Programming Languages
CPSC-354 Report

Laila Hardisty
Chapman University

December 22, 2021

Abstract

Language is inherent to our world, how we take in information and communicate with those around us. This same idea applies to Computer Science through the concept of Programming Languages. They are central to defining the computability of problems, allowing us to communicate with the computer. In this paper, we will be looking at Haskell as a programming language, a couple of the theories behind programming languages, as well as exploring a project to understand Haskell further. This report serves to enhance certain aspects of programming languages and provides a brief overview of important ideas.

Contents

1 Introduction

Throughout this paper, we'll be taking a look at Programming Languages as a concept central to the study of Computer Science. This will be accomplished through an overview of Haskell, a look at theories that are essential for understanding programming languages, and a simple project completed in Haskell. In the first section of this report, we will be exploring Haskell. Specifically, how to get started using Haskell, what the key features of Haskell are, a few examples using Haskell to explain its features better, and recommended resources with which to explore Haskell further. Haskell features will include Types and Type Classes, Laziness, Recursion, Monads. These are features that are a distinctive part of Haskell but are also indicative of other similar programming languages. The second part will cover two essential programming language theories: parsing and lambda calculus. The parser section will include discussions regarding the structure of a parser, abstract syntax trees, and ambiguous and non-ambiguous grammar. For the lambda calculus section, the topics will include binding and scope, church numerals, and the idea of a language being Turing complete. Finally, the paper will end with a project written using Haskell. Using Roman numerals, the project will be about rewriting to create a calculator along with other conversion functions.

2 Haskell

The theories of Programming Languages can be clearly illustrated in Haskell as it is a functional language. Haskell is also Turing complete a theory that will defined later in this paper, that makes it quite an adept programming language.

2.1 Getting Started

The first step to begin coding in Haskell, before you even know what Haskell is, is to install Haskell. This was one of the harder parts for me as there were a lot of confusing resources and it all became a labyrinth of trying to download something and then having an error, and then repeating the process again. For this part, I would recommend Dr. Kurz tutorial which can be found in the resources section. If you have a Mac, like me it's pretty simple: `curl -sSL [https://get.haskellstack.org/](https://get.haskellstack.org/)| sh` . Once Haskell is installed you should then be able to use the interactive console of the Glasgow Haskell Compiler, this is where you'll be able to interact with, load, and run your scripts.

Important commands to know:

- `stack exec ghci` or `ghci` to open the console
- `:quit` to exit the console
- `:l scriptName` or `:load scriptName` to load the Haskell file before running it
- `:r` will reload the current script
- `:main` to run what code is contained in main

Once you have installed Haskell and are able to open the ghci console you are now ready to start learning about Haskell.

2.2 Key Features

The most important thing to know about Haskell is that it is a purely functional programming language, in all previous classes imperative programming languages have been used when coding. So what is a purely functional language, well in short it means that you don't tell the computer what to do but rather you tell it what stuff is. This means that you can't set a variable as something and then redefine it later. Also that functions have no side effects and if you give it the same input it will always produce the same output,

the latter phenomenon is known as referential transparency. A purely functional language is a language that only supports functional paradigms, while impure functional languages support functional paradigms and imperative style programming [FPI]. Functional programming was specifically designed to work with mathematical functions that use conditional expressions and recursion. It works especially well with high school algebra, lambda calculus, and problems where equational reasoning is used. Functional programming also abstracts away from memory as it requires such a large memory space due to the fact that it uses immutable data [FPI].

2.2.1 Types and Type Classes

There are a few crucial points about Haskell in regards to types. The first is that it is statically typed which means that the variable types are explicitly declared and thus are determined at compile time [TT]. Haskell has type inference as well so that the type system is able to deduce the type of variable. There is no need to label every piece of code with the type because the type system is intelligent enough to figure it out itself. All this allows Haskell to be elegant and concise this leads to shorter programs than the imperative equivalent. Haskell shares many of the same common types with other programming languages. The command `:t` can be used on any expression to tell us the type.

Haskell Supported Types

- Int: a bounded integer that is used for whole numbers. The maximum is 2147483647 and the minimum is -2147483648.
- Integer: same as Int but it can express bigger numbers, however, Int is more efficient.
- Float: for floating points with single precision.
- Double: for floating points with double the precision.
- Bool: the typical boolean type of true or false.
- Char: a single character, denoted by single quotes ' '. A list of characters makes up a string.

Data Types

Lists: Represented as `[a]` is a homogeneous data structure as they store several elements of the same type, similar to most other programming languages list except that they can't be indexed.

Common functions:

- `++` used to concatenate two list
- `:` used to prepend a list as in `5:[1,2,3,4,5]` → `[5,1,2,3,4,5]`
- `!!` to get an item out of a list. For example, `[5,8,3,3] !! 2` returns 8
- `head` takes a list and returns its head, the first item in a list
- `tail` takes a list and returns the tail, which is everything but the head
- `last` takes a list and returns the last element
- `init` takes a list and returns everything except the last element
- `length` takes a list and returns its length
- `null` checks if a list is empty; returns true if it is, returns false if its not
- `elem` takes a thing and a list of things and tells us if that thing is an element of the list for example, `4 ‘elem‘ [3,4,5,6]` → `True`

Ranges: You can take lists and use them to create a patterned set of values that can be enumerated. This can be accomplished a few different ways, `[1..10] → [1,2,3,4,5,6,7,8,9,10]` this can also be applied to char and be reversed, this can also be done to create sequences such as `[3,6..20] → [3,6,9,12,15,18]` it is not recommended however to do this with floats. Similar to creating sequences you can also use the keyword `take` to get a specified amount of a range as in `take 24 [13,26..]` to get the first 24 multiples of 13. The keyword can also be used in combination with `cycle` in order to create a repetitive list since `cycle` takes a list and cycles it into an infinite list `take 12 (cycle "hi ") → "hi hi hi hi "`

Type Classes

Basically, type classes are an interface that defines some sort of behavior. They allow you to create your own interface as well that provides a common feature set over a wide variety of types. The `:t` keyword can again be used in this case, if you use it on a function instead of a type it shows you the behavior that the typeclass describes. This can be shown using the equality operator, `:t ==` will give you `(==) :: (Eq a)=> a -> a -> Bool`. This introduces us to `=>` symbol which is a class constraint, it tells you what the constraints are type-wise for the function. It is basically telling you in this case that the equality operator can take two values that are of the same type and the type is a member of the Eq typeclass that the equality operator is a part of. In this case, the class constraint is E, this can also be called context since it is placed at the front of the expression and lets you what the rest of the expression is in reference to [TCO]. An important typeclass to know for the Programming Languages course is the **Num** typeclass which is the numeric typeclass containing such types as Int and Integer meaning that they act like numbers. For an operator like `*` the typeclass tells you that, `(*) :: (Num a)=> a -> a -> a` it takes two values of type Num and returns type Num, this can come in handy when you need to define a function and you want take in one type and return another. I used this for a function where I wanted to take in an Integer and then return a list of Integers, so I had to define it as `hailstone :: Integer -> [Integer]`. While most of the numerical operators are the same in Haskell there is one crucial one that is different, that would be divide. Instead of using the symbol `/` like most programming languages, Haskell uses `` `div` ``. It is important to note that when I'm using `` ` `` this symbol is not an apostrophe, as in `'`, it is actually an accent mark.

