CPSC-354 Report

Jillian Anderson
Chapman University

February 21, 2022

Abstract

This report will be an in depth analysis on Haskell and functional programming languages as a whole. We will touch on the basics of Haskell and how it differs from imperative programming, recursion, abstract reduction systems, string rewriting, a real world example of abstract reduction systems, and a doubly linked list mini-project.

Contents

1 Introduction

Haskell is a functional programming language that was named after Haskell Brooks Curry, a United States mathematician. Haskell was based on lambda calculus, which is a mathematical logic system implemented to research function definitions and the ways to utilize and apply them. The following will be a discussion of the basics of Haskell, recursion, abstract reduction systems, and the implementation of a doubly linked list as a mini project.

2 Haskell

For this part of the paper we will be diving into the specifics of Haskell. When first learning Haskell, I had only had experience with imperative programming languages such as Java, Python and C++. Haskell however is a functional programming language. In this section, I will be addressing Imperative vs Functional programming languages and the basics of Haskell.

2.1 Imperative vs Functional Programming Languages

There are many differences between Functional and Imperative programming languages. Functional programming is a version of declarative programming. It is implemented using a pure functional approach. Although there is no finite definition of "functional", we tend to accept that a functional programming language uses functions as values and excels at evaluating expressions instead of executing commands. There are many advantages to using this approach including:

- Easier code readability due to functions being designed to execute a specific task, while not relying on external values
- Easier code design changes because Haskell code is easier to refactor
- Easier debugging process because pure functional programming languages can have its functions tested in isolation, making errors easier to spot

Functional programming does not take advantage of mutable data and instead focuses on how functions are applied. This means that in Haskell, once data is created it cannot be changed. There are a few reasons that Haskell implements immutable variables as such. Firstly, immutability helps Haskell mimic a mathematical approach, which is the basis for functional programming. Objects in mathematics do not change. For example, taking an expression $1 + 2 = 3$, the function of adding 1 and 2 to get 3 does not change the values of 1 and 2. These values are immutable despite the fact that an addition "function" is applied. Haskell works in a similar manner. If I had a function $sum(1, 2)$ the function would return 3 without disturbing the input values. In addition, immutability helps protect the integrity of your data. Multiple threads of a program can access immutable data without the concern that it would have been altered in some way at some point. It allows for structure and simplicity within Haskell programs. [HB]

Imperative programming languages however rely heavily on mutable data. Variables are declared that serve the sole purpose of changing value. For example given this python code, the variable i is constantly changing:

```
--Python while loop
i = 1
while(i < 5):
    print("i is equal to: " + str(i))
    i += 1
```

This code will print out statements saying i is equal to 1,2,3,4 and 5. The value of i changes with each iteration of the while loop. In this circumstance, the value of i is clearly mutable and it is necessary for it to be. If i was immutable, it would be stuck at the value of 1 and attempting to run the line "i += 1" would return an error.

There are, however, ways to declare variables with values that cannot be changed in imperative programming languages. For example, in java you can declare a variable using the keyword "final" to instruct that the variable is in it's "final" state. It would look as follows:

```
--Declaring an immutable variable in Java
private final String name = "Jill";
```

Although at first glace it may seem that the "final" keyword and immutability are the same thing, they do have a few key differences. Final means that you cannot alter that particular objects reference point but can still mutate its state. What that means is that although you cannot perform reassignments to a final variable, you can still perform a change to the corresponding object using setter methods. A setter method is a function that is used to modify the value of a instance variable. Immutability, however, means that the state of the object cannot be changed once it is created. [FVIJ]

In preparation for completing this report, my colleague, Jessica Roux, and I decided to implement a basic 5 function calculator in Python. This calculator will act as the complement to the calculator we have already implemented in Haskell for Assignment 1. We did this to demonstrate some key differences between a functional programming language such as Haskell and an imperative programming language such as Python.

Our approach to the Python calculator was incredibly simplistic. It takes in a basic one step calculation in the command line and returns the answer. Our calculator can complete 6 functions: addition, subtraction, multiplication, division, modulo, and power. Our calculator thus far does not account for multi-step equations and therefore does not follow the order of operations. Although it is our plan to implement a calculator in Python that does account for these things, we thought it was incredibly interesting how much more complex the implementation of of the Python calculator was when compared to the Haskell calculator, despite the Python calculator's shortcomings.

Our technique for the Python implementation was to design 6 functions that execute each of the 6 operations and then create a main function to take in the command line argument and call the appropriate function.

```
--Python Calculator Code Snippet
def add(first, second):
    return first + second

--located in main
if sys.argv[2] == "+":
       print(add(sys.argv[1], sys.argv[3]))
```

Using the technique shown above, implementing 6 different operations resulted in approximately 52 lines of code.

```
--Haskell Calculator Code Snippet
```

```
eval (Plus n m) = (eval n) + (eval m)

Plus. Exp ::= Exp "+" Exp1 ;
```

As you can probably see the implementation using python proved to be slightly more lengthy then the Haskell implementation. We were able to implement addition in only two lines in Haskell. Although the complete Haskell calculator ended up being roughly the same number of lines as the Python calculator, the Haskell calculator can do so much more. It can compute multi-operation equations and accounts for the order of operations. Haskell seems the ideal choice to implement a program such as a calculator.

The code for the python calculator can be found at: https://github.com/jianderson/PythonCalculator.

2.2 What Makes Haskell Different?

Haskell is a "lazy" programming language. This means that it will not execute anything unless the execution was specifically requested. Python, on the other hand is an eager evaluation programming language. This type will evaluate an expression automatically when it is assigned relation to a variable. Lazy evaluation can be very efficient when it comes to run time and computational cost. In Haskell, since data is only created as needed, it can ensure that your program is as efficient as possible. [**?**]

Haskell is also considered statically typed, which means that the data types used within a program are known to the compiler at compile time. Haskell has type inference which means that it will "guess" variable types without the user explicitly specifying them. The compiler knows the difference between strings, integers and other data types, meaning that type specific errors will be caught by the compiler. This aspect of Haskell can come in handy because it allows the programmer to write a function that takes in parameters and call that function on many different types of data. This is another way that Haskell code stays clean and concise.

