Recursion & the Caesar Cipher

Informatics 1 – Functional Programming: Tutorial 2

Due: The tutorial of week 4 (12-13 Oct.)

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.

Recursion

In this first part we revisit several functions from last week's tutorial. You will be asked to reimplement them using *recursion* instead of *list comprehensions*.

To test your new recursive functions you should compare them against your implementation from last week and/or against the version from the solutions posted on the course website.

Exercises

1. (a) Write a function `halveEvensRec :: [Int] -> [Int]` that returns half of each even number in the given list. For example,

   ```
   halveEvensRec [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
   ```

   Your definition should use *recursion*, not list comprehension. You may use the functions `div, mod :: Int -> Int -> Int`.

   (b) To confirm your function is correct, write a test function `prop_halveEvens` that tests that `halveEvensRec` returns the same result as `halveEvens` from last week. You can use your implementation of `halveEvens` from last week, or the solution from the course website, or both. You might need to change the name of one of them. You can write more than one test function or combine two properties with `&&`.

   Run your test function(s) using QuickCheck.

2. (a) Write a function `inRangeRec :: Int -> Int -> [Int] -> [Int]` to return all numbers in the input list within the range given by the first two arguments (inclusive). For example,

   ```
   inRangeRec 5 10 [1..15]  ==  [5,6,7,8,9,10]
   ```

   Your definition should use *recursion*.

(b) To confirm your function is correct, write a test function `prop_inRange` that tests that `inRangeRec` returns the same result as `inRange` from last week. You can use your implementation of `inRange` from last week, or the solution from the course website, or both. You might need to change the name of one of them. You can write more than one test function or combine two properties with `&&`.

Run your test function(s) using QuickCheck.

3. (a) Write a function `countPositivesRec` to count the positive numbers in a list (the ones strictly greater than 0). For example,

```
countPositivesRec [0,1,-3,-2,8,-1,6]  ==  3
```

Your definition should use *recursion*. You will need a specific library function. A list of library functions is available from the course website.

(b) To confirm your function is correct, write a test function `prop_countPositives` that tests that `countPositivesRec` returns the same result as `countPositives` from last week. You can use your implementation of `countPositives` from last week, or the solution from the course website, or both.

Run your test function(s) using QuickCheck.

4. (a) Write a function `multDigitsRec :: String -> Int` that returns the product of all the digits in the input string. If there are no digits, your function should return 1. For example,

```
multDigitsRec "The time is 4:25"  ==  40
multDigitsRec "No digits here!"   ==   1
```

Your definition should use *recursion*. You will need a library function to determine if a character is a digit, one to convert a digit to an integer, and one to do the multiplication.

(b) To confirm your function is correct, write a test function `prop_multDigits` that tests that `multDigitsRec` returns the same result as `multDigits` from last week. You can use your implementation of `multDigits` from last week, or the solution from the course website, or both.

Run your test function(s) using QuickCheck.

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. You can use the list of library functions linked from the course webpage, as well as those in the Appendix of this tutorial sheet.

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.

We wrote a function `makeKey :: Int -> [(Char, Char)]` for you which will generate such a key for a given offset. Example:

```
Tutorial2> 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 shows how to encrypt all of the uppercase English letters. There are no duplicates: each letter appears just once amongst the pairs' first components (and just once amongst the second components).

Exercises

5. 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:

   ```
   Tutorial2> lookUp 'B' [('A', 'F'), ('B', 'G'), ('C', 'H')]
   'G'
   Tutorial2> lookUp '9' [('A', 'X'), ('B', 'Y'), ('C', 'Z')]
   '9'
   ```

   Use list comprehension, then write an equivalent function `lookUpRec` using recursion. Test that the two functions are equivalent with a test function `prop_lookUp`.

6. Write a function

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

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

   ```
   Tutorial2> encipher 5 'C'
   'H'
   Tutorial2> encipher 7 'Q'
   'X'
   ```

7. 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:

   ```
   Tutorial2> normalize "July 4th!"
   "JULY4TH"
   ```

8. Write a function

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

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

   ```
   Tutorial2> 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

9. Decrypting an encoded message is easiest if we transform the key first. Write functions

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

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

   ```
   Tutorial2> reverseKey [('A', 'G'), ('B', 'H') , ('C', 'I')]
   [('G', 'A'), ('H', 'B') , ('I', 'C')]
   ```

   First use list comprehension, then write an equivalent function `reverseKeyRec` using recursion. Check that your two functions are equivalent with a test function `prop_reverseKey`.

