a halting app that checks if another application, I, runs forever or halts. If the input I terminates, then the halting app accepts and determines that the application will not run forever. If the input I runs forever, then the halting application rejects. However, like other problems, this real world application at its core reduces back down to the halting problem, if an application can determine if another application runs forever. The answer is still no as the halting application can receive the halting application as its input and creates another paradox that would result in the halting problem of true = false and is not applicable in the real world.

3.7 A Brief Summary Of Abstract Reduction Systems

An abstract reduction system is a set of objects and the rules that can transform them. It is an abstraction based on term rewriting systems where for

a,b $\subset$ A if A $\rightarrow$ R

This means that a is said to reduce or be rewritten to b.

Confluence is a property where a system can be written in multiple ways and yield the same result. This can be important for languages such as Haskell, where pattern matching can result in different rule applications but the result will converge. Confluence is generally visualized as an expression branching outwards into multiple branches that then converge back into the same expression in the next step. This visually is known as peak, diverging, and valley, the rejoining step. A common example of confluence is with arithmetic. The expression

$$(1 + 2) * (3 + 4)$$

could be evaluated as

$$3 * (3 + 4) or (1 + 2) * 7$$

and they both converge at the value 3*7 showing that for there is a peak for every valley.

Termination is relatively straightforward meaning that a sequence will eventually return a final result or has gotten to a point where no rules can apply. In the above expression, regardless of which rule was applied first, the above expression terminates with the value of 21. An expression that could not terminate could be with the expression a with the rules (a $\rightarrow$ b) and (b $\rightarrow$ a). This would not terminate as the "a" would become "b" and the "b" would become an "a" and endlessly repeat. An even more reduced form could simply be (a $\rightarrow$ a) as the expression "a" would become "a" and endlessly repeat.

Confluence and termination lead to the existence of normal forms. A normal form is when an expression cannot be reduced any further. In this case, it is an expression that cannot have any more rules applied to it. A unique normal form is if all elements in the ARS reduce to the same normal form. Examples of normal forms will be seen in the String Rewriting section.

3.8 String Rewriting

String rewriting is an abstract reduction system reduces a large string into a smaller string with a series of rules. A set S consisting of a string or alphabet is reduced through a series of rules a $\rightarrow$ b. These rules are often written as a tuple of form (a,b) where the substring a is rewritten into substring b. A very simple example of a string rewriting system could be seen in the example below.

```
"ab" with the rule, (b, c), yields "ac".
```

Now this is a small string with only one possible normal form, the unique normal form. To demonstrate some attributes of the string rewriting system, a function was created in Python below.

```
def rewrite(rewrite_rules, output):
    for rule in rewrite_rules:
        if rule[0] in output:
            print(output,":", rule[0], "->", rule[1], "=", output.replace(rule[0], rule[1], 1))
            output = output.replace(rule[0], rule[1], 1)
            output = rewrite(rewrite_rules, output)
    return output
```

Above is an implementation in Python incorporating a series of rules and the input string. The rewrite rules are a list of tuples in the form (a,b) that rewrite substring a into substring b. The program first iterates through the rules and checks if the substring is in the input string and then applies the rule if it is. Once it applies the rule, it calls the function again applying rules until the program finishes when no more rules can be applied.

For the purpose of making a string rewrite system with a more real world application, a string was created where the characters and rules represent building a sandwich with the string,"mbmbbbmmm". In this case, the character "m" represents meat, "b" represents bread, and "s" represents the final sandwich that can be made with one meat and one bread. The first two rules, ("mb","s"), ("bm","s"), are used to show that any combination of meat and bread, "mb", or "bm" becomes a sandwich "s". The last two rules, ("ms","sm"), ("bs","sb"), are used to sort the "s" to the left so that there aren't issues with a "s" getting in between a "m" and "b" so that the normal rules can apply.

```
print(rewrite([("mb","s"), ("bm","s"), ("ms","sm"), ("bs","sb") ],"mbmbbbmmm") )
mbmbbbmmm : mb -> s = smbbbmmm
smbbbmmm : mb -> s = ssbbmmm
ssbbmmm : bm -> s = ssbsmm
ssbsmm : bs -> sb = sssbmm
sssbmm : bm -> s = ssssm
ssssm
```

