Session 5

Functions as data: Higher-order functions

This session

interpreter

```
map sqrt [2, 3, 4, 5, 6]
map (*2) [1..10]
map (^2) [1..10]
map (2^) [1..10]
filter even [3, 1, 4, 5, 9, 2, 6]
```

Higher-order functions (functions that take functions as arguments), with list functions as an example:

- general-purpose higher-order functions of lists in the standard library, bundling common patterns
- by passing simple functions are parameters to these functions, we can express useful transformations

Aim: to be able to use higher-order functions effectively.

General patterns over lists

Some common patterns of computation over lists

- apply to all (mapping)

```
ordAll :: [Char] -> [Int]
ordAll cs = [ord c | c <- cs]

doubleAll :: [Int] -> [Int]
doubleAll ns = [2*n | n <- ns]
```

- selecting elements (filtering)

```
pickEven :: [Int] -> [Int]
pickEven ns = [n | n <- ns, even n]
```

```
letters :: [Char] -> [Char]
letters cs = [c | c <- cs, isAlpha c]
```

How can we avoid writing similar functions over and over?

Generalized mapping

Prelude

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

(Later we'll see that actually **map** has a more elementary definition.)

map applies **f** to each element:

map f $[x_1, x_2, \ldots] = [$**f** $x_1,$ **f** $x_2, \ldots]$

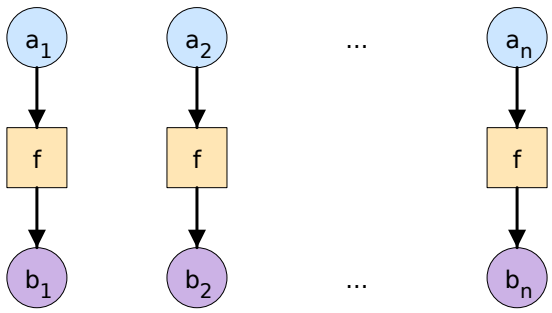
Examples:

interpreter

```
:m + Data.Char
map sqrt [2, 3, 4, 5, 6, 7]
map ord "Hello"
map toUpper "hello"
map length (words "I am not a number")
```

A diagram of the map function

```
map :: (a -> b) -> [a] -> [b]
```



Special cases of map

Given the higher-order **map** function:

Prelude

```
map :: (a -> b) -> [a] -> [b]
```

We can rewrite the previous functions as

```
ordAll cs = map ord cs

doubleAll ns = map double ns
  where double n = 2*n
```

Selecting elements from a list

Another function:

Prelude

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Examples:

interpreter

```
:m + Data.Char
filter odd [3,1,4,5,9,2,6]
filter even [3,1,4,5,9,2,6]
filter isUpper "Hello"
filter isAlpha "Hello world"
```

Given `filter`, the previous functions become

```
pickEven ns = filter even ns
letters cs = filter isAlpha cs
```

Generalized splitting of lists

Two more library functions with the same type as `filter`:

Prelude

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Examples

interpreter

```
:m + Data.Char
takeWhile isAlpha "Hello world"
dropWhile isAlpha "Hello world"

filter odd [1..]
takeWhile odd [1..]
```

Anonymous functions

Function values

There are several ways of making functions that can be used as arguments to higher-order functions:

- defined names, e.g. `max`, `(+)`, `double`.
- partially applied functions, e.g. `max 0`
- functions that return functions, e.g. `map`
- anonymous functions (a.k.a. lambda expressions), e.g. `\n -> n*n`
- operator sections, e.g. `(2/)`, `(/2)`

These are all kinds of expression, with function types.

Historical aside: the λ-calculus

In the λ-calculus, a form of computation devised by Alonzo Church in the 1930s, there were only three kinds of expression:

- variables x
- applications $e_1\ e_2$
- anonymous functions $\lambda\, x.\, e$

with a single computation step:

$$(\lambda\, x.\, e)\ a \quad \Longrightarrow \quad e \text{ with each } x \text{ replaced by } a$$

For example,

$$(\lambda\, x.\, f\ x\ x)\ (g\ y) \quad \Longrightarrow \quad f\ (g\ y)\ (g\ y)$$

Anonymous functions in Haskell

Anonymous functions have the form `\`x `->` `e` (approximating $\lambda\, x.\, e$).

