IN3043 Functional Programming
Solutions to Exercises 2

1. The function takes three arguments, so we make up names for them (**x**, **y** and **z**), and build the required Boolean expression from them:

```
threeDifferent :: Int -> Int -> Int -> Bool
threeDifferent x y z  =  x /= y && y /= z && x /= z
```

   An alternative definition is

```
threeDifferent x y z  =  not (x == y || y == z || x == z)
```

   This is equivalent to the previous version, by de Morgan's law.

2. It returns **True** if its arguments are not all the same. (So this is subtly different from the previous question.)

   This is because **x == y && y == z** is **True** if all three arguments are equal to each other, and the function returns the negation of that.

3. We can use de Morgan's law:

```
mystery :: Int -> Int -> Int -> Bool
mystery x y z = x /= y || y /= z
```

4. Floating point numbers are represented as binary fractions of limited precision. Just as you can't represent one third as an exact decimal fraction, binary fractions and cannot represent decimal fractions exactly. The closest approximation is chosen, but it is off by a tiny amount. Sometimes these errors cancel out, but often they accumulate, resulting in the tiny difference between the two numbers here.

   You would get the same effect in Java or C. The moral of the story is that you should be very careful about comparing floating point numbers for equality, in any language.

5. We can use **floor** to get nearest integer below **x**, so we want to write **x - floor x**. However this isn't allowed, because the minus function needs two arguments of the same type, but **x** has type **Double** and **floor x** has type **Integer**, and Haskell doesn't convert types without us asking it to. So we need to convert the **Integer** to **Double**, which we can do with the function **fromIntegral**:

```
fractional :: Double -> Double
fractional x = x - fromIntegral (floor x)
```

6. We can obtain a version using guards directly from the description:

```
clamp :: Double -> Double -> Double -> Double
clamp lo hi x
  | lo <= x && x <= hi = x
  | x < lo = lo
  | x > hi = hi
```

Since these three guards are mutually exclusive, we can arrange them in any order, and replace the last one by **otherwise**. Here I've chosen to replace the most complex guard:

```
clamp :: Double -> Double -> Double -> Double
clamp lo hi x
  | x < lo = lo
  | x > hi = hi
  | otherwise = x
```

Alternatively, we could define the function more concisely:

```
clamp :: Double -> Double -> Double -> Double
clamp lo hi x = max lo (min x hi)
```

7. We need a function mapping from **Char** to **Int**. The **ord** function does this, but it gives the wrong **Int**. Fortunately the codes for digits are contiguous and in order, so we can get the correct number by subtracting the code for '0'.

```
import Data.Char

charToNum :: Char -> Int
charToNum c
  | isDigit c   = ord c - ord '0'
  | otherwise   = 0
```

8. (a) This is an enumerated type:

```
data Direction = North | South | East | West
    deriving Show
```

We've added **deriving Show** so that we can display these values.

(b) We need four equations, one for each input value:

```
turnLeft North = West
turnLeft South = East
turnLeft East = North
turnLeft West = South
```

(c) This function has the same structure:

```
turnRight :: Direction -> Direction
turnRight North = East
turnRight South = West
turnRight East = South
turnRight West = North
```

9. (a) There are several possible answers. One is

```
isLeapYear :: Int -> Bool
isLeapYear y
  | y `mod` 400 == 0 = True
  | y `mod` 100 == 0 = False
  | y `mod` 4   == 0 = True
  | otherwise        = False
```

Note that putting the clauses in a different order would be different (and give the wrong answer).

We could reduce that to

```
isLeapYear y
  | y `mod` 400 == 0 = True
  | y `mod` 100 == 0 = False
  | otherwise        = y `mod` 4 == 0
```

or even

```
isLeapYear y = y `mod` 400 == 0 ||
    y `mod` 100 /= 0 && y `mod` 4 == 0
```

(b) It is easiest to use the function we've just defined:

```
daysInYear :: Int -> Int
daysInYear y
  | isLeapYear y  = 366
  | otherwise     = 365
```

(c) We define an enumerated type listing the twelve months:

```
data Month
    = Jan | Feb | Mar | Apr | May | Jun
    | Jul | Aug | Sep | Oct | Nov | Dec
    deriving Show
```

(d) Our function has a case for each month, except that there are two cases for February, depending on whether or not the year is a leap year:

```
daysInMonth :: Month -> Int -> Int
daysInMonth Jan y = 31
daysInMonth Feb y
  | isLeapYear y  = 29
  | otherwise     = 28
daysInMonth Mar y = 31
daysInMonth Apr y = 30
daysInMonth May y = 31
daysInMonth Jun y = 30
daysInMonth Jul y = 31
daysInMonth Aug y = 31
daysInMonth Sep y = 30
daysInMonth Oct y = 31
daysInMonth Nov y = 30
daysInMonth Dec y = 31
```

We could shorten this a bit by observing that the most common result is **31**:

```
daysInMonth :: Month -> Int -> Int
daysInMonth Feb y
  | isLeapYear y  = 29
  | otherwise     = 28
daysInMonth Apr y = 30
daysInMonth Jun y = 30
daysInMonth Sep y = 30
daysInMonth Nov y = 30
daysInMonth m   y = 31
```