In the first and second step, the rule ("mb","s") is applied yielding the string "smbbbmmm" then "ssbbmmm". From there the ("bm","s") rule is applied yielding "ssbsmm". From there two more rules are applied yielding the final string, "ssssm". With this final string, there are no more rules that can apply. This means with the starting string of "mbmbbbmmm" the string reduction system will yield "ssssm". The string "ssssm" means that four sandwiches can be created with the initial combination of meat and bread with one meat remaining. In addition, the string "ssssm" is also known as a normal form. Now in a normal string rewrite system, the rules could apply in any order. An example of rearranging the rules below can be shown.

```
print(rewrite([("bm","s"), ("mb","s"), ("ms","sm"), ("bs","sb") ],"mbmbbbmmm") )
ssssm
```

With a different order of rules being applied, the end result of "ssssm" is still reached. In the case of this example, the order of the rules does not matter and "ssssm" will always be reached making "ssssm" known as the unique normal form. However, the final normal form being the same with different rule applications isn't always the case. There may be situations where rules being applied in different order could result in different results as seen as seen in the example below.

```
rewrite([("ab","a"), ("ba","b")],"abbaba")
aaa

rewrite([("ba","b"), ("ab","a")],"abbaba")
a
```

Based on the order of the rules these two strings are rewritten to have different values. The same starting input can have different final inputs based on the order of rules being applied with "a" and "aaa" being final input. One solution to this specific SRS would be to introduce additional rules, such as "aaa" reducing to "a". However, in other strings there may be several extra rules needed to reduce a string to the same value.

To handle this problem, the Knuth-Bendix completion algorithm is used to attempt to transform a set of equations into a finitely terminating, confluent term rewriting system and the reductions preserve identity [KBA]. In this situation, the algorithm attempts to handle strings that can be reduced in two ways. The pair of rules generated by these two reductions are called critical pairs [CP]. As these two critical pairs reduce to the same expression, all critical pairs of a term rewriting system are joinable meaning the area is locally confluent. The Knuth-Bendix algorithm works by introducing new rules until there are no more critical pairs [SKB].

Returning to the example from before, a rule could be introduced with the Knuth-Bendix algorithm that sees how these two strings can be reduced in two ways and introduces the rule "aaa" into "a" making the two strings "abbaba" reduced to "a"

```
print(rewrite([("ab","a"), ("ba","b"), ("aaa","a")],"abbaba"))
a
print(rewrite([("ba","b"), ("ab","a"), ("aaa","a")],"abbaba"))
a
```

The Knuth-Bendix algorithm also applies to algebra as when it succeeds it solves the word problem for algebra.

$$0 + \mathrm{x} = \mathrm{x}$$

$$0 + \mathrm{x} \rightarrow \mathrm{x}$$

The rules from the Knuth-Bendix algorithm can result in the algorithm terminating with a terminating confluent set of rules, loop without terminating, or terminating with a failure. The algorithm can loop forever if a rule is introduced that causes it to be stuck in a loop such as ("ab, a") and ("a", "ab") which will just cycle endlessly[KBA]. In addition, there is also the possibility of terminating with a failure as the algorithm could not find a set of rules that reduction ordering can fix.

3.9 Equivalence

Equivalence is an important aspect of abstract reduction systems. In an abstract reduction system, two elements can be shown as equivalent if they reduce to the same element. In the case of a string rewriting system, this would be equivalent if the unique normal forms of each string were equivalent. The formal definition of equivalence would be the following.

