School of Computer Science, University of Nottingham

G51FUN Functional Programming, Spring 2015

Graham Hutton

Coursework I, Part 5/5

Tuesday 17th February 2015

Deadline: Monday 9th March 2015, 11am

This exercise sheet covers material from Lecture 6 (Recursive Functions), and is worth 20% of the first coursework. You should attempt to complete the exercises in your own time, but if you get stuck you can ask for help during the lab sessions.

Assessment will be carried out by oral examination during the lab sessions (nothing needs to be handed in). When you have completed the exercises you should ask a tutor to examine your solution. The tutor will then ask you some questions to test your understanding. You will not receive any marks if you cannot explain your solution. Note that tutors will only be available to assess your solution during the official lab session (Mondays, 9am to 11am, A32).

Ensure that your script type checks before submitting it to a tutor. If you are unable to successfully complete some of the definitions then comment out those that are incomplete.

1. Define a recursive function

   $triangle :: Int \to Int$

   that returns the sum of all natural numbers between its argument and zero (inclusive). For example, *triangle* 4 should return $4 + 3 + 2 + 1 + 0$, i.e. the value 10.

2. Add the following code at the top of your script (before the definition of *triangle*):

   import *Prelude hiding* $(or, minimum, lookup)$

   Now use recursion to define the following functions:

   (a) Determine if any Boolean in a list is true:

   $or :: [Bool] \to Bool$

   (b) Find the minimum element in a non-empty list of numbers.

   $minimum :: [Int] \to Int$

   Hint: the appropriate base case for the definition is a singleton list $[n]$ rather than the empty list, and you may find the library function *min* useful.

   (c) Count how many times a number occurs in a list:

   $count :: Int \to [Int] \to Int$

   (d) Locate the value associated with a given key in a list of key/value pairs:

   $lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to b$

   For example, *lookup* 5 $[(2, \texttt{"Red"}), (5, \texttt{"Blue"}), (1, \texttt{"Yellow"})]$ should return `"Blue"`. Your may assume that the key is always present in the list.

3. *Euclid's Algorithm* returns the greatest common factor of two numbers. Given two natural numbers, the algorithm repeatedly subtracts the smaller number from the larger until the two numbers are equal. The resulting number is the greatest common factor of the original numbers. Define a recursive function implements Euclid's Algorithm.

   $euclid :: Int \to Int \to Int$

   For example, *euclid* 6 27 should return 3.

4. **Bonus Exercise**. (Optional, but you will earn you additional marks if you complete it.)

   (a) Define a recursive function

   $altMap :: (a \to b) \to (a \to b) \to [a] \to [b]$

   that takes as arguments two functions f and g and a list xs, and applies f to all elements at even positions in xs and g to all elements at odd positions in xs. For example, *altMap* (+10) (+100) $[1, 2, 3, 4, 5]$ should return $[11, 102, 13, 104, 15]$.

   (b) Using *altMap*, define a more general version of the *luhn* function (from Part 3, question 4) that works for bank card numbers of any length.

   $luhn :: [Int] \to Bool$

   Hint: there is no need to use recursion in the definition, and you may find the library functions *reverse*, *id*, *sum* and *mod* useful.

   Test your function on your own bank card!