2.2.1 Types and Typeclasses

Since Haskell is a statically typed programming language, the data types are all established at compile time. Haskell uses a variety of different data types that are similar to many other programming languages. To summarize, Haskell utilizes inbuilt type classes such as Int (an integer), Integer (a data type for a long number), Float (a floating point number with single precision), Double (a floating point number with double precision), Bool (a Boolean type), and Char (a character). [HB]

Types for functions are declared in formats that resemble mathematical expressions. These are called type signatures and reveal a lot about how the function works. For example lets analyze:

```
--Haskell plusTwo function
plusTwo :: Int -> Int
plustwo a = a + 2
```

This functions purpose is to take in an integer and add 2 to that integer. So for example, if I ran $plusTwo$ 7 the result should be 9. The type signature is $plusTwo :: Int -> Int$. This tells us that the input will be an Int and the output will also be an Int. This is a very simplistic example. Can you imagine how the type signature might look if there were two inputs instead of just one?

In case you are stuck, we will go through another example. Take a look at this function:

```
--Haskell member function
member :: Int -> [Int] -> Bool
member _ [] = False
member x (y:ys) = if y == x then True else member x ys
```

Let's discuss what this function does. This is a member function, meaning that it takes in a value and a list and returns either true or false based on if the list contains the specified value. We can see the the type signature for this function looks as follows: $member :: Int- > [Int]- > Bool$. This means that for the member function, it will take in an Int and a list of Ints (represented by $[Int]$) and return a Boolean.

There are also cases where a function can take in another function. Take a look at this example:

```
--Haskell map function
map :: (Int -> Int) -> [Int] -> [Int]
map func [] = []
map func (x:xs) = (func x) : (map func xs)
```

A map function is one that takes in another function and a list and performs that function on each element of the list. So for example if I were to run $map\ plusTwo\ [1,2,3]$ the output would be $[3,4,5]$. The type signature for this function looks like $map :: (Int- > Int)- > [Int]- > [Int]$. This means that the map function will take in another function and a list of Ints and return a list of Ints. We can tell by the type declaration that the input function takes in an Int and returns an Int.

Type signatures for functions in Haskell are extremely important because they give the compiler the knowledge of how to use the function. They are also extremely helpful for programmers that are attempting to write a function. If you can decide what the inputs and outputs of your function will be, the compiler error messages will help you code your function. For example, if you write your code in a way that the expected output does not match the actual output, the Haskell compiler will make that known. The error message will display what type the output should be and also what type the output currently is. This is extremely helpful because it can help you catch errors in your code. [TS]

2.3 Recursion

Recursion is a programming technique in which a function is called within it's own definition. An example of recursion in an imperative programming language would be:

```
def countToInfinity(num):
    print(countToInfinity(num+1))
```

This function will count concurrent numbers indefinitely, which makes its "countToInfinity" name very fitting. This is an example of recursion because the function countToInfinity is called within itself. Recursion has many uses within imperative programming and is a way to execute a function multiple times until a specified condition is met. In imperative programming however, there are many cases where recursion can be replaced with for or while loops. For example, you can program the countToInfitinity function as follows:

```
def countToInfinity(num):
    while (true):
        print(num)
        num += 1
```

It's interesting because we can clearly see that the while implementation of countToInfinity requires more lines of code then the recursive version, but many imperative language programmers tend to make great use out of loops and shy away from utilizing recursion.

However, in a functional programming language like Haskell, recursion is even more crucial. This is because Haskell programming is based on defining what something is. Many times recursion is used to create these definitions. There are no loops in Haskell meaning that you have to instead rely on being very intentional with how you define something. With recursion however, you need a way for the function to eventually

terminate. In the recursive countToInfinity example above, the function would not be able to terminate. It would continue calling the function infinitely. When using recursion in Haskell it is crucial to define an edge condition, in order to prevent infinite looping from happening. [LYAH]

An edge condition is a definition that is responsible for terminating the recursion.

```
-- add natural numbers
addN :: NN -> NN -> NN
addN O m = m
addN (S n) m = S (addN n m)
```

The recursive definition for addN in this example is addN (S n) m = S (addN n m). You can see that addN is called within the definition. The edge case is addN O m = m. The way this works is first the recursive definition will execute. It will continue to call addN until finally the function will be called on O and m. This will match our edge case, which means it will equal m. The function is terminated and the answer is found.

This is made possible through a feature of Haskell called pattern matching. [HPM] In pattern matching, input values attempt to match with specific function patterns. A heavily utilized example is the $x : xs$ pattern. In this pattern x is the head of the list and xs is the tail of the list. So for example, if the input was $[1, 2, 3, 4]$ x would be equal to 1 and xs would be equal to $[2, 3, 4]$. Using this pattern in a function would look as follows:

```
-- function to calculate the sum of numbers in a list
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

In order to understand how pattern matching works it can be help to follow an example through step by step. Let's say we ran the function $sumList[1, 2, 3]$. Eventually, the output should be 6, but pattern matching is a huge part of why this function works. We can see in the function that there are two $sumList$ definitions: $sumList[]$ and $sumList(x : xs)$. Looking at our example $sumList[1, 2, 3]$ can you guess which pattern our input falls into?

If you guessed $sumList(x : xs)$ you would be correct! $(x : xs)$ represents a list rewritten as the head and the tail while [] represents an empty list. So initially we would have:

$$sumList[1, 2, 3] = 1 + sumList[2, 3]$$

Next, we can look at our input for the first recursive call: $sumList[2, 3]$. This call still pattern matches with $sumList(x : xs)$ meaning that we can execute the function again as follows:

$$sumList[1, 2, 3] = 1 + (sumList[2, 3] = 2 + sumList[3])$$

This can be rewritten as:

$$sumList[1, 2, 3] = 1 + 2 + sumList[3]$$

You can make your first addition between the 1 and the 2. Additionally, we can perform the finally application to $sumList[3]$. [3] is still technically a list meaning that it pattern matches to $sumList(x : xs)$ as $(3 :\#)$.

$$sumList[1, 2, 3] = 3 + (sumList[3] = 3 + sumList[])$$

This can be rewritten as:

$$sumList[1,2,3] = 3 + 3 + sumList[]$$

First, we can add together 3 and 3. Next, lets look at $sumList[]$ and attempt to pattern match it. We can see that it does not line up with $sumList(x : xs)$ which we have been using up until now because it is an empty list. It does, however, pattern match to $sumList[] = 0$. Finally, we will have:

$$sumList[1,2,3] = 6 + (sumList[] = 0)$$

This can be rewritten as:

$$sumList[1,2,3] = 6 + 0$$

As a final answer we can see that $sumList[1,2,3]$ is equal to 6. This matches our expected answer so we can tell the recursion executed successfully.

2.3.1 Towers of Hanoi

The Towers of Hanoi is a mathematical logic game that is made up of 3 rods and an assortment of disks of different sizes. The key of the game is to move the tower from the first rod to the third rod. The rules are that you can only move 1 disk at a time and disks can only be placed on disks of a larger size or else the tower will collapse. This puzzle can also be seen as an algorithm that utilizes recursion.

The two rules of the algorithm can be described as:

```
hanoi 1 x y = move x y
```

```
hanoi (n+1) x y =
     hanoi n x (other x y)
      move x y
      hanoi n (other x y) y
```

Let's break down the meaning of these rules. $hanoinxy$ takes in n (the number of disks), x (the place where the tower currently is), and y (the place where the tower should go). So for example given $hanoi102$ I would be saying that there is currently a tower made up of only 1 disk located a rod position 0 that eventually needs to be moved to rod position 2. Additionally, $movexy$ is a function that moves the top most disk from rod x to rod y. Given $move12$ the topmost disk in rod position 1 would be moved to rod position 2. The last function $otherxy$ gives the third rod which isn't x or y. So for example, given $other02$ the position in question would be rod position 1.

Lets say we are given 5 disks at rod position 0 that need to be moved to rod position 2. How would we do that? Below is a jumping off point to solve which collection of moves would allow you to move all 5 disks to rod position 2. Try completing the execution on your own and list the moves that a Towers of Hanoi user would have to make.

```
hanoi 5 0 2
      hanoi 4 0 1
            hanoi 3 0 2
                  hanoi 2 0 1
                        hanoi 1 0 2 = move 0 2
                        move  0 1
```

```
                    hanoi 1 2 1 = move 2 1
              move 0 2
              hanoi 2 1 2
                    hanoi 1 1 0 = move 1 0
                    move  1 2
                    hanoi 1 0 2 = move 0 2
```

In case you are stuck, lets walk through it now. Here are my solutions to extend the solution:

```
hanoi 5 0 2
      hanoi 4 0 1
            hanoi 3 0 2
                  hanoi 2 0 1
                        hanoi 1 0 2 = move 0 2
                        move  0 1
                        hanoi 1 2 1 = move 2 1
                  move 0 2
                  hanoi 2 1 2
                        hanoi 1 1 0 = move 1 0
                        move  1 2
                        hanoi 1 0 2 = move 0 2
           move 0 1
           hanoi 3 2 1
                 hanoi 2 2 0
                        hanoi 1 2 1 = move 2 1
                        move 2 0
                        hanoi 1 1 0 = move 1 0
                  move 2 1
                  hanoi 2 0 1
                        hanoi 1 0 2 = move 0 2
                        move 0 1
                        hanoi 1 2 1 = move 2 1
     move 0 2
     hanoi 4 1 2
            hanoi 3 1 0
                   hanoi 2 1 2
                          hanoi 1 1 0 = move 1 0
                          move 1 2
                          hanoi 1 0 2 = move 0 2
                   move 1 0
                   hanoi 2 2 0
                          hanoi 1 2 1 = move 2 1
                          move 2 0
                          hanoi 1 1 0 = move 1 0
             move 1 2
             hanoi 3 0 2
                   hanoi 2 0 1
                          hanoi 1 0 2 = move 0 2
                          move 0 1
                          hanoi 1 2 1 = move 2 1
                   move 0 2
                   hanoi 2 1 2
                          hanoi 1 1 0 = move 1 0
                          move 1 2
                          hanoi 1 0 2 = move 0 2
```

The structure of this algorithm is actually quite intuitive. Starting with $hanoi502$ each $hanoi$ call will need to have 3 parts: $hanoinx(otherxy)$, $movexy$, and $hanoin(otherxy)y$. While computing this I kept comments of the values of n, x, and y for each $hanoi$.It made it easier to keep track of. Looking at this algorithm, we can see that it relies heavily upon recursion. The $hanoi$ function is continuously called and only breaks for instances such as $hanoi102 = move02$. The reason is for this particular $hanoi$ call, only 1 disk is being moved from 0 to 2. We have a function move to achieve that for us, which is why $hanoi102$ can terminate into $move02$ which translates to move the top disk from 0 to 2. Comparing this to the way that we have seen recursion in the past, we can determine that $hanoi1\_\_$ would be our edge case. Any variable that would pattern match into that form, would break the recursion.

Now lets collect the move functions that we established in order to create instructions for how to solve the 5 disk hanoi puzzle. They are as follows:

move 0 2, move 0 1, move 2 1, move 0 2, move 1 0, move 1 2, move 0 2, move 0 1, move 2 1, move 2 0, move 1 0, move 2 1, move 0 2, move 0 1, move 2 1, move 0 2, move 1 0, move 1 2, move 0 2, move 1 0, move 2 1, move 2 0, move 1 0, move 1 2, move 0 2, move 0 1, move 2 1, move 0 2, move 1 0, move 1 2, move 0 2

If you want to check your work, you can head to https://www.mathsisfun.com/games/towerofhanoi.html and choose the 5 disk puzzle to execute your moves. Another reason I love this example is that is shows the real world application of algorithms with recursion. Following the patterns of the equations leads us to the result for the Towers of Hanoi puzzle.

3 Programming Languages Theory

3.1 Introduction on Abstract Reduction Systems

An Abstract Reduction System, abbreviated as ARS, (A,R) is a set A together with a relation $R \subseteq A \times A$. [F]

This definition above means that R is a set of pairs (a, b) with $a, b \in A$. Abstract Reduction Systems are used to transform data according to a specified set of rules. In layman's terms, an ARS is a set of objects that have relationships with one another, denoted by the symbol $\Longrightarrow$. It utilizes term rewriting, also referred to as reduction. There are a few definitions that are important to go over before jumping into examples.

Terminating:

- An abstract reduction system is considered terminating if it does not admit an infinite computation.
- This definition above means that there is no situation in which one can infinitely apply rules to a given string.

Confluent:

- Confluence is an attribute of some Abstract Reduction Systems.
- If a term is confluent it means that using the rules provided by the abstract reduction system, the term can be rewritten in more than one different way while still giving the same end result.
- A real world example of confluence is the equation $1 + 2 + 4$. This equation can be rewritten as $3 + 4$ or $1 + 5$ but it will still yield the same end result of 6.

Normal Form:

- A term b is a normal form is there is no reduction c according to the rules of the Abstract Reduction System.

- A term b is considered a normal form of a if b itself is a normal form and there exists rules to conduct a computation from a to b.

Unique Normal Forms:

- An Abstract Reduction System is said to have Unique Normal Forms if all elements of the system have a singular, or unique, normal form.
- This definition means that for any given element of a ARS with Unique Normal Forms there exists only one normal form which this element can reduce to.

Equivalency Class:

- An equivalency class is a subset of strings that are "equivalent" to each other based on the fact that each string is accessible via the others by applying rules of the Abstract Reduction System either forwards or backwards.

$$a \implies b$$

$$aa \implies bb$$

- An example of this would be that given the above Abstract Reduction System rules, a and b would be in one equivalency class while aa and bb would be in a second equivalency class.
- The reasoning behind this is that from a it is possible to apply rules in order to get to b but it is not possible to apply rules and end up with aa or bb.

Invariant:

- Invariants can be viewed as properties that do not change when rules are applied.
- Invariants can come in many different forms such as string length, quantity of objects and combinations of objects.
- There can be more then one invariant for any given Abstract Reduction System.
- Relating Invariants to Equivalency Classes, in order to confirm that 2 strings are in the same equivalency class these two string must hold the same value for each of the Invariants for the Abstract Reduction System.
- For example, if the Invariant is the quantity of a's one equivalency class could be filled with objects that have an even number of a's and the other equivalency class filled with objects that have an odd number of a's.

3.2 String Rewriting

Next, in order to better understand Abstract Reduction Systems, we will be walking through some string rewriting exercises. String rewriting is utilizing the rules of an ARS $(A, \implies)$ where A is a set of words in order to perform reductions. Rules on smaller words can be applied to any part of longer words. [R]

Rules on smaller words can be applied to any part of longer words. For example, when given the rule $aa \implies ab$, if we have a string $bbbaa$ our rule can be applied to the end of the word at aa. The reduction will produce the new string $bbbab$.

3.2.1 Exercise 1:

$$aa \implies a$$

$$bb \implies a$$

$$ab \implies b$$

$$ba \implies b$$

This example is a terminating Abstract Reduction System but the important question to consider is how can we tell? First, we must keep in mind that the definition of termination is that there exists no situation where one can infinitely apply rules to a word. Looking at the structure of the rules in this ARS, it is easy to see that the infinite application of rules would not be possible. Each rule reduces the string in length. What this means is that a word will continue to get smaller as rules are applied until it is too small to apply any further rules. At this point, the world will consist of one letter (either a or b) and it will terminate.

Next, we can identify the normal forms of this Abstract Reduction System. As a reminder, the definition of a normal form is a word in your system which you cannot apply any more rules to. With that in mind, the normal forms of this ARS would be a and b. These two are normal forms because given either of these words, there are no further reductions to be done. Additionally, we can also say that a is a normal form of aa and bb and b is a normal form of ab and ba. We can say this because a and b are normal forms and there exists rules to reduce aa and bb to a and ab and ba to b.

The invariant for this example is $(2(\#a) + \#b)\%2$. This means that there are 2 equivalency classes: one where $(2(\#a) + \#b)\%2 = 0$ and another where $(2(\#a) + \#b)\%2 = 1$. Lets attempt to determine which normal form goes with which equivalency class. Let't take one of the rules as an example. If we have the string ab we can see how this applys to the invariant. The number of a's is 1 and the number of b's is 1. $(2 + 1)\%2 = 3\%2 = 1$. So we can tell that the b normal form associates with the invariant computing to 1 while the a normal form associates with the invariant computing to 0.

3.2.2 Exercise 2:

$$aa \implies a$$

$$bb \implies b$$

$$ba \implies ab$$

$$ab \implies ba$$

This Abstract Reduction System is slightly different then the one from the first example. One difference is that this ARS does not terminate. We know this due to a few different factors.

First, one thing that should stick out is the observation that not all the rules reduce the string length. The second two rules $ba \implies ab$ and $ab \implies ba$ maintain the length of the string instead of shortening it. While this doesn't always indicate that the Abstract Reduction System won't terminate, it is something to pay attention to. Next, we should take a closer look at those two rules. They are the inverse of each other. This is the thing that indicates the ARS will not terminate. Termination, as we know, means that there can be no situation where rules can be applied infinitely. Given the string ba, the rules $ba \implies ab$ and $ab \implies ba$ can be applied an infinite amount of times, changing the string back and forth from ab to ba. This is not permitted for a terminating Abstract Reduction System which is how we know that this ARS does not terminate.

Next, we can attempt to identify the normal forms of this Abstract Reduction System. The normal forms are a and b. One thing that can sometimes be confusing is why ab and ba are not considered normal forms,