Let (A,$\rightarrow$) be an ARS and let a,b range over A

a,b are equivalent if a$\Leftrightarrow$b where $\Leftrightarrow$ is a relation that is reflexive, symmetric, and transitive.

```
print(rewrite([("mb","s"), ("bm","s"), ("ms","sm"), ("bs","sb") ],"mmmmbbbb") )
ssss

print(rewrite([("mb","s"), ("bm","s"), ("ms","sm"), ("bs","sb") ],"mbmbmbmb") )
ssss
```

It can be shown for each string there exists a unique normal form meaning that regardless of the order of each rule being applied, the same normal form is reached. This is important as it implies that.

$$\text{mmmmbbbb} \rightarrow \text{ssss}$$

$$\text{mbmbmbmb} \rightarrow \text{ssss}$$

With both values having the same unique normal form, it means they are equivalent or in more formal terms seen below.

$$\text{mmmmbbbb} \Leftrightarrow \text{mbmbmbmb}$$

3.10 Invariants

An invariant is a condition or relation that is always true.

Definition: A function P:A $\rightarrow$ B is an invariant for an ARS (A,$\rightarrow$) if

$$a \rightarrow b \Rightarrow P(a) = P(b)$$

$$\text{for all a,b} \subset A$$

A simple example of an invariant can be seen with our earlier string rewrite system where we have a system that rewrites "aa" into "a".

```
print(rewrite([("aa","a")],"aaaaaa") )
a
```

The invariant for this example is that as long as there is a string of only "a"s, it will reduce to a single a. This process can be expanded into more complicated string rewriting systems but the basic concept is the same where the system will have a relation that is always true in regards to its inputs and outputs.

One example of a more complex invariant can be seen with bubble sort. Bubble sort is a sorting algorithm that swaps adjacent elements if they are in the wrong order. The invariant of bubble sort is that after n passes of the algorithm over the array, the last n elements of the array should be in order [ICD]. If at any point this invariant were to be false, the array would not be sorted and bubble sort would fail. In this case, if the invariant were ever to be false, the program would not function correctly.

Invariants are an important part of programming as it can be seen as the assumptions a piece of code takes before it may compute anything. In other words, invariants are set of assertions that a part of the program should always hold true. By ensuring that these properties remain true, you can debug programs more easily.

4 Roman Numeral Arithmetic with String Rewriting

4.1 Introduction to the Project

In this project, string rewriting is used to incorporate arithmetic for Roman numerals including addition, subtraction, and multiplication. The file for the project can be found here [ML].

4.2 Roman Numeral Grammar

Roman numerals are a number system designed by the Romans. The values of each numeral are the following; I, V, X, L, C, D, and M, which have the integer values of 1, 5, 10, 50, 100, 500, and 1000 respectively. There are a few key rules to the grammar of Roman numerals. For adding Roman numerals, two numbers are placed side by side or for two strings, they are concatenated. If a numeral follows another numeral that is larger or equal in value it is added to the numeral before it. Lastly, there is a rule that a value cannot be repeated more than three times and gets a unique grammatical form. One example of this is seen with roman numeral four as the value "IIII" becomes "IV".

4.3 Initial Exploration In Python

The Roman numeral rewrite system was first explored with the Python string rewrite system. The first attempt was to simplify a roman numeral into a series of I's where it could be added and subtracted more easily. Concatenation was used to add the two numbers. For the first attempt, one rule was introduced, ("V","IIIII").

```
reduced = (rewrite([("V","IIIII")],"III"+"V") )
IIIV : V -> IIIII = IIIIIIII
```

Above the first expression is concatenated yielding "IIIV" which was then reduced into "IIIIIIII", 8 ones. From here, another set of rules was applied to sort the largest number to the left and normalize the number.

```
print(rewrite([("IIIII", "V"), ("IV", "VI")], reduced))
IIIIIIII : IIIII -> V = VIII
VIII
```

This yielded the correct value of 8. However, this system does not take into account numbers that are not perfect multiples of 1 or 5. To do this, each Roman numeral would need to be converted into the correct amount of "I"s before being added together and reassembled.