Instead of writing

```
doubleAll :: [Int] -> [Int]
doubleAll ns = map double ns
  where double n = 2*n
```

we can write

```
doubleAll :: [Int] -> [Int]
doubleAll ns = map (\n -> 2*n) ns
```

For example: **interpreter**

```
map (\n -> 2*n) [1..10]
```

However, giving the function a meaningful name often makes the program clearer.

A special case: operator sections

A special syntax is available for infix operators like `*`:

`(*` e`)`	is equivalent to	`\ x -> x *` e
`(`e `*)`	is equivalent to	`\ x ->` e `* x`

for some variable `x` that does not occur in e.

Examples: **interpreter**

```
map (2*) [1..10]
map (+1) [1..10]
map (2^) [1..10]
map (^2) [1..10]
takeWhile (< 20) [1..]
filter (< 20) [1..]
```

Exception: **(-1)** isn't a section, it's a negative number.

To get the corresponding effect, one must use the function **subtract** instead:

interpreter

```
subtract 1 5
map (subtract 1) [1..10]
```

Partial application

How many arguments does a function take?

A function type like

Prelude

```
max :: Int -> Int -> Int
```

is actually an abbreviation for

Prelude

```
max :: Int -> (Int -> Int)
```

Similarly, the expression **max 0 y** is equivalent to **(max 0)y**.

That is, the partial application **max 0 :: Int -> Int** is a function, and we can pass it as an argument:

interpreter

```
map (max 5) [1..9]
map (min 5) [1..9]
```

Special case: functions that return functions

We can also view **map** as a function returning a function:

Prelude

```
map :: (a -> b) -> ([a] -> [b])
```

so that the previous definitions

```
ordAll :: [Char] -> [Int]
ordAll cs = map ord cs

letters :: [Char] -> [Char]
letters cs = filter isAlpha cs
```

may be abbreviated as

```
ordAll = map ord

letters = filter isAlpha
```

Function composition

This is the counterpart of the mathematical operator $f \circ g$.

$$(f \circ g)(x) = f(g(x))$$

Prelude

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

Note the type could also be written

Prelude

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Examples:

interpreter

```
filter (not . isAlpha) "Hello world!"
filter not (map isAlpha "Hello world!")
map (chr . ord) "Hello world!"
map chr (map ord "Hello world!")
```

More list functions

Operating on lists (including infinite ones)

For many list operations, the initial part of the output depends only on the initial part of the input, so these operations are also useful on infinite lists:

- **map** also works on infinite lists:

  interpreter

  ```
  take 20 (map (^2) [1..])
  takeWhile (<1000) (map (2^) [1..])
  ```

- **filter** also works, if it produces enough elements:

  interpreter

  ```
  take 20 (filter even [1..])
  take 5 (filter (<6) [1..])
  take 20 (filter (<6) [1..])
  ```

- similarly **takeWhile**:

  interpreter

  ```
  takeWhile (< 100) [1..]
  takeWhile ((< 1000) . (^2)) [1..]
  ```

Combining lists elementwise

Recall

Prelude

```
zip :: [a] -> [b] -> [(a,b)]
```

zip $[x_1, x_2, \ldots]$ $[y_1, y_2, \ldots] = [(x_1, y_1), (x_2, y_2), \ldots]$

interpreter

```
zip [1..] "abcdef"
```

The library includes a higher-order version:

Prelude

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

zipWith f $[x_1, x_2, \ldots]$ $[y_1, y_2, \ldots] = [f\ x_1\ y_1, f\ x_2\ y_2, \ldots]$

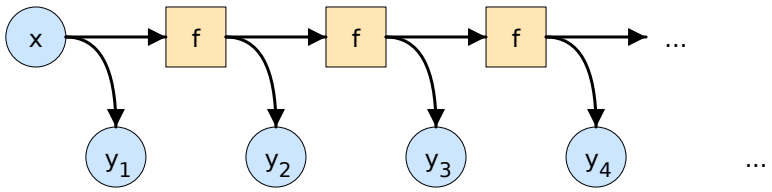
interpreter

```
zipWith (*) [1..] [10,9..1]
zipWith (<) [1..] [10,9..1]
take 20 (zipWith (*) [1..] [2..])
```

Repeatedly applying a function

This higher-order function takes a function and applies it repeatedly to an argument:

```
iterate :: (a -> a) -> a -> [a]
```



`iterate` $f\ x = [x, f\ x, f\ (f\ x), \ldots$

For example:

interpreter

```
take 20 (iterate (+1) 1)
takeWhile (< 3000) (iterate (*2) 1)
```

Example: linear congruential generator

A simple scheme for generating pseudo-random numbers from a seed value:

```
module Generator where

generate :: Int -> [Int]
generate seed = iterate (\ n -> 224149*n + 1) seed
```

(Other numbers could be used, but some choices are better than others.)