as it can be a misconception that all terms that are to the right of the $\implies$ are normal forms. However, in order to answer this, we must draw on the definition of a normal form, which is that a term is considered a normal form if it is a word of the ARS that no further rules can be applied to. ab and ba are both terms that rules can be applied to meaning that they are not considered normal forms.

The next question to address for this Abstract Reduction System is if there is a way to change the rules so that the new ARS has unique normal forms but still the same equivalence relation. There is, and the way to achieve this is to remove either the rule that $ba \implies ab$ or $ab \implies ba$. For our purposes lets assume that we remove the last rule in the Abstract Reduction System $ab \implies ba$. Now, the loop that would allow ba to be rewritten as ab and then oscillate between the two values has been ended. Now, ba will terminate to ab. This ARS now has unique normal forms because any element within the system has only one normal form it can reduce down to. aa reduces to a, bb reduces to b, and ba reduces to ab.

3.2.3 Exercise 3:

$$ab \implies ba$$

$$ba \implies ab$$

This Abstract Reduction system is comprised of only the last two lines in the previous example.

First let's take a look at what this Abstract Reduction System actually does. Upon first look, I can tell that this ARS will allow me to put my string into any order desired. Let's take a string $aabb$ and attempt to rewrite it as $abab$:

$$aabb \equiv abab$$

Next, lets see if we can rewrite the same string $aabb$ to $bbaa$:

$$aabb \equiv abab \equiv baab \equiv baba \equiv bbaa$$

This Abstract Reduction System can rewrite a string of a's and b's to any formulation of the string, as long as it contains the same number of a's and b's. For example, $aabb$ cannot be rewritten to $abbb$.

Next, with the information we discussed above, it should be easy to decide if the Abstract Reduction System is terminating or not. If you answered not terminating, you would be right! This ARS is non-terminating because the string ab can have infinite rules applied to it. In other words, there is a loop in the rules of this ARS.

The next question to consider are what are the normal forms of this ARS. In this example, there are actually no normal forms. We remember that a normal form is a term in the system that cannot have any further rules applied to it. Therefore, it should be simple to see that, in this ARS, there are no terms that cannot be applied any further rules. ba can reduce to ab and ab can reduce to ba.

Next, we can ask if there is a way to change the rules of the Abstract Reduction System so that the new ARS has unique normal forms. There is, and it is extremely similar to the example above. By taking out either of the rules, the new ARS would have unique normal forms. For example, if we removed the second rule, we would be left with the ARS:

$$ab \implies ba$$

In this Abstract Reduction System, it would be considered terminating, the normal form would be ba, and the ARS would have unique normal forms because ab would always reduce to and terminate at ba.

3.2.4 Exercise 4:

$$ab \implies ba$$

$$ba \implies ab$$

$$aa \implies$$

$$b \implies$$

This Abstract Reduction System is slightly more complicated, as it includes empty strings.

First, lets get a feel of the rules for the ARS but reducing some example strings. Lets start with *abba*:

$$abba \equiv baba \equiv baab \equiv bb \equiv b \equiv <>$$

This particular string will reduce down to an empty string. Next lets try the string *bababa*:

$$bababa \equiv baabba \equiv bbba \equiv bba \equiv ba \equiv a$$

This string reduced down to a single a. This will be important later.

This Abstract Reduction System is non terminating but the question is why? The answer can be found in the first and second lines of the ARS. We have seen rules similar to these before so we know that they result in a loop and can also be used to order the string in any particular order, as long as the initial and final strings have the same number of a's and b's.

Next we should address the equivalence classes for this Abstract Reduction System. There appears to be 2 equivalency classes. If we remember, an equivalency class is a subset of strings that are equivalent to each other, meaning that each string is accessible via the others by applying the rules of the ARS to the strings either backwards or forwards. The two equivalency classes for this ARS are:

$$b \equiv <> \equiv aa$$

$$ab \equiv ba$$

Lets talk about why these are the two equivalency classes for the Abstract Reduction System. The first one involves the rules $aa \implies$ and $b \implies$. Starting from b you can reduce to an empty string $<>$ and then from that empty string $<>$ you can reverse reduce to aa. This is why the first equivalency class is $b \equiv <> \equiv aa$. It is important to note that this reverse reduction is not valid for the actual string rewriting, just for the equivalency class definitions. For example, given this Abstract Reduction System if I had an empty string $<>$ I would not be permitted to rewrite it as aa.

The next question to ask is where the invariants come in. If we recall, an invariant is a property that does not change when rules are applied. Invariants can be discovered easily by examining the equivalency classes because strings in the same equivalency class must hold the same value for each invariant.

Looking back on our example, the two equivalency classes are:

$$b \equiv <> \equiv aa$$

$$ab \equiv ba$$

The invariant, or the thing that distinguishes whether a term will be in the first or second equivalency class is the number of a's. Lets take an example, *aaab* and determine which equivalency class it is in.

$$aaab \equiv ab \equiv ba \equiv ab$$

$aaab$ has three a's, an odd number, and belongs to the second equivalency class. Lets try another, $aabb$.

$$aabb \equiv aab \equiv aa \equiv <>$$

$aabb$ has two a's, an even number, and belongs to the first equivalency class. It is clear that depending on if the number of a's is even or odd, the string belongs to the first equivalency class if the number is even and the second equivalency class if the number is odd.

Next, we can look at how we would change the rules of the Abstract Reduction System so it becomes terminating without changing its equivalence classes. In order to do this, we would remove either the first or second rule. For our purposes, lets remove the first, making the new ARS rules:

$$ba \implies ab$$

$$aa \implies$$

$$b \implies$$

Now lets confirm that this new Abstract Reduction System has the same equivalency classes as the old one. The first equivalency class has not been changed and will stay:

$$b \equiv <> \equiv aa$$

The second equivalency class lost a rule but it appears that this rule was redundant anyways! The second equivalency class can stay:

$$ab \equiv ba$$

Even without the initial first rule, the equivalency class $ab \equiv ba$ will stay the same due to our ability to use reverse reduction values when defining the equivalency classes. Therefore, it is possible to change the rules of the first Abstract Reduction System to make it terminating while not changing it's equivalency classes.

3.2.5 Exercise 5:

$$ba \implies bbaa$$

$$aa \implies$$

$$ba \implies ab$$

$$ab \implies ba$$

This example is different from all the ones we seen before due to the rule $ba \implies bbaa$. This is the first time we have seen a rule that increases the length of the string. Lets begin with some sample reductions. First, lets work through if it is possible to reduce ab to $aabb$:

$$ab \equiv ba \equiv bbaa \equiv baba \equiv abba \equiv abab \equiv aabb$$

We were successfully able to rewrite $abtoaabb$. Next we are going to try a longer, more complicated example. Can we reduce ba to $abbaababbab$:

$$ba \equiv bbaa \equiv bbbaaa \equiv bbbbaaaa \equiv bbbbbaaaaa \equiv bbbbbbaaaaaa$$

We can stop here. We already have our answer. The target string was 11 characters long. We will not be able to perform any reductions in order to produce a string that has 11 characters. If we start with a string of an even number of characters such as ab and we are only able to grow our string by 2 characters every rule application ($ba \implies bbaa$) then it makes sense that we would not be able to reach a string of 11 characters. If we had starting with the string bab we would have successfully be able to reduce to the string $abbaababbab$.

Next, we should look at if we can reduce ba to $abbaababbaba$. First we need to rewrite the string so it is the appropriate number of characters long:

$$ba \equiv bbaa \equiv bbbaaa \equiv bbbbaaaa \equiv bbbbbaaaaa \equiv bbbbbbaaaaaa$$

The string above is 12 characters which is good because the target string has 12 characters. All that is left is manipulating the string using $ba \implies ab$ and $ab \implies ba$ to correct the order of the characters:

$$\begin{gathered} bbbbbbaaaaaa \implies bbbbbabaaaaa \implies bbbbabbaaaaa \implies bbbabbbaaaaa \implies bbabbbbaaaaa \implies \\ babbbbbaaaaa \implies abbbbbbaaaaa \implies abbbbbabaaaa \implies abbbbabbaaaa \implies abbbabbbaaaa \implies \\ abbabbbbaaaa \implies abbabbbabaaa \implies abbabbabbaaa \implies abbababbbaaa \implies abbaabbbbaaa \implies \\ abbaabbbabaa \implies abbaabbabbaa \implies abbaababbbaa \implies abbaababbaba \end{gathered}$$

We were able to rewrite ba as $abbaababbaba$. When looking at these examples and the rules of the Abstract Reduction System we can identify that strings with no b's and an even number of a's will reduce to an empty string ¡¿, while strings with no b's and an odd number of a's will reduce to a. Strings with at least 1 b and 1 a can reduce to a string of only b's. For example the string ba will reduce as follows:

$$ba \equiv bbaa \equiv bb$$

The number of b's and a's don't matter. If there is at least one of each, the string will always reduce to b's only. Take the longer example $bbbbaaa$:

$$bbbbaaa \equiv bbbba \equiv bbbbbaa \equiv bbbbb$$

This string still reduces down to b's only! As long is there is at least 1 b we can use the first rule ($ba \implies bbaa$) to generate enough a's so we can use the second rule ($aa \implies$) to remove the a's from the string, leaving only b's.

3.3 Real World Example

When Abstract Reduction Systems are discussed in regards to real world examples, a popular path to explore is that of adding Roman Numerals using string rewriting. For example if you wanted to add the Roman Numeral 3, written as III, to the Roman Numeral 10, written as X, then you can simple concatenate them and then rewrite them into normal form:

