IN3043 Functional Programming
Solutions to Exercises 5

1. (a) We want to do the same thing to each element of the list, so we use **map**:

   ```
   capitalize cs = map toUpper cs
   ```

   (b) To select some elements from the string, we use **filter**:

   ```
   capitals cs = filter isUpper cs
   ```

   (c) To do this, we must combine **filter** and **map**:

   ```
   capitalizeLetters cs = map toUpper (filter isLetter cs)
   ```

2. (a) This is a simple **map**:

   ```
   map (^2) [1..20]
   ```

   (b) We use **takeWhile** rather than **filter**, because we don't want to keep examining numbers forever:

   ```
   takeWhile (< 500) (map (^2) [1..])
   ```

   (If we had used **filter**, we'd get the same answer, but not the end of the list, because GHCi has no way of knowing that there won't be any more.)

   (c) Here we combine **takeWhile** with **dropWhile**:

   ```
   takeWhile (< 1000) (dropWhile (<= 500) (map (^2)
      [1..]))
   ```

3. It returns the number of times that **x** occurs in the list **ys**. We can rewrite it as

   ```
   count x ys = length (filter (== x) ys)
   ```

   or equivalently

   ```
   count x = length . filter (== x)
   ```

4. (a) This is equivalent to

   ```
   f = filter p
       where p n = n^2 < 20
   ```

   That is, it takes a list and selects the numbers whose squares are less than 20.

(b) First 1 is added to **x**, and then the result is divided by 2:

```
f = map ((/2) . (+1))
```

5. **tail xs** yields all the elements of **xs** but the first. Zipping this with **xs** pairs each element with its predecessor. By applying **(-)** to each of these pairs, we compute the difference between each number in the list and its predecessor. This list of differences will be one shorter than the input list **xs**.

6. (a) This is a straight translation of the description:

```
collatz :: Int -> Int
collatz n
  | even n = n `div` 2
  | otherwise = 3*n + 1
```

(b) We can use **iterate** to generate the sequence, and **takeWhile** to cut it when **1** is reached:

```
collatzSteps :: Int -> Int
collatzSteps n = length (takeWhile (> 1) (iterate
   collatz n))
```

(c) We use the same sequence as before, but now we want the largest element:

```
collatzMax :: Int -> Int
collatzMax n
  | n <= 1 = 1
  | otherwise = maximum (takeWhile (> 1) (iterate
     collatz n))
```

If the argument is **1**, the list will be empty and **maximum** will fail, so we handle that case specially.

7. The function to construct a row from it predecessor is

```
nextrow :: [Int] -> [Int]
nextrow ns = zipWith (+) ([0] ++ ns) (ns ++ [0])
```

The definition of Pascal's triangle follows immediately:

```
pascal :: [[Int]]
pascal = iterate nextrow [1]
```

8. (a) The list **[1,3..]** is the infinite list of odd numbers. Plugging this into the diagrams given in the lecture, **scanl (+)0 [1,3..]** computes the infinite list

```
[0, 0+1, (0+1)+3, ((0+1)+3)+5, ...]
```

which evaluates to `[0, 1, 4, 9, 16, 25, ...]`, that is, the infinite list of square numbers.

(b) `scanl (*)1 [1..]` computes the infinite list

```
[1, 1*1, (1*1)*2, ((1*1)*2)*3, ...]
```

which evaluates to `[1, 1, 2, 6, 24, ...]`, that is, the infinite list of factorials.

9. First, we define a function to construct the history of a turtle executing a list of commands:

```
turtleHistory :: [Command] -> [Turtle]
turtleHistory cmds = scanl action startTurtle cmds
```

We can shorten that a little:

```
turtleHistory :: [Command] -> [Turtle]
turtleHistory = scanl action startTurtle
```

This is because applying `turtleHistory` to a list of commands expands to applying `scanl action startTurtle` to that list of commands.

For the desired function, we only need the distance of the location of each state. We could do this by calculating the number of steps until 200 or more is reached, as in the lecture, doing the same for 400, and subtracting the two counts.

