The Caesar Cipher

Informatics 1 – Functional Programming: Tutorial 3

Heijltjes, Wadler

Due: The tutorial of week 5 (23/24 Oct.)
Reading assignment: Chapters 8 and 9 (pp. 135-166)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

The Caesar Cipher

When we talk about cryptography these days, we usually refer to the encryption of digital messages, but encryption actually predates the computer by quite a long period. One of the best examples of early cryptography is the Caesar cipher, named after Julius Caesar because he is believed to have used it, even if he didn't actually invent it. The idea is simple: take the message you want to encrypt and shift all letters by a certain amount between 0 and 26 (called the *offset*). For example: encrypting the sentence "THIS IS A BIG SECRET" with shifts of 5, would result in "YMNX NX F GNL XJHWJY".

In this exercise you will be implementing a variant of the Caesar cipher.

Encrypting text

A character-by-character cipher such as a Caesar cipher can be represented by a *key*, a list of pairs. Each pair in the list indicates how one letter should be encoded. For example, a cipher for the letters A–E could be given by the list `[('A', 'C'), ('B', 'D'), ('C', 'E'), ('D', 'A'), ('E', 'B')]`. Although it's possible to choose any letter as the ciphertext for any other letter, this tutorial deals mainly with the type of cipher where we encipher each letter by shifting it the same number of spots around a circle, for the whole English alphabet.

Exercises

1. We can rotate a list by taking some items off the front of it and putting them on the end. For example:

```
Main> rotate 3 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"DEFGHIJKLMNOPQRSTUVWXYZABC"
```

Write the function `rotate :: Int -> [Char] -> [Char]`. Your function should return an error (using the library function `error`) if the number is too small (less than 0) or too large (greater than *the length of the list*).

2. Look at the test function `prop_rotate`.

   (a) What precisely does it test?

   (b) How does it make sure that `rotate` does not produce an error?

3. Using the function `rotate` from the previous question, write a function

   ```
   makeKey :: Int -> [(Char, Char)]
   ```

   that returns the cipher key with the given offset. See above for the description of how the cipher key is represented as a list of pairs. Example:

   ```
   Main> makeKey 5
   [('A','F'),('B','G'),('C','H'),('D','I'),('E','J'),('F','K'),
    ('G','L'),('H','M'),('I','N'),('J','O'),('K','P'),('L','Q'),
    ('M','R'),('N','S'),('O','T'),('P','U'),('Q','V'),('R','W'),
    ('S','X'),('T','Y'),('U','Z'),('V','A'),('W','B'),('X','C'),
    ('Y','D'),('Z','E')]
   ```

   The cipher key should show how to encrypt all of the uppercase English letters, and there should be no duplicates: each letter should appear just once amongst the pairs' first components (and just once amongst the second components).

4. Write a function

   ```
   lookUp :: Char -> [(Char, Char)] -> Char
   ```

   that finds a pair by its *first* component and returns that pair's *second* component. When you try to look up a character that does not occur in the cipher key, your function should leave it unchanged. Examples:

   ```
   Main> lookUp 'B' [('A', 'F'), ('B', 'G'), ('C', 'H')]
   'G'
   Main> lookUp '9' [('A', 'X'), ('B', 'Y'), ('C', 'Z')]
   '9'
   ```

5. Write a function

   ```
   encipher :: Int -> Char -> Char
   ```

   that encrypts the given single character using the key with the given offset. For example:

   ```
   Main> encipher 5 'C'
   'H'
   Main> encipher 7 'Q'
   'X'
   ```

6. Text encrypted by a cipher is conventionally written in uppercase and without punctuation. Write a function

   ```
   normalize :: String -> String
   ```

   that converts a string to uppercase, removing all characters other than letters and digits (remove spaces too). Example:

   ```
   Main> normalize "July 4th!"
   "JULY4TH"
   ```

7. Write a function

   ```
   encipherStr :: Int -> String -> String
   ```

that normalizes a string and encrypts it, using your functions `normalize` and `encipher`. Example:

```
Main> encipherStr 5 "July 4th!"
"OZQD4YM"
```

Decoding a message

The Caesar cipher is one of the easiest forms of encryption to break. Unlike most encryption schemes commonly in use today, it is susceptible to a simple brute-force attack of trying all the possible keys in succession. The Caesar cipher is a *symmetric key* cipher: the key has enough information within it to use it for encryption as well as decryption.

Exercises

8. Decrypting an encoded message is easiest if we transform the key first. Write a function

```
reverseKey :: [(Char, Char)] -> [(Char, Char)]
```

to reverse a key. This function should swap each pair in the given list. For example:

```
Main> reverseKey [('A', 'G'), ('B', 'H') , ('C', 'I')]
[('G', 'A'), ('H', 'B') , ('I', 'C')]
```

9. Write the functions

```
decipher :: Int -> Char -> Char
decipherStr :: Int -> String -> String
```

that decipher a character and a string, respectively, by using the key with the given offset. Your function should leave digits and spaces unchanged, but remove lowercase letters and other characters. For example:

```
Main> decipherStr 5 "OZQD4YM"
"JULY4TH"
```

More QuickCheck tricks

To test the `rotate` function we had to make sure that the test function did not generate any errors. The input, randomly generated by QuickCheck, had to obey certain criteria—you found out which in exercise (2).