2.2.2 Laziness

Since Haskell is a functional programming language this means that it only evaluates computations when they're needed which makes the program more efficient [WFP]. Programs will reuse results from previous computations and save on run time [WFP]. Haskell is able to accomplish this by something known as lazy graph reduction. This does not mean that there aren't functional programming languages that don't evaluate lazily, for example OCaml. It is able to evaluate only when needed by using leftmost outermost reduction which is the same thing as normal order reduction [CSE]. This system along with pattern-matching, call-by-need, and thunks. A thunk is a function that returned from another function, or in more complex terms a subroutine used to inject an additional calculation into another subroutine [WT]. A thunk helps to delay calculations until they're actually needed, which is why it is so helpful in lazy programming. Pattern-matching and thunks are also able to help out with evaluating just enough. Call-by-need also helps out with evaluating at most once, which requires the most techniques [LEI]. Evaluating at most once uses: substitute pointers, updating redex root with result, and a self-updating model [LEI]. While all the techniques are important you really only need to know the three main points when it comes to how Haskell is lazy. To sum up lazy programming:

- it evaluates only when needed
- it evaluates just enough
- it evaluates at most once

2.2.3 Recursion

Functional programming tends to avoid using if-else statements and loops in general [WFP]. This makes sense based on our previous knowledge of functional programming as these constructors can easily create different outputs with the same input and that would not be up to functional programming standards. Haskell can use if-then-else statements sparingly, the do notation, or case statements to achieve the same effect as an if-else statement. Since loops and if-else statements aren't as common, functional programs often use recursion in place of these constructors for iterative tasks. Haskell, in particular, is really meant for recursive programming, since once you understand the components it is pretty simple to get. As with most instances of recursion, it is easiest to start thinking of recursion in terms of the base case and what it would take to get there. It aligns with how you would apply recursion in other programming languages except that for Haskell recursion becomes one of its most important features. This goes along with the fact that Haskell works especially with mathematical ideas as definitions in mathematics are often defined recursively. It is helpful in some cases when working with recursion to define edge conditions, these are elements that are not defined recursively but help to terminate the function [R]. Often edge conditions are scenarios where applying the recursive function doesn't make sense. Haskell also uses techniques like pattern matching which makes recursive programming much easier, edge conditions, and base cases are used to make patterns clearer for the recursive function to follow [R]. Recursive functions are imperative when it comes to programming in Haskell which is why Haskell was made to, as simply as possible when it comes to recursion, implement them.

2.2.4 Monads

Basics

Without getting into the technical details, monads are strategies for solving coding problems that recur often, regardless of what you're writing [AAM]. Monad strategies allow you to combine computations into more complex computations, it helps to organize the computations as values and sequences of computations as typed values [AAM]. A monad allows you to not have to code a combination each time it is needed. Once you understand monads, which are the hardest part of Haskell so far, you'll have access to features that will make your programming easier. There are three features that monads possess that make them useful when it comes to functional programs.

- Modularity: this is what it means by creating more complex computations from other computations while keeping the combination strategy separate.
- Flexibility: when using monads for functional programming, the program becomes more adaptable. All of a monads strategy can be found in a singular place allowing for such flexibility.
- Isolation: monads allow imperative-style computations while using a purely functional language. They can deal with side-effects that Haskell doesn't allow outside of the main part of the program.

Operators

The monad class defines two basic operators, `>>=` known as a bind and `return` [AM].

```
infixl 1 >>, >>=

class Monad m where

    (>>=)           :: m a -> (a -> m b) -> m b

    (>>)           :: m a -> m b -> m b

    return          :: a -> m a

    fail            :: String -> m a

    m >> k          =  m >>= \_ -> k
```

- `>>=` or `>>` is used to combine two monadic values, it takes a monadic value and a function takes a normal value and returns a monadic value. `<<` is used when the function does not need the value produced by the first monadic operator.
- `return` is used to inject a value into the monad. It takes something and surrounds it with a monad.
- `fail` never necessary to use it explicitly in code. Used by Haskell to enable failure for special monads.

The precise meaning of such operators as the bind depends upon the monad. These operators help when trying to apply values that come with certain contexts, how we have to account for their behavior and feed them into other functions. An important keyword to know when it comes to monads is `Nothing` which represents the absence of a value or that the computation has failed [FM].

do notation

Syntax like `do` helps to simplify chains of monadic operations, by bringing together monadic values in sequence [AM]. It can help when there is a failure context, is able to check values before the bind operator worries about their context. When creating a `do` expression, every line is a monadic value whose result can be inspected using `<-`. The formatting of a `do` expression is partly what makes monads more similar to imperative programming. The expression is sequential though because each line relies on the value that was found in the previous line [FM]. Pattern matching can be used with the `do` notation when binding monadic values to names. While pattern matching can fail in some cases often the next pattern will be matched, if every pattern fails then the `fail` function is called. The fail function only allows the current monad to fail so that the whole program doesn't crash. Some monads that incorporate the concept of failure into their programming so that we get a result that isn't a crashed program.

Laws

Just because something is an instance of the monad typeclass does not mean it is a monad, it has to abide by certain laws in order to be considered a monad. These laws are what allow us to make reasonable assumptions about all monads. The first two laws describe how `return` should function, while the third law describes how a chain of monadic functions should act.

Left Identity: States that if we take a value and it is put it in default context with `return` then fed to a function using bind, that's the same thing as taking a value and applying the function to it [FM].

Right Identity: States that if we have a monadic value and we use bind to feed it to `return`, then we have the original monadic value [FM].

Associativity: States that when we have a chain of monadic values, it shouldn't matter how they're nested [FM]. As the monadic values are bounded together throughout the function so it doesn't how these values line up, it just matters what they mean.

Types of Monads

The three most common monads are:

- Maybe: a monad for building computations that may fail to return a value.
- List: the Haskell list type constructor is a monad. Allows us to build computations that can return none, one, or more values [AAM].
- I/O: Haskells input/output system functions as a monad that allows it to be separated from the purely functional part of the language. Common actions include the ability to read and set global variables, write files, and read input.

2.3 Examples

Simple Arithmetic

Simple arithmetic can be done within the Glasgow Haskell Compiler:

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

Boolean Algebra

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Testing for Equality

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

Prefix vs Infix Functions

- Prefix Function: the function is applied at the beginning of the expression. Most functions in Haskell are in this form. They are called by writing the function name, a space and then the parameters, separated by spaces. For example, `min 9 10 → 9`
- Infix Function: the function is being applied in the middle. More commonly found in imperative languages. Such functions like, `3 * 8 → 24` or `24 ‘div‘ 8 → 3`.

