# CPSC-354 Report

Sharon Chang
Chapman University

October 16, 2022

**Abstract**

Placeholder

# Contents

# 1 Introduction

Placeholder

# 2 Homework

## 2.1 Week 1

```
def numInput():
    while True:
        numStr = input("Enter a number: ")
        try:
            num = int(numStr)
            if num > 0:
```

```python
                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) + ".")
```

I chose to write my GCD program in Python.

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

```haskell
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

```
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.
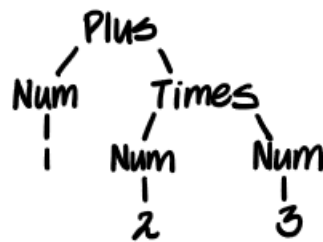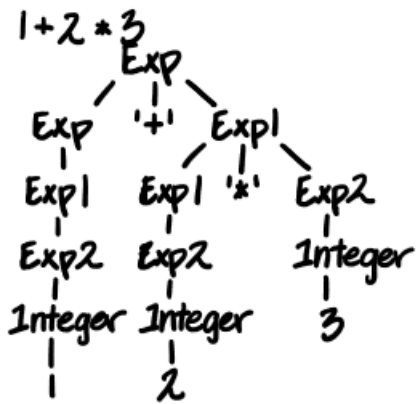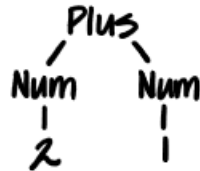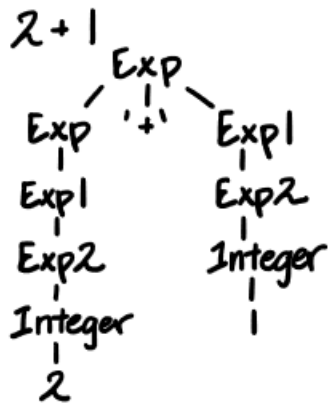
Move the top disk from Pole 1 to Pole 3. Next, move the next top disk from Pole 1 to Pole 2. Take the top disk from Pole 3 and move it to Pole 2. Then, move the top disk from Pole 1 to Pole 3. Follow by moving the top from Pole 2 to Pole 1. Then, move the top disk from Pole 2 to Pole 3. Afterwards, move 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.

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

$$\text{hanoi}(n+1)(x)(y) = \text{hanoi}(n)(x)(\text{other}(x)(y))$$
$$= \text{move}(x)(y)$$
$$= \text{hanoi}(n)(\text{other}(x)(y))(y)$$
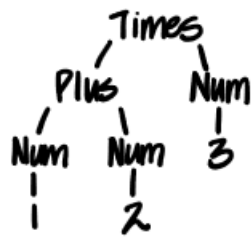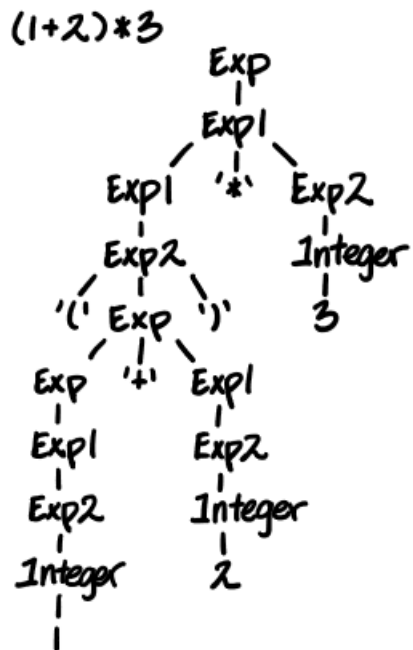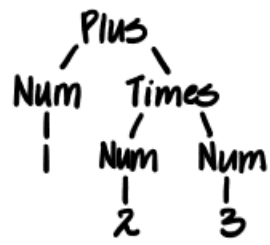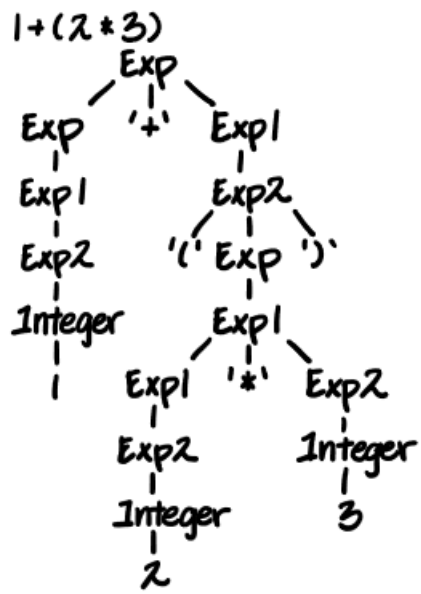
## 2.4 Week 4

