CPSC-354 Report

Sharon Chang
Chapman University

December 18, 2022

Abstract

The following report documents how I learned the various concepts needed to understand how functional and imperative programming languages work. I first learned about recursion, then Lambda calculus and abstract reduction trees. These concepts built on one another in order to help me understand the computer logic behind programming languages by the end of the semester. This report also culminates in a final project where I explore various methods of implementing infinite lists in Haskell and discuss the various pros and cons associated with each approach.

Contents

1 Introduction

In order to work as an effective programmer, it is best to understand the minutia working in the shadows that allows your code to work. In this interest, I was taught various concepts core to the understanding of how computer logic parses different types of programming languages. As a culmination of what I learned, I also created a final project centered around the Haskell programming language that I had been using throughout the semester in my coursework.

2 Homework

2.1 Week 1

In order to familiarize myself with LaTex, I was instructed to create a program to calculate the greatest common denominator of any two numbers in any programming language of my choice. I chose to write my GCD program in Python.

```python
def numInput():
    while True:
        numStr = input("Enter a number: ")
        try:
            num = int(numStr)
            if num > 0:
                return num
            else:
                print("The number must be greater than 0.")
                continue
        except:
            print("Invalid input.")

def gcd(numA, numB):
    if (numA > numB):
        result = gcd(numA-numB, numB)
    elif (numA < numB):
        result = gcd(numA, numB-numA)
    else:
        result = numA
    return result

# Main
numA = numInput()
print("First number acquired.")
numB = numInput()
print("Second number acquired.")

result = gcd(numA, numB)
print("The GCD of " + str(numA) + " and " + str(numB) + " is: " + str(result) + ".")
```

The numInput function first gets inputs from the user and makes sure that they are valid inputs to perform the gcd function on. The while loop traps the user until they input a valid non-zero integer. This repeats twice to obtain two numbers.

After that, the two numbers are put into the gcd function. If number A is greater than number B, it recursively calls on the function again to find the GCD between the difference of A and B and number B. If number B is greater than number A, it recursively calls on the function again to find the GCD between number A and the difference of B and A. This repeats until the two numbers being compared are equal, after which the function returns the equal number. This equal number is the GCD of the two given numbers. This recursive function works because the method of subtraction will repeatedly lower the numbers until they inevitably equal out to yield a result.

2.2 Week 2

After familiarizing myself with LaTex, I then had to familiarize myself with Haskell. I created a series of recursive functions in Haskell which are listed below, followed by explanations for how they work.

```
select_evens [] = []
select_evens (x:xs) | length (x:xs) >= 2 = (head xs) : (select_evens (tail xs))
                    | length (x:xs) == 1 = []
                    | otherwise = x:xs

select_odds [] = []
select_odds (x:xs) | length (x:xs) >= 1 = x : (select_odds (tail xs))
                   | otherwise = x:xs

member i [] = False
member i (x:xs) | i == x = True
                | otherwise = member i xs

append [] [] = []
append xs [] = xs
append [] ys = ys
append xs ys = head xs : (append (tail xs) ys)

revert [] = []
revert xs = append (revert (tail xs)) (head xs : [])

less_equal [] [] = True
less_equal xs [] = False
less_equal [] ys = True
less_equal xs ys | head xs <= head ys = less_equal (tail xs) (tail ys)
                 | otherwise = False
```

For my select evens function, I set the base case to return an empty list. The recursive function has 3 cases. In the first case, provided the given list has a length of 2 or greater, the head of the list xs is taken as the first element of the list. This is the second element of the given list because x is the head of list x:xs. After that the tail of the list xs is put back into select evens, which removes the first two elements from the list x:xs. In the second case, if the length of the list is 1, it also returns an empty list. Otherwise, it just returns the given list.

```
select_evens [1,2,3,4,5] =
   2 : (select_evens [3,4,5]) =
   2 : 4 : (select_evens [5]) =
   2 : 4 : [] =
   [2,4]
```

My select odds function is similar to my select evens function, with a base case of an empty list. If the length of the given list is greater than or equal to 1, the head of list x:xs, which is the value x, is taken as the first element of the list. The tail is then passed back into select odds for the same reasons as the select evens function. If the length is too short, it returns the given list.

```
select_odds [1,2,3,4,5] =
   1 : (select_odds [3,4,5]) =
   1 : 3 : (select_odds [5]) =
   1 : 3 : 5 : [] =
   [1,3,5]
```

The member function returns a base case of False given any input with an empty list. When given a list, as long as the list is greater than or equal to 1, it checks to see if the given input value matches the head of the given list. If it matches, it returns true, otherwise, it passes the tail of the list back into the member function.

```
member 3 [1,2,3,4,5] =
   member 3 [2,3,4,5] =
   member 3 [3,4,5] =
   True
```

When given an empty set of lists, the append function returns an empty list. If only one of the given lists is empty, it returns the non-empty list. When given two non-empty lists, the head of the first list is taken as the first element of the list. It is attached to the resulting list from appending the tail of the first list with the whole of the second list. Eventually, this empties out the first list, which will cause the entirety of the second list to be appended onto the elements of the first list.

```
append [1,2,3] [4,5,6] =
   1 : (append [2,3] [4,5,6]) =
   1 : 2 : (append [3] [4,5,6]) =
   1 : 2 : 3 : (append [] [4,5,6]) =
   1 : 2 : 3 : [4,5,6] =
   [1,2,3,4,5,6]
```

The base case of the revert function returns an empty list when given an empty list. When given a non-empty list, the revert function uses the previous append function to create a new list where the head of the given list is placed at the back. The remaining list elements are passed back into the revert function, which will result in a list where the elements are in reversed order.

```
revert [1,2,3,4,5] =
   (append (revert [2,3,4,5])) : 1 : [] =
   (append (revert [2,3,4,5])) : [1] =
   (append (revert [3,4,5])) : 2 : [] : [1] =
   (append (revert [3,4,5])) : [2] : [1] =
   (append (revert [4,5])) : 3 : [] : [2] : [1] =
   (append (revert [4,5])) : [3] : [2] : [1] =
   (append (revert [5])) : 4 : [] : [3] : [2] : [1] =
   (append (revert [5])) : [4] : [3] : [2] : [1] =
   (append (revert [])) : 5 : [] : [4] : [3] : [2] : [1] =
   (append (revert [])) : [5] : [4] : [3] : [2] : [1] =
   [] : [5] : [4] : [3] : [2] : [1] =
   [5,4,3,2,1]
```

When the two lists given to the less equal function are both empty, the function returns a true. If the first list is non-empty but the second list is, the function returns false. If the second list is non-empty but the first one is, the function returns true. If both lists are non-empty lists, the head of both lists is compared to one another. If the head of the first list is less than or equal to the second list, the function continues by passing on the tails of both lists back into itself. Otherwise, if the head of the first list is greater than the head of the second list, the function stops and returns a false.

```
less_equal [1,1,4] [1,2,3] =
   less_equal [1,4] [2,3] =
   less_equal [4] [3] =
   False
```

2.3 Week 3

The hanoi function takes 3 number inputs, which are the number of disks, the starting location, and the end location, in that order. It recursively calls on itself until the number of disks is reduced to one, upon which the move function is called. The move function has 2 inputs, the starting location and the ending location. It only moves the topmost disk of that location.