$$IIIX$$

$$IIXI$$

$$IXII$$

$$XIII$$

However, since this example is so widely explored, I wanted to come up with a different real world application to analyze. I sat down with a colleague of mine, Jessica Roux, and we devised an Abstract Reduction System that deals with dollars and coins of American currency.

3.3.1 Currency Abstract Reduction System

Our Abstract Reduction System is made up of 2 types of rules, ordering rules and equivalency rules. Each letter represents a certain type of currency:

$$c = \text{Dollar}$$

$$q = \text{Quarter}$$

$$d = \text{Dime}$$

$$n = \text{Nickle}$$

$$p = \text{Penny}$$

Using these symbols we can begin to create the equivalency rules:

- $ppppp \implies n$
- $nn \implies d$
- $ddn \implies q$
- $ddddd \implies qq$
- $qqqq \implies c$

These rules will be used to help normalize a certain amount of money. For example, say you have two dimes and one nickle, or ddn. This can be rewritten in a more efficient way by saying that we actually have the equivalent of one quarter or q. However, what if we have a coin string that is equivalent to 30 cents and is written as two nickels and 2 dimes, or $nndd$. Order matter for reductions so in its current state, the string $nndd$ could not be reduced to qn. In order to combat this we devised a set of rules to act as sorting rules to put the strings into optimized ordered. The intended order is from largest quantity to smallest quantity, so Dollars, Quarters, Dimes, Nickles, Pennies (c,q,d,n,p). So for example, given the string $ppdndqqqc$ we would want our rewriting rules to return a string $cqqqddnpp$. From there we can make our $ddn \implies q$ resulting in $cqqqqpp$ and then another reduction using $qqqq \implies c$ to end up with the string $ccpp$, which is the most normalized way of writing \$2.02. The string ordering rules are as follows:

- $qc \implies cq$
- $nc \implies cn$
- $dq \implies qd$
- $pq \implies qp$
- $pd \implies dp$
- $dc \implies cd$
- $pc \implies cp$
- $nq \implies qn$
- $nd \implies dn$
- $pn \implies np$

3.3.2 Example

Let's try an example. If we go to the bank with 6 Dimes, 3 Dollars, 6 Pennies, 11 Quarters, and 2 Nickels and we want to exchange it for an equal amount but using the smallest number of coin/dollar combinations as possible, what will the bank give us? Keep in mind that this value is \$6.51 so we can compare it to our final answer.

Right now we have $ddddddcccppppppqqqqqqqqqqnn$. Right off the bat, I see some quantity simplification that can be done. We know that $qqqq \implies c$ so we can rewrite $qqqqqqqq$ as cc, leaving $ddddddcccppppppccqqqnn$. I also see that using $ppppp \implies n$, we can turn 5 of the pennies into a nickel, leaving $ddddddcccnpccqqqnn$. Lastly, using $ddddd \implies qq$ we end up with a final string of $qqdcccnpccqqqnn$

The next step is to order the remaining string:

$$qqdcccnpccqqqnn \equiv qqcdccnpccqqqnn \equiv qcqdccnpccqqqnn \equiv cqqdccnpccqqqnn \equiv cqqcdcnpccqqqnn \equiv$$
$$cqcqdcnpccqqqnn \equiv ccqqdcnpccqqqnn \equiv ccqqcdnpccqqqnn \equiv ccqcqdnpccqqqnn \equiv cccqqdnpccqqqnn \equiv$$
$$cccqqdncpcqqqnn \equiv cccqqdcnpcqqqnn \equiv cccqqcdnpcqqqnn \equiv cccqcqdnpcqqqnn \equiv ccccqqdnpcqqqnn \equiv$$
$$ccccqqdncpqqqnn \equiv ccccqqdcnpqqqnn \equiv ccccqqcdnpqqqnn \equiv ccccqcqdnpqqqnn \equiv cccccqqdnqpqqnn \equiv$$
$$ccccqqdqnpqqnn \equiv cccccqqqdnpqqnn \equiv ccccqqqdnqpqnn \equiv cccccqqqdqnpqnn \equiv cccccqqqqdnpqnn \equiv$$
$$ccccqqqqdnqpnn \equiv cccccqqqqdqnpnn \equiv ccccqqqqqdnpnn \equiv cccccqqqqqdnnpn \equiv ccccqqqqqdnnnp$$

Next, we can continue equivalency reductions:

$$ccccqqqqqdnnnp \equiv cccccqdnnnp \equiv cccccqddnp \implies cccccqqp$$

The final answer is *cccccqqp* as there are no further reductions to be done. When looking at this string we can clearly see that its monetary value is \$6.51. This is great news because it means that we correctly performed our reductions and ended up with the same dollar amount with far less dollars and coins.

3.3.3 ARS Analysis

In this section we will be analyzing the properties of this real life Abstract Reduction System.

Confluence First, we should discuss confluence as it relates to this example. If we recall, confluence is an attribute of some ARS. If an ARS if confluent, it means that, using the rules, a term can be rewritten in more then one way while still achieving the same end result. Our Currency Abstract Reduction System is confluent. We can show this by looking at an example.

Let's say we have the dollar amount of \$0.50. This can be reduced in a few different ways starting with *ddddnn*:

$$ddddnn \equiv ddqn \equiv dqdn \equiv qddn \equiv qq$$

$$ddddnn \equiv ddddd \equiv qq$$

Both of these rule applications yield the same end result, *qq*. However, how the end result is achieved is different. In the first reduction we apply the rules in this order:

$$ddn \implies q,\ dq \implies qd,\ dq \implies qd, \text{ and } ddn \implies q$$

In the second reduction we apply the rules in this order:

$$nn \implies d \text{ and } ddddd \implies qq$$

Different rules are being applied in each reduction yet it still yields the same end result meaning that the Abstract Reduction System is confluent. This makes sense because each dollar amount has its own equivalency class meaning it has a final reduced form. Performing reductions in one part of the string doesn't mean that other reductions are impossible.

Termination We can use a measure function to prove termination in our Currency Abstract Reduction System. For example, let's assign numerical values to our strings in order to visualize the measure function:

- $p = 4$
- $n = 3$
- $d = 2$
- $q = 1$
- $c = 0$

Next, let's do an example to show that through reductions, the numerical value will continually decrease, proving that the ARS terminates. Let's reduce *ncqnp*.

$$30134 \equiv 03134 \equiv 01334 \equiv 0124$$

$$ncqnp \equiv cnqnp \equiv cqnnp \equiv cqdp$$

As shown above, the numerical value will continue to decrease as rules are applied because each rule either shortens the length of the string or orders the string in a way that rewrites the smaller numbers to the left hand side or the larger numbers to the right-hand side. Through this measure function, we can prove that our Currency Abstract Reduction System will terminate.

Equivalency Classes Next we can look into identifying the equivalency classes for our Abstract Reduction System. For the Currency ARS, the equivalency classes are a little different then anything we have seen in this document prior. There are an infinite number of equivalency classes. Each dollar amount represents its own equivalency class. For example, d, nn, and $ppppppppppp$ would all be in the same equivalency class because they all represent the dollar amount \$0.10. Since, you can have an infinite number of dollar amounts, there are an infinite number of equivalency classes. I have listed some examples below:

$$\$1.36 \equiv ncqnp \equiv cnqnp \equiv cqnnp \equiv cqdp$$

$$\$1.25 \equiv cnnnnn \equiv cdnnn \equiv cddn \equiv cq$$

$$\$0.15 \equiv pppppppppppppppp \equiv nnn \equiv dn$$

Invariant Since the equivalency classes all depend on the monetary amount it is easy to deduce what the invariant would be. If you need a refresher, an invariant is something that stays the same even when rules are applied. Therefore, the invariant for the Currency Abstract Reduction System is the monetary amount. In the equivalency class for \$0.5 you can apply rules such as $ppppp \implies n$ and this will not change the monetary value.

4 Project

My college, Jessica Roux, and I worked together to create our mini project for this report. We used an variation of Lambda Calculus, titled Lambda Fun, created using Haskell to implement our mini project. For our third assignment in our Programming Languages class, we were tasked with implementing a round robin linked list using Lambda Fun. For our mini project, Jessica and I extended this implementation and used Lambda Fun to create a function doubly linked list.

The following will be a discussion on the intricacies and process for our implementation. The entirety of the code for the project can be found at: https://github.com/jianderson/Mini-Project

4.1 Project Background

There is some background that will be good to discuss before diving into our procedure for this mini project. First, assignment 3 was an implementation of a circular list. This circular list consisted of nodes that held data and 1 pointer. The pointer lead to the address of the next node in the list and so on. At the last node of the list, the pointer looped back to point to the front of the list, completing the circular shape. The nodes were stored as $[e, a]$ where e was the element of the node and a was the pointer. The implementation of this list had six different functions associated with it: newCList, next, get, update, insert, and delete.

"newCList" takes in one element and creates a new circular list containing only the one element. The pointer a points back around to itself. "next" takes in the address of a circular list and returns the address of the next element in the list. "get" takes in the address of a circular list and returns the element (e) at the current node. "update" takes in an element (e) and a circular list and replaces the element at the current position in the list with the new element e. "insert" takes an element (e) and a circular list and inserts the element e into the list after the current element. "delete" takes in a circular list and deletes the element after the current element. These functions are all crucial to the way the circular list can be created.

The circular lists we had been dealing with are a variation of a linked list. A linked list is a data structure that consists of nodes that are connected to each other via pointers. A doubly linked list is an extension of a linked list where each node has an element, a pointer pointing to the next node, and also a pointer pointing backwards to the previous node. A doubly linked list is extremely useful in situations where forward and backward navigation of the data is helpful.

4.2 Project and Memory Model

Once we decided that our plan was to implement a doubly linked list using Lambda Fun, we had to discuss the memory model of the language in order to ensure that we were coding the data structure in the correct way.

