Comprehension and Recursion
Informatics 1 – Functional Programming: Tutorial 2

Heijltjes, Wadler

Due: The tutorial of week 4 (16/17 Oct.)
Reading assignment: Chapters 6 and 7 (pp. 96-134)

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.

Comprehension and Recursion

In these problems you'll be asked to define several functions in two ways: first with a list comprehension and second with recursion. These two definitions should *not* depend on one another. The recursive version should not mention the list-comprehension version, and vice-versa.

To allow the two solutions to these problems to co-exist in one file, you need to give them different names. For this tutorial, use the given name for the list-comprehension version, and append `Rec` to the name for the recursive version. For example, `halveEvens` should be a function using a list comprehension and `halveEvensRec` should be a recursive function that behaves the same.

In addition, to verify that both functions work in the same way, you will write and run an appropriate QuickCheck test. For the first two exercises the test properties are given as an example.

You will find the skeletons of the functions in the file `tutorial2.hs` on the website. **Note:** for these exercises you may not use any library functions other than the ones stated. If you have an additional solution using other library functions, you're welcome to discuss it during the tutorial.

Exercises

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

```
GHCI> halveEvens [0,2,1,7,8,56,17,18]
[0,1,4,28,9]
```

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

   (b) Write the function `halveEvensRec` with the same behaviour, but this time using *recursion*, not a list comprehension.

   (c) Use the test property `prop_halveEvens` to check that both functions behave similarly.

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

   ```
   GHCI> inRange 5 10 [1..15]
   [5,6,7,8,9,10]
   ```

   Your definition should use a *list comprehension*, not recursion.

   (b) Write a similar function `inRangeRec` using *recursion*.

   (c) Verify both functions with the test function `prop_inRange`.

3. (a) Write a function `sumPositives` to sum up all the positive numbers in a list (the ones strictly greater than 0).

   ```
   GHCi> sumPositives [0,1,-3,-2,8,-1,6]
   15
   ```

   Your definition should use a *list comprehension*. You may not use recursion, but you will need a specific library function (try to find out which one).

   (b) Why do you think it's not possible to write `sumPositives` without library functions?

   (c) Write a recursive function `sumPositivesRec` without using any library functions.

   (d) Write a test function `prop_sumPositives` and run the appropriate QuickCheck test.

4. (a) Professor Pennypincher will not buy anything if he has to pay more than £199.00. But, as a member of the Generous Teachers Society, he gets a 10% discount on anything he buys. Write a function `pennypincher` that takes a list of prices and returns the total amount that Professor Pennypincher would have to pay, if he bought everything that was cheap enough for him.

   ```
   GHCi> pennypincher [45.00, 199.00, 220.00, 399.00]
   417.6
   ```

   Your solution should use a *list comprehension*, and you may use the same library function as in exercise (3a).

   (b) Write recursive function `pennypincherRec` without using library functions.

   Note: we skip the QuickCheck test, because even if your functions are correct, it would fail on rounding errors.

5. (a) Write a function `sumDigits :: String -> Int` that returns the sum of all the digits in the input string.

   ```
   GHCI> sumDigits "CA 90210"
   12
   GHCI> sumDigits "No digits here!"
   0
   ```

   Your definition should use a *list comprehension*. You'll need a library function to determine if a character is a digit, one to convert a digit to an integer, and the same function as in (3a) and (4a).

   (b) Write recursive function `sumDigitsRec` according to the same specification. You may use library functions that act on single characters or integers, but you may not use library functions that act on a list.

   (c) Write a test function `prop_sumDigits` and run the appropriate QuickCheck test.

6. (a) Write a function `capitalized :: String -> String` which, given a word, capitalizes it. That means that the first character should be made uppercase and any other letters should be made lowercase. Use a *list comprehension* and library functions that change the case of a character.

   E.g.:

   ```
   GHCI> capitalized "edINBurgH"
   "Edinburgh"
   ```

   (b) Write a recursive function `capitalizedRec`. (Hint: you may need to write a helper function; of the helper function and the main function only one needs to be recursive.)

   (c) Write a test function `prop_capitalized` and run the appropriate QuickCheck test.

7. (a) Using the function `capitalized` from the previous problem, write a function

   ```
   title :: [String] -> [String]
   ```

   which, given a list of words, capitalizes them as a title should be capitalized. The proper capitalization of a title (for our purposes) is as follows: The first word should be capitalized. Any other word should be capitalized if it is at least four letters long. E.g.:

   ```
   GHCI> title ["tHe", "sOunD", "ANd", "thE", "FuRY"]
   ["The","Sound","and","the","Fury"]
   ```

   Your function should use a *list comprehension*, and not recursion. Besides the `capitalized` function, you will probably need some other auxiliary functions. You may use library functions that change the case of a character and the function `length`.

   (b) Write a *recursive* function `titleRec`. You can use `capitalized` or `capitalizedRec` and the auxiliary functions you used for `title` — even the list-comprehension ones, as long as the function `titleRec` is recursive (in a meaningful way).

   (c) Write a test function `prop_title` and compare the two functions.

8. (a) Dame Curious is a crossword enthusiast. She has a long list of words that might appear in a crossword puzzle, but she has trouble finding the ones that fit a slot. Write a function

   ```
   crosswordFind :: Char -> Int -> Int -> [String] -> [String]
   ```

   to help her. The expression

   `crosswordFind letter inPosition len words`

   should return all the items from `words` which (a) are of the given length and (b) have `letter` in the position `inPosition`. For example, if Curious is looking for seven-letter words that have `'k'` in position 1, she can evaluate the expression:

   ```
   crosswordFind 'k' 1 7 ["funky", "fabulous", "kite", "icky", "ukelele"]
   ```

   which returns `["ukelele"]`. (*Remember that we start counting with 0, so position 1 is the second position of a string.*)

   Your definition should use a *list comprehension*. You may also use a library function which returns the nth element of a list, for argument n, and the function `length`.

   (b) Write a recursive function `crosswordFindRec` to the same specification (you can use the same library functions).

   (c) Write a test function `prop_crosswordFind` and run the appropriate QuickCheck test.

9. (a) Write a function `search :: String -> Char -> [Int]` that returns the positions of all occurrences of the second argument in the first. For example

```
GHCI> search "Bookshop" 'o'
[1,2,6]
GHCI> search "senselessness's" 's'
[0,3,7,8,11,12,14]
```

Your definition should use a *list comprehension*, not recursion. You may use the function `zip :: [a] -> [b] -> [(a,b)]`, the function `length :: [a] -> Int`, and the term forms `[m..n]` and `[m..]`.

(b) Write the recursive function `searchRec`. You may like to use an auxiliary function in your definition, but you shouldn't use any library functions.

(c) Write a test function `prop_search` and run the appropriate QuickCheck test.

10. This final exercise has a slightly different format. The problem does not lend itself very well for list comprehension, so we will start with the recursive version straight away. In addition, you are encouraged to reflect on why exactly a solution using list comprehension is not suitable to this problem.

    (a) Write a *recursive* function `containsRec` that takes two strings and returns `True` if the first contains the second as a substring. Use the library function `isPrefixOf` which returns `True` if the second string begins with the first string (try it out!).

    ```
    GHCi> containsRec "United Kingdom" "King"
    True
    GHCi> containsRec "Appleton" "peon"
    False
    GHCi> containsRec "" ""
    True
    ```

    Note: pay attention to the last case of the above three (`containsRec "" ""`).

    (b) Try to describe why it is difficult to write this function with a list comprehension. If you want, you can try this anyway (but you don't have to). A hint: use the library function `drop` to create a list of all possible suffixes ("last parts") of a string.