The following demonstrates the hanoi and move functions as mathematical equations.

$$\text{hanoi}(1)(x)(y) = \text{move}(x)(y)$$

$$\begin{aligned}\text{hanoi}(n+1)(x)(y) &= \text{hanoi}(n)(x)(\text{other}(x)(y)) \\ &= \text{move}(x)(y) \\ &= \text{hanoi}(n)(\text{other}(x)(y))(y)\end{aligned}$$

The following code demonstrates the recursive steps used when executing a function to solve a Hanoi tower problem of 5 disks.

```
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
```

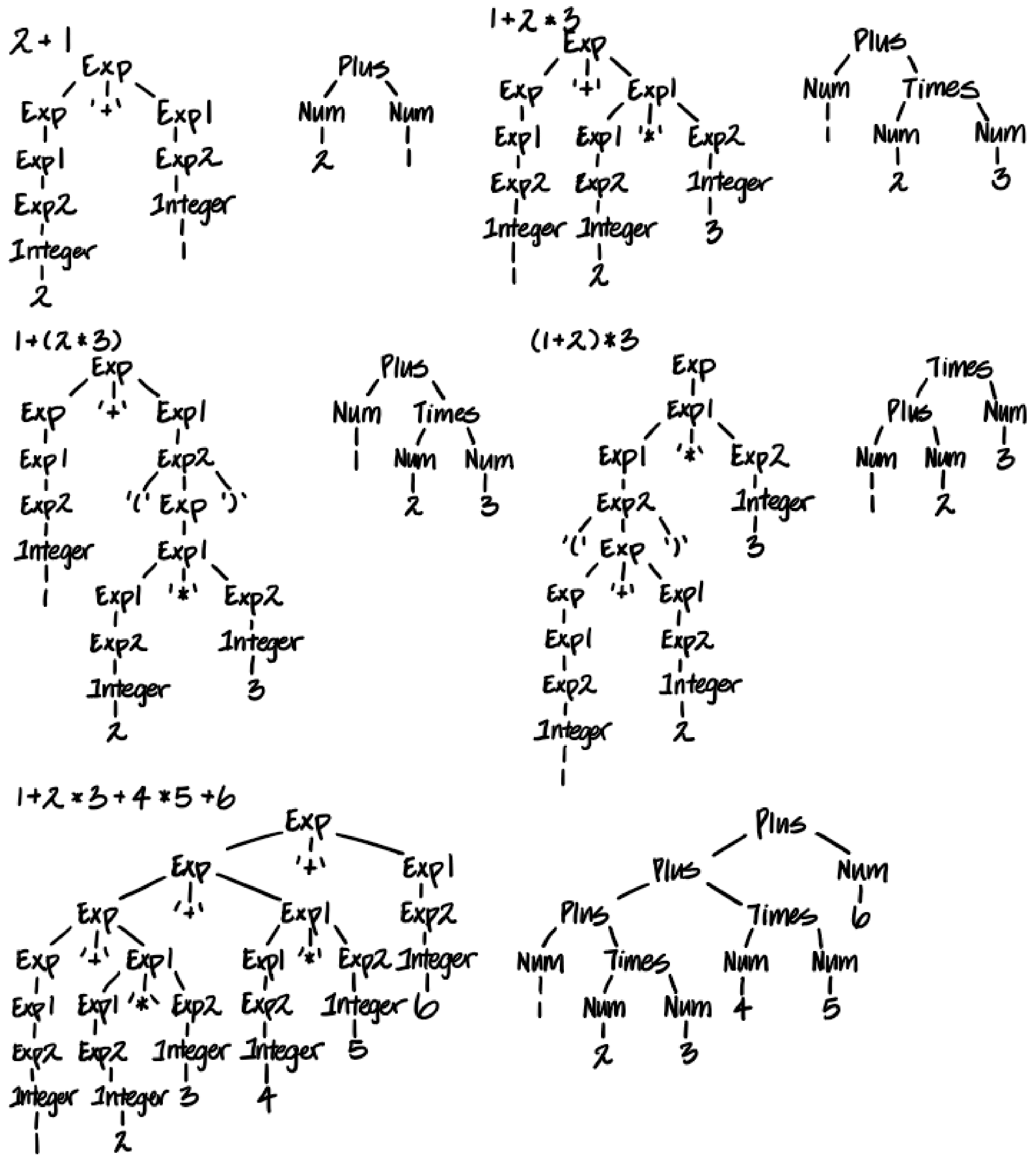
These instructions count the leftmost pole as the Pole 1, the middle pole as Pole 2, and the rightmost pole as Pole 3.

Move the top disk from Pole 1 to Pole 3. Next, move the next top disk from Pole 1 to Pole 2. Take the top disk from Pole 3 and move it to Pole 2. Then, move the top disk from Pole 1 to Pole 3. Follow by moving the top from Pole 2 to Pole 1. Then, move the top disk from Pole 2 to Pole 3. Afterwards, move the top disk from Pole 1 to Pole 3. Move the next top disk from Pole 1 to Pole 2. Next, take the top disk from Pole 3 and move it to Pole 2. Take the next top disk from Pole 3 and move it to Pole 1. Then, take the top disk from Pole 2 and move it to Pole 1. Follow by moving the top disk from Pole 3 to Pole 2. Next, move the top disk from Pole 1 to Pole 3. Then, move the next top disk from Pole 1 to Pole 2. Return to the top disk of Pole 3 and move it to Pole 2. Now, take the top disk of Pole 1 and move it to Pole 3. Next, take the top disk from Pole 2 and move it to Pole 1. Then, take the top disk from Pole 2 and move it to Pole 3. Follow by taking the top disk from Pole 1 and moving it to Pole 3. Return to Pole 2 and move its top disk to Pole 1. Now, take the top disk from Pole 3 and move it to Pole 2. Next, take the top disk from Pole 3 and move it to Pole 1. Then, take the top disk from Pole 2 and move it to Pole 1. Take the next top disk from Pole 2 and move it to Pole 3. Now, take the top disk from Pole 1 and move it to Pole 3. Return to the next top disk from Pole 1 and move it to Pole 2. Then, take the top disk from Pole 3 and move it to Pole 2. Next, take the top disk from Pole 1 and move it to Pole 3. Then, take the top disk from Pole 2 and move it to Pole 1. Move the next top disk from Pole 2 to Pole 3. Then, take the top disk from Pole 1 and move it to Pole 3. A 5-disk Tower of Hanoi has now been successfully moved from Pole 1 to Pole 3.