To compare the two, `div 92 10` is the same as `92 ’div’ 10`

Type Classes

```
ghci> :t 20
20 :: (Num t) => t
ghci> :t head
head :: [a] -> a
ghci> :t read
read :: (Read a) => String -> a

ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Maybe Monad

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++"!") Nothing
Nothing
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

A Simple Function

To be implemented in a firstScript.hs, (.hs is the Haskell file type): `doubleMe x = x + x`. This can then be called while in the Glasgow Haskell Compiler

```
ghci> :l firstScript
[1 of 1] Compiling Main          ( firstScript.hs, interpreted )
Ok, one module loaded.
*Main> doubleMe 9
18
*Main> doubleMe 8.3
16.6
```

All the examples in this section were either adaptations, compiled from Learn You a Haskell for Great Good! or [PL] and my own knowledge, or purely from my knowledge.

2.4 Resources

For more information about any particular section feel free to look at the references.

- For installing Haskell: Installing Haskell
- My favorite Haskell tutorial: Learn You a Haskell for Great Good!

3 Programming Languages Theory

This section will investigate two theories that are central to the study of Programming Languages, the first of which is also a concept concerning Languages at large. Parsing and Lambda Calculus, these two notions both help to give a basic understanding of what it means to be a Programming Language.

3.1 Parsing

Parsing is the method of transforming a linear sequence of characters into an abstract syntax tree [P]. Another more exact way of looking at it is that parsing is a way of interpreting the input stream as terms in the language at hand [AS]. Or in more simple terms it's about knowing where to put parentheses so that the parser will break down the input in an intended way. A parser is able to piece together what the input

sequence means by breaking it down into an abstract syntax tree/parse tree. This is done by identifying the structure based upon the predefined grammar and extracting the data. It takes the concrete syntax and then parses it to create the abstract syntax which can later be used to create a value by an interpreter [PL]. This interpretation of the abstract syntax is done using recursion upon the abstract syntax tree. Parsing is necessary for programming languages as there are different forms of how the data can be understood by the software, and ways the computer might need to use them [GP].

3.1.1 Introduction

Humans, ourselves, parse every day using ingrained grammar gained from living. For example, if you were to look at a picture of a person's face you could come up with a few guesses about what type of emotion they're feeling based on their facial expression. While you might not always be right, you are able to take the prior knowledge you have about emotions and compare that to the photo in front of you in order to arrive at a conclusion. This same phenomenon is also present in math expressions, when evaluating $9-4*2$ you know due to previous teachings that you should use the grammar PEMDAS when deciding in what order you should evaluate the expression. This would lead to you completing the multiplication $4*2$ term giving you a value of 8 before evaluating the subtraction term $9-8$ giving you a value of -1. If you did not have this grammar with which to base your evaluation upon, you would have two options when it came to how you might approach this expression. While the former evaluation is the current agreed-upon method, an alternative process would be to evaluate the subtraction $9-4$ term first resulting in an multiplication expression of $5*2$ concluding in a value of 10. This is why parentheses are such a powerful symbol as they are easily able to denote which term in the expression should be evaluated first without relying on an extensive predetermined grammar or to go agains. In order to evaluate using the first method it could be expressed as $9-(4*2)$ or if it was intended to be evaluated in the manner of the second $(9-4)*2$. This is why grammar for how to interpret input sequences is extremely useful when parsing. Abstract syntax trees are a way of visualizing the breakdown of expressions. However, the usefulness and specificity of the grammar is dependent upon whether it is ambiguous or non-ambiguous.

Structure of a Parser

A parser is comprised of a lexer, also known as a lexical analyzer or a tokenizer, and the proper parser. The overall function of the lexer is to take the input sequence and break it down into tokens, this is what the proper parser will work with [AS]. The proper parser will scan the provided tokens to create an abstract syntax tree that will then be fed to the interpreter [PL]. Some parsers do not break down this process into two steps, instead, they combine the lexer and proper parser into one called a scannerless parser [GP]. If the parser was given an input of $2+2$ the lexer would then find the tokens which in this case would be NUM 2, PLUS $+$, and NUM 2, with the titles differing depending on what the grammar defined it as. These are the tokens the proper parser would receive and then create an abstract syntax tree out of.

3.1.2 Abstract Syntax Trees

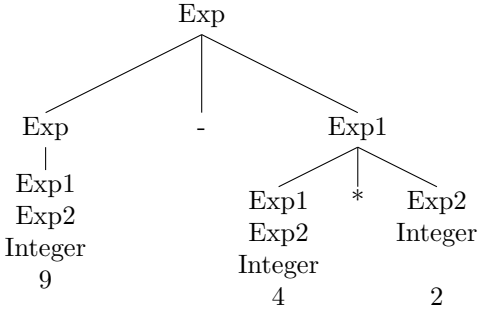
Above is an example of an abstract syntax tree (AST), a more refined version of a parse tree, or a concrete syntax tree, referring to the output generated by the parser. While they are both of the data structure type known as a tree and have root nodes, as well as child nodes and subtrees that represent the tokenized input they do represent different levels of abstraction. The parse tree is generated first, it contains all the tokens given by the lexer as well as the rules given by the grammar [GP]. A parse tree is a more direct representation of the concrete syntax including all the intermediary parts [PL]. The abstract syntax tree

then condenses this information into only the most important parts, leaving only the necessary information for the interpreter. The differences between the two and how the tree works are best represented visually. The grammar for the following trees would be:

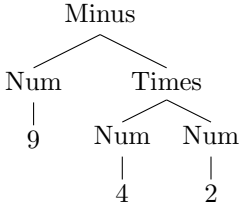
```
Exp -> Exp '-' Exp1
Exp -> Exp1
Exp1 -> Exp1 '*' Exp2
Exp1 -> Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'