A handy operator

The `($)` operator is equivalent to function application:

Prelude

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

but because it is an infix operator, we can write things like

interpreter

```
putStr $ unlines $ map show $ take 9 $ iterate (+1) 0
```

instead of

interpreter

```
putStr (unlines (map show (take 9 (iterate (+1) 0))))
```

Folding lists

Folding lists to summary values

Several library functions reduce a list to a single value:

Prelude

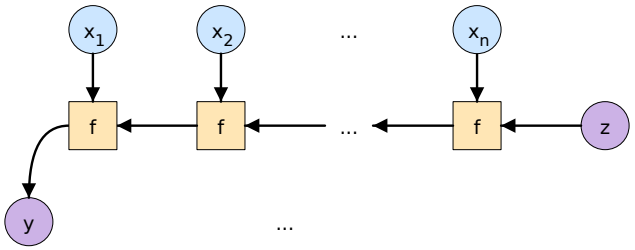
```
sum :: Num a => [a] -> a
product :: Num a => [a] -> a
and :: [Bool] -> Bool
or :: [Bool] -> Bool
concat :: [[a]] -> [a]
```

interpreter

```
sum [1..5]
product [1..5]
and [True, False, True]
or [True, False, True]
concat [[1], [2,3], [], [4,5]]
```

Generalized folding

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```



Examples:

interpreter

```
foldr (+) 0 [1..5]
foldr (*) 1 [1..5]
foldr (++) [] [[1], [2,3], [], [4,5]]
```

Definitions using `foldr`

Given the higher-order **`foldr`** function:

Prelude

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

We can rewrite the previous functions as

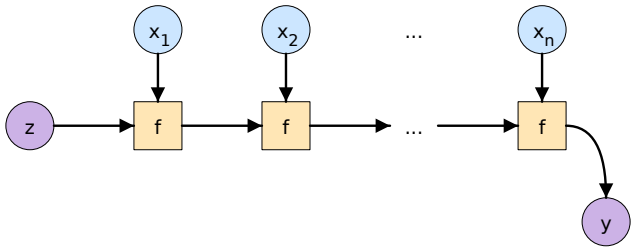
Prelude

```
sum xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
and xs = foldr (&&) True xs
or xs = foldr (||) False xs
concat xs = foldr (++) [] xs
```

Folding from the left

There is also another version that folds from the left:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```



Examples (same answers as **foldr**):

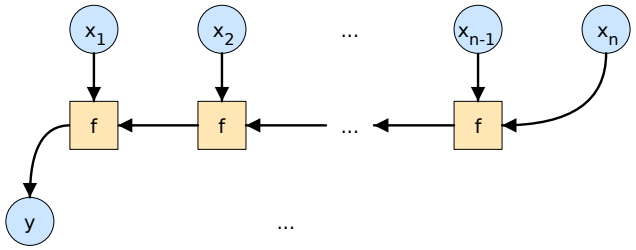
interpreter

```
foldl (+) 0 [1..5]
foldl (*) 1 [1..5]
foldl (++) [] [[1], [2,3], [], [4,5]]
```

Folding non-empty lists

For situations when there is no answer in the empty list case:

```
foldr1 :: (a -> a -> a) -> [a] -> a
```



Examples:

interpreter

```
foldr1 max [3,1,4,5,9]
foldr1 min [3,1,4,5,9]
```

Special cases

The library function

Prelude

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

is used to define

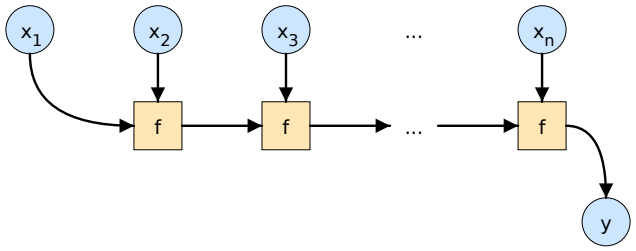
Prelude

```
maximum :: Ord a => [a] -> a
maximum xs = foldr1 max xs

minimum :: Ord a => [a] -> a
minimum xs = foldr1 min xs
```