The word hanoi appears 31 times in the computation.

2.4 Week 4

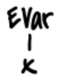
The following demonstrates the syntax trees for various arithmetic expressions. The expression being demonstrated is at the top right corner of each section, with two trees below it. The right tree is the concrete syntax tree, and the left is the abstract syntax tree.



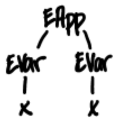
2.5 Week 5

There is a folder called "hw5" in another folder labelled "src" that can be found in the repository that this file is contained in. Using the grammar and interpreter contained within, I generated linearized abstract syntax trees for various Lambda expressions, then made 2D trees of them as seem below.

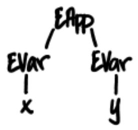
Expression: x
Abstract Syntax Tree: Prog (EVar (Id "x"))



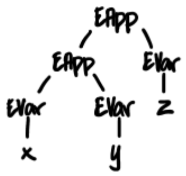
Expression: x x
Abstract Syntax Tree: Prog (EApp (EVar (Id "x")) (EVar (Id "x")))



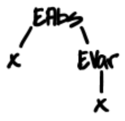
Expression: x y
Abstract Syntax Tree: Prog (EApp (EVar (Id "x")) (EVar (Id "y")))



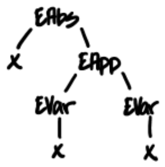
Expression: x y z
Abstract Syntax Tree: Prog (EApp (EApp (EVar (Id "x")) (EVar (Id "y"))) (EVar (Id "z")))



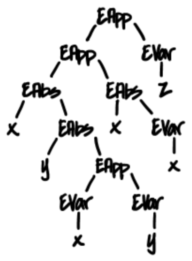
Expression: \ x.x
Abstract Syntax Tree: Prog (EAbs (Id "x") (EVar (Id "x")))



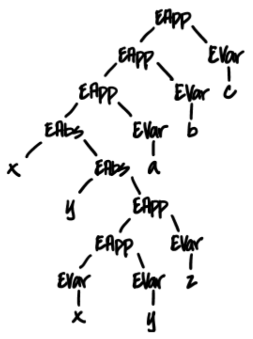
Expression: \ x.x x
Abstract Syntax Tree: Prog (EAbs (Id "x") (EApp (EVar (Id "x")) (EVar (Id "x"))))



Expression: (\ x . (\ y . x y)) (\ x.x) z
Abstract Syntax Tree: Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EVar (Id "x")) (EVar (Id "y"))))) (EAbs (Id "x") (EVar (Id "x")))) (EVar (Id "z")))



Expression: (\ x . \ y . x y z) a b c
Abstract Syntax Tree: Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EApp (EApp (EVar (Id "x")) (EVar (Id "y"))) (EVar (Id "z"))))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))



Additionally, the following includes examples of various lambda expressions evaluated by hand.

```
Expression: (\x.x) a =
a
```

```
Expression: \x.x a =
\x.x a
```

```
Expression: (\x.\y.x) a b =
(\y.a) b =
a
```

```
Expression: (\x.\y.y) a b =
(\y.y) b =
b
```

```
Expression: (\x.\y.x) a b c =
(\y.a) b c =
(a) c =
a c
```

```
Expression: (\x.\y.y) a b c =
(\y.y) b c =
(b) c =
b c
```

```
Expression: (\x.\y.x) a (b c) =
(\y.a) (b c) =
a
```

```
Expression: (\x.\y.y) a (b c) =
(\y.y) (b c) =
b c
```

```
Expression: (\x.\y.x) (a b) c =
(\y.(a b)) c =
(a b) =
a b
```

```
Expression: (\x.\y.y) (a b) c =
(\y.y) c =
c
```

```
Expression: (\x.\y.x) (a b c) =
(\y.(a b c)) =
\y.(a b c)
```

```
Expression: (\x.\y.y) (a b c) =
(\y.y) =
\y.y
```

Now, I will demonstrate how the interpreter from LambdaNat0 of Assignment 2 (also found in the same repository as this report) evaluates Lambda expressions via the evalCBN function. Each step is accompanied by the line number of the Interpreter.hs code that performs that step.

```
evalCBN ((\x.x)((\y.y)a)) =
evalCBN (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "a")))) = Line 13
evalCBN (subst (Id "x") (EAbs (Id "y") (EVar (Id "a"))) (EVar (Id "x"))) = Line 67
evalCBN (EAbs (Id "y") (EVar (Id "a"))) = Line 13
evalCBN (subst (Id "y") (EVar (Id "a")) (EVar (Id "y"))) = Line 64
evalCBN a = Line 50
a
```

2.6 Week 6

The following demonstrates the evaluation of the exponent expression 2^3.

```
(\exp . \two . \three . exp two three)
(\m.\n. m n)
(\f.\x. f (f x))
(\f.\x. f (f (f x)))
=
(\m.\n. m n) (\f.\x. f (f x)) (\f2.\x2. f2 (f2 (f2 x2)))
=
(\n. (\f.\x. f (f x)) n) (\f2.\x2. f2 (f2 (f2 x2)))
=
((\f.\x. f (f x)) (\f2.\x2. f2 (f2 (f2 x2))))
=
((\x. (\f2.\x2. f2 (f2 (f2 x2))) ((\f2.\x2. f2 (f2 (f2 x2))) x)))
=
((\x. (\x2. ((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2
    x2))) x) x2)))))
=
(\x. (\x2. ((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2
    x2))) x) x2))))
=
(\x. (\x2. ((\x2. x (x (x x2)))) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x)
    x2))))
=
(\x. (\x2. (\x2. x (x (x x2))) (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x)
    x2))))
=
(\x. (\x2. (x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
=
(\x. (\x2. (x (x (x (((\x2. x (x (x x2)))) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
=
(\x. (\x2. (x (x (x ((\x2. x (x (x x2))) (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))
=
(\x. (\x2. (x (x (x ((x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) x2))))))))))
=
(\x. (\x2. (x (x (x (x (x (x (((\f2.\x2. f2 (f2 (f2 x2))) x) x2)))))))))
=
(\x. (\x2. (x (x (x (x (x (x (((\x2. x2 (x2 (x2 x2))) x))))))))))
=
(\x. (\x2. (x (x (x (x (x (x (((x (x (x x)))))))))))))
=
(\x. (\x2. (x (x (x (x (x (x (x (x (x x))))))))))
```

2.7 Week 7

Returning to the evalCBN function in LambdaNat0 of Assignment 2, I will now examine the variables of that function.

