Chapter 15

```
primes :: [Int]
primes = sieve [2 ..]
  where
    sieve (p : xs) = p : sieve (filter (\x -> x `mod` p /= 0) xs)
```

Exercise 1: Identify the redexes in the following expressions, and determine whether each redex is innermost, outermost, neither, or both:

```
1 + (2*3)
(1+2) * (2+3)
fst (1+2, 2+3)
(\x -> 1 + x) (2*3)
```

The expression `1 + (2*3)` has one redex `2*3` which is both innermost and outermost.

The expression `(1+2) * (2+3)` has two redexes, `1+2` and `2+3`. The redex `1+2` is both innermost and outermost.

The expression `fst (1+2, 2+3)` has three redexes, `1+2` is innermost and the whole expression is outermost.

The expression `(\x -> 1 + x) (2*3)` has two redexes, `2*3` which is innermost and the whole expression which is outermost. The expression `1 + x` inside the lambda is not a redex since we do not reduce under lambdas.

Exercise 2: Show why outermost evaluation is preferable to innermost for the purposes of evaluating the expression `fst (1+2,2+3)`.

Innermost evaluation results in:

$$\begin{aligned}\texttt{fst}\,(1+2,2+3) &= \texttt{fst}\,(3,2+3)\\ &= \texttt{fst}\,(3,5)\\ &= 3\end{aligned}$$

Outermost evaluation results in:

$$\begin{aligned}\texttt{fst}\,(1+2,2+3) &= 1+2\\ &= 3\end{aligned}$$

So outermost evaluation does not evaluate `2+3` which is discarded anyway by `fst`.

Exercise 3: Given the definition `mult = \x -> (\y -> x * y)`, show how the evaluation of `mult 3 4` can be broken down into four separate steps.

$$\begin{aligned}\texttt{mult}\;3\;4 &= (\lambda x \to (\lambda y \to x \times y))\;3\;4\\ &= (\lambda y \to 3 \times y)\;4\\ &= 3 \times 4\\ &= 12\end{aligned}$$

Exercise 4: Using a list comprehension, define an expression `fibs :: [Integer]` that generates the infinite sequence of Fibonacci numbers

$$0, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

using the following simple procedure:

1. the first two numbers are 0 and 1;
2. the next is the sum of the previous two;
3. return to the second step.

Hint: make use of the library functions `zip` and `tail`. Note that numbers in the Fibonacci sequence quickly become large, hence the use of the type `Integer` of arbitrary-precision integers above.

```
fibs :: [Integer]
fibs = [0, 1] ++ zipWith (+) fibs (tail fibs)
```

Exercise 5: Define appropriate versions of the library functions

```
repeat :: a -> [a]
repeat x = xs where xs = x : xs

take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x : xs) = x : take (n - 1) xs

replicate :: Int -> a -> [a]
replicate n = take n . repeat
```

for the following type of binary trees:

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show

repeatT :: a -> Tree a
repeatT x = Node (repeatT x) x (repeatT x)

takeT :: Int -> Tree a -> Tree a
takeT 0 _ = Leaf
takeT _ Leaf = Leaf
takeT n (Node tl x tr) = Node (takeT (n - 1) tl) x (takeT (n - 1) tr)

replicateT :: Int -> a -> Tree a
replicateT n = takeT n . repeatT
```

Exercise 6: *Newton's method* for computing the square root of a (non-negative) floating point number `n` can be expressed as follows:

- start with an initial approximation to the result;
- given the current approximation `a`, the next approximation is defined by the function `next a = (a + n/a) / 2`;

- repeat the second step until the two most recent approximations are within some desired distance of one another, at which point the most recent value is returned as the result.

Define a function `sqroot :: Double -> Double` that implements this procedure. Hint: first produce an infinite list of approximations using the library function iterate. For simplicity, take the number `1.0` as the initial approximation, and `0.00001` as the distance value.

```
approx :: Double -> [Double]
approx n = iterate next 1.0
  where
    next a = (a + n / a) / 2

sqroot :: Double -> Double
sqroot x =
    snd $
        head $
            dropWhile (\(x, y) -> abs (x - y) >= 0.00001) $
                zip a (tail a)
  where
    a = approx x
```