coercions Exp 2
```

Let's look at the expression $9 - (4 * 2)$ as a parse tree, and as an abstract syntax tree.

Parse Tree

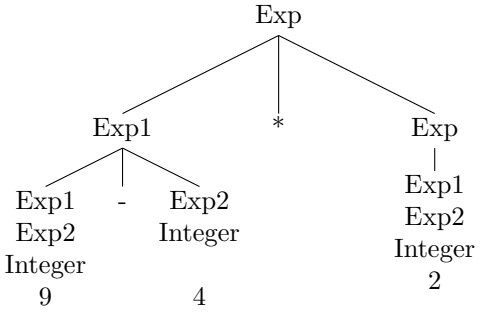


AST

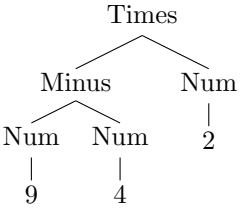


Compared to the expression $(9 - 4) * 2$ as a parse tree, and as an abstract syntax tree.

Parse Tree



AST



While these trees do not explicitly show the parentheses they do affect how the parse tree and therefore the abstract syntax tree is interpreted. These trees when going through the interpreter are computed bottom-up [PL]. It is the grammar that allows the rules to be analyzed as an arithmetic expression. In this instance, it is a context-free grammar that is used to generate the above trees, an important and commonly used grammar for parsers. This type of grammar is also known as left-recursive grammar since the expressions are defined in inverse order of precedence resulting in leftmost derivation trees. Looking more closely at context-free grammar it can be divided into ambiguous and non-ambiguous grammar.

3.1.3 Ambiguous Grammar

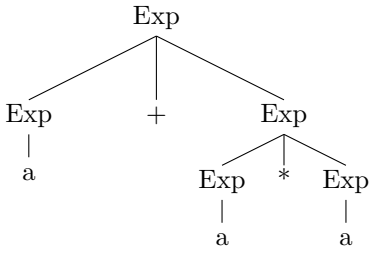
When using context-free grammar (CFG) to generate trees, it is known as ambiguous if it generates more than one AST or parse tree for the same input string [DAU]. It is clear to see why a grammar that generates more than one tree for an input sequence would contain some ambiguity. Ambiguous grammar is easily

created when there aren't levels attached to the rules, or in other words, there is only 1 coercion [PL]. This might lead to the parser getting stuck if it is unsure of how to generate the expression although the parse tree generated would shorter and faster than the one found using non-ambiguous grammar [DAU]. This type of grammar also generates different trees depending on if the leftmost or rightmost derivation is used, and may even have multiple for each [AUG]. An example of ambiguous grammar would be:

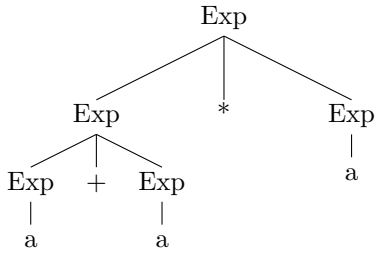
```
Exp ::= Exp "+" Exp ;
Exp ::= Exp "*" Exp ;
Exp ::= a ;
```

If this grammar were to be used on $a + a * a$, there would be two different resulting parse trees.

Parse Tree 1



Parse Tree 2

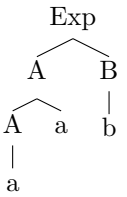


Another example of an ambiguous grammar is:

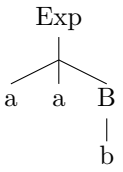
```
Exp ::= AB | aaB;
A ::= a | Aa ;
B ::= b ;
coercions Exp 2 ;
```

Using this grammar on the string aab, two different parse trees can be derived.

Parse Tree 1



Parse Tree 2



An easy way to identify ambiguous grammar is either by looking at the rules and seeing that they're all defined using the same starting point, or level. Another way is by noticing that there are multiple options for how a level can be defined or interpreted.

3.1.4 Non-Ambiguous Grammar

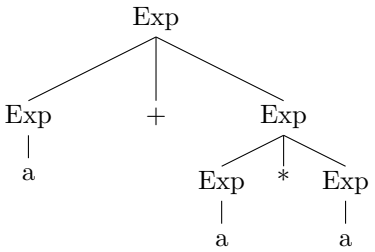
Non-ambiguous or unambiguous grammar is when only one parse can be derived from the context-free grammar [DAU]. Meaning that there are no other possible ways to interpret the input sequence using the CFG, that only one derivation is reached no matter if the leftmost or rightmost derivation is used. Each of these unique input sequences is able to lead to the creation of a unique parse tree. Non-ambiguous grammar creates a larger parse tree and takes a longer time to generate than ambiguous grammar would [AUG]. Often

non-ambiguous grammar is the desired result when creating CFG since the program will then give you the same intended derivation each time. Using the first ambiguous grammar example from earlier but rewritten to be non-ambiguous we get:

```
Exp ::= Exp "+" Exp1 ;
Exp ::= Exp1 ;
Exp1 ::= Exp1 "*" Exp2 ;
Exp1 ::= Exp2 ;
Exp2 ::= a ;
coercions Exp 2 ;
```

This would result in a singular parse tree when parsing $a + a * a$.

Parse Tree

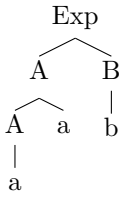


Taking the second example of ambiguous grammar from above and translating it into non-ambiguous CFG gives us:

```
Exp ::= AB;
A ::= Aa | a ;
B ::= b ;
coercions Exp 2 ;
```

Applying this non-ambiguous CFG to aab results in the derivation of a singular tree from the parse trees generated by the ambiguous grammar.

Parse Tree



It is fairly easy to take ambiguous grammar and translate it into non-ambiguous grammar, as long as there is a clear idea of how the input sequence should be evaluated. This translation can be accomplished by defining an order of precedence for how the sequence should generated into a parse tree. This can also be accomplished by removing some of the ambiguity through the removal of the multiple options that a certain sequence can be tokenized into.

All the examples in this section were either adaptations, compiled from multiple sources, [AUG], [DAU], and [PL], and my own knowledge, or purely from my knowledge.

3.2 Lambda Calculus

Lambda calculus is known as one of the most powerful programming languages as it has extensive functionality. It is a system of mathematical logic developed by Alonzo Church in the 1930s as a foundation for functional programming [LC]. It is one of the simplest languages, being able to denote functions and applications; another defining feature would be that it is Turing complete and can also work with Church numerals.

3.2.1 Introduction

There are three basic components of lambda calculus:
in these definitions, e is an expression

- variables: $e ::= x$
- function application: $e_1 e_2$
- function abstraction: $\lambda x.\ e$

For application, this would be applying the function e_1 to the argument e_2 as an entire program. For abstraction, x would be a variable, so in the case of this function, x is a parameter, yet in this function, e does not depend on x; therefore, it is abstracted [PL]. While lambda calculus has no built-in functions, it is with these constructs that lambda calculus can define any function. While each lambda can only take one argument and one parameter, it can handle multiple arguments by having nested lambdas or arguments as in, $(\lambda x.\ (\lambda y.\ x + y)\ 2)\ 3$ [PL]. This would then become $2 + 3$ using substitution because it is also being simplified while evaluating it. Writing a function in this way, with the nested arguments, would be an example of currying.

3.2.2 Binding and Scope

Binding

In lambda calculus, every variable can be defined as either free or bound in a lambda expression.

- bound variable: a variable that is tied to a λ

  $\lambda x.$

  x is a bound variable
- free variable: a variable that is not tied to a λ

  $\lambda x.\ y$

  y is a free variable

Often the lambda part of the expression, $\lambda x.$, will be called a binder since it is the λ that determines what a variable will be characterized as. A variable can be both bound and free within the same function, as in the case of $(\lambda x.\ xy)(\lambda y.\ y)$, where y is both bound and free as the two functions do not interact in that way. An expression with no free variables, like $(\lambda y.\ y)$ can also be defined as a closed expression; however, any expression with more than just bound variables would be called an open expression, as in $(\lambda x.\ xy)$ [LC].

Scope

Scope functions in lambda calculus expressions the same way it would in other programming languages. In particular lambda calculus uses lexical scoping. Lexical scoping is when the program's text defines the scope of the variable, this means the text simply has to be inspected to determine the scope of a variable [TLC]. This expression, $(\lambda x.\ xy)(\lambda y.\ y)$, can also show us how scope affects an expression. As y is not within the scope of $\lambda x.$ in the first function, so it becomes free. The x is bound to $\lambda x.$ meaning it is in the scope of x. The y represents something different in the first function from the y in the second function. In order to not be confused when dealing with binding and scope, it can be helpful to name variables differently depending on if they have local or global scope.

3.2.3 Examples

An example from [PL].

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

The y outside the parentheses is a free variable. While the y's inside are bound variables.

Because the y's are bounded differently, this means they cannot be evaluated as if they were the same variable. Shown below is an incorrect way of evaluating the expression.

$$(\lambda y.y + y)2$$

$$2 + 2$$

This expression, when evaluated correctly, should result in $y + 2$. This is because the one y is a free variable, so it cannot be substituted the way the y's attached to the binder are. This result leaves the free variable as a free variable. A free variable can not become a bound variable.

By renaming the variables in the function, it can become clearer which variables are free and which are bound. Once the characterization of the variables and their scope is clear, it can become much more apparent how this function should be evaluated.

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

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

$$y + 2$$

Another example of a call-by-value expression, in this case the `y` expression is the value getting substituted into the `x` function.

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

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

Lambda calculus expression evaluated using call-by-name, here the `y` function is substituted into the `x` function.

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

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

The second call-by-value example is also an example of a non-terminating expression, it doesn't matter if we evaluate it using call-by-name or call-by-value, we'll continue to get the same result $(\lambda y.y)(\lambda y.y)$.

Sometimes though, call-by-name can terminate an expression that would otherwise not terminate if it was only evaluated using call-by-value. This happens in the case of the following expression:

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

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

If this were to be evaluated using call-by-name it would result in:

$$\lambda y.y$$

The previous examples were taken from slides provided by [CCL].

3.2.4 Church Numerals

Introduction

Church numerals are another invention of Churches' used to represent natural numbers as functions. They allow lambda calculus to interact with natural numbers without actually working with them directly [CN]. This representation is accomplished by using lambda calculus to denote zero and any successor number. All Church numerals are represented using two parameters `s`, is the successor function and `o`, is a value representing zero [CR].

$$\lambda s.\lambda o.something$$

Using this argument zero will be represented as $\lambda s.\lambda o.o$, once we have this as a base we are free to use the successor function to represent any other natural number. The natural number n can be represented as the church numeral n.

$$1 : \lambda s.\lambda o.so$$
$$2 : \lambda s.\lambda o.s(so)$$
$$3 : \lambda s.\lambda o.s(s(so))$$
$$\underline{n} : \lambda s.\lambda o.s^n o$$

To discover the natural number that the church numeral denotes, all you have to do is count the number of times the function is applied. There is an actual successor function that can be applied to n in order to find its successor; S(n) this function would result in the same thing as n + 1 [CR].

$$S = \lambda n.\lambda s.\lambda o.(s(nso))$$

Using Church numerals, simple arithmetic functions can also easily be defined in lambda calculus. They are able to perform these types of expressions as everything is relative to the Church numeral of zero [CR]. Using the successor function, we can get addition, then from there, multiplication. Using multiplication, we can obtain a function for exponentiation.

- Add: $\lambda m.\lambda n.\lambda s.\lambda o.ms(nso)$
  using the successor function, this becomes
  $\lambda mn.(mSn)$
  from here we get the function:
  add m n = m + n [CR]

- Mult: $\lambda m.\lambda n.\lambda s.\lambda o.m(sn)o$
  utilizing the add function,
  $\lambda mn.m(addn)\underline{0}$
  our final function will be:
  mult m n = m * n [CR]

- Exp: $\lambda m.\lambda n.\lambda s.\lambda o.mnso$
  with the multiplication function, exponentiation can be simplified to
  $\lambda mn.m(multn)\underline{1}$
  this makes a function of:
  exp m n = $\underline{m^n}$ [CR]

Church numerals can also be used in functions for predecessors; from there, arithmetic functions such as subtraction and division are able to be created. While Church numerals only expressly denote natural numbers, Church numerals can also be used for more complex numbers like integers and floating-point numbers [TLC].

Church Numerals and Haskell

Church numerals can be implemented in Haskell by making use of Haskell typecasting. The basic type of a Church numeral in Haskell is `type Church a = (a -> a)-> (a -> a)` [CE]. This typecasting is doing the same thing as the lambda calculus denotation of the above Church numerals. One parenthesis of `a` is used to represent the first function for successors, while the second parentheses of `a` is used to typecast the zero value. To actually obtain a Church numeral from an integer, a church function will need to be implemented. The church function is simply applying the lambda calculus to the church type obtained from the integer. This means that the church function only needs to be applied two ways, as it will use recursion. Firstly by defining a base case of `0`, and then for any natural number `n`. The `λ` is represented by `\` in Haskell.

```
type Church a = (a -> a) -> (a -> a)

numToChurch :: Integer -> Church

numToChurch 0 = \s -> \o -> o

numToChurch n = \s -> \o -> s(churchNum (n - 1) s o)
```

To take a Church numeral back to integer another function must be created. Once again we can rely upon lambda calculus to explain what is going on in the function, this would be a variation of the successor function that relies upon recursion and a base case of 0: $\lambda x.x(\lambda y.y + 1)(0)$.