A better method was discovered which was to sort the numerals by their value by sorting the greatest value roman numeral to the left.

```
a = "II"
b = "VI"
print(rewrite([("IIIII", "V"), ("IV", "VI")], a+b))
IIVI : IV -> VI = IVII
IVII : IV -> VI = VIII
VIII
```

This example worked as it was able to move the greatest value, "V", to the left and yielded the correct value of 8 without the need of breaking the value into "I"s.

4.4 Haskell Implementation

With some attempts explored in Python, the next part of the project was to adapt the Python model into Haskell. To do this, various text packages in Haskell were explored that could make string rewriting easier. The Data.Text package from Haskell was chosen as it offered a variety of useful functions for string manipulation with the Text type [DT]. In particular, the most useful function was the replace function, which would take a string as an input and swap substring a to substring b. This was exactly like a replacement rule above. To make swapping from String to Text more easily understandable, new functions were made that utilized the pack and unpack functions from the Text package.

```
txtToStr a = T.unpack(a)
strToTxt a = T.pack(a)
txtConcat a b = T.concat[a,b]
```

The concat function would be useful as adding two roman numerals requires concatenating them.

With the ability to convert to and from the Text package, the first step was to make a function that applies a single rule to a word. In this case the function applies the rule "IV" to "VI" which sorts the higher value. The function takes a tuple of a rule, (l,r), and a word that replaces the substring "l" in the word with the substring "r" similar to the Python implementation. The function, applyRule, is seen below.

```
--applies a single rule to the word and returns the word
applyRule rule word = applyRule' rule word where
   applyRule' (l,r) word = (T.replace (strToTxt(l)) (strToTxt(r)) (word))


applyRule ("IV","VI") (strToTxt "IVI")
"VII"
```

This rule now sorts "V" and "I" and makes it so the higher value, "V" is on the left of any "I". However, adding roman numerals requires several replacement rules. The next step was creating a list of rules that would be applied to a word which led to the creation of the applyRuleSet function.

```
--applies a set of rules to a word then returns the word
applyRuleSet rules word = applyRuleSet' rules word where
   applyRuleSet' ([]) xs = xs
   applyRuleSet' (l:r) xs = if xs == (applyRule (l) (xs)) then applyRuleSet' r (applyRule (l)
       (xs)) else applyRuleSet' rules (applyRule (l) (xs))
```

With this function, a list of rules can be applied to a word. The function goes through each tuple in the list and calls the applyRule function applying the rule to the word. If the word is unchanged with the rule application of the rule seen in the line "if xs == (applyRule (l) (xs))" it will apply the next rule. With every application of a rule, the function starts over from the beginning of the rule set to ensure that all rules are applied. This is necessary as earlier replacement rules may now be valid now that the string has changed. Once all rules are applied, the function terminates with the line, "applyRuleSet' ([]) xs = xs" as there are no rules remaining.

The next step is to now write out the rules for string rewriting. First, there needs to a rule set that sorts values so that the greater values are always on the left of the smaller roman numerals.

```
--rules to move greater values to left and package up values.
rulesList = [("IV","VI"), ("IX","XI"), ("IL","LI"),("IC","CI"), ("ID","DI"), ... ,("IIIII","V"),
    ("VV","X"), ("XXXXX","L"), ("LL","C"), ("CCCCC","D"), ("DD","M") ]
```

The first rules list can be seen above that swaps all greater values to the left. It is abbreviated to conserve space as there were many possible combinations of lesser and greater Roman numeral values that can be swapped. The last few rules in the list are the first attempt at simplifying smaller values into greater values such as in the case of "IIIII" to "V".

```
print $ applyRuleSet rulesList (strToTxt ("IV"++"I"))
"VII"
```