Folding non-empty lists from the left

There is a similar counterpart to **`foldl`**:

```
foldl1 :: (a -> a -> a) -> [a] -> a
```



Examples:

interpreter

```
foldl1 max [3,1,4,5,9]
foldl1 min [3,1,4,5,9]
```

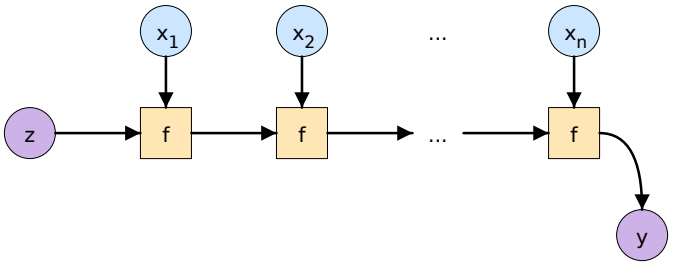
Example: driving the turtle

To test our turtle, generate a sequence of pseudo-random turtle commands from a seed:

```
genCommand :: Int -> Command
genCommand n
  | r == 0 = TurnLeft
  | r == 1 = TurnRight
  | r == 2 = RaisePen
  | r == 3 = LowerPen
  | otherwise = Move (r - 3)
  where
    r = n `mod` 11

genCommands :: Int -> [Command]
genCommands seed = map genCommand (generate seed)
```

`foldl` and turtle functions

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```



Turtle operations:

```
startTurtle :: Turtle
action :: Turtle -> Command -> Turtle
```

so **`foldl action startTurtle :: [Command] -> Turtle`**

Applying a list of commands

What is the turtle state after 100 commands?

interpreter

```
foldl action startTurtle $ take 100 $ genCommands 37
```

(The seed value could be anything.)

Other questions:

- What's the furtherest the turtle gets from the origin in the first 100 commands?
- How many commands does it take to get the turtle 200 steps from the origin?

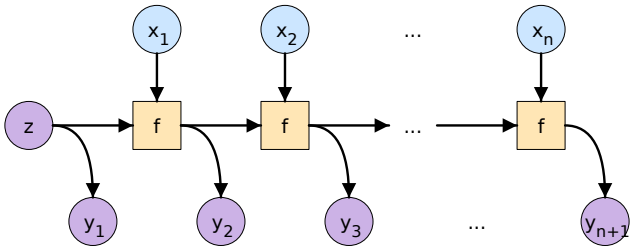
To answer these, we need the whole history of the turtle's state.

Accumulating lists

Partial sums (and products, etc)

Another library function:

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```



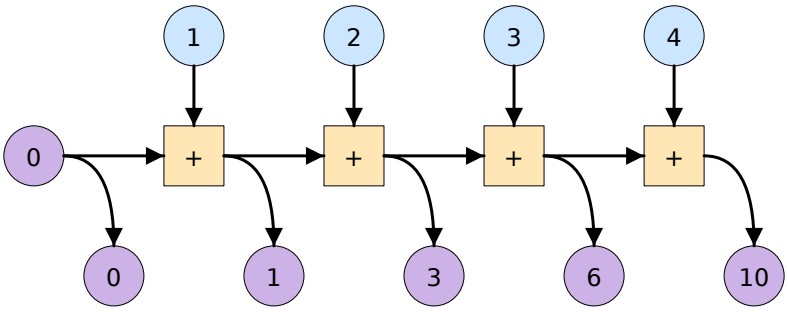
interpreter

```
scanl (+) 0 [1..6]
scanl (*) 1 [1..6]
```

The list output by `scanl f z xs` is one longer than `xs`. Its first element is `z`, and its last is equal to `foldl f z xs`.

