School of Computer Science, University of Nottingham

G51PGP Programming Paradigms, Spring 2019

Graham Hutton

Haskell Coursework I, Part 3/3

Deadline: Wednesday 6th March 2019, 12 noon

This exercise sheet covers the material from Lectures 4–6 (Defining Functions, List Comprehensions, Recursive Functions), and is worth 2% of the overall module mark. You should attempt to complete the exercises in your own time, but if you get stuck you can ask for help from the tutors during the weekly lab sessions.

Assessment will be carried out by oral examination during the lab session – nothing needs to be handed in. When you have completed the exercises ask a tutor to examine your solution. The tutor will then ask you some questions to test your understanding. Note that tutors will only be available to answer questions and assess your solution during the official G51PGP lab sessions (Wednesdays, 11am to 1pm, A32).

---

1. Define a function `second :: [a] -> a` that returns the second element in a list that contains at least this many elements using:

   (a) `head` and `tail`

   (b) list indexing `!!`

   (c) pattern matching

   Name your functions `second1`, `second2` and `second3`.

2. The *exclusive or* operator takes two logical values and returns `True` when exactly one is `True`, and `False` otherwise. Define a function `xor :: Bool -> Bool -> Bool` that implements this operator using:

   (a) pattern matching

   (b) `if then else`

   (c) the operator `/=` (*not equal to*)

   Name your functions `xor1`, `xor2` and `xor3`. Try to simplify your definitions as much as possible, e.g. by not using *any* other library functions or operators.

3. Using a list comprehension, define a function `sumsqr :: Int -> Int` that calculates the sum $1^2 + 2^2 + \cdots + n^2$ of the first n integer squares.

4. Using a list comprehension, define a function `grid :: Int -> [(Int,Int)]` that returns a list of all (x, y) coordinate pairs on an $n \times n$ square grid, excluding the diagonal running from $(0, 0)$ to (n, n).

5. Define a recursive function `euclid :: Int -> Int -> Int` that implements *Euclid's algorithm* for calculating the greatest common divisor of two non-negative integers: if the two numbers are equal, this number is the result; otherwise the smaller number is subtracted from the larger, and the same process is then repeated. For example, `euclid 6 27` should return the result 3.

6. The definition of `reverse :: [a] -> [a]` given in lecture 6 is potentially inefficient for long lists. Complete the following definition (fill in the gaps `?`) of a fast version of reverse that uses an extra parameter to *accumulate* the result:

```
fastrev :: [a] -> [a]
fastrev xs = rev xs []

rev :: [a] -> [a] -> [a]
rev []     ys  = ?
rev (x:xs) ys  = rev xs ?
```

   Your new definition should not use the `++` operator.