```
--visualization of the memory model
Env:
l = <address 0>
Memory:
0 -> 7
```

The memory model of Lambda Fun was separated into an immutable stack and a mutable heap. The stack is represented by "env" and the heap is represented by "memory". It was crucial to have ample practice with this memory model before attempting the doubly linked list so we could ensure that we were saving our data the right way.

In traditional memory models. The stack is a linear data structure that has very quick access. It allocates local variables only and is great for holding addresses that point to locations in the heap. The heap has a hierarchical data structure that has slower access speeds. The heap allows you to access variables globally and memory is randomly allocated. It is important to distinguish between the two to ensure that the location you choose to store data makes sense in accordance with the pros and cons of the stack and the heap.

"new" is the keyword that alerts the compiler to allocate memory on the heap. For our Lambda Fun memory model we would create a new variable by running "val c = new [];;". If you take a look at the memory model after running this command, it will look as follows:

```
--memory model
Env:
c = <address 0>
Memory:
0 -> un-initialized
```

We can see that so far c has not been initialized. The only thing that has happened is that we have told the memory that there is a variable c that will point to allocated memory in the heap located at address 0. To assign the value 5 to c we would run: "c := 5;;". Checking the memory model after this will reveal that a change has been made:

```
--memory model
Env:
c = <address 0>
Memory:
0 -> 5
```

Another important thing to touch on before we dive into the project implementation is the function of the ! symbol. For our purposes, asking the memory what the value of c is will return address 0. Lambda Fun is coded in such a way that accessing a variable means accessing it from the stack which does not reveal the content of the variable, only the address where that content is stored. In order to access the actual value, we would need to find !c. This comes in handy once we attempt to insert and delete from the doubly linked list because in order to do that, we have to change their pointers which can only be done by accessing the value of each variable on the heap.

4.3 Implementation of the Project

Below will be a walk-through of our implementation and the through process behind it.

4.3.1 nil

The first function we implemented for our doubly linked list was nil. Here is the function below:

```
--nil function
val nil =
    let val a = new [] in
    a := "NULL";
    a ;;
```

We borrowed this function from the linked list file provided to us from assignment 3. This function has no inputs, meaning that it executes at compile time. This function allocates data in the heap that holds the value "NULL". This is a crucial part of the code because the front and back nodes of the list will point to null.

4.3.2 newDNode

The next function that we wrote was called newDNode. This function takes in an element and creates a doubly linked list node. Here is the implementation:

```
--newDNode function
val newDNode = \e.
    let val x = new [] in
    x := [nil, e, nil];
    x;;
```

This function uses the heap allocation and the ":=" assignment to create a new node with element e and set both the forward and backward pointers to point to nil. This function will be important during insert because in order to insert an element into the doubly linked list, it first has to be inserted into a node with forward and backward pointers.

4.3.3 newDList

The next function we implemented was newDList. Here is the code:

```
--newDList function
val newDList = \e.
    let val x = newDNode e in
    x;;
```

This function takes in an element and uses the newDNode function to create the first node of a new doubly linked list. We thought it was important to utilize both a newDNode function and a newDList function because they would serve different purposes. Our vision for the newDList function was for it to be the way the user created a new doubly linked list. We imagined that the newDNode function would act as an abstracted back end function to utilize in definitions such as insert or newDList.

4.3.4 cons

The next function we have is cons. Here is the implementation:

```
--cons function
val cons = \e. \a.
    let val b = newDList e in
    b := [nil, e, a];
    a := [b, head(tail (!a)), nil];
    a;
    b ;;
```

This function was included in the linked list file with assignment 3. It does not serve a huge purpose in our implementation but we included it for testing reasons. It was a way to test some of our functions on lists longer than 1 element before we wrote our insert function. It is slightly altered from the cons function in the linked list file because we had to allow for the backwards pointer.

4.3.5 hd

The next function we implemented was hd. Here is the code:

```
--hd function
val hd = \a.
    case !a of {
        "NULL"-> "tried to take the head of an empty list",
        [b,e,a'] -> b
    } ;;
```

This function is technically a "back" function that we decided to call hd. We named it this because the function returns the head of the node, which is the backwards pointer. For example if we called hd on a node that was [nil,6,1] it would return nil. We knew this function would be important for insert and delete.

4.3.6 get

The next function we implemented was get:

```
--get function
val get = \a.
    case !a of { [b,e,a] -> e};;
```

We knew that this function would come in handy for the insert and delete implementations. It returns the element of any given node from the doubly linked list.

4.3.7 tl

The next function we created was tl:

```
--tl function
val tl = \a.
    case !a of {
        "NULL" -> "tried to take the tail of an empty list",
        [b,e,a'] -> a'
    } ;;
```

This function looks very similar to hd. Instead of returning the backwards pointer for any given node, it returns the forwards pointer. This function is similar to a next function. We utilize this in the insert and delete implementations.

4.3.8 insertBack

Here is the insert function we created:

```
--insertBack function
val insertBack = \e. \a.
    let val x = newDNode e in
    let val nodePtr = new [] in
    let val currNodePtr = new [] in
    currNodePtr := a;
    while (!currNodePtr) != nil do
        (nodePtr := (!currNodePtr);
        currNodePtr := tl(!currNodePtr));
    x := [!nodePtr, e, nil];
    !nodePtr := [hd (!nodePtr), get(!nodePtr), x];
    x;
    nodePtr;;
```

This function takes in a list and an element, makes that element into a node, and inserts the node at the back of the list. This was somewhat difficult to code because we had to ensure that we were inserting after the last node.There is a chance that the current node of the input list a is either a node at the front or the middle of the doubly linked list. We implemented a while loop in order to ensure that the insert was taking place in the right location. The while loop iterates towards the back of the doubly linked list until it hits a null pointer. From there, we are able to reassign pointers to insert the new node.

4.3.9 deleteBack

Here is the delete function we created:

```
--deleteBack function
val deleteBack = \a.
    let val nodePtr = new [] in
    let val currNodePtr = new [] in
    currNodePtr := a;
    while (!currNodePtr) != nil do
        (nodePtr := (!currNodePtr);
        currNodePtr := tl(!currNodePtr));
    hd !nodePtr := [hd(hd (!nodePtr)), get(hd(!nodePtr)), nil];;
```

This function deletes the last node of the doubly linked list. It works in a similar fashion to insert back, using a while loop to iterate until we reach the last node of the list. From there, we reassign pointers to "delete" the last node. However, deleting with doubly linked lists does not imply that the last node gets erased. Instead the second to last node's forward pointer merely changes to null, meaning that the last node can no longer be accessed by the doubly linked list. From there, garbage collection takes care of the wasted allocated data taken up by the "deleted" last node.

4.4 Running our Program

To run our program open the terminal and navigate into the folder labeled "test". To start the program run "stack exec LamFun". Then run ":setLang LamMem" to ensure you are running on the right language. Next, load the doubly linked list file by running ":load doubly-linked-list.lc". Now you are ready to create your list. Type "val c = newDList 5;;". This will create a new doubly linked list with a pointer c that points to the first element of the list which is a node that contains the value 5. Next run "insertBack 6 c;;". This command inserts a node with the value 6 to the back of the doubly linked list at address c. Finally, test the delete function by running "deleteBack c;;". This command will delete the last node of c and you should be left with a single node doubly linked list. Feel free to play around with the implementation.

4.5 Roadblocks/ Future work

There were a few challenges Jessica and I encountered while completing this mini project. First, we were not able to implement insertFront and deleteFront functions. The reason was that we were unsure about how to reassign the pointer in the stack that points to the first node in the list in the case that the first node is deleted or changed. We attempted to create an insertFront that reassigns the pointer but unfortunately since memory on the stack is immutable this attempt was not a success. Although at the current moment Jessica and I are unsure about how to solve this issue, we believe that if given more time with this project we would be able to come to a solution.

The potential we see for future work includes the implementation of a successful insertFront and deleteFront. If given more time we would also be interested in pursuing the implementation of even more complicated doubly linked list functions. For example, a delete by element function could prove to be extremely helpful and we even have current ideas on how we might create such a function. We would use a while loop to iterate through the doubly linked list, using "get" to check the value of each node against the target value until we either reach the end of the list or reach the first instance of the target value for deletion. In addition, we would choose to incorporate a head and tail function that would return either the first or last node of the doubly linked list respectively. Lastly, we would add a count function that would return the number of nodes currently in the doubly linked list.

5 Conclusions

As a final note, learning the basics of a functional programming language has been engaging and challenging but not without it's rewards. Seeing our mini project come together, gave me the confidence to know that I can use Haskell as a tool to code complex and useful programs. Haskell taught me to become comfortable with recursion which is a skill that I suspect will carry over to my work in imperative programming. Diving into abstract reduction systems gave me greater knowledge of the mathematical theory behind every programming language and these tools seem applicable to so much more than just building languages. I have enjoyed my time working with Haskell and I plan to continue practicing my functional programming skills in the future.

References

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

[LYAH] Learn You A Haskell

[LE] Lazy Evaluation

[LLMS] List of LaTeX Mathematical Symbols

[R] Rewriting

[F] Function

[TS] Type-Signature

[TTC] Types and Type Classes

[HPM] Haskell Pattern Matching

[URIH] Understanding Recursion in Haskell

[FVIJ] Final vs Immutability in Java

[IP] Imperative Programming

[HB] Haskell Basics