- TD -
Caesar code

A well-known method for encoding a string is the *Caesar cipher*. To encode a string, Caesar simply replaced each letter in the string by the letter 3 places further down in the alphabet, wrapping around at the end of the alphabet. For example, the string "haskell is fun" is encoded as "kdvnhoo lv ixq". The shift factor of 3 used by Caesar can be replaced by any integer between 1 and 25, giving twenty-five different ways of encoding a string.

The goal here is to define functions implementing the Caesar cipher, and respectively cracking a string encoded using the cipher. For simplicity, we encode only lower case letters, the other characters being left unchanged.

This exercise is strongly inspired from the one proposed in Chapter 5 of the book Programming in Haskell, Graham Hutton, Cambridge University Press, 2016.

1. Auxiliary functions

Define a function `let2int` which converts a lower-case letter between `'a'` and `'z'` into the corresponding integer between 0 and 25

```
let2int :: Char -> Int
```

together with a function int2let that performs the opposite conversion:

```
int2let :: Int -> Char
```

For example, `let2int 'c'` produces `2` and `int2let 4` produces `'e'`.

Hint: you can use the function `ord` to convert `Char` into `Int` (`import Data.Char`) and the function `chr` to convert `Int` into `Char`.

Define a function `rotate` which rotates the elements of a list n places to the left, wrapping around at the start of the list, and assuming that n is between zero and the length of the list:

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

For example, `rotate 3 [1..5]` produces `[4,5,1,2,3]`.

2. Encode strings

Define the function `encode` that encodes a string using a given shift factor:

```
encode :: Int -> String -> String
```

For example, `encode 3 "james bond is the best in town"` produces `"mdphv erqg lv wkh ehvw lq wrzq"`.

3. Code cracker

The key to cracking the Caesar cipher is the observation that some letters are used more frequently than others in English text. By analysing a large volume of text, one can derive the following table of approximate percentage frequencies of the 26 letters of alphabet:

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2,
0.8, 4.0, 2.4, 6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0,
2.4, 0.2, 2.0, 0.1]
```

We use a function that calculates the percentage of one integer with respect to another, returning the result as a floating-point number:

```
percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

Using `percent` together with the functions `lowers` and `count` from the course, define a function that returns a frequency table for any string:

```
freqs :: String -> [Float]
```

For example, `freqs "james bond is the best in town"` produces `[4.166667,8.333334,0.0,4.166667,12.5,0.0,0.0,4.166667,8.333334,4.166667,0.0,0.0,4.166667,12.5,8.333334,0.0,0.0,0.0,12.5,12.5,0.0,0.0,4.166667,0.0,0.0,0.0]`.

A standard method for comparing a list of observed frequencies *os* with a list of expected frequencies *es* is the *chi-square* statistic, defined by the following summation in which *n* denotes the length of the two lists, and xs_i denotes the i-th element of a list *xs*:

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

The smaller the value it produces the better the match between the two frequency lists. Translate the above formula into a function definition:

```
chisqr :: [Float] -> [Float] -> Float
```

To decode a string we can produce the frequency table of the encoded string, calculate the chi-square statistic for each possible rotation of this table with respect to the table of expected frequencies, and using the position of the minimum chi-square value as the shift factor. Define a function which implements this behaviour:

```
crack :: String -> String
```

Hint: you can use the function `positions` defined in the course.