2 + 1

```
        Exp
      /  |  \
   Exp  '+'  Exp1
    |         |
   Exp1      Exp2
    |         |
   Exp2     Integer
    |         |
 Integer      1
    |
    2
```

```
        Plus
       /    \
    Num     Num
     |       |
     2       1
```

1 + 2 * 3

```
          Exp
       /   |   \
    Exp   '+'   Exp1
     |        /  |  \
    Exp1   Exp1 '*' Exp2
     |      |        |
    Exp2   Exp2    Integer
     |      |        |
  Integer Integer   3
     |      |
     1      2
```

```
          Plus
        /      \
     Num       Times
      |        /    \
      1      Num     Num
              |       |
              2       3
```

6

1+(2*3)

```
                    Exp
              /      |      \
          Exp       '+'      Exp1
           |                  |
          Exp1               Exp2
           |               /   |   \
          Exp2           '('  Exp  ')'
           |                   |
        Integer              Exp1
           |              /    |    \
           1           Exp1   '*'   Exp2
                        |             |
                       Exp2         Integer
                        |             |
                     Integer          3
                        |
                        2
```

```
            Plus
          /      \
        Num      Times
         |       /    \
         1     Num    Num
                |      |
                2      3
```

(1+2)*3

```
              Exp
               |
              Exp1
           /    |    \
        Exp1   '*'   Exp2
         |            |
        Exp2        Integer
      /  |  \         |
    '('  Exp  ')'     3
       /   |   \
     Exp  '+'  Exp1
      |         |
     Exp1      Exp2
      |         |
     Exp2     Integer
      |         |
   Integer      2
      |
      1
```

```
            Times
          /       \
        Plus      Num
       /    \       |
     Num    Num     3
      |      |
      1      2
```

## 2.5 Week 5

Placeholder

## 2.6 Week 6

```
(\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) 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))))))))))))
```
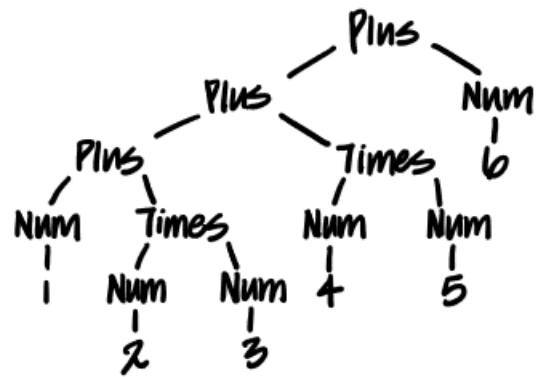
```
=
(((\x. (\x2. (x (x (x ((x (x (x (((\x2. x (x (x x2))) x) x2)))))))))))
=
(((\x. (\x2. (x (x (x ((x (x (x (((x (x (x x)))) x2)))))))))))
=
((\x. (\x2. (x (x (x (x (x (x ((x (x (x x))) x2)))))))))
```