In the test `prop_rotate` we made sure the input was of the right kind by *changing* it. But this is not always the best solution, and sometimes it is not even possible. A more general way to ensure the input of a function has a certain property, is to use an *implication* '`==>`'.

The QuickCheck implication is a lot like a logical implication. It takes two Boolean expressions as arguments, for example `expr1` and `expr2` (its resulting type is called `Property`):

```
expr1, expr2 :: Bool

prop_test :: Property
prop_test = expr1 ==> expr2
```

In general, the property described above holds if `expr1` is `False` or `expr2` is `True`. However, to make sure that all tests are *relevant*, QuickCheck ignores the test if `expr1` is `False`, and only counts the tests in which both `expr1` and `expr2` are `True`:

```
*Main> quickCheck (True ==> True)
OK, passed 100 tests.
*Main> quickCheck (False ==> True)
Arguments exhausted after 0 tests.
```

As you can see, QuickCheck does not continue to generate values forever; if after a certain amount of tests `expr2` still isn't `True`, it will stop with the message '*arguments exhausted*'.

Exercises

10. To see if your encryption works, write a QuickCheck test `prop_cipher` to verify that decoding an encoded string with the same key returns the original message — but then in uppercase and without spacing or punctuation ("normalized"). Use '`==>`' to make sure your test doesn't generate any errors.

Breaking the encryption

One kind of brute-force attack on an encrypted string is to decrypt it using each possible key and then search for common English letter sequences in the resulting text. If such sequences are discovered then the key is a candidate for the actual key used to encrypt the plaintext. For example, the words "the" and "and" occur very frequently in English text: in the *Adventures of Sherlock Holmes*, "the" and "and" account for about one in every 12 words, and there is no sequence of more than 150 words without either "the" or "and".

The conclusion to draw is that if we try a key on a sufficiently long sequence of text and the result does not contain any occurrences of "the" or "and" then the key can be discarded as a candidate.

Exercises

11. Write a function

    ```
    candidates :: String -> [(Int, String)]
    ```

    that decrypts the input string with each of the 26 possible keys and, when the decrypted text contains "THE" or "AND", includes the decryption key and the text in the output list.

    ```
    Main> candidates "DGGADBCOOCZYMJHZYVMTOJOCZHVS"
    [(5,"YBBVYWXJJXUTHECUTQHOJEJXUCQN"),
    (14,"PSSMPNOAAOLKYVTLKHYFAVAOLTHE"),
    (21,"ILLFIGHTTHEDROMEDARYTOTHEMAX")]
    ```

    Hint: Use the function `contains` you defined in Tutorial 2.

Appendix: Utility function reference

Note: To use these functions you'll need to add this line to the top of your program:
`import Char`

`ord :: Char -> Int` (Thompson p. 42)
Return the numerical code corresponding to a character
Examples: `ord 'A' == 65` `ord '1' == 49`
See also: `http://www.zvon.org/other/haskell/Outputchar/ord_f.html`

`chr :: Int -> Char` (Thompson p. 42)
Return the character corresponding to a numerical code
Examples: `chr 65 == 'A'` `chr 49 == '1'`
See also: `http://www.zvon.org/other/haskell/Outputchar/chr_f.html`

`mod :: Int -> Int -> Int` (Thompson p. 36)
Return the remainder after the first argument is divided by the second
Examples: `mod 10 3 == 1` `mod 25 5 == 0`
See also: `http://www.zvon.org/other/haskell/Outputprelude/mod_f.html`

`isAlpha :: Char -> Bool`
Return `True` if the argument is an alphabetic character
Examples: `isAlpha '3' == False` `isAlpha 'x' == True`
See also: `http://www.zvon.org/other/haskell/Outputchar/isAlpha_f.html`

`isUpper :: Char -> Bool`
Return `True` if the argument is an uppercase letter
Examples: `isUpper 'x' == False` `isUpper 'X' == True`
See also: `http://www.zvon.org/other/haskell/Outputchar/isUpper_f.html`

`isLower :: Char -> Bool`
Return `True` if the argument is a lowercase letter
Examples: `isLower '3' == False` `isLower 'x' == True`
See also: `http://www.zvon.org/other/haskell/Outputchar/isLower_f.html`

`toUpper :: Char -> Char` (Thompson p. 42)
If the argument is an alphabetic character, convert it to upper case
Examples: `toUpper 'x' == 'X'` `toUpper '3' == '3'`
See also: `http://www.zvon.org/other/haskell/Outputchar/toUpper_f.html`

`isPrefixOf :: String -> String -> Bool`
Return `True` if the first list argument is a prefix of the second
Examples: `isPrefix "has" "haskell" == True` `isPrefix "has" "handle" == False`
See also: `http://www.zvon.org/other/haskell/Outputlist/isPrefixOf_f.html`

`error :: String -> a` (Thompson p. 61)
Signal an error
Examples: `error "Function only defined on positive numbers!"`
See also: `http://www.zvon.org/other/haskell/Outputprelude/error_f.html`