```
churchToNum :: Church -> Integer

churchToNum cn = \x -> x (\y -> y +1) (0)
```

These above examples are adapted from [CE] and [CR].

3.2.5 Turing Complete

To call a programming language Turing complete means that it can solve any computational problem given enough time and memory no matter how complex [TC]. A machine that was Turing complete became known as a Turing machine; if the machine was programmed using the right instructions, it could solve any problem. Lambda calculus is known to be Turing complete since it is a powerful model of computation and can be used to implement a Turing machine. Turing himself proved that Church's and his models, lambda calculus and Turing machines, were equivalent, remarking upon the universality of the two [OTM]. As they are equivalent, this means that this idea also works in reverse, that a Turing machine can simulate lambda calculus. In particular, lambda calculus can simulate a Universal Turing Machine, allowing the Universal Turing Machine to evaluate problems using lambda calculus [LC]. In Church's Thesis, he asserts that lambda calculus can represent any computable function, reinforcing that it is a Turing complete language [FFP]. This assertion also implies that lambda calculus is just as powerful as recursive functions since recursive

functions are computable and therefore saying that all lambda calculus definable functions are recursive [FFP].

The Halting Problem

Lambda calculus faces the Halting problem as it is normalizing but not terminating. The Halting problem is, in simple terms, the problem of determining if, considering a program and an input, the program will terminate or continue to run forever [HP]. As we saw earlier in this section in certain cases lambda calculus expression do not terminate, $((\lambda x.(xx))(\lambda x.(xx)))$, they will continue to reach the same result each time the expression is computed. Though it doesn't necessarily have to reach a result to be considered terminating, it could get some error and terminate and accordingly be classified as terminating [LC]. Even if an expression doesn't terminate, it can still normalize by reaching a normal form. For lambda calculus, a normal form is an equivalent form in which no more reductions or simplifications can be made [LC]. A normal form is $\lambda x.x$ as it can't be reduced anymore yet, $((\lambda x.(xx))(\lambda x.(xx)))$ is not since it still possible to reduce it [LC]. Because of all this, it has been decided that the Halting problem with regards to Turing machines, and lambda calculus as well, is undecidable [LC]. As Turing proved, there is no program that can definitively prove the Halting problem making it unsolvable and undecidable [HP]. The Halting problem can be solved in lambda calculus with the addition of types; this ensures that a lambda calculus problem will terminate [LC]. This is why it is important to typecast lambda calculus within Haskell so that it is terminating.

4 Project

4.1 Introduction

This section shows a project that consists of rewriting Roman numerals, creating arithmetic using Roman numerals, and converting between roman numerals and integers. This entire project was created by following a project conceived of by [RNC] and then building upon it using the understanding gained throughout it. This project emphasizes such ideas as the grammar or the order of rewriting, as it is the basis of the project. It involves three different functions: `order`, `reduce`, and `extend`. The second part would be arithmetic, where it is necessary to understand how Roman numerals interact with each other and concepts of basic arithmetic. This involves four functions: `addRN`, `subRN`, `minusRN`, and `multRN`. The third section concerns conversions; how to switch between roman numerals and integers. This section enforces concepts of recursion as well as the idea of how to use modules. There is also a fourth part which is simply normalization, taking the roman numeral and making it syntactically correct.

4.2 The Project

To run this project simply open terminal and navigate to the folder within which the romanNumerals.hs file is contained. Then use the `stack exec ghci` to open the interactive console. From there load it by `:l romanNumerals.hs`, then `:set -XOverloadedStrings`, and finally `:main` to see some of the results of the test code executed. Also in the interactive console specific functions can be run.

4.2.1 Section 1

Order

The first function `order` is simply taking a string of Roman numerals and rewriting the string so that the string is ordered from the left to the right with the largest numeral as the leftmost character and the smallest numeral as the rightmost character. The idea behind this is very similar to how grammar is

defined or the rules behind an abstract syntax tree. In this case, we know that most roman numerals are written with the largest numeral as the leftmost character, so those are the implemented rules. Since this was to be achieved, that meant that the function needed to be ordered in such a way that this is actually what happens when the string is rewritten. That means that the first line of code that the string should encounter would be the character that we want to end up as the rightmost part of the string, the would mean the code begins with rewriting I and ends with M as it is the biggest we are going with this example. Of course, there needs to be a catch-all case which is the `otherwise` where the string is just set to empty.

```
order :: T.Text -> T.Text
order s
    | let ind = M.fromJust (T.findIndex ('I' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'I') s = T.snoc (order (T.append bef (T.tail aft))) 'I'
    | let ind = M.fromJust (T.findIndex ('V' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'V') s = T.snoc (order (T.append bef (T.tail aft))) 'V'
    | let ind = M.fromJust (T.findIndex ('X' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'X') s = T.snoc (order (T.append bef (T.tail aft))) 'X'
    | let ind = M.fromJust (T.findIndex ('L' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'L') s = T.snoc (order (T.append bef (T.tail aft))) 'L'
    | let ind = M.fromJust (T.findIndex ('C' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'C') s = T.snoc (order (T.append bef (T.tail aft))) 'C'
    | let ind = M.fromJust (T.findIndex ('D' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'D') s = T.snoc (order (T.append bef (T.tail aft))) 'D'
    | let ind = M.fromJust (T.findIndex ('M' ==) s), let (bef, aft) = T.splitAt ind s, T.isInfixOf
        (T.singleton 'M') s = T.snoc (order (T.append bef (T.tail aft))) 'M'
    | otherwise = T.empty
```

Each line looks for the index of the first occurrence of the single Roman numeral in the input string using the Maybe monad function `fromJust`; this function works by extracting an element out of a Just, which in this case is the function `findIndex`. Next, the string is split at the index point, with the `before` becoming the string before the index of occurrence. The after is the rest of the string, including the index, this part of the string is then checked to find if there are other occurrences of the character in the string. If so, the two separate strings are called recursively after being concatenated without the specific character that was split upon. This continues until there are no more occurrences, at which point all of the previously encountered characters will be appended at the end. While this idea of rewriting works, for the most part, Roman numerals are not that simple, and there are exceptions to this standard grammar.

Reduce

Can create exception handling using a function called `reduce`, this function will come after the order function because if we think about it in terms of rules, this rewriting will be applied to whatever result was created. Once again, the order of the lines of code is crucial since the function will be called recursively, and in some instances, the `order` function will be called. It is not enough, though, to format this function exactly like the `order` function where all you need to do is list the rules and include an otherwise. This is because not every string needs to go through this function, and in certain cases, there will be characters within the string that can't be reduced. To account for this, there is the line `not (T.null s)= T.snoc (reduce (T.init s))(T.last s)`; this test to see if the string is empty, if not then we save the character that prevented our string from fitting into any of the other rules and call refine on the rest of the string until the rest of the string is organized and then we add that character back onto the end. It is also important that when dealing with suffixes of similar types, such as ***IIII*** and ***IIIII*** to have the longer one be listed earlier, because if the shorter one is listed earlier than, the longer version will match up with that shorter one even tho that's not what it actually represents. Another thing to keep in mind is that `order` shouldn't be called on functions that go against the traditional order as in nine, ***IX***. Part of discovering the order is thinking through how the numerals interact but, part of it is trial and error, seeing

what the rewrite gives you once it gets to a point where it doesn't follow the `order` rules.