## 2.7   Week 7

In lines 5-7 the variables e1 and e2 are bound by evalCBN with the scope being the body of the evalCBN function. The variables i and e3 are free. In lines 18-22 the variables id1, e1, and s are bound by subst with the scope being body of the subst function. The variables f and e2 are free.

```
evalCBN (\x.x)((\y.y)a) = line 6
evalCBN ((\y.y)a) = line 7
evalCBN (a) = line 8
a
```
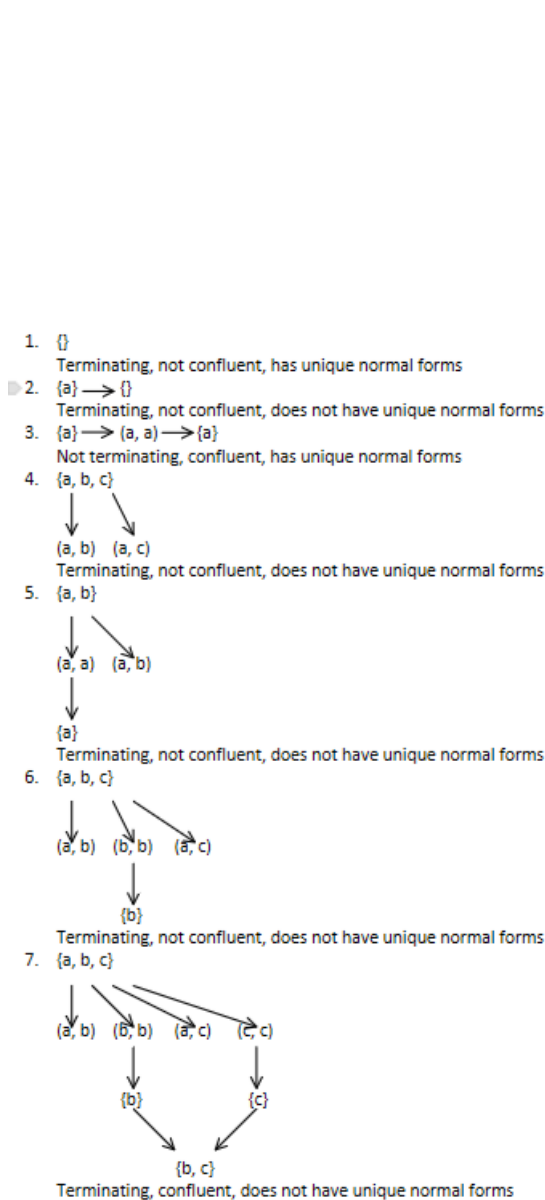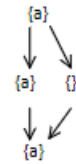
```
evalCBN (\x.\y.x) y z = line 6
evalCBN (\y.y) z = line 7
evalCBN (z) = line 8
z
```
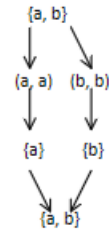
1. Confluent, terminating, has unique normal forms (I think this wrong in hindsight)

{a}

{a}    {}

{a}

2. Confluent, terminating

{a, b}

(a, a)   (b, b)

{a}     {b}

{a, b}

3. Confluent, has unique normal forms

{a}

{a}    {}

{a}

4. Confluent

{a}

(a, a)

{a}

5. Terminating, has unique normal forms

{a, b, c}

{a}    {c}

6. Terminating

{a, a}

{a}

7. Has unique normal forms

{a}

{a}

8. None
   Impossible (?)

1. {}
   Terminating, not confluent, has unique normal forms

2. {a} $\longrightarrow$ {}
   Terminating, not confluent, does not have unique normal forms

3. {a} $\longrightarrow$ (a, a) $\longrightarrow$ {a}
   Not terminating, confluent, has unique normal forms

4. {a, b, c}

   (a, b)   (a, c)
   Terminating, not confluent, does not have unique normal forms

5. {a, b}

   (a, a)   (a, b)

   {a}
   Terminating, not confluent, does not have unique normal forms

6. {a, b, c}

   (a, b)  (b, b)   (a, c)

   {b}
   Terminating, not confluent, does not have unique normal forms

7. {a, b, c}

   (a, b)  (b, b)  (a, c)  (c, c)

   {b}        {c}

   {b, c}
   Terminating, confluent, does not have unique normal forms

# 3   Project

Placeholder

## 3.1   Specification

I want to make a program involving infinite lists in Haskell in order to generate lists of variable size and manipulate them. There are a variety of sequences in math similar to the Fibonacci sequence, and I will implement them as separate functions. The sequences being implemented will be arithmetic, geometric, triangular, square, cube, and Fibonacci.

Triangle, cube, square, and Fibonacci will work by taking in 1 integer number and generating a list of the

sequence with the length of the given integer. For example, (fib 5) would generate a list containing the first 5 Fibonacci numbers. Arithmetic and geometric will take in 3 integer numbers. The first integer works the same as the other sequences, where it dictates the list's length. The 2nd number will be the first entry of the list and the 3rd number will be the number added to each entry. For example, (arit 3 1 6) will produce the list [1, 7, 13]. In geometric, the 2nd number works the same but 3rd number will be the number multiplied against the given value to continue the list. For example, (geo 3 1 6) will produce the list [1, 6, 36].

These lists can have mathematical operations performed between them, which will result in a new list. The available operations will be addition, subtraction, multiplication, and integer division. When performing mathematical operations between lists of different lengths, the resulting list will be the length of the longer list. The shorter list will be extended with 0 values to fill it out to match the length of the longer list. For example, adding the lists [2, 4, 8, 16] and the list [1, 2, 3] will be the same as adding the lists [2, 4, 8, 16] and [1, 2, 3, 0], resulting in the list [3, 6, 11, 16].

## 3.2 Prototype

Placeholder

## 3.3 Documentation

Placeholder

## 3.4 Critical Appraisal

Placeholder

# 4 Conclusions

Placeholder

# References

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