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
With a shift factor of 4, for example:

- "abc " would be encoded to "def"

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

With a shift factor of 4, for example:

- "abc " would be encoded to "def"

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)
```

We can see them called in Figure 1

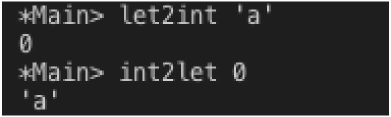

Figure: Calling int2let and let2int

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)
```

We can see them called in Figure 1

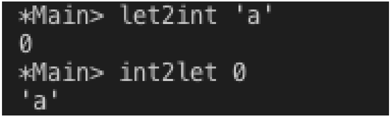

Figure: Calling int2let and let2int

Encoding and Decoding contd.

We define a function *shift*
as follows:

- ```
  shift :: Int -> Char -> Char
  shift n c  | isLower c = int2let (
                 (let2int c + n ) `mod` 26)
               | 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.

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

We call this as shown in Fig 2

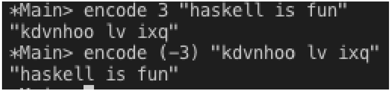

Figure: Calling encode with positive and negative values

Frequency Tables

Table of approximate percentage frequencies of the twenty-six letters of the alphabet :

```
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.

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

We can see how it's called in Fig 3

Figure: Calling freqs on a string

the letter 'a' occurs with a frequency of approximately 6.6%, the letter 'b' with a frequency of 13.3% etc.

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]
```

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
```

Now, suppose that we are given an encoded string, but not the shift factor that was used to encode it, and wish to determine this number in order that we can decode the string. This can usually be achieved by producing the frequency table of the encoded string, calculating the chi-square statistic for each possible rotation of the table with respect to the table of expected frequencies, and using the position of the minimum chi-square value as the shift factor. For example, if we let table

```
table' = freqs "kdvnhoo␣lv␣ixq"
```

```
[chisqr (rotate' n table') table | n <- [0
will give us

 [1409.1558,639.92175,612.2969,
 202.32024, 1440.2488, 4247.621, 650.8992
..]
```

```
crack :: String -> String
crack xs = encode (-factor) xs
    where
     factor = head (positions
          ( minimum chitab) chitab )
     chitab = [chisqr (rotate' n table') table
              n <- [0..25]]
     table' = freqs xs
```

For example:

```
crack "kdvnhoo lv ixq"
```

```
"haskell is fun"
```