MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 3: Pattern Matching and Recursion

Introduction

This worksheet is intended to give you practice in defining functions using pattern matching and recursion. Begin by copying the Week3.hs file from the unit website to your file-space. This file defines a module Week3 containing re-implementations of the `||` operator (using pattern matching) and recursive functions `fact`, `mult` and `divide` from the lecture. Load this script into GHCi, and experiment by evaluating some expressions. Add your solutions to the programming exercises to this file. Make sure that you thoroughly test the solutions to each of the exercises.

Pattern matching

1. Following the examples of the re-implementations of the `||` (or) operator, write three definitions of the `&&` (and) operator using pattern matching. Remember to first hide the definition given in the standard prelude, and to include the fixity declaration:

   ```
   infixr 3 &&
   ```

   to force complex Boolean expressions to be correctly interpreted. Make sure your three definitions mirror those of the definitions of `||` given in the file and described in the lecture notes:

   (i) on slide 6

   (ii) at the top of slide 7

   (iii) at the bottom of slide 7

   After writing and testing each operator, comment out the definition using `--` to allow you to give the next definition.

2. Using pattern matching, define an exclusive or function:

   ```
   exOr :: Bool -> Bool -> Bool
   ```

   which gives True when exactly one of its arguments is True.

3. Using pattern matching, define a function:

   ```
   ifThenElse :: Bool -> Int -> Int -> Int
   ```

   which gives its second argument if the condition (the first argument) is True, and the third argument if the condition is False (for example, `ifThenElse (3 > 5) 7 12` gives 12).

4. Use pattern matching to write a function:

   ```
   daysInMonth :: Int -> Int
   ```

   which takes an integer (assumed to be between 1 and 12) and returns the number of days in the corresponding month (e.g. `daysInMonth 1` is 31 and `daysInMonth 2` is 28.) Try to make your solution as short as possible. Using the `daysInMonth` function,

write a new (simpler) version of your `validDate` function from the previous worksheet (no guards should be necessary).

Recursion

5. Write a recursive function:

   ```
   sumNumbers :: Int -> Int
   ```

   that gives the sum of the first n positive integers (e.g. `sumNumbers 3` is 1 + 2 + 3 = 6).

6. Write a recursive function:

   ```
   sumSquares :: Int -> Int
   ```

   that gives the sum of the first n squares (e.g. `sumSquares 3` is 1 + 4 + 9 = 14).

7. Using multiplication, write a recursive function:

   ```
   power :: Int -> Int -> Int
   ```

   which raises its first argument to the power of the second (e.g., `power 2 3` is $2^3 = 8$).

8. Write a recursive function:

   ```
   sumFromTo :: Int -> Int -> Int
   ```

   that gives the sum of all integers between and including its two arguments. For example, `sumFromTo 5 8` is 5 + 6 + 7 + 8 = 26. If the first argument is greater than the second, your function should return 0.

9. One definition of the greatest common divisor (GCD) of two non-negative integers is:

   - the GCD of two equal integers is their common value;
   - the GCD of two non-equal integers is the GCD of their positive difference and the smaller integer.

   Write a function:

   ```
   gcd :: Int -> Int -> Int
   ```

   for finding the GCD of two non-negative integers, **based on the above definition**. (Note: youll need to hide the definition of `gcd` given in the prelude.)

10. The integer square root of a positive integer n is the largest integer whose square is less than or equal to n. (E.g. the integer square root of 7 is 2, and that of 9 is 3). Use recursion to write a function:

    ```
    intSquareRoot :: Int -> Int
    ```

    that computes the integer square root of a number.

11. If many of your answers to questions 5 to 10 involve guards, write versions of three of them that use pattern matching and no guards. Similarly, if many already use pattern matching, write versions of two of them that use guards.