In line 5 the variables e1 and e2 are bound on the left of the equal sign with their scope being the body of the evalCBN function (through the end of line 7). The variables i and e3 in line 6 are bound on the left of the arrow sign and their scope is to the end of that line. Similarly, in line 7, e3 is bound to the left of the arrow sign and the scope is also to the end of that line. In line 8, x is then bound to the left of the equal sign and its scope is the end of line 8.

In line 18 the variables id, id1, e1, and s are bound on the left of the equal sign with the scope being body of the subst function (through the end of line 22). The variable f in line 20 is also bound to the left of its equal sign. The variable e2 in line 21 is also bound to the left of its equal sign. Both f and e2 have scopes reaching the end of line 22.

The following demonstrates an example of the the evalCBN function at work using abstract syntax in the same manner as demonstrated in the section Week 5. The line numbers refer to Interpreter-fragment.hs within LambdaNat0 this time.

```
evalCBN ((\x.\y.x) y z) =
evalCBN ((EAbs (Id "x") (EAbs (Id "y") EVar (Id "x"))) EVar (Id "y") EVar (Id "z")) = Line 6
evalCBN (subst (Id "x") (EVar (Id "y")) (EAbs (Id "y") EVar (Id "x")) = Line 18
evalCBN (EAbs (Id "a") (EVar (Id "y"))) = Line 21
evalCBN ((EAbs (Id "a") (EVar (Id "y"))) (EVar (Id "z"))) = Line 22
evalCBN (subst (EVar (Id "a")) (EVar (Id "z")) (EVar (Id "y))) = Line 6
evalCBN (EVar (Id "z")) = Line 15
z
```

The following demonstrates the various properties of abstract reduction sequences.

1. A = {}

   Terminating, confluent, has unique normal forms

   No elements given for R, therefore this is a tree consisting of empty elements that automatically satisfy all conditions.

2. A = {a} & R = {}

   Terminating, confluent, has unique normal forms

   R is empty, therefore this is a tree consisting of empty elements that automatically satisfy all contitions.

3. A = {a} & R = {(a, a)}

   Non-terminating, confluent, no unique normal forms

   $$\begin{tikzcd} a \arrow[loop right] \end{tikzcd}$$

4. A = {a, b, c} & R = {(a, b), (a, c)}

   Terminating, non-confluent, no unique normal forms

   $$\begin{tikzcd} & b \\ a \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

5. A = {a, b, c} & R = {(a, b), (a, c)}

   Terminating, non-confluent, no unique normal forms

$$\begin{tikzcd} a \arrow[loop above] \arrow[r] & b \end{tikzcd}$$

6. A = {a, b, c} & R = {(a, b), (b, b), (a, c)}

   Non-terminating, non-confluent, no unique normal forms

$$\begin{tikzcd} & b \arrow[loop right] \\ a \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

7. A = {a, b, c} & R = {(a, b), (b, b), (a, c), (c, c)}

   Non-terminating, non-confluent, no unique normal forms

$$\begin{tikzcd} & b \arrow[loop right] \\ a \arrow[ur] \arrow[dr] & \\ & c \arrow[loop right] \end{tikzcd}$$

Here are examples fulfilling every combination of ARS properties, where possible.

1. Confluent, terminating, has unique normal forms

$$\begin{tikzcd} & b \arrow[dr] & \\ a \arrow[ur] \arrow[dr] & & d \\ & c \arrow[ur] & \end{tikzcd}$$

2. Confluent, terminating, no unique normal forms

   Does not exist. A confluent and terminating ARS must reach a point of reduction on account of the termination needing an endpoint and the confluence inevitably reducing an element. Therefore, such a tree must always have unique normal forms.

3. Confluent, non-terminating, has unique normal forms

$$\begin{tikzcd} & b \arrow[dr] & \\ a \arrow[loop above] \arrow[ur] \arrow[dr] & & d \\ & c \arrow[ur] & \end{tikzcd}$$

4. Confluent, non-terminating, no unique normal forms

$$\begin{tikzcd} & b \arrow[dl, bend right] \\ a \arrow[ur] \arrow[dr] & \\ & c \arrow[ul, bend left] \end{tikzcd}$$

5. Non-confluent, terminating, has unique normal forms

   Does not exist. A terminating ARS with unique normal forms must have a point of divergence for the unique forms, but must come together to a point of termination in order to be terminating. Therefore, such an ARS must always be confluent.

6. Non-confluent, terminating, no unique normal forms

$$\begin{tikzcd} & b \\ a \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

7. Non-confluent, non-terminating, has unique normal forms

   Does not exist. An ARS that is non-confluent and non-terminating cannot maintain all elements in a reduced state, and must inevitably have some elements with share normal forms. This means that such an ARS cannot have unique normal forms.

8. Non-confluent, non-terminating, no unique normal forms

$$\begin{tikzcd} & b \\ a \arrow[loop above] \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

2.8 Week 8

The following section is an analysis on the ARS system seen below.

```
aa -> a
bb -> b
ba -> ab
ab -> ba
```

This ARS does not terminate on account of the last two rules. These rules can infinitely loop into each other, forming the following computation.

```
ba -> ab -> ba -> ...
```

This ARS has 3 normal forms where the computation can no longer be reduced. These forms are a, b, and [] (empty list).

In order to modify the ARS to make it terminate, I need to introduce a fourth normal form. By editing the ARS to the form below, the ARS now terminates and has the unique normal forms a, b, [], and ab.

```
aa -> a
bb -> b
ba -> ab
ab -> ab
```

This modified ARS now terminates and breaks the loop created by the original ruleset. This maintains the equivalency relationship from the original rules as well, as seen below.

```
Original:
ba -> ab -> ...
ab -> ba -> ab -> ...

Modified:
ba -> ab -> ab
```

Looking at the 4 normal forms a, b, [], and ab, I can find specification being implemented by the ARS. The normal form a has the invariant that there is at least one a in the input. The normal form b has the same invariant, but for there being at least one b. The empty string results from there being no a or b. Then, there is the normal form ab, which results from the combination of either ba or ab. That is, the invariant that there is at least one a and b. These invariants communicate that this ruleset decides whether the input is empty, has ”a”s but no ”b”s, ”b”s but no ”a”s, or has both ”a”s and ”b”s.

