Haskell Functions (continued)

Lazy Evaluation

1. **Infinite list**

```
main :: IO ()
main = do
    putStrLn "Enter how many numbers to generate:"
    n <- readLn
    print (take n [1..])  -- Lazy evaluation ensures only `n`
elements are generated
```

2. **Lazy Fibonacci Sequenc**e: This program takes a user input n and prints the first n Fibonacci numbers.

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)  -- Infinite
Fibonacci series

main :: IO ()
main = do
    putStrLn "Enter the number of Fibonacci numbers to display:"
    n <- readLn
    print (take n fibs)  -- Only computes the first `n` Fibonacci
numbers
```

3. **Lazy Evaluation with map:** This program takes a list of numbers from user input, doubles each number lazily, and prints the first n results.

```
main :: IO ()
main = do
    putStrLn "Enter a list of numbers separated by spaces:"
    input <- getLine
    let numbers = map read (words input) :: [Int]
    putStrLn "Enter how many results to display:"
    n <- readLn
    print (take n (map (*2) numbers))  -- Lazily doubles each
number
```

4. **Lazy Boolean Evaluation (&&):** This program demonstrates short-circuiting with lazy evaluation.

```
lazyAnd :: Bool -> Bool -> Bool
lazyAnd False _ = False  -- If first argument is False, second is
never evaluated
lazyAnd True y = y

main :: IO ()
main = do
    putStrLn "Enter first boolean value (True/False):"
    x <- readLn
```

```
    putStrLn "Enter second boolean value (True/False):"
    y <- readLn
    print (lazyAnd x y)
```

5. **Lazy File Reading:** This program reads a file lazily and prints only the first n lines.

```
main :: IO ()
main = do
    putStrLn "Enter file name:"
    fileName <- getLine
    putStrLn "Enter number of lines to display:"
    n <- readLn
    contents <- readFile fileName
    putStr (unlines (take n (lines contents)))  -- Lazily reads
only required lines
```

Sample output:

```
Enter file name:
file.txt
Enter number of lines to display:
3
(Line 1)
(Line 2)
(Line 3)
```

6. **Lazy Evaluation in an Infinite Prime Generator:** This program generates prime numbers using **lazy evaluation** and the **Sieve of Eratosthenes**.

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]

main :: IO ()
main = do
    putStrLn "Enter the number of prime numbers to display:"
    n <- readLn
    print (take n primes)
```

Example Run

```
Enter the number of prime numbers to display:
10
[2,3,5,7,11,13,17,19,23,29]
```

7. **Lazy Evaluation with the Collatz Sequence:** This program generates the **Collatz sequence** for a given number.

```
collatz :: Integer -> [Integer]
collatz 1 = [1]
```

```
collatz n
    | even n    = n : collatz (n `div` 2)
    | otherwise = n : collatz (3 * n + 1)

main :: IO ()
main = do
    putStrLn "Enter a number to generate its Collatz sequence:"
    n <- readLn
    print (collatz n)
```

Example Run

```
Enter a number to generate its Collatz sequence:
6
[6,3,10,5,16,8,4,2,1]
```

8. **Infinite List of Factorials:** This program generates an **infinite list of factorials** lazily.

```
factorials :: [Integer]
factorials = scanl (*) 1 [1..]  -- [1, 1, 2, 6, 24, 120, ...]

main :: IO ()
main = do
    putStrLn "Enter the number of factorials to display:"
    n <- readLn
    print (take n factorials)
```

```
Example Run

Enter the number of factorials to display:
6
[1,1,2,6,24,120]
```

9. **Infinite Haskell Pascal's Triangle:** This program generates **Pascal's Triangle** using **lazy evaluation.**

Pascal's triangle, in algebra, a triangular arrangement of numbers that gives the coefficients in the expansion of any binomial expression, such as $(x + y)^n$.

```
pascal :: [[Integer]]
pascal = iterate nextRow [1]
  where
    nextRow row = zipWith (+) (0:row) (row++[0])

main :: IO ()
main = do
    putStrLn "Enter the number of rows of Pascal's Triangle to
display:"
    n <- readLn
```

```
    mapM_ print (take n pascal)
```

```
Example Run

Enter the number of rows of Pascal's Triangle to display:
5
[1]
[1,1]
[1,2,1]
[1,3,3,1]
[1,4,6,4,1]
```

10. **Generating the Lazy List of Hamming Numbers:** This program generates **Hamming numbers**, which are numbers whose only prime factors are 2, 3, or 5.

```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x:xs) (y:ys)
    | x < y     = x : merge xs (y:ys)
    | x > y     = y : merge (x:xs) ys
    | otherwise = x : merge xs ys

hammingNumbers :: [Integer]
hammingNumbers = 1 : merge (map (2*) hammingNumbers)
                           (merge (map (3*) hammingNumbers)
                                  (map (5*) hammingNumbers))

main :: IO ()
main = do
    putStrLn "Enter the number of Hamming numbers to display:"
    n <- readLn
    print (take n hammingNumbers)

Example Run

Enter the number of Hamming numbers to display:
10
[1,2,3,4,5,6,8,9,10,12]
```

11. **Infinite Lazy List of Triangular Numbers:** This program generates **triangular numbers**.

```
triangularNumbers :: [Integer]
triangularNumbers = scanl1 (+) [1..]  -- 1, 3, 6, 10, 15, ...

main :: IO ()
main = do
    putStrLn "Enter the number of triangular numbers to display:"
    n <- readLn
    print (take n triangularNumbers)

Example Run
```

```
Enter the number of triangular numbers to display:
10
[1,3,6,10,15,21,28,36,45,55]
```