Exercises 1

Please, write your solutions on a new Haskell file. Rename it to Ex1.hs and include a comment at the beginning of the file filling in the following information:

```
-------------------------------------------------------------------------------
-- Data Structures. ETSI Informática. UMA
--
-- (please, fill in the following form)
-- Degree: Grado en Ingeniería ........................ [Informática | del Software | de Computadores].
-- Student: SURNAME, NAME
-- Date:  DAY | MONTH | YEAR
--
-- Exercises 1. I solved the following exercises: .........
--
-------------------------------------------------------------------------------

import Test.QuickCheck
```

1. Three positive integer values (*x, y, z)* are a Phytagoran triple if $x^2+y^2=z^2$, that is to say, if they correspond to lengths for sides of a triangle.

a) Define a function

```
isTriple :: Integer -> Integer -> Integer -> Bool
```

to test whether three values are a Phytagorean triple. For instance:

```
Main> isTriple 3 4 5        Main> isTriple 3 4 6
True                        False
```

b) For any *x* and *y* positive integers such that $x>y$, triple (x^2-y^2, $2xy$, x^2+y^2) is Phytagorean. Accordingly, write a function named `triple` taking two numbers and returning a Phytagorean triple. For instance:

```
 Main> triple 3 1
(8,6,10)
 Main> isTriple 8 6 10
True
```

c) Read and understand the following property, that states that all triples generated by `triple` function are Phytagorean:

```
p_triples x y = x>0 && y>0 && x>y ==> isTriple l1 l2 h
  where
    (l1,l2,h) = triple x y
```

d) Check this property using *QuickCheck* (recall to import `Test.QuickCheck` at the beginning of your program and to copy the property into your file). You should observe a result similar to the following one:

```
Main> quickCheck p_triples
*** Gave up! Passed only 62 tests
```

Which means that, although only 62 test cases fulfilling the precondition where generated by *QuickCheck*, all of them passed the test.

2. Define a polymorphic function

```
swap :: (a,b) -> (b,a)
```

for swapping components in a two components tuple:

```
Main> swap (1,True)                Main> swap ('X','Y')
(True,1)                           ('Y','X')
```

3. This exercise is on sorting tuples.

a) Define an overloaded function for ordered types

```
sort2 :: Ord a => (a,a) -> (a,a)
```

taking a two components tuple (whose elements have the same type) and returning the corresponding ascending sorted tuple:

```
Main> sort2 (10,3)                 Main> sort2 ('a','z')
(3,10)                             ('a','z')
```

b) Copy the following properties for function `sort2` into your script:

```
p1_sort2 x y = sorted (sort2 (x,y))
  where sorted (x,y) = x<=y

p2_sort2 x y = sameElements (x,y) (sort2 (x,y))
   where
      sameElements (x,y) (x’,y’) = (x==x’ && y==y’) || (x==y’ && y==x’)
```

Understand each of these properties and test them using *QuickCheck*.

c) Define an overloaded function

```
sort3 :: Ord a => (a,a,a) -> (a,a,a)
```

taking a tuple with three components of the same type and returning it sorted in ascending order:

```
Main> sort3 (10,3,7)
(3,7,10)
```

d) Write similar properties to the ones in paragraph b) but for `sort3` function, and test them using *QuickCheck*.

4. Although there already exists a predefined function (`max :: Ord a => a -> a -> a`) returning the maximum of two values, in this exercise you will define your own version.

a) As it is not allowed to define a function whose name coincides with a predefine done, define a function `max2 :: Ord a => a -> a -> a` such that:

```
Main> 10 `max2` 7                  Main> max2 'a' 'z'
10                                 'z'
```

b) Define the following properties that function `max2` should fulfill and test them using *QuickCheck* (recall to import `Test.QuickCheck` at the beginning of your program):

i. `p1_max2`: maximum of *x* and *y* is either *x* or either *y*.
ii. `p2_max2`: maximum of *x* and *y* is greater than or equal to *x* and greater or equal than *y.*
iii. `p3_max2`: if *x* is greater than or equal to *y*, then maximum of *x* and *y* is *x*.
iv. `p4_max2`: if *y* is greater than or equal to *x*, then maximum of *x* and *y* is *y*.