2.9 Week 9

For my final project, the two major parts will be the list generating functions and the list arithmetic functions. My milestones will be as follows:

November 13th: Complete the 4 list generating functions (triangle, cube, square, and fibonacci)

December 4th: Complete the list manipulation functions, but at this point they will only work with lists of the same length

December 11th: Figure out how to make lists of different lengths compatible in my arithmetic functions.

This schedule will give me all of finals week to do any final edits that need to be made and complete the report details.

I will now analyze the following ARS.

```
ba -> ab
ab -> ba
ac -> ca
ca -> ac
bc -> cb
cb -> bc

aa -> b
ab -> c
ac ->
bb ->
cb -> a
cc -> b
```

By drawing the rules out, the properties of the ARS become apparent.

$$\begin{tikzcd}
ba \arrow[r, bend left] & ab \arrow[l, bend left] \arrow[r] & c & ca \arrow[r, bend left] & ac \arrow[l, bend left] \arrow[r] & {} & bc \arrow[r, bend left] & cb \arrow[l, bend left] \arrow[r] & a \\
aa \arrow[r] & b & bb \arrow[r] & {} & cc \arrow[r] & b
\end{tikzcd}$$

This ARS is non-terminating because the loops formed by ab, ba, ac, ca, bc, and cb can run forever. It is also non-confluent as there are no visible peaks or valleys formed by connecting the rules together. The ARS also does not have unique normal forms. The normal forms here are c, [] (empty), a, and b, but the normal forms [] and b are both shared between multiple rules.

The invariants of this ARS are:

- Any pair of letters with the letter b and a different letter will evaluate into the missing third letter
- Any pair of two of the same letter that is not the letter b will evalate into the letter b
- Any exceptions to the above two rules will evaluate into empty

2.10 Week 10

The following code demonstrates the computation steps for the factorial function $fix_F 2$.

$$fix_F 2 = \text{(def of fix)}$$

$$F(fix_F)2 = \text{(def of F)}$$

$$(\Lambda \text{ n. if n} == 0 \text{ then } 1 \text{ else } fix_F(n-1)_n)2 = \text{(beta reduction)}$$

$$(\text{if } 2 == 0 \text{ then } 1 \text{ else } fix_f(n-1)_n)2 = \text{(if then else)}$$

$$(fix_F 1) * 2 = \text{(def of F)}$$

$$(\Lambda \text{ n. if n} == 0 \text{ then } 1 \text{ else } fix_F(n-1)_n)2 = \text{(beta reduction)}$$

$$(\text{if } 1 == 0 \text{ then } 1 \text{ else } fix_F(1-1)_n)2 = \text{(if then else)}$$

$$(fix_F 0) * 2 = \text{(def of F)}$$

$$(\Lambda \text{ n. if n} == 0 \text{ then } 1 \text{ else } fix_F(n-1)_n)2 = \text{(beta reduction)}$$

$$(\text{if } 0 == 0 \text{ then } 1 \text{ else } fix_F(n-1)_n)2 = \text{(if then else)}$$

$$1 * 2 = \text{arithmetic}$$

$$2$$

2.11 Week 11

The following short essay discusses this paper on a prototype for a programming language based on writing and implementing contracts.

A tailor-made language for writing and executing contracts would open the door to creating specialized contract creation tools. Similar to specialized languages, such as SQL being a database language, it helps to have a specialized tool one can use that is built to handle a specific purpose. Even if it is not perfect, it could still be used to handle common, simple contracts. Having a preexisting groundwork also helps to build more complicated projects on top of it later on. Additionally, because current contracts are lengthy pieces of writing, to make a program that could successfully replace that could perhaps be a step towards better natural language processing.

However, while I feel that a specialized contract language could be useful, computers would likely be severely limited in the types of contracts they can interpret. Contracts that rely on hard concepts such as "payment of X amount by Y date" could be easily run through a computer. However, when accounting for human error, computer-run contract systems must be extremely complex to keep up. Payment contracts, such as rent, are not often subject to late or partial payments. A human administrator would be able to provide the necessary leeway for such circumstances, but a computer is unlikely to care. One could use a program to retroactively inform the computer that an unwritten change was made to the contract, but with how common such scenarios are, this might be introducing unnecessary complexity to the situation.

Similarly, contracts that rely on abstract concepts, such as NDAs, would be difficult to administer via a computer program. A computer would be unable to understand the concept of information leaking and would be unreliable in enforcing an NDA as a contract. In the same vein, many contracts rely on multiple conditions that can overlap and affect other conditions, ultimately affecting the outcome. Lawyers are specially trained to learn how to parse contracts, which are often beyond the scope of understanding for normal people. Contracts of such complexity, which are already a challenge for human understanding, would pose a great obstacle for a computer to parse via a programming language. One would have to conduct an order of operations for conditions that are flexible enough to work with a wide range of contracts.

However, because of this complexity issue, should this project succeed, it would open the possibility for contract parsing to become more open to the general public. Not everyone is a lawyer, and not everyone has one on hand. Given the prolific nature of contracts in the online world, from agreeing to be subject to tracking cookies or signing up for a new website, having a computer program that could parse any given contract and reduce it to understandable terms would be very useful. Making the language of contracts more accessible could also change how lawyers and contract writers approach writing contracts, as the parser would make it impossible for them to obfuscate their intentions through complicated language. This development could force transparency upon these institutions and, in turn, generate more fair contracts for the general population.

2.12 Week 12

The following section demonstrates Hoare logic analysis as applied to the code below.

```
while (x!=0) do z:=z*y; x:= x-1 done
```

We know that the program must terminate because the condition of the while loop is x not equaling 0, which x will inevitably reach on account of it being decremented with each loop. Let us assume that the variables are initialized as x = 100, y = 2, and z = 1. We will treat the variable t as a count for how many times the loop has been executed. It follows that x + t must always equal 100, as x is decremented by 1 while t is incremented by 1 with each loop. The variable y is never edited during the loop, so it will always stay at 2. The variable z is multiplied by y with each loop, being initialized as 1 but increasing to 2, then 4, then 8, then 16, and so on and so forth until the loop completes itself. This pattern reflects the exponents of 2, therefore one can write the relationship between z and y as z = y^t. This gives us two equations to work with.

$$t + x = 100$$

$$z = y^t$$

Logically, we can put these equations to yield a single invariant equation.

$$z = y^{(100-x)}$$

Now that we have the invariant, we will move on to discuss the preconditions and postconditions of our example code. The precondition of this loop running is that x must equal some positive integer above 0 and z must equal 1. The equation presented above is our postcondition that results from the termination of our program. It is worth noting that the 100 in our above invariant equation is merely a stand-in for whatever value x is initialized with. This results in the following Hoare triple.

