# Exercises

**You should check your solutions using GHCI or writing a .hs script and running it.**

**Exercise 1:**
Using list comprehension define the following list

```
[1,2,3,4,5,6]
```

**Solution 1:**

```
[x | x <- [1..6]]
```

**Exercise 2:**
Using list comprehension define the following list

```
[10,20,30,40,50,60]
```

**Solution 2:**

```
[x*10 | x <- [1..6]]
```

**Exercise 3:**
Using list comprehension define the following list

```
[(1,1),(2,2),(3,3),(4,4)]
```

**Solution 3:**

```
[(x,x) | x <- [1..4]]
```

**Exercise 4:**
Using list comprehension define the following list

```
[(1,2),(2,3),(3,4),(4,5)]
```

**Solution 4:**

```
[(x,x+1) | x <- [1..4]]
```

**Exercise 5:**

Using list comprehension define the following list (note that the second element in the 2-tuple is always 1).

```
myConstFunc = [(1,1),(2,1),(3,1),(4,1),(5,1)]
```

**Solution 5:**

```
myConstFunc :: [(Int, Int)]
myConstFunc = [(x, 1)| x <- [1..5]]
```

**Exercise 6:**

Using list comprehension define the list of tuples, containing a number (between 1 and 10 inclusive) and its square.

```
squares = [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),
           (7,49),(8,64),(9,81),(10,100)]
```

**Solution 6:**

```
squares :: [Int]
squares = [x^2 | x <- [1..10]]
```

**Exercise 7:**

Write down the values as defined in the following lists f1, f2, f3. Check your answers.

```
f1 :: [(Int, Int)]
f1 = [(x, y) | x <-[1..3], y<- [4..5]]

f2 :: [(Int, Int)]
f2 = [(x, y) | y<- [4..5], x <-[1..3]]

f3 :: [(Int, Int)]
f3 = [(y, x) | x <-[1..3], y<- [4..5]]
```

**Solution 7:**

```
l1 = [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
l2 = [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
l3 = [(4,1),(5,1),(4,2),(5,2),(4,3),(5,3)]
```

**Exercise 8:**

Given the following definition of

```
isEven :: Integer -> Bool
isEven n = (n `mod` 2 == 0)
```

Write down the values as defined in the following list: Check your solution.

```
[2*n | n <- [2,4,7], isEven n, n>3]
```

**Solution 8:**

```
[8]
```

**Exercise 9:**

Give a definition of a function

```
doubleAll :: [Integer] -> [Integer]
```

which doubles all the elements of a list of integers.

**Solution 9:**

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

**Exercise 10:**

Give a definition of a function

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

which converts all small letters in a String into capitals.
***Hint:*** You can use the following function (having imported Data.Char):

```
import Data.Char
toupper :: Char -> Char
```

**Solution 10:**

```
capitalize :: String -> String
capitalize xs = [toUpper(c) | c<- xs ]
```

**Exercise 11:**

Using a list comprehension, write a function **sigma** that calculates the sum of

$$\sum_{i=1}^{i=100} i^2$$

**Solution 11:**

```
sigma :: Int
sigma = sum [x^2 | x <- [1..100]]
```

**Exercise 12:**

Using a list comprehension,write a function **sigma'**

```
sigma' :: Int-> Int
```

that takes an integer n and calculates

$$\sum_{i=1}^{i=n} i^2$$

**Solution 12:**

```
sigma' :: Int-> Int
sigma' n = sum[x^2 | x <- [1..n]]
```

**Exercise 13:**

Define the function

```
matches :: Integer -> [Integer] -> [Integer]
```

which picks out all occurences of an integer in a list. For instance:

```
*Main> matches 1 [1,2,3,4,1]
[1,1]
*Main> matches 1 [2,3,4]
[]
*Main>
```

Using matches or otherwise (**Hint:** e.g. the **patterns** functions seen in class), define a function

```
elem':: Integer -> [Integer] -> Bool --elem is already defined in Prelude
```

which is True is the Integer is an element of the list, and False otherwise.

**Solution 13:**

```
matches :: Eq a => a -> [a] -> [a]
matches x xs = [x' | x'<- xs, x == x']

--using matches
myElem :: Eq a => a -> [a] -> Bool
myElem x xs = matches x xs /=[]

--using patterns
myElem' :: Eq a => a -> [a] -> Bool
myElem' x xs = patterns x xs /=[]
```

**Exercise 14:**

Suppose that a *coordinate grid* of size $m \quad x \quad n$ is given by the list of all pairs $(x, y)$ of integers such that $0 \leq x \leq m$ and $0 \leq y \leq n$. Using a list comprehension, define a function:

```
grid :: Int -> Int -> [(Int, Int)]
```

that returns a coordinate grid of a given size. For example:

```
*Main> grid 1 2
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
*Main>
```

**Solution 14:**

```
grid :: Int-> Int -> [(Int, Int)]
grid x y = [(x', y')| x' <- [0..x], y'<- [0..y]]
```

**Exercise 15:**

Using a list comprehension and the function **grid** above, define a function

```
square :: Int -> [(Int, Int)]
```

that returns a coordinate square of size n, excluding the diagonal from $(0, 0)$ to $(n, n)$. For example:

```
*Main> square 2
[(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)]
*Main>
```

**Solution 15:**

```
square :: Int -> [(Int, Int)]
square x = [(x', y')| x'<- [0..x], y' <- [0..x], x' /= y']
```

**Exercise 16:**

In a similar way to the function ***length***, show how the library function

```
replicate :: Int -> a -> [a]
```

that produces a list of identical elements can be defined using list comprehension. (Call your version **myReplicate**) For example:

```
*Main> myReplicate 3 True
[True,True,True]
```

**Solution 16:**

```
myReplicate :: Int -> a -> [a]
myReplicate x y = [y| _ <- [1..x]]
```

**Exercise 17:**

A triple $(x, y, z)$ of positive integers is called pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int -> [(Int,Int,Int)]
```

that returns a list of all such triples whose components are at most a given limit. For example

```
*Main> pyths 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
*Main>
```

**Solution 17:**

```
pyths ::Int -> [(Int, Int, Int)]
pyths n = [(x, y,z) | x <- [1..n], y<- [1..n], z <- [1..n], x^2 + y^2
    == z^2]
```

**Exercise 18:**

A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension and the function **factors**, define a function

```
perfects :: Int -> [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
*Main> perfects 500
[6,28,496]
*Main>
```

**Hint:** Note that the list of factors of x includes x...

**Solution 18:**

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x ==0]

perfects :: Int -> [Int]
perfects n = [x | x <- [1..n], x == sum (factors x) - x]

perfects' :: Int -> [Int]
perfects' n = [x | x <- [1..n], x == sum ( init (factors x))]
```