5. Define an overloaded function for ordered types

```
between :: Ord a => a -> (a,a) -> Bool
```

taking a value *x* and a tuple with two components (*max*,*min*) and testing whether *x* is in the interval determined by *min* and *max*, i.e., if *x* ∈ [*min*,*max*]. For instance:

```
Main> 5 `between` (1,10)                Main> between 'z' ('a','d')
True                                    False
```

6. Define an overloaded function for types with equality

```
equals3 :: Eq a => (a,a,a) -> Bool
```

taking a tuple with three components of the same type and returning `True` if all of them are equal. For instance:

```
Main> equals3 ('z','a','z')
False
Main> equals3 (5+1,6,2*3)
True
```

7. Recall that the quotient and the modulo for an integer division can be calculated by using predefined functions `div` and `mod`.

a) Define a function named `decompose` taking an integer value representing seconds, and returning a three components tuple with corresponding hours, minutes and seconds, so that returned minutes and seconds are in the range 0 to 59. For instance:

```
decompose 5000 => (1,23,20)          decompose 100 => (0,1,40)
```

Define this function by filling in the following outline:

```
type Hour       = Integer
type Minute     = Integer
type Second     = Integer
decompose :: Second -> (Hour,Minute,Second)
decompose x = (hours, minutes, seconds)
  where
    hours = ...
    ...
```

b) Read and test the following property in order to check correctness for your function:

```
p_decompose x = x>=0 ==> h*3600 + m*60 + s == x
                         && between m (0,59)
                         && between s (0,59)
  where (h,m,s) = decompose x
```

8. Let us consider the following definition to represent that one euro is 166.386 pesetas:

```
oneEuro :: Double
oneEuro = 166.386
```

a) Define a function named `pesetasToEuros` changing an amount of pesetas (expressed as a `Double`) into corresponding amount of euros. For instance:

```
pesetasToEuros 1663.86 => 10.0
```

b) Define a function named `eurosToPesetas` changing euros into pesetas. For instance:

```
eurosToPesetas 10  => 1663.86
```

c) Let us consider the following property, stating that by changing pesetas into euros and then changing back those euros into pesetas, we should obtain the original amount of pesetas (i.e., those are inverse functions):

```
p_inverse x = eurosToPesetas (pesetasToEuros x) == x
```

Test this property with *QuickCheck* and check that it does not hold. Why? (**hint**: these functions are defined on floating point numbers).

9. Let us consider the following operator to test whether two `Double` values are approximately equal:

```
infix 4 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
 where epsilon = 1/1000
```

For instance:

```
(1/3) ~= 0.33 => False            (1/3) ~= 0.333 => True
```

Copy this operator definition into your file, and restate property p_inverse so that it holds. Test that your new definition for the property does indeed hold using *QuickCheck*.

10. Let us consider the quadratic equation $ax^2 + bx + c = 0$.

a) Define a function named `roots` taking three arguments (corresponding to equation coefficients *a*, *b* and *c*) and returning a tuple with two real solutions to the equation (you can use predefined function `sqrt` in order to compute the square root of a floating number). Recall that the discriminant is defined as b^2-4*ac* and that a quadratic equation has real solutions if the corresponding discriminant is non-negative. For instance:

```
roots 1 (-2) 1.0 => (1.0,1.0)
roots 1.0 2 4 => Exception: Non real roots
```

b) Let us consider the following property to test that values returned by `roots` are indeed roots of the equation:

```
p1_roots a b c  = isRoot r1 && isRoot r2
  where
   (r1,r2) = roots a b c
   isRoot r = a*r^2 + b*r + c ~= 0
```

Test this property using *QuickCheck* and check that it fails. Find out the reason, and add appropriate preconditions so that it does not fail by filling in interrogations in the following outline:

```
p2_roots a b c  = ??????? && ?????? ==> isRoot r1 && isRoot r2
  where
    (r1,r2) = roots a b c
    isRoot r = a*r^2 + b*r + c ~= 0
```

so that:

```
Main> quickCheck p2_roots
+++ OK, passed 100 tests
```

11. Define an overloaded function named `isMultiple` over integral numbers taking to values *x* and *y*, and returning `True` if *x* is a multiple of *y*. For example:

```
isMultiple 9 3 => True            isMultiple 7 3 => False
```