$$\{x \geq 0 \, \Lambda \, z = 1\} \text{ while } (x! = 0) \text{ do } z := z * y; \; x := x - 1 \text{ done } \{z = y^x\}$$

However, this only accounts for certain values of x and z. Taking the prior example initialization values of x = 100 and y = 2 but changing z to z = 5, we would see the pattern of 10, 20, 40, etc. One sees that this pattern reflects the earlier pattern except everything is now multiplied by 5. Therefore, it is better rewritten as follows to account for the variables having different values.

$$\{x = a \wedge y = b \wedge z = c\} \text{ while } (x! = 0) \text{ do } z := z * y;\ x := x - 1 \text{ done } \{z = c * b^a\}$$

The following lines demonstrate the proof tree demonstrates the above steps in Hoare logic:

$$\{z * y = c * y^{(a-(x-1))} \wedge y = b\}\ z := z * y\ \{z = c * y^{(a-(x-1))} \wedge y = b\}\ x := x - 1\ \{z = c * y^{(a-x)} \wedge y = b\}$$

$$\{z = c * y^{(a-(x))} \wedge y = b \wedge x \neq 0\}\ z := z * y;\ x := x - 1\ \{z = c * y^{(a-x)} \wedge y = b\}$$

$$\{z = c * y^{(a-x)} \wedge y = b\} \text{ while } (x! = 0) \text{ do } z := z * y;\ x := x - 1 \text{ done } \{x = 0 \wedge z = c * y^{(a-x)} \wedge y = b\}$$

$$\{x = a \wedge y = b \wedge z = c\} \text{ while } (x! = 0) \text{ do } z := z * y;\ x := x - 1 \text{ done } \{z = c * b^a\}$$

3 Project

The final project I have created goes into the variety of methods with which Haskell allows the user to create and manipulate infinite lists. These methods are demonstrated by implementing the same list generation and manipulation functions in different ways. These functions generate mathematical patterns as lists and also perform various forms of list manipulation between two lists.

3.1 Specification

My project specification changed radically in the last week of the project. The initial project will be discussed in the prototype section. This section will describe the project as it currently exists.

My goal was to produce 5 different Haskell files demonstrating 5 different methods of approaching infinite lists in Haskell. There are 5 files in my final project folder and each one is named after the infinite list method used for the functions.

- seqRec: Uses recursive functions.
- seqComp: Uses list comprehension.
- seqMap: Uses the built-in Haskell map function.
- seqZip: Uses the built-in Haskell zipwith function.
- seqScan: Uses the built-in Haskell scan function.

All 5 of these files contain list generation functions that generate a variety of mathematical patterns as lists. The list generation functions work as follows:

- Arithmetic: Takes in inputs dictating the length of the list, the starting number, and the number to be added to each previous number. Outputs the corresponding arithmetic number sequence.
- Geometric: Takes in inputs dictating the length of the list, the starting number, and the number to be multiplied against each previous number. Outputs the corresponding geometric number sequence.
- Triangle: Takes in an input dictating the length of the list. Generates a sequence of triangular numbers.
- Square: Takes in an input dictating the length of the list. Generates a sequence of square numbers.
- Cube: Takes in an input dictating the length of the list. Generates a sequence of cubed numbers.

- Fibonacci: Takes in an input dictating the length of the list. Generates a sequence of Fibonacci numbers.

Of the 5 files, 3 of them also contain list manipulation functions. seqZip and seqScan do not contain these functions on account of issues with implementation and time constraints which will be elaborated upon further later on. The list manipulation functions are as follows:

- List Filter: Takes in 2 list inputs. Removes all elements present in the second list from the first list and outputs the cleaned version of the first list.
- List Match: Takes in 2 list inputs. Outputs a list of all elements present in both lists.
- List Sort: Takes in 2 list inputs and a string. The string must say either "asc" or "desc" and is case sensitive. If the string is "asc", the two lists are combined and their elements sorted in ascending order. If the string is "desc" then they are sorted in descending order. Duplicate elements are removed.
- List Extension: Takes in 2 list inputs. If the two lists are of different lengths, it will output the shorter list with extra 0s added to the end in order to match its length with the longer list. Otherwise, both lists are output as is. Regardless, these two lists are output as elements of a tuple.
- List Arithmetic: Takes in 2 list inputs and an integer. The integer indicates which of 4 basic arithmetic methods to use (addition, subtraction, multiplication, or integer division). The output list is the result of performing the chosen arithmetic function between the two lists. In subtraction and division, the element of the first list will be reduced by its corresponding element in the second list. If dividing by 0, it returns a 0 for that element.

3.2 Prototype

My initial prototype consisted of a single file that implemented the various functions as I saw fit. All of the list generation functions were present in this prototype, but there were only 2 list manipulation functions: list extension and list arithmetic. List arithmetic was initially planned as 4 separate functions for each arithmetic operation, but upon noticing that I was reusing code between almost all of them, I decided to merge the 4 together into 1 function and use an integer input to differentiate between each input. The project was greatly expanded in order to better demonstrate the various methods of using infinite lists that I had found in Haskell.

This first prototype was largely built off of knowledge I had learned through class, or from basic infinite list introductions that I found online. One of these was the 2007 article on TechRepublic and the other was a 2019 blog post by Elias Hernandis. Upon attempting to implement them myself, I found myself frequently referring to Zvon and HaskellWiki in order to find the specifics of how various functions work, as well as find other related functions that I could perhaps put to use. In order to better demonstrate the information I learned from these sources, I expanded the initial prototype into my current 5-file setup to show various different methods of list implementation.

seqRec can be treated as the second prototype of the project, but the first protoype of the project as it currently exists, on account of it building off knowledge I had already learned from class. In doing this, I was able to map out the logic of the functions from a familiar approach before tackling them in new ways. Often I found myself reusing structures that I had previously learned in order to build functions using the following structure, such as how list comprehension became very useful for my later work with the built-in Haskell functions.

3.3 Documentation

3.3.1 List Comprehension

In Haskell, list comprehension is a built-in method of allowing one to construct complex lists directly through declaring a list, without needing extraneous functions. According to Nick Gibson, the writer of the TechRepublic article, such an approach saves money because the list and its items are built as it is declared, allowing it to construct infinite lists without needing infinite memory to store everything in. My use of list comprehension comes down to these two basic implementations.

```
[1..10] = [1,2,3,4,5,6,7,8,9,10] -- Linear Generation
[x | x <- [1..10]]       -- Input Generation
```

Linear generation was used for all of the mathematical sequences on account of it being very suited to handle mathematical patterns. The list comprehension implementation of Fibonacci was also the only one to use a where clause to indicate certain base case scenarios. This will be elaborated upon later when I write about the map function, but I thought it was interesting to note that all the other functions could be declared within a single line using either linear or input generation.