`scanl` on sums

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```



interpreter

```
scanl (+) 0 [1..4]
```

Scanning infinite lists

This function

Prelude

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

$$\texttt{scanl} \oplus z\ [x_1, x_2, \ldots] = [z, z \oplus x_1, z \oplus x_1 \oplus x_2, \ldots]$$

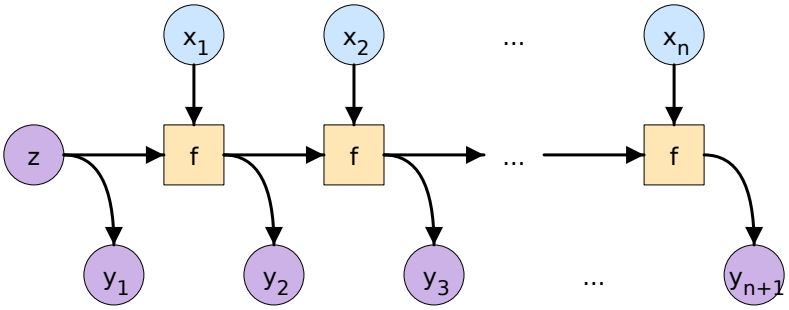
also works on infinite lists:

interpreter

```
take 20 (scanl (+) 0 (repeat 1))
take 20 (scanl (+) 0 [1..])
take 20 (scanl (*) 1 (repeat 2))
take 20 (scanl (*) 1 [1..])
```

`scanl` and turtle functions

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```




```
startTurtle :: Turtle
action :: Turtle -> Command -> Turtle
```

so `scanl action startTurtle :: [Command] -> [Turtle]`

Turtle history

interpreter

```
putStr $ unlines $ map show $ take 20 $
    scanl action startTurtle $ genCommands 37
```

What's the furtherest the turtle gets from the origin in the first 100 commands?

interpreter

```
maximum $ map normPoint $ map location $
    scanl action startTurtle $
    take 100 $ genCommands 37
```

How many commands does it take to get the turtle 200 steps from the origin?

interpreter

```
length $ takeWhile (< 200) $
    map normPoint $ map location $
    scanl action startTurtle $ genCommands 37
```

Looking back

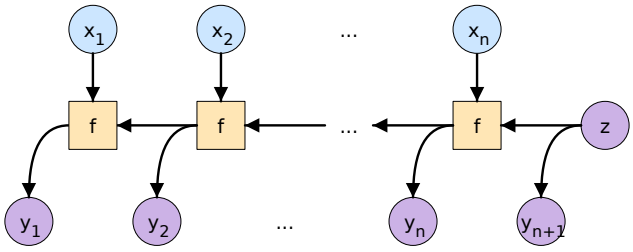
What have we done here?

- We defined simple functions like `\ n -> 224149*n + 1`, `genCommand` and `action`, which we can test in isolation.
- Passing those functions as parameters to higher-order functions like `map`, `iterate`, `foldl` and `scanl` yields powerful transformations.
- Functions like `iterate` and `scanl` yield entire histories, as infinite lists.
- We can use functions like `take` and `takeWhile` to select the initial portion of interest.
- We can use assemble these components is many ways to answer different questions.
- Operating on the history as a list is more flexible than procedural solutions, where other calculations are mixed in with movement.

Scanning from the right

There is also `scanr`, but it is mainly useful with finite lists.

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```



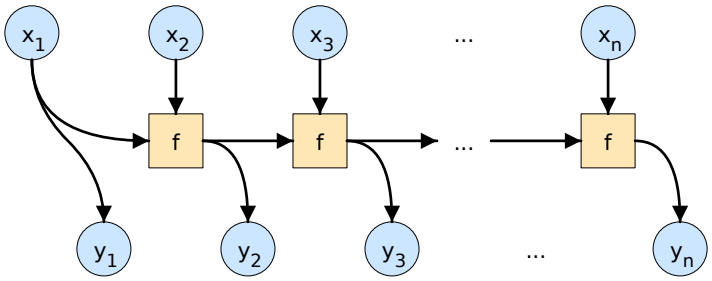
interpreter

```
scanr (+) 0 [1..6]
scanr (*) 1 [1..6]
```

