LABORATORY 9

Exercise 9.4.1: What will be printed by the **:sprint l** after running the following commands:

```
ghci> l = [1,3..] :: [Int]
ghci> take 3 (filter (\x -> mod (x - 1) 2 == 0) l)
[1,3,5]
ghci> :sprint l
l = 1 : 3 : 5 : _
ghci>
```

Exercise 9.4.2: What will be printed by the **:sprint l** after running the following commands:

```
ghci> l = [1,10..] :: [Int]
ghci> take 3 (filter (\x -> mod x 10 == 0) l)
[10,100,190]
ghci> :sprint l
l = 1 : 10 : 19 : 28 : 37 : 46 : 55 : 64 : 73 : 82 : 91 : 100 :
    109 : 118 : 127 : 136 : 145 : 154 : 163 : 172 : 181 : 190 : _
ghci>
```

TRACING:

- twos :: [Integer]

  twos = 2:twos

2:twos -> 2:(2:twos) -> 2:(2:(2:twos)) -> ... -> [2,2,2,...] (the output is a list full of 2s)

- rep :: t -> [t]

  rep e = e:(rep e)

This is a generalization of the example from the above (the output will be a list full of element e (instead of 2 as above))

e:(rep e) -> e:(e:rep e) -> e:(e:(e:rep e)) -> ... -> [e,e,e,...]

- fibs :: [Integer]

fibs = 0:1:(zipWith (+) fibs (tail fibs))

The output will be a list of fibonacci numbers.

take 3 (0:1:(zipWith (+) (0:1:(zipWith (+) fibs (tail fibs))) (1:(zipWith (+) fibs (tail fibs))))) -> am pus doar 1:(zipWith (+) fibs (tail fibs)) fara 0: pt ca am facut tail

[0,1,1]

- count :: [Integer]

  count = 1:(map (+1) count)

The output is a list of consecutive numbers.

1:(map (+1) (1:(map (+1) (1:(map (+1) count))))) -> 1:(map (+1) (1:(map (+1) [1]))) -> 1:(map (+1) [1,2]) -> 1:[2,3] -> [1,2,3]

- powsOf2 :: [Integer]

  powsOf2 = 2:(map (*2) powsOf2)

The output will be a list of powers of 2 (starting from 2).

2:(map (*2) (2:(map (*2) (2:(map (*2) powsOf2)))))

2:(map (*2) (2:(map (*2) [2])))

2:(map (*2) (2: [4]))

2:(map (*2) [2,4])

2:[4,8]

[2,4,8]

- oneList :: [[Integer]]

  oneList = [1]:(map (1:) oneList)

The output will be a list of 1s.

[1]:(map (1:) ([1]:(map (1:) ([1]:(map (1:) oneList)))))

[1]:(map (1:) ([1]:(map (1:) [1])))

[1]:(map (1:) ([1]:[1,1]))

[1]:(map (1:) ([1,1,1])) -> concatenate 1 at each 1 from the list (because map applies at each element)

[1]:[1,1,1,1,1,1]

[1,1,1,1,1,1,1]

- primes :: [Integer]

  primes = sieve [2..] where

sieve (x:xs) = x:sieve [y | y <- xs, mod y x /= 0]

The output will be a list of prime numbers.

take 3 (primes [1,2,3,4,5,6,7,8,9]) -> [2,3,5]

- se va face sieve doar pe lista [2,3,4,5,6,7,8,9]:

se ia pe rand primul elem din lista si apoi se verifica pt fiecare elem din tail daca mod y x diferit de 0 pt ca doar atunci il concateneaza