```
reduce :: T.Text -> T.Text
reduce s
    | T.isSuffixOf (T.pack "VV") s = reduce (order (T.append (T.dropEnd 2 s)(T.pack "X") ))
    | T.isSuffixOf (T.pack "IIIII") s = reduce (order (T.append (T.dropEnd 5 s) (T.pack "V")))
    | T.isSuffixOf (T.pack "VIIII") s = reduce (T.append (T.dropEnd 5 s) (T.pack "IX"))
    | T.isSuffixOf (T.pack "IIII") s = reduce (T.append (T.dropEnd 4 s) (T.pack "IV"))
    | T.isSuffixOf (T.pack "VIX") s = reduce (T.append (T.dropEnd 3 s) (T.pack "XIV"))
    | T.isSuffixOf (T.pack "LL") s = reduce (order (T.append (T.dropEnd 2 s) (T.pack "C")))
    | T.isSuffixOf (T.pack "XXXXX") s = reduce (order (T.append (T.dropEnd 5 s) (T.pack "L")))
    | T.isSuffixOf (T.pack "LXXXX") s = reduce (T.append (T.dropEnd 5 s) (T.pack "XC"))
    | T.isSuffixOf (T.pack "XXXX") s = reduce (T.append (T.dropEnd 4 s) (T.pack "XL"))
    | T.isSuffixOf (T.pack "LXC") s = reduce (T.append (T.dropEnd 3 s) (T.pack "CLX"))
    | T.isSuffixOf (T.pack "DD") s = reduce (order (T.append (T.dropEnd 2 s) (T.pack "M")))
    | T.isSuffixOf (T.pack "CCCCC") s = reduce (order (T.append (T.dropEnd 5 s) (T.pack "D")))
    | T.isSuffixOf (T.pack "DCCCC") s = reduce (T.append (T.dropEnd 5 s) (T.pack "CM"))
    | T.isSuffixOf (T.pack "CCCC") s = reduce (T.append (T.dropEnd 4 s) (T.pack "CD"))
    | T.isSuffixOf (T.pack "DCM") s = reduce (T.append (T.dropEnd 3 s) (T.pack "MCD"))
    | not (T.null s) = T.snoc (reduce (T.init s)) (T.last s)
    | otherwise = T.empty
```

Each line looks at the end of the string `s`, to find the numerals that can be rewritten. T.pack is frequently used throughout this function and others as the type String needs to be type T.Text. If the specific suffix is found, then `reduce` is called recursively on the rewritten string. The rewritten string consists of the removed suffix and the appended rewritten Roman numeral.

Extend

While `extend` is included in this section as it is part of rewriting a string of Roman numerals, it is actually used in the arithmetic section. A necessary addition, as Roman numerals use subtraction to represent certain numbers which would lead to inaccurate arithmetic. This function is laid out, in the same way, as the `reduce` function but does the exact opposite of it. So instead of having a non-empty string case that reduces the front of the string and saves the back to be appended, it extends the back of the string and saves the frontmost character that cannot be extended using the rules to prepend to the rewritten string. Conceptually, it looks at the beginning of the string, searching for Roman numeral combinations that can be expanded. It doesn't have to worry about the order that the Roman numerals are in, as that is not a concern when adding, so it does not include cases of pure reordering like in `reduce`. It does follow a very set order. You have the largest Roman numerals at the top of the cases since those will need to be expanded first before encountering the extending of smaller Roman Numerals.

```
extend :: T.Text -> T.Text
extend s
    | T.isPrefixOf (T.pack "CM") s = T.append (T.pack "DCCCC") (extend (T.drop 2 s))
    | T.isPrefixOf (T.pack "CD") s = T.append (T.pack "CCCC") (extend (T.drop 2 s))
    | T.isPrefixOf (T.pack "XC") s = T.append (T.pack "LXXXX") (extend (T.drop 2 s))
    | T.isPrefixOf (T.pack "XL") s = T.append (T.pack "XXXX") (extend (T.drop 2 s))
    | T.isPrefixOf (T.pack "IX") s = T.append (T.pack "VIIII") (extend (T.drop 2 s))
    | T.isPrefixOf (T.pack "IV") s = T.append (T.pack "IIII") (extend (T.drop 2 s))
    | not (T.null s) = T.cons (T.head s) (extend (T.tail s))
    | otherwise = T.empty
```

4.2.2 Section 2

addRN

Arithmetic using Roman numerals is a fairly simple process as at the most basic level, all that needs to happen is that the two strings are concatenated. To get a more exact and syntactically correct version of the addition, `reduce` and `order`, or when used together `normalize` are called. To add more precisely, `extend` is also called since Roman numerals use a form of subtraction; that way, the exact version of the number is the one the function interacts with. While there is a lot of rewriting to be done, there is really only one step to the addition process of Roman numerals.

```
addRN :: T.Text -> T.Text -> T.Text
addRN n m = normalize (T.concat [extend n, extend m])
```

subRN

This is critical for the computation of subtraction and multiplication using Roman numerals since both of those functions will be completed recursively. We can't just write a subtraction function since you can't just do a concatenation like for addition. Instead, we'll continue to subtract one from each side until one becomes empty. To do this, we need a minus one function which is what `subRN` does. It is able to do this by once again listing out a bunch of cases and including an `otherwise`. This function uses many of the same cases as `reduce` however, it doesn't need to account for a simple reorganization. It only focuses on the combined reduced characters and the single characters. Starting with the ***I*** as the topmost case because at the very last one that can be subtracted is one, and then each case thereafter increases in value until reaching ***M***.

```
subRN :: T.Text -> T.Text
subRN s
    | T.isSuffixOf (T.pack "I") s = T.dropEnd 1 s
    | T.isSuffixOf (T.pack "IV") s = T.append (T.dropEnd 2 s) (T.pack "III")
    | T.isSuffixOf (T.pack "V") s = T.append (T.dropEnd 1 s) (T.pack "IV")
    | T.isSuffixOf (T.pack "IX") s = T.append (T.dropEnd 2 s) (T.pack "VIII")
    | T.isSuffixOf (T.pack "X") s = T.append (T.dropEnd 1 s) (T.pack "IX")
    | T.isSuffixOf (T.pack "XL") s = T.append (T.dropEnd 2 s) (T.pack "XXXIX")
    | T.isSuffixOf (T.pack "L") s = T.append (T.dropEnd 1 s) (T.pack "XLIX")
    | T.isSuffixOf (T.pack "XC") s = T.append (T.dropEnd 2 s) (T.pack "LXXXIX")
    | T.isSuffixOf (T.pack "C") s = T.append (T.dropEnd 1 s) (T.pack "XCIX")
    | T.isSuffixOf (T.pack "CD") s = T.append (T.dropEnd 2 s) (T.pack "CCCXCIX")
    | T.isSuffixOf (T.pack "D") s = T.append (T.dropEnd 1 s) (T.pack "CDXCIX")
    | T.isSuffixOf (T.pack "CM") s = T.append (T.dropEnd 2 s) (T.pack "DCCCXCIX")
    | T.isSuffixOf (T.pack "M") s = T.append (T.dropEnd 1 s) (T.pack "CMXCIX")
    | otherwise = T.empty
```

Each case looks for the suffix containing a certain Roman numeral character combination. Once finding one, it will remove the suffix from the string and append the reduced version to the rewritten string. One way of thinking about these key Roman numeral combinations is that they need to be accounted for if they are out of order of the original grammar.

minusRN

Once the backbone of `subRN` is in place, `minusRN` is very easy to conceptualize. This form of the function only accounts for the subtraction of a smaller number from a larger number. The first case asserts

that if the first Roman numeral is an empty string, then the result is an empty string. This also means that the first Roman numeral becomes a negative number through subtraction, resulting in an empty string. This case could be modified to account for negative numbers using second case and changing it so that if `n` is empty then it will equal `m`. In which case, m would be the leftover of the second number and therefore be negative. This is the same line as the second case, except the second case signifies a positive number since the second number was not greater than the first. The third case is the actual action of subtraction.