Scanning non-empty lists

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

```
module Turtle where

import Geometry
import Generator

-- state of a turtle
data Turtle = Turtle Point Direction PenState
    deriving Show

-- state of a turtle's pen
data PenState = PenUp | PenDown
    deriving Show

-- start at the origin, facing North, with pen up
startTurtle :: Turtle
startTurtle = Turtle origin North PenUp

-- location of the turtle
location :: Turtle -> Point
location (Turtle pos dir pen) = pos

-- a command for a turtle
data Command
    = TurnLeft | TurnRight | Move Int | RaisePen | LowerPen
    deriving Show

-- action of a turtle command
action :: Turtle -> Command -> Turtle
action (Turtle pos dir pen) TurnLeft =
    Turtle pos (turnLeft dir) pen
action (Turtle pos dir pen) TurnRight =
    Turtle pos (turnRight dir) pen
action (Turtle pos dir pen) (Move n) =
    Turtle (plusPoint pos (timesPoint n (oneStep dir))) dir pen
action (Turtle pos dir _) RaisePen =
    Turtle pos dir PenUp
action (Turtle pos dir _) LowerPen =
    Turtle pos dir PenDown

-- convert a number into a command
genCommand :: Int -> Command
genCommand n
  | r == 0 = TurnLeft
  | r == 1 = TurnRight
  | r == 2 = RaisePen
  | r == 3 = LowerPen
  | otherwise = Move (r - 3)
  where
    r = n `mod` 11

-- generate an infinite list of commands, for testing
genCommands :: Int -> [Command]
genCommands seed = map genCommand (generate seed)
```

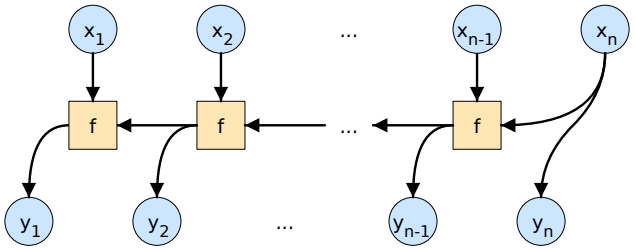


interpreter

```
scanl1 max [3,1,4,5,9,2,6]
scanl1 min [3,1,4,5,9,2,6]
```

Scanning non-empty lists from the right

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
```



interpreter

```
scanr1 max [3,1,4,5,9,2,6]
scanr1 min [3,1,4,5,9,2,6]
```

Summary

We have seen a number of higher-order functions on lists:

- value to list: **`iterate`**
- list to list, elementwise: **`map`** and **`zipWith`**
- list to list, selecting elements: **`filter`**, **`takeWhile`** and **`dropWhile`**
- list to value: **`foldr`**, **`foldl`**, **`foldr1`** and **`foldl1`** (with special cases **`sum`**, **`product`**, **`and`**, **`or`**, **`concat`**, **`maximum`** and **`minimum`**)
- list to list, incremental accumulation: **`scanr`**, **`scanl`**, **`scanr1`** and **`scanl1`**

Exercises

1. The following three functions are similar to exercises from session 4 (you need `import Data.Char`), but this time use **`map`** and **`filter`** instead of list comprehensions.

   (a) Give a definition of the function

   ```
   capitalize :: String -> String
   ```

   that returns the input list with the lower case letters capitalized and the others unchanged.

   (b) Give a definition of the function

   ```
   capitals :: String -> String
   ```

   that selects the capital letters from the input string.

   (c) Write a function that does the same as **`capitalize`**, but discards non-letters.

2. Use higher order functions to write expressions to compute

   (a) the squares of all the numbers up to 20.

   (b) all the square numbers less than 500.

   (c) all the square numbers between 500 and 1000.

3. Work out what the following function does, and rewrite it using higher order functions and operator sections instead of a list comprehension:

   ```
   count :: Eq a => a -> [a] -> Int
   count x ys = length [y | y <- ys, y == x]
   ```

4. (a) What does this function do?

   ```
   f :: [Integer] -> [Integer]
   f = filter ((< 20) . (^2))
   ```

   (b) Rewrite the following function using sections:

   ```
   f :: [Float] -> [Float]
   f xs = map (\x -> (x+1)/2) xs
   ```

5. What does the following function do?

   ```
   foo :: [Int] -> [Int]
   foo xs = zipWith (-) (tail xs) xs
   ```

6. (a) Write a function

   ```
   collatz :: Int -> Int
   ```

that takes a number n and returns half of n if n is even, and $3n + 1$ if n is odd.

If you start from 3, the sequence eventually reaches 1:

$$3, 10, 5, 16, 8, 4, 2, 1$$

The sequence seems to eventually reach 1 whatever number you start with, but no-one has been able to prove this. (This is the *Collatz conjecture*.)

(b) Write a function