12. Define a left associative logical implication operator `(==>>) :: Bool -> Bool -> Bool` so that its precedence is lower than the one corresponding to conjunction and disjunction operators:

```
Main> 3 < 1  ==>> 4 > 2
True
Main> 3 < 1 || 3 > 1  ==>> 4 > 2 && 4 < 2
False
```

Hint: you can directly define it using a single equation:

```
???   ==>>  ???  =  ???
```

Or you can define it by using multiple equations and patterns:

```
False ==>>  y    =  y
```

13. Leap years are those which are multiple of 4. One exception to the rule are those years that are multiples of 100, which are only considered leap if they are additionally multiple of 400. Define a function named `isLeap` taking a year and returning `True` if it is a leap year. For example:

```
isLeap 1984 => True          isLeap 1985 => False
isLeap 1800 => False         isLeap 2000 => True
```

Hint: use the logical implication operator defined in previous exercise in order to represent the following sentence: *"n* is a leap year if it fulfills the following two conditions: (a) it is a multiple of 4, and (b) if *n* is a multiple of 100 then *n* should be a multiple of 400".

14. Although there already exists a predefined operator (^) in Haskell to compute raising of a number to a power, the aim of this exercise is that you define your own version of this operator.

a) By using the following property

$b^n = b \cdot b^{n-1}$

define a recursive `power` function taking an integer *b* and a natural exponent *n* and returning b^n. For example:

```
power 2 3 => 8
```

b) Let us now consider the following property:

$$b^n = \begin{cases} \left(b^{\frac{n}{2}}\right)^2, & \text{if } n \text{ is even} \\ b \cdot b^{n-1}, & \text{if } n \text{ is odd} \end{cases}$$

Turn this property into a new recursive definition, and accordingly, define a recursive `power'` function taking an integer *b* and a natural exponent *n* and returning b^n (don't use previous `power` function in your new definition). For example:

```
power' 2 3 => 8                  power' 2 4 => 16
```

c) Test using *QuickCheck* correctness of both functions by means of the following property:

```
p_power b n = n>=0 ==> power b n == sol
                        && power' b n == sol
  where sol = b^n
```

d) Assuming that doing a square operation corresponds to a doing product, determine the number of products that both functions do in order to raise some base to an exponent *n*.

Hint: in order to analyze efficiency for function `power'`, consider first exponents that are power of 2.

15. For a set whose elements are all different, a permutation of it is each possible rearrangement of its elements. For instance, for set {1,2,3}, there exists 6 different permutations of its elements: {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2} y {3,2,1}. For a set with *n* elements, the number of permutations is the factorial of *n* (usually written as *n*!), and defined as the product of natural numbers from 1 to *n*. Define a `factorial` function taking a natural number and returning its factorial. As factorial function grows very quickly, use `Integer` type in order to avoid overflows. For instance:

```
factorial 3 => 6            factorial 20 => 2432902008176640000
```

16. This exercise studies integer division of numbers.

a) Define `divides` function taking two numbers and testing whether its first argument exactly divides its second one. For instance:

```
2 `divides` 10 =>  True    4 `divides` 10 => False
```

b) Read, understand and test using *QuickCheck* the following property for `divides` function:

```
p1_divides x y =  y/=0 && y `divides` x ==> div x y * y == x
```

c) Define a property `p2_divides` to test using *QuickCheck* that if a number exactly divides two numbers, it also exactly divides their sum.

17. The median of a set of numbers is the value for which half the numbers are larger and half are smaller. Define a function to compute the median for a tuple with 5 components

```
median :: Ord a => (a,a,a,a,a) -> a
```

so that, for instance, `median (3,20,1,10,50) => 10`

Observe that 1,3 ≤ 10 ≤ 20,50. According to this observation, define this function using guards by filling in the following outline:

```
median (x,y,z,t,u)
       | x > z   = median (z,y,x,t,u)
       | y > z   = median (x,z,y,t,u)
       . . .
```