```
minusRN :: T.Text -> T.Text -> T.Text
minusRN n m
    | n == T.empty = T.empty
    | m == T.empty = n
    | otherwise = minusRN (subRN n) (subRN m)
```

The `otherwise` works by calling `minusRN` recursively while one is subtracted from each Roman numeral until one becomes an empty string. Through this process of subtracting by one, the string is also rewritten and remains syntactically correct. Once one of the numerals is empty, the operation is complete, and the result is achieved.

multRN

The multiplication of Roman numerals accounts for four different cases, with the first three being base cases. The first case states that if either is empty, therefore, equal to zero, then the result will be zero. The second and third cases check if either of the numerals being multiplied is equal to one, and if that is the case, then the result will be the numeral that is not one.

```
multRN :: T.Text -> T.Text -> T.Text
multRN n m
    | (n == T.empty) || (m == T.empty) = T.empty
    | m == T.pack "I" = n
    | n == T.pack "I" = m
    | otherwise = addRN n (multRN n (subRN m))
```

The fourth case, `otherwise`, is the multiplication operation. The multiplication operation can be thought of as adding the first numeral as many times as the second numeral calls for. This is completed by adding the first numeral and then calling `multRN` recursively on the first numeral and the second numeral until the second numeral is equal to one.

4.2.3 Section 3

intRN

This function converts an Integer into a Roman numeral using recursion. This starts by defining a base case where if the Integer is zero then the string will be empty. If not then an ***I*** will be added to the string each time one is subtracted from the original integer resulting in a string of ***I***'s equal to the Integer. `normalize` does have to be called on this function to change the result into a syntactically Roman numeral. This is due to the fact that `normalize` could not be called within the function as a ***I*** would still be added on even if the string was rewritten to a subtraction Roman numeral. Calling `normalize` after the entire recursive computation ensures that the number is accurate

```
intRN :: Integer -> T.Text
intRN 0 = T.empty
intRN n = T.append (T.pack "I") (intRN(n - 1))
```

rnInt

While the temptation is there to format the conversion from a Roman numeral to an Integer the same as the opposite conversion, that would be impossible as far as I could understand. That is because there is no way to represent a base case using the type Data.Text since that would be putting a qualified name in a binding position. To work around this, a variable and the case format were used; from there, the base case and recursive definition could be achieved. The base case was that the Integer would be zero if the variable were empty. If not, one would be added each time the recursive definition was called, and the Roman numeral was subtracted by one.

```
rnInt :: T.Text -> Int
rnInt m
    | m == T.empty = 0
    | otherwise = 1 + rnInt (subRN m)
```

Normalize

Normalize is the simplest function in this program, it only normalizes whatever Roman numeral string is given to it. It accomplishes this by rewriting the string, first using the `order` function and then by using the `reduce`, resulting in a syntactically correct Roman numeral string.

```
normalize :: T.Text -> T.Text
normalize x = reduce (order x)
```

To implement many of the functions in this project, the documentation of the Maybe monad and Text module were used [DM] and [DT].

5 Conclusion

An exploration of Programming Languages in three different parts, from a Programming Language itself to a project using a Programming Language. This paper explored ideas such as Haskell as a Programming Language and essential Programming Language theories. It also furthered these ideas through a project concerning Roman numerals. It showed how to use Haskell, features of Haskell, how Haskell functions, and further resources. Then it examined such theories of Programming Languages as, Parsing and Lambda Calculus. The paper delved into Parsing through abstract syntax trees and ambiguous and non-ambiguous grammar. It then used binding and scope, Church numerals, and the concept of Turing complete to explain Lambda Calculus. Ending with a project on a Roman numeral calculator that uses Haskell to analyze the ideas of recursion and rewriting along with others. This paper can only cover a small part of the study of Programming Languages yet is able to give a brief introduction.

References

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

[FPI] Functional Programming - Introduction, tutorialspoint, 2021

[TT] Types and Typeclasses, Learn You a Haskell

[TCO] Type Classes and Overloading, A Gentle Introduction to Haskell, Version 98

[WFP] What is functional programming?, Ryan Thelin, Oct 15, 2020

[CSE] CSE 130 Lecture Notes, University of California, San Diego, January 24, 2001

[WT] What is a Thunk?, Nick Scialli, January 31, 2020

[LEI] Lazy evaluation illustrated for Haskell divers, Takenobu T.

[R] Recursion, Learn You a Haskell

[AAM] All About Monads, Haskell Wiki

[AM] About Monads, A Gentle Introduction to Haskell, Version 98

[FM] A Fistful of Monads, Learn You a Haskell

[P] Introduction to Programming Languages/Parsing, Wikibooks, 2020

[AS] Parsing, Princeton, Feb 8, 1996

[GP] A Guide to Parsing: Algorithms and Terminology, Gabriele Tomassetti

[DAU] Difference between Ambiguous and Unambiguous Grammar, Geeks for Geeks, July 15, 2020

[AUG] Difference Between Ambiguous and Unambiguous Grammar, Lithmee, August 3, 2018

[LC] Lambda Calculus, Brilliant Wiki, December 15, 2021

[TLC] The Lambda Calculus, Cornell, 2008

[CCL] Call-by-name, Call-by-value, and Lazy Evaluation, David Walker and Andrew W. Appel, 2015

[CN] Church Numerals, Martin Oldfield, June 5, 2013

[CR] Comp 311 - Review 2, Robert Cartwright, 2008

[CE] Church Encoding, Wikipedia, December 15, 2021

[RC] Church Numeral: Haskell, Rosetta Code, October 3, 2021

[TC] Glossary: Turing Complete, Binance Academy, 2021

[FFP] Foundations of Functional Programming, Lawrence C Paulson, 2021

[OTM] λ-Calculus: The Other Turing Machine, Blelloch and Harper, October 25, 2015

[HP] Halting problem, Wikipedia, December 15, 2021

[RNC] Roman Numeral Calculator (Project), Dan Haub, December 23, 2020

[DM] Data.Maybe, Haddock, version 2.26.0

[DT] Data.Text, Haddock, 2.24.0