10. 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:

    ```
    Tutorial2> decipherStr 5 "OZQD4YM"
    "JULY4TH"
    ```

Optional Material

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 `contains :: String -> String -> Bool` that returns `True` if the first string contains the second as a substring (this exercise is the same as the last of the optional exercises of the previous tutorial).

    ```
    Tutorial2> contains "Example" "amp"
    True
    Tutorial2> contains "Example" "xml"
    False
    ```

12. 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.

```
Tutorial2> candidates "DGGADBCOOCZYMJHZYVMTOJOCZHVS"
[(5,"YBBVYWXJJXUTHECUTQHOJEJXUCQN"),
(14,"PSSMPNOAAOLKYVTLKHYFAVAOLTHE"),
(21,"ILLFIGHTTHEDROMEDARYTOTHEMAX")]
```

Strengthened Ceasar

As you have seen in the previous section, the Caesar Cipher is not a very safe encryption method. In this section, security will be upgraded a little.

Exercises

13. Write a function `splitEachFive :: String -> [String]` that splits a string into substrings of length five. Fill out the last part with copies of the character 'X' to make it as long as the others.

    ```
    Tutorial2> splitEachFive "Secret Message"
    ["Secre", "t Mes", "sageX"]
    ```

14. The library function `transpose` switches the rows and columns of a list of lists:

    ```
    Tutorial2> transpose ["123","abc","ABC"]
    ["1aA","2bB", "3cC"]
    Tutorial2> transpose ["1","22","333"]
    ["123","23","3"]
    ```

    If the rows in a list of lists are of the same length, transposing it twice returns the original one. Use your `splitEachFive` function to write a quickCheck property to test this. Also, show with an example that this is not always the case when the rows are of different lengths.

15. Write a function `encrypt :: Int -> String -> String` that encrypts a string by first applying the Caesar Cipher, then splitting it into pieces of length five, transposing, and putting the pieces together as a single string.

16. Write a function to decrypt messages encrypted in the way above.

    Hint: The last action of the previous function is to put the transposed list of strings back together. You will need a helper function to undo this (it is not `splitEachFive`).

Appendix: Utility function reference

Note: for most of these functions you will need to import `Data.Char` or `Data.List`. (This has already been done for you in tutorial2.hs.)

`ord :: Char -> Int`
Return the numerical code corresponding to a character
Examples: `ord 'A' == 65` `ord '1' == 49`

`chr :: Int -> Char`
Return the character corresponding to a numerical code
Examples: `chr 65 == 'A'` `chr 49 == '1'`

`mod :: Int -> Int -> Int`
Return the remainder after the first argument is divided by the second
Examples: `mod 10 3 == 1` `mod 25 5 == 0`

`isAlpha :: Char -> Bool`
Return `True` if the argument is an alphabetic character
Examples: `isAlpha '3' == False` `isAlpha 'x' == True`

`isDigit :: Char -> Bool`
Return `True` if the argument is a numeric character
Examples: `isDigit '3' == True` `isDigit 'x' == False`

`isUpper :: Char -> Bool`
Return `True` if the argument is an uppercase letter
Examples: `isUpper 'x' == False` `isUpper 'X' == True`

`isLower :: Char -> Bool`
Return `True` if the argument is a lowercase letter
Examples: `isLower '3' == False` `isLower 'x' == True`

`toUpper :: Char -> Char`
If the argument is an alphabetic character, convert it to upper case
Examples: `toUpper 'x' == 'X'` `toUpper '3' == '3'`

`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`

`error :: String -> a`
Signal an error
Examples: error "Function only defined on positive numbers!"