With the first ruleset, came testing and with testing came bugs. If "IV and "I" were added the answer should be "V" but instead with this rule list, the answer is VII, which is 7. This is due to the fact that the rewriting system does not take into account that "IV" is equal to "IIII" and sorts the "V" to the left creating the wrong number. To fix this, another rule list was needed and the aptly named removeProblemsList was created which simplifies edge case values into numbers that can be properly added.

```
removeProblemsList = [("IV","IIII"), ("IX","VIIII"), ("XL","XXXX"), ("XC","LXXXX"), ("CD","CCCC"),
    ("CM","DCCCC")]
```

This list of rules is simple, it takes the edge case grammar values and converts them into more simplified versions that can be properly added with the string rewriting system. The value of "IV" becoming "IIII" now allows the addition of "I" and "IV" to become "IIIII", which is now the correct value of 5 but with incorrect grammar. This required the addition of one more rule list, the normalizing rule list, to normalize values into correct roman numeral grammar.

```
normalizeList = [("VIIII","IX"),("IIII","IV"), ("LXXXX","XC"), ("XXXX","XL"), ("DCCCC","CM"),
    ("CCCC","CD")]
```

The normalize list was created to apply the roman numeral grammar rules and normalize the values. The value "IIII" now becomes "IV" and other edge case values now become their proper form.

The next function created was a function that would apply each of these rule sets to a string, removing problem values, sorting, then normalizing.

```
--converts string to roman numeral by removing problem values, sorting, and then normalizing
cleanAndNormalize x = applyRuleSet normalizeList (applyRuleSet rulesList (applyRuleSet
    removeProblemsList (strToTxt x)))
```

This function first applies the removeProblemsList which simplifies edge case values. Then it applies rulesList to sort and bundle up similar values. The value is then normalized with the normalizeList.

4.5 Roman Numeral Addition

With the ability to apply several rule sets complete, addition was next to implement. To add roman numerals, concatenation was used to create a single string of the two values. However, edge case values such as "IV" required a bit more forethought than just immediate concatenation. This required a more careful application of the rules.

```
normalize x = applyRuleSet normalizeList (applyRuleSet rulesList ((strToTxt x)))

--Adds two string roman numerals
add a b = normalize (txtToStr(txtConcat ( applyRuleSet removeProblemsList (strToTxt(a)) )
    (applyRuleSet removeProblemsList (strToTxt(b)) )))
```

To allow for proper addition, removeProblemsList had to be applied to both values before concatenating them to remove problem values. Once those values were removed, the values were concatenated then normalized which completed the add function. The creation of a new function, normalize, was needed as the inputs have already had problem values removed. In this function, strToTxt and txtToStr as needed to ensure the types matched up properly. If cleanAndNormalize was used again, it resulted in a rare error where it would remove incorrect edge cases that should not be removed.

Below is a demonstration of some addition with edge case values.

```
ghci> add "I" "IV"
"V"
ghci> add "IV" "V"
"IX"
ghci> add "V" "IV"
"IX"
```

With these examples, it can be seen that addition now properly works and normalizes the final value.

4.6 Roman Numeral Subtraction

Subtraction was approached differently than addition by utilizing other aspects of the Data.Text kit. One possible solution was to convert each Roman numeral into singular ones, "I", similar to the initial exploration in Python, then subtract the difference of the values. A new list was created, convertToOnesList, that reduces all Roman numeral values to "I" and another function that returns the length of the string of "I"s which is effectively the integer value of the roman numeral.

```
convertToOnesList = [("V", "IIIII"), ("X", "VV"), ("L", "XXXXX"), ("C", "LL"), ("D","CCCCC"),
    ("M","DD")]

returnIntegerValue x = T.length(applyRuleSet convertToOnesList (applyRuleSet removeProblemsList
    (strToTxt(x)) ))
```

The return integer value works by applying the convertToOnes list and then checking the length of the text which would be the value in decimal. With these two functions, the subtraction function can be created.

```
--Applies subtraction, x-y
sub x y = if returnIntegerValue y > returnIntegerValue x then error "Error: Negative Number"
   else cleanAndNormalize (txtToStr(T.drop (returnIntegerValue y)((convertToIs x))))

ghci> sub "V" "I"
"IV"
```

The subtraction functions first checks if the second value can be properly subtracted from the first. If the value would be negative, an error is thrown for a negative value. If subtraction can proceed normally, it converts the first value into all "I"s then drops the amount of "I"s that should be subtracted. It then normalizes the final value.

