COMP105: Programming Paradigms
Lab Sheet 3

Filename: Lab3.hs

This lab covers material from Lectures 7 through 9. Although most of these problems can be solved with library functions, the purpose of this lab is to practice recursion, so you should write recursive functions for all of these questions. Fill out your answers in a file named Lab3.hs, and upload it to Codegrade when you have finished.

1. **Basic recursion.** These questions cover recursion on numbers, which we saw in Lecture 7. You should use recursion to solve the questions below.

   (a) Write a recursive function `mult13 n` that computes $13 * n$ by adding 13 to itself n times.

   (b) Write a recursive function `pow3 n` that computes 3^n by multiplying 3 by itself n times.

   (c) Write a recursive function `odd_sum n` that computes the sum of all odd numbers less than or equal to `n`. Hint: modify the `even_sum` function from Lecture 7 if you are stuck.

   (d) The *Lucas numbers* (denoted by L_i) are defined so that $L_0 = 2$, $L_1 = 1$, and $L_n = L_{n-1} + L_{n-2}$ for all $n > 1$. Write a simple recursive function `lucas n` that computes the nth Lucas number. Your solution does not have to be computationally efficient.

2. **Recursion on lists.** These questions cover recursion on lists, which we say in Lecture 8.

   (a) Write a recursive function `half_sum list` that computes the sum of all elements in the list divided by 2.

   (b) Write a recursive function `mult2 list` that multiplies all elements of the input list by 2.

   (c) Write a recursive function `drop_evens list` that returns a new list with only the odd elements from the input list.

   (d) Write a recursive function `triple list` that repeats each element of the input list three times, so `triple [1,2]` should return `[1,1,1,2,2,2]`.

   (e) Write a recursive function `mult_adjacent list` that takes a list with an even number of elements, and returns a new list where adjacent pairs have been multiplied together. So, `mult_adjacent [1,2,3,4]` should return `[2, 12]`.

(f) Write a recursive function `get_ele i list` that returns the element at position `i` in `list`. Your function should return an error if the list does not contain `i` elements.

(g) Write a recursive function `drop_ele i list` that returns a copy of the input list with the element at position `i` removed. Your function should return an error if the list does not contain `i` elements.

3. **More complex recursion on lists.** These questions cover the more complex recursive patterns that we saw in Lecture 9.

   (a) Write a recursive function `div_list list1 list2` that takes two lists of the same length, and divides each element in `list1` by the corresponding element in `list2`. So the function returns a list where the element in position `i` in the output is `(list1 !! i) / (list2 !! i)`.

   (b) Write a recursive function `longer list1 list2` that returns `True` if `list1` is longer than `list2`, and `False` otherwise.

   (c) Write a recursive function `div3_and_not list` that returns a pair of lists. The first element of the pair should contain all elements of `list` that are divisible by 3, and the second element should contain all elements of the list that are not divisible by `3`.

   (d) Write a recursive function `vowels_and_consonants string` that takes a string of characters and returns a pair of strings: the first string in the pair should contain all vowels in the string (which are the letters `"aeiou"`), and the second string in the pair should contain all letters that are not vowels.

   (e) Write a function `fast_lucas n` that computes the `n`th Lucas number in a computationally efficient way, like we did for Fibonacci numbers in Lecture 9.

   (f) (*) Write a function `mult_by_pos list` that multiplies each element in a list by its position in the list. One way of doing this uses a helper function and the `zip` library function.

4. (*) **The Collatz problem (in Haskell!).** Suppose we take a number n and do the following transformation:

   - $n \to n/2$ if n is even,
   - $n \to 3n + 1$ if n is odd.

   Applying this rule starting with $n = 13$ and stopping when we reach 1 generates the following sequence

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

   We call this the *Collatz sequence* for 13, and it has 10 terms (including 13 and 1 themselves). Starting at different numbers gives Collatz sequences of different lengths. For example, the sequence starting at $n = 97$ has 119 terms, which is the longest sequence with starting number less than 100.

It is thought that every Collatz sequence will eventually arrive at 1 (the Collatz conjecture,) but it has not yet been proved!

Write a function `longest_collatz n` that computes the starting number less than `n` that has the longest Collatz sequence. If there is a tie, then return the largest number that is tied for the longest sequence length. Your solution will probably need to use one or more helper functions. The `div` function does integer division.

Lab complete. Don't forget to upload Lab3.hs to Codegrade!