Recursion
Since variables in Haskell are immutable, our only way of achieving iteration is through recursion.
For example, the reverse function receives a list as its input and outputs the same list but with its elements in reverse order:

\[
\text{reverse} :: \text{[a]} \rightarrow \text{[a]} \\
\text{reverse} [] = [] \\
\text{reverse} (x:xs) = \text{reverse} xs ++ [x]
\]

Currying
As you know, you can turn any infix operator into a prefix operator by putting it in parentheses:

\[
(+) 3 4 \\
= 7
\]

Now currying allows us to place the parentheses differently:

\[
(+ 3) 4 \\
= 7
\]

By “fixing” the first input to (+) to be 3, we created a new function (+ 3) that receives only one (further) input.

Lambda Expressions
More examples for lambda expressions:

\[
\text{zipWith} \ (\lambda \ y \rightarrow x^2 + y^2) \text{[1..10]} \text{[11..20]} \\
= [122, 148, 178, 212, 250, 292, 338, 388, 442, 500]
\]

\[
\text{map} \ (\lambda \rightarrow (x, x^2, x^3)) \text{[1..10]} \\
= [(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64), (5, 25, 125)]
\]

Recursion
Another example: The function zip takes two lists and outputs a list of pairs with the first element taken from the first list and the second one from the second list. Pairs are created until one of the lists runs out of elements.

\[
\text{zip} :: \text{[a]} \rightarrow \text{[b]} \rightarrow \text{[(a,b)]} \\
\text{zip} _ [] = [] \\
\text{zip} [ ] _ = [] \\
\text{zip} (x:xs) (y:ys) = (x,y):\text{zip} \ xs \ ys
\]

Currying
We can check this:

\[
:t \ (+ 3) \\
\text{Int} \rightarrow \text{Int}
\]

This (+ 3) function can be used like any other function, for example:

\[
\text{map} \ (+ 3) \text{[1..5]} \\
= [4, 5, 6, 7, 8]
\]

Or:

\[
\text{map} \ (\text{max} \ 5) \text{[1..10]} \\
= [5, 5, 5, 5, 5, 6, 7, 8, 9, 10]
\]

Folds
Folds take a binary function \( f \), a start value \( z \), and a list:

\[
\text{foldr} \ f \ z \ [] = z \\
\text{foldr} \ f \ z \ (x:xs) = f \ x \ (\text{foldr} \ f \ z \ xs) \\
\text{foldl} \ f \ z \ [] = z \\
\text{foldl} \ f \ z \ (x:xs) = \text{foldl} \ f \ (f \ z \ x) \ xs
\]
Folds
You can think of these functions as using an accumulator that receives value z. Its value is updated by applying f to it and the next list element, and the output is the final accumulator value after the whole list has been processed.
To understand these two functions and their difference, it is best to do some sample computations by hand.
Folds can be used to build a variety of functions, e.g.:
sum = foldl (+) 0
(Point-free style: The last argument xs is identical on both sides of the equation and can be omitted.)

The $ Operator
The $ operator is defined as follows:
f $ x = f x
It has the lowest precedence, and therefore, the value on its right is evaluated first before the function on its left is applied to it.
As a consequence, it allows us to omit parentheses:
negate (sum (map sqrt [1..10]))
can be written as:
negate $ sum $ map sqrt [1..10]

Function Composition
Similarly, we can use function composition to make our code more readable and to create new functions. As you know, in mathematics, function composition works like this:
(f ° g) (x) = f(g(x))
In Haskell, we use the "." character instead:
map (\xs -> negate (sum (tail xs)))
[[1..5],[3..6],[1..7]]
Can be written as:
map (negate . sum . tail)
[[1..5],[3..6],[1..7]]

Data Types
Data types can be declared as follows:
data Bool = false | true
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
Then we can construct values of these types like this:
x = Circle 3 4 5
To make these values printable, we need to use:
derive Show Shape

Records
If we want to name the components of our data types, we can use records:
data Car = Car {company :: String, model :: String, year :: Int}
derive Show Car
myCar = Car {company="Ford", model="Mustang", year=1967}
Car.company myCar
Ford

Data Types
We can use pattern matching on our custom data types:
surface :: Shape -> Float
surface (Circle _ _ r) = 3.1416 * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
surface $ Circle 10 20 10
314.16
Input/Output with “do”

Purely functional code cannot perform user interactions such as input and output, because it would involve side effects. Therefore, we sometimes have to use impure functional code, which needs to be separated from the purely functional code in order to keep it (relatively) bug-safe. In Haskell, this is done by so-called Monads. To fully understand this concept, more in-depth study is necessary. However, in this course, we do not need to perform much input and output. We can use a simple wrapper (or “syntactic sugar”) for this – the “do” notation.

In a do-block, we can only use statements whose type is “tagged” IO so that they cannot be mixed with purely functional statements.

Example for a program performing input and output:

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Homework

Now let us finally look at the homework solutions!

(1) Write a function `divides a b` that tells you whether `a` divides `b`. Examples:

- \( \text{divides } 7 \ 14 \) true
- \( \text{divides } 2 \ 9 \) false

Solution:

`divides a b = (mod b a == 0)`

(2) Make use of your `divides` function and write another function `divList n d` that returns a list of all integers from 1 to `n` that are divisible by `d`. Example:

```
divList 30 7
```

Output:

```
[7, 14, 21]
```

Solution:

`divList n d = filter (divides d) [1..n]`

(3) Write a function `isPrime n` that tells you if `n` is prime. (This one is tricky.) Examples:

- `isPrime 5`
  
  true
- `isPrime 20`
  
  false

Solution:

`isPrime n = n > 1 && null [x | x <- [2 .. n `div` 2], n `mod` x == 0]`