4.7 Roman Numeral Multiplication

Multiplication followed a similar logic to subtraction. To multiply a value, you simply add in n times. Utilizing Data.Text's replicate function, the text is copied n times similar to multiplication. The final value is then cleaned and normalized.

```
--Applies multiplication
mult x y = cleanAndNormalize (concat(replicate (returnIntegerValue y) (x)))

ghci> mult "IV" "V"
"XX"
```

In the example above, IV is replicated 5 times and then the value is normalized to get the final multiplied value.

4.8 Converting Between Decimal and Roman Numerals

With addition, subtraction, and multiplication complete, a decimal to roman numeral converter was created.

```
convertToRomanNumeral x = txtToStr (cleanAndNormalize (concat(replicate (x) ("I"))))
```

The function works by copying the value of "I" x times and then normalizing the value into the final roman numeral value.

```
ghci> convertToRomanNumeral 3999
"MMMCMXCIX"
```

To convert a roman numeral into an integer, a similar process was used in reverse where the number was cleaned of edge case values such as "IV" by turning them into "IIII" then the entire number was turned into "I"s. The length of the final string is then the integer value.

```
returnIntegerValue x = T.length(applyRuleSet convertToOnesList (applyRuleSet removeProblemsList
    (strToTxt(x)) ))
ghci> returnIntegerValue "XV"
15
ghci> returnIntegerValue "MMMCMXCIX"
3999
```

4.9 Final Thoughts On Project

The project contained an interesting exploration of string rewriting systems to implement addition, subtraction, and multiplication of Roman numerals. It required thought into what rules were needed and the order of application of the different rules. The order of rules was discovered to be removing edge case values, sorting, and then normalizing to obtain the final value. The project also functioned as a good exploration of the different Haskell packages.

5 Conclusion

In conclusion, Haskell is a functional programming language with many diverse applications in the industry. It can be used to explore the fundamental aspects of programming theory including applications in lambda calculus, abstract reduction systems, and string rewriting systems. One example of a string rewrite system was explored in Haskell to successfully create a program for Roman numeral arithmetic.

References

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

[TC] The Trick Of The Mind - Turing Complete, I-Programmer, 2016.

[TM] Turing Machines, Stanford Encyclopedia of Philosophy, 2018.

[MS] Functional programming vs. imperative programming, Microsoft, 2015.

[HA] Haskell Features, 2021.

[LYH] Introduction To Haskell, 2021.

[IMT] Introduction to Monad Theory, 2021

[HAV] Values, Types, and Other Goodies, 2021

[HAF] Haskell Functions, 2021

[SOH] Haskell Laziness, 2013

[IOH] Basic Input/Output (Haskell), 2013

[LCC] Lambda Calculus and Computations, 2019.

[THU] Why is Turing's halting problem unsolvable?, 2013

[KBA] Knuth-Bendix Completion Algorithm, 2021

[CP] Critical Pairs, 2021

[SKB] String rewriting and Knuth-Bendix completion, 2010

[HP] Halting Problem, 2021

[FBV] Free and Bound Variables, 2021

[FPC] Fixed-point combinator, 2021

[DT] Haskell Data.Text package, 2021

[ICD] Invariants In Code Design, 2016

[HCPL] Haskell History of a Community Powered Language, 2019

[HHLC] A History of Haskell Being Lazy with Class, 2007

[HII] Haskell In Industry, 2021

[DBI] De Bruijn index, 2021

[MRLC] More Readable Lambda Calculus, 2021

[DM] Data.Maybe, 2021

[HW] Haskell Wiki, 2021

[ML] Haskell Report Michael Lewson, 2021