An alternative approach, which avoids scanning the history twice, is to drop initial entries until 200 or more is reached, and then take entries from there until 400 is reached:

```
further :: [Command] -> Int
further cmds =
    length $
    takeWhile (< 400) $
    dropWhile (< 200) $
    map normPoint $ map location $
    turtleHistory cmds
```

We can use the fact that mapping by one function and then mapping by another is equivalent to mapping by their composition:

$$\texttt{map}\ f\ .\ \texttt{map}\ g = \texttt{map}\ (f\ .\ g)$$

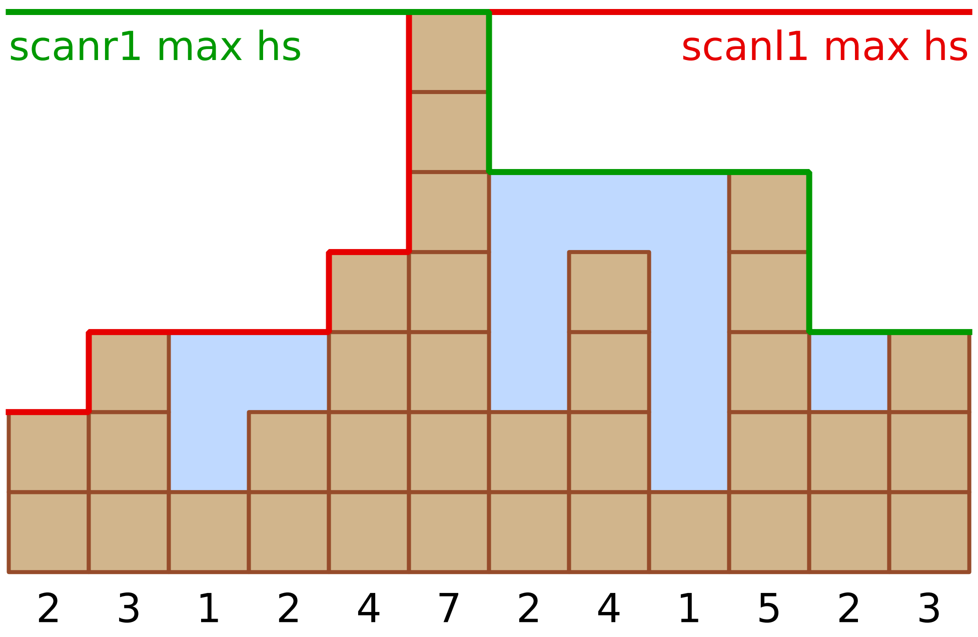
to rewrite our function as

```
further :: [Command] -> Int
further cmds =
    length $
    takeWhile (< 400) $
    dropWhile (< 200) $
    map (normPoint . location) $
    turtleHistory cmds
```

We can even write the whole thing as a composition:

```
further :: [Command] -> Int
further =
    length .
    takeWhile (< 400) .
    dropWhile (< 200) .
    map (normPoint . location) .
    turtleHistory
```

10. We first compute the maximum water level at each point, because subtracting the height of the bars will give the amount of water. For this, we need the height of the tallest bar to the left and to the right of each point. We can compute each of these, for all positions, with scans from the left and right:



Computing the lesser at each point with **`zipWith min`** yields the water level at each point. Similarly we can obtain the amount of water at each point by subtracting the bar heights from the water levels with **`zipWith (-)`**. The sum of these is the total amount of water:

```
waterHeld :: [Int] -> Int
waterHeld hs
  | null hs = 0
  | otherwise = sum (zipWith (-) water_level hs)
  where
    water_level = zipWith min (scanl1 max hs) (scanr1 max
       hs)
```

The Prelude also has an 3-way version of **`zipWith`**:

Prelude

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

which could be used to combine the two **`zipWith`**s:

```
waterHeld :: [Int] -> Int
waterHeld hs
  | null hs = 0
  | otherwise =
    sum (zipWith3 water (scanl1 max hs) (scanr1 max hs)
       hs)
  where
    water lmax rmax h = min lmax rmax - h
```