The Caesar Cypher

Graham Hutton

February 4, 2019

Caesar Cipher

Example of string encoding with constant shift factor of 3 . . .

- "abc " would be encoded to "def"
- "haskell is fun" would be encoded to "kdnnhoo lv ixq"

More Generally

So, more generallly with a shift factor of 4, for example:

"abc" would be encoded by "efg"

How will we use Haskell to implement the Caesar and more . . .

Caesar Cipher

Example of string encoding with constant shift factor of 3 . . .

- "abc " would be encoded to "def"
- "haskell is fun" would be encoded to "kdnnhoo lv ixq"

More Generally

So, more generallly with a shift factor of 4, for example:

"abc" would be encoded by "efg"

How will we use Haskell to implement the Caesar and more . . .

Encoding and Decoding

```haskell
import Data.Char    -- imports standard functi
```

For simplicity, we will only encode the lower-case characters within a string and leave the other characters unchanged. Firstly

```haskell
let2Int :: Char -> Int
let2Int c = ord c - ord 'a'

int2Let :: Int -> Char
int2Let n = chr (ord 'a' + n)
```

```
*Main> let2int 'a'
0
*Main> int2let 0
'a'
```

Encoding and Decoding

```
import Data.Char    -- imports standard functic
```

For simplicity, we will only encode the lower-case characters within a string and leave the other characters unchanged.
Firstly

```
let2Int :: Char -> Int
let2Int c = ord c - ord 'a'

int2Let :: Int -> Char
int2Let n = chr (ord 'a' + n)
```

```
*Main> let2int 'a'
0
*Main> int2let 0
'a'
```

Encoding and Decoding contd.

We define a function *shift*
as follows:

- ```
  shift :: Int -> Char -> Char
  shift n c    | isLower c = int2let ( (let2in
                 | otherwise = c
  ```

  (The library function

  isLower :: Char -> Bool

  returns True if it's a lower-case letter.)
  ```

Encoding and Decoding contd.

Using *shift* within a list comprehension, it is now easy to define a function that encodes a string using a given string factor.

```haskell
encode :: Int -> String -> String
encode n xs = [shift n x | x <- xs]
```

We call this as follows:

```
*Main> encode 3 "haskell is fun"
"kdvnhoo lv ixq"
*Main> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

Frequency Tables

```
table :: [Float]
table = [8.1, 1.5, 2.8, 4.1, 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, 9.0, 2.8,
         1.0, 2.4, 0.2, 2.0, 0.1]
```

we define a percent function

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

Frequency Tables cont.

We now look at producing a frequency table for a string. We use *count* and *lowers* as follows:

```
count :: Eq a => a-> [a] -> Int
count x xs = length [ x' | x' <- xs, x==x']

lowers :: [Char] -> Int
lowers xs =
    length [x| x <- xs,
               x >= 'a' && x <= 'z']
```

Frequency Tables cont.

```haskell
freqs :: String -> [Float]
freqs xs = [percent (count x xs) n |
                          x <- ['a'..'z']]
      where n = lowers xs
```

when called . . .

Frequency Tables cont.

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.

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

Frequency Tables cont.

Using *zip* and list comprehension we translate the previous formula into code

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o-e)^2)/e |
                        (o,e) <- zip os es]
```

We will continue to see how this cracks the code.. Now, we define a function that rotates the elements of a list *n* places the left, wrapping around the start of the list, and assuming that the integer arguments *n* is between 0 and the *length* of the list

```
rotate : Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs
```

We will continue to see how this cracks the code..