The use of input generation was better suited for the list manipulation functions where I wanted more complex reasoning for the construction of the lists. Of the list manipulation functions, listFilter and listMatch were particularly easy to implement by simply checking whether or not an element was present. This is also the only implementation where I was able to extend lists by simply tacking on a separate list consisting of the necessary amount of 0s. In both recursion and map implementations of the same function, I had to take very roundabout approaches that I personally disliked. My list arithmetic function also shows how complexity can be introduced into the input generation approach of list comprehension on both sides of the guard rail. This opens up the possibility for much more complex list manipulation.

The list comprehension feature of Haskell is both useful and flexible, proving itself to be capable of handling all the functions with relative ease.

However, despite the input generation implementation being very effective at building list manipulation functions, I also found it strongly lacking on account of limitations with its ability to parse individual elements. This is most obvious looking at my implementation of the listSort function. I could not find a method of parsing each individual element and deeming whether or not it belong in the output. Therefore, I ended up taking the much clunkier approach of generating a massive list running from the minimum of one list to the maximum of the other, and then using my listMatch function to remove all the extraneous values. This is obviously a terribly inefficient way of approaching the function, but when limited to the list comprehension approach, this was the best that I could come up with.

3.3.2 Haskell map Function

On account of being a function, when building the project with the map function I frequently found myself having to declare lists as part of the input instead of simply building one to my specifications, as seen in the example below.

```
func l = map func [1..l]
  where func 0 = 0
        func 1 = 1
        func n = n
```

All of my mathematical sequences involve me declaring a generic list using list comprehension and using the map function to modify it to fit my desired output. This was quite useful, and I was very glad that I first built out the list comprehension implementations and learned that concept prior to moving onto this one. I found it interesting that for every map function, I had to individually declare certain outputs with where clauses, whereas in my list comprehension implementation, only the Fibonacci function needed a where clause on account of its unique starting sequence. While this is objectively less efficient than using list comprehension, which accomplished the same things in singular lines while using less memory, I found it interesting in how it reflected mathematical equational reasoning that many of these sequences are traditionally written with.

However, I found it interesting that this approach was generally much better at list manipulation than list comprehension, especially when combined with the filter and flip functions. Additionally, these map functions relied heavily on recursion, with only extension and length not using them. However, unlike the recursive equvalents of these functions, the map functions were much easier to read and I did not need to build out every base case for them.

Additionally, while building out the extension and arithmetic functions with the map implementation, I found that it was very useful that I could build the map function in a manner that strong resembled Lambda calculus.

```
func y = map (\x -> y) [0..10] = [y, y, y, y, y, y, y, y, y, y]
```

This approach allowed me to use conditionals and perform operations in the extension and arithmetic functions in a way that reflected material that we had learned in class.

3.3.3 Haskell zipWith and scanl Functions

On account of zipWith and scanl functioning in similar ways that led to similar roadblocks, I have elected to speak of them together rather than give each a separate section because I feel I would find myself reiterating many points between the two. Both functions were exceptionally good at building the list generation functions, albeit relying on the generation of other lists in order for me to have the necessary inputs to execute them with.

The zipWith function works as in the following example.

```
func l = zipwith (+) [0..l] [0..l] = [0, 1, 2, ... l+l]
```

This is very useful for quickly performing simple operations between two lists, but on account of all my generation functions taking in integer inputs, I had to use list comprehension to build out these lists in order to achieve the desired output. I initially assumed that this would mean it would excel at list manipulation, but I encountered so many errors that it became too time consuming and I ultimately had to toss it. The explanation for why is further below in the Critical Appraisal.

Similarly, the scanl function works as follows:

```
func l = scanl (+) 2 [0..l] = [2, 4, 6, ... l+2]
```

This is also a very efficient method of performing a simple list operation upon a list, except it runs into the similar problem of my functions not providing any lists on input, therefore requiring me to construct lists for the scanl function to work with. Unlike the zipWith function, I was under no illusion that this could perform well with the list manipulation functions, and ultimately those got scrapped for similar reasons as the list manipulations of the zipWith implementation.

3.3.4 Other Functions and Features

Here I will detail various other interesting built-in Haskell functions and items that I encountered while working on this project. Some things I learned, while new, were not quite interesting enough to talk about at length, and therefor have been excluded from this list.

- Tuples: I encountered tuples as a solution to the list extension function needing to output 2 lists. I initially thought I could just output the fixed list, but then that meant that an input of 2 lists with matching lengths would have nothing to output. Additionally, there would be no way for any following functions to determine which of the two input lists needs to be replaced with the returning output list. This led me to the concept of tuples in Haskell, which would allow me to return my two lists and then separate them with the fst and snd functions. Unlike lists, tuples are denoted by parentheses, e.g., (a, b, c). Tuples are able to store different data types, unlike lists which can only work with whatever singluar data type they have been declared with. However, they are also fixed in length, unlike the infinite lists of Haskell. While this means that we can reliably use tuples in our Haskell programs without being concerned about variability in its size, this also means that it lacks the flexibility in storing variable amounts of information that a list has.

- Filter: The filter function works similar to my listFilter function except instead of taking 2 lists as inputs, it takes a condition and 1 list. All elements satisfying the condition are returned. This was very useful in removing extraneous elements from my lists when building my map implementations where I found myself sometimes forced to use roundabout approaches to building certain list manipulation functions.

- Flip: Filp is a very unique function on account of it working somewhat strangely in comparison to conventional functions. It takes in a function and arguments, then returns the result of the function with the reverse order of arguments. While this does not appear significant, it was greatly helpful in reducing the map implementations of listFilter and listMatch down to singluar lines by flipping the functions elem and notElem. These uses would output lists of elements that the filter function could then remove from those lists and produce the desired result. My use of the flip function often occurred paired alongside the filter function and with the use of recursion, as demonstrated below.

```
func as bs = map func (filter (flip elem bs) as) where func x = x
```

- Iterate: Iterate is a built-in infinite list function that will generate an infinite list according to the pattern that was input to it. This was very useful in generating lists to for the calculations needed to perform my mathematical sequences with the zipWith function, on account of zipWith automatically outputting a list to the length of the shorter input list. This meant that iterate's infinite output would not affect my outputs. The patterns generated by iterate can be as simple as addition to resembling the Lambda calculus inputs of other functions, as seen in the following example adapted from the Zvon page about the iterate function:

```
iterate (\x -> (x+3)*2) = [1, 8, 22, 50, 106, ...]
```