```
collatzSteps :: Int -> Int
```

that takes a positive number n and returns the number of times that the function must be applied to reach 1. In the above example, **`collatzSteps 3`** is **`7`**.

(c) Write a function

```
collatzMax :: Int -> Int
```

that takes a positive number n and returns the largest number encountered by repeatedly applying the function until it reaches 1. In the above example, **`collatzMax 3`** is **`16`**.

7. Consider Pascal's triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
...
```

where each number is the sum of the two just above it. (Some versions omit the top row.)

Write a definition

```
pascal :: [[Int]]
```

of a list representation: **`[[1], [1,1], [1,2,1], [1,3,3,1], ...`**

Hint: write a function of type **`[Int] -> [Int]`** to construct a row from the one immediately above it.

8. What do the following expressions compute:

   (a) **`scanl (+)0 [1,3..]`**

   (b) **`scanl (*)1 [1..]`**

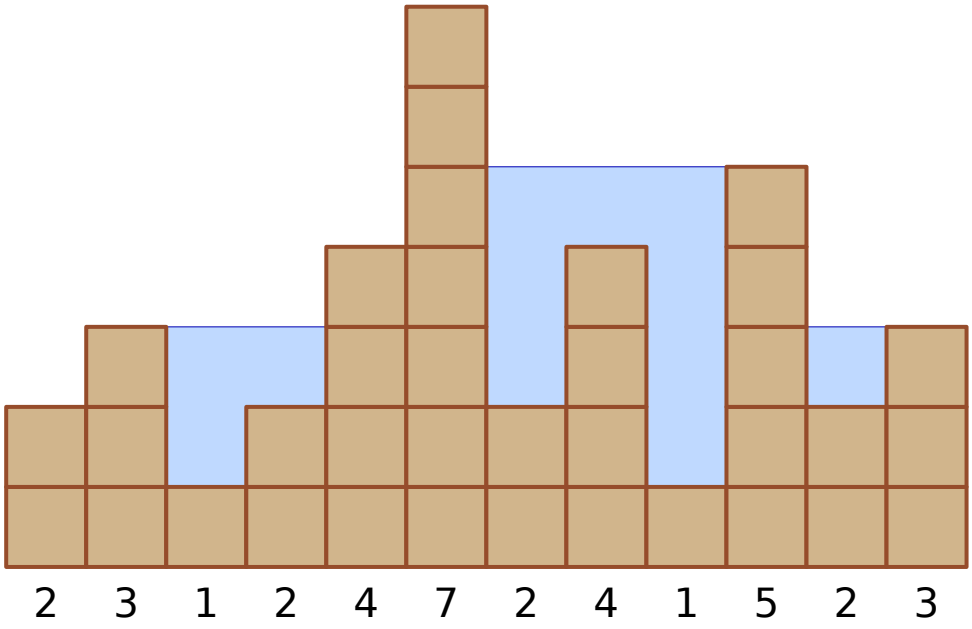
9. Returning to the turtle example from the lecture, write a function to compute the following:

   Given an infinite list of commands, after the turtle reaches 200 steps away from the origin, how many more commands does it take to reach 400 steps from the origin?

   Here are some test cases:

Input	Output
`genCommands 37`	`122`
`genCommands 39`	`1906`
`genCommands 42`	`1222`

10. (optional extra challenge) Consider the question of how much water would be held if we poured water over a histogram of blocks like this:



This is a two-dimensional world, so the water can't run off at the front or back, but it does need to be held in by both sides. Thus the histogram in the above example can hold 12 units of water. Your task is to write a function

```
waterHeld :: [Int] -> Int
```

to compute this amount. For example, if applied to `[2,3,1,2,4,7,2,4,1,5,2,3]` (the above histogram), it should return `12`. There is a concise answer that runs in linear time.

Hint: To find the water level at a given point, we need the height of the tallest bar from that point left, and the height of the tallest from that point right. You can work out each of these separately, but for all points at once, *i.e.* construct a list where each entry is the greatest height of any bar from the start to that point, and another list for the same in the opposite direction.