IN3043 Functional Programming
Solutions to Exercises 4

1. (a)
```
[1..100]
```
```
[n*n | n <- [1..20]]
```
(b)
```
[n, n <- [1..100], 100 `mod` n == 0]
```

2. This is a simple list comprehension:
```
squareAll ns = [n*n | n <- ns]
```
Note the use of a variable name like **ns** to refer to a list of things.

3. (a) The capitalization function is a another list comprehension with the same structure as **doubleAll**:
```
capitalize cs = [toUpper c | c <- cs]
```
(b) To restrict the list to letters, we can add a guard:
```
capitalizeLetters cs = [toUpper c | c <- cs, isAlpha c]
```

4. (a) We can put this together from library functions:
```
backwards s = unwords (reverse (words s))
```
(b) We can do this with a list comprehension:
```
revwords :: String -> String
revwords s = unwords [reverse w | w <- words s]
```

5. The candidate divisors are numbers between 1 and n, i.e. **[1..n]**. From these, we use a guard to select the actual divisors:
```
divisors n = [d | d <- [1..n], n `mod` d == 0]
```

6. We can do this by dividing the results of two library functions, **sum** and **length**. However, we have to use **fromIntegral** (from session 2) to convert the **Int** returned by **length** to a **Double** to match the type of the result of **sum**:

```
average xs = sum xs / fromIntegral (length xs)
```

7. (a) No list comprehension here, this is easy:

```
palindrome :: String -> Bool
palindrome w = reverse w == w
```

The right-hand side here is an expression denoting the Boolean value we want. You might have written the equivalent

```
palindrome w = if reverse w == w then True else False
```

or

```
palindrome w
  | reverse w == w = True
  | otherwise      = False
```

but the above is much simpler (and clearer, once you're used to it).

(b) We need to ignore non-letters and turn all letters to the same case before comparing. Fortunately we already have function to do this: **capitalizeLetters**, so we can just write

```
palindrome2 :: String -> Bool
palindrome2 w = palindrome (capitalizeLetters w)
```

8. Applying **sort** to the list brings identical characters together, and we can form them into groups by applying **group** to the sorted list. Now for each group like "aaaaa", which is equivalent to

```
['a', 'a', 'a', 'a', 'a']
```

we can extract the number of occurrences using **length** and the characters itself using **head**. (This is safe, because we know that each group has at least one element.) Doing this on each group, we get:

```
-- list of unique items in the input list, each paired
   with the
-- number of times it occurs
frequency :: [Char] -> [(Char, Int)]
frequency ws = [(head g, length g) | g <- group (sort ws)]
```

In fact, this function doesn't require characters – we can use it on any type we can compare, so we can generalize its type to

```
frequency :: Ord a => [a] -> [(a, Int)]
```

9. Using the **frequency** function defined above, we can express this idea directly:

```
palindromic :: [Char] -> Bool
palindromic xs = length [x | (x, n) <- frequency xs, odd
   n] <= 1
```

10. We use **lines** to break the string into a list of lines, and then **zip** twice, to assign x-coordinates for positions on each line, and to to assign y-coordinates for successive lines:

```
readGrid :: String -> [(Point, Char)]
readGrid s =
    [(Point x y, c) |
        (y, cs) <- zip [0,-1..] (lines s),
        (x, c) <- zip [0..] cs]
```

11. (a) **tails** returns a list containing the input list, its **tail**, the **tail** of that, and so on down to the empty list. So it produces all the suffixes of the input list, longest first. **inits** does the same for **init**, producing all the prefixes of the input list, but with the shorter lists first.

    If we apply **inits** to an infinite list, *e.g.* **inits [1..]**, we get an infinite list so we need to select the front part of it: **interpreter**

    ```
    take 8 (inits [1..])
    ```

    We can select as long a prefix as we like, and we can see that the function is producing all the prefixes of **[1..]**.

    If we apply **tails** to an infinite list, we expect an infinite list of suffixes, each of which will be infinite. To examine them, we need to trim both the outer list and each of the lists it contains, *e.g.* **interpreter**

    ```
    take 6 [take 5 xs | xs <- tails [1..]]
    ```

    Trying different cutoff points, we find that the function is producing the results we expect.

    (b) The **foo** function produces all the ways of splitting the input list into a pair of lists, with the earliest first. A descriptive name would be **splits**:

    ```
    -- all the ways of splitting a list, earliest first
    splits :: [a] -> [([a], [a])]
    splits xs = zip (inits xs) (tails xs)
    ```

    The **bar** function produces all the non-empty sublists of the input list. A descriptive name would be **sublists**:

```
-- all non-empty sublists of a list
sublists :: [a] -> [[a]]
sublists xs =
    [ys | ts <- tails xs, ys <- inits ts, not (null
      ys)]
```

12. The **sublists** function gives us all the adjacent sets of bars. For each of these, **length** gives the width of the largest contained rectangle, and **minimum** gives its height. Multiplying these gives the area of the rectangle. Then we use **maximum** to get the largest such area:

```
largestRectangle :: [Int] -> Int
largestRectangle hs =
    maximum [length l * minimum l | l <- sublists hs]
```