- Replicate: Replicate is a list generating function that generates a list as long as the first input and consisting only of elements matching the second input. This was useful for generating dummy lists to give zipWith and scanl to work with. These dummy lists would have the desired length and be populated with the same value throughout, as seen below.

```
replicate 5 0 = [0, 0, 0, 0, 0]
```

- Repeat: Repeat functions similar to replicate, except it generates an infinite list and does not take a length input. This was used instead of replicate in places where the generated list length did not matter.

```
repeat 0 = [0, 0, 0, ...]
```

- Take: Take uses 2 inputs, one indicating list length, and the other being a list. It takes the first x amount of values from the list, according to the indicated length, and outputs that. I used this in scenarios where I only needed a section of a list, or in situations where the list would otherwise print forever (as occurred when writing the scanl application of the Fibonacci sequence.

```
take 5 (iterate (+1) 0) = [0, 1, 2, 3, 4]
```

3.4 Critical Appraisal

There are no known bugs in my final project, however it is worth explaining why seqZip and seqScan had all of the list manipulation functions removed.

Both seqZip and seqScan are missing half the functions present in the other 3 files; all of the list manipulation functions. This is on account of having great difficulty wrangling the two functions to do as I desired. Both zipWith and scanl are built to take in 3 inputs, one of which is a function to be performed between the other two functions. I found that it was very difficult to make these produce filtered or sorted lists on account of those being boolean comparisons that are meant to entirely remove the irrevlevant elements. Instead, I found that if I were to implement these functions, I would need to essentially merge my two input lists together and then insert a placeholder value to mark elements that I want to remove, and then scrub my list of those placeholder values. This could only work if I had a good placeholder value to work with. In the case of listFilter, this was simple, I could simply insert a value from the second list and then scrub those out because they are supposed to disappear anyways. However, when building listMatch, I realized that there was no practical way to recreate this. I would need to find a way to always designate a placeholder value that is not in either of the two input lists, and given the variability of my input lists, I could not come up with an appropriate solution to this problem. I wanted to put in some sort of null value, only to discover that Haskell does not have that capability, so I ultimately scrapped those functions. Sort was scrapped for similar implementation issues, where I could not come up with a good formulaic approach to implementing listSort using those two functions without ending up creating more problems for myself. listExtension could not work on account of zipWith consistently reducing to the smaller list, so I attempted creating longer dummy lists and zipping those, only to encounter errors at compile time. Similarly, I ran into issues attempting to implement a conditional when builting the arithmetic functions for the divide-by-0 case, and with limited time at my hands, I decided it was best to scrap the entire list manipulation section of both seqZip and seqScan.

3.4.1 Observations

Overall, I feel that list comprehension was the best general approach to the mathematical sequence functions, on account of taking up very little space both visually in the code, but also by not requiring me to conjure up additional lists just to reach my desired output as I did with the map, zipWith, and scanl functions. However, I feel like the map function was best suited to performing the list manipulations. zipWith and scanl seemed very interesting at first, but when put into implementation, I feel that they are best used for more complex sequences when necessary, and, at least at the scale of my project, proved to be not that much better than general list comprehension.

I was very surprised by how good it felt to work with list comprehension in Haskell, as it initially seemed like it might be very limited. However, its ability to perform both traditional equational reasoning and also evaluate expressions similar to Lambda calculus made it feel like a good general-use tool when generating and manipulating lists in Haskell.

3.4.2 Questions

I wonder if there could still be some way to accomplish the list manipulation functions in Haskell using the zipWith and scanl functions. I found rather hamfisted approaches to them, and while those approaches ultimately didn't work, I did not feel as if they were an impossible task. Rather, I felt that if I had approached the project from this approach sooner and given myself more time, I could perhaps reach a solution to those problems.

Additionally, Haskell's library of list manipulation functions and the existence of tuples in Haskell have left me to wonder about how it holds up in comparison to other programming languages and how they handle lists. What are the benefits and drawbacks of these different approaches to data storage between programming language? Why does Haskell offer so much support for lists?

3.4.3 Further Comments

The variety of list functions I found in Haskell as well as the flexibility of Haskell's list comprehension have left me wondering what more could be done with lists in Haskell. I can see how these could be greatly useful and the potential these functions have for higher-complexity work involving lists, but I myself am at a loss for possible ideas and implementations of these tools.

4 Conclusions

Overall, the course touched on subjects that are strongly present throughout programming and helped build a strong logical foundation for students to become better programmers.

Recursion is such an important aspect of programming that it is taught early in the coursework for the Computer Science major and enforced throughout. Implementing recursion equationally through Haskell helped to visualize recursion from a new lens and led me to view recursive programming from a more mathematical standpoint than a purely logical one. Practicing recursive programming through Lambda calculus further reinforced this new understanding of programming as a mathematical exercise. While initially confusing, it felt very intuitive after grasping the basic logical rules at play.

Using grammar and trees throughout the course was also very helpful for learning how computer logic works. The trees were especially useful in giving a strong visual for how something as basic as arithmetic operations has many steps to go through for a computer to understand and parse it. Combining that with our brief interaction with abstract reduction trees helped to understand better how a computer breaks down these elements.

I had previously encountered these concepts during NLP, but spending more time on them in this course helped me understand them better. While I previously walked away from NLP because I did not understand how to construct grammar and trees, my experience in this course has made me better grasp those concepts that previously confused me.

Aside from NLP, I feel that understanding these basic logical building blocks is greatly helpful to understanding programming at large. While it is understood that computers cannot reason the same way people do, it is hard to truly grasp this concept unless you are forced to confront it head-on. While I was frustrated by the redundancy of many steps and how it felt like many simple things were being drawn out to the point of being unnecessarily complicated, I also understood that these frustrations highlight the difference between a person and a computer. The computer has no sense of intuition and must perform complex logical reasoning to grasp even the most basic tasks. Therefore, as a programmer, it is imperative for me to understand what steps the computer is going through to write efficient code for it. It is easy to forget how much logical reasoning a computer must be instructed to do and, in turn, lose sight of your subject while writing your code. If novelists and authors write for people, then programmers write for computers. A writer, regardless of their craft, has to understand their audience to communicate effectively with them, whether the audience is human or otherwise.

References

[EH] Defining Infinite Structures in Haskell, Elias Hernandis, 2019.

[HW] HaskellWiki, HaskellWiki, 2013.

[PL] Programming Languages 2022, Chapman University, 2022.

[TR] Infinite List Tricks in Haskell, TechRepublic, 2007.

[ZV] Zvon, Zvon.