CPSC 449: Assignment 1

Fall 2020

Consult D2L for the official due date/time

Each of the questions below is worth 25% of the total grade. Make sure your submission follows the *Assignment Submission Guidelines*.

1. Implement a function **myLog** that computes $\log_b x$. More specifically, your function shall have the following type:

   ```
   myLog :: Integer -> Integer -> Integer
   ```

   The first argument is the base (b), and the second argument is the number whose logarithm is to be computed (x). The return value should be the biggest non-negative integer y such that $b^y \le x$. Here are some examples:

   myLog 2 16 $\rightsquigarrow$ **4**
   myLog 2 17 $\rightsquigarrow$ **4**
   myLog 2 31 $\rightsquigarrow$ **4**
   myLog 2 32 $\rightsquigarrow$ **5**
   myLog 2 1 $\rightsquigarrow$ **0**

   You may assume that the two arguments are both positive, and the first argument is at least 2. Your solution shall be recursive. **Hint:** If applicable, show off you know how to introduce local definitions using either **where** or **let**.

   The TA will use QuickCheck to test your code. In particular, the TA will check for compliance to the following two properties.

   $$\forall b, x \in \texttt{Integer}\,.\,(b > 1 \land x > 0) \rightarrow (b^{(\texttt{myLog}\ b\ x)} \le x) \tag{1}$$
   $$\forall b, x, z \in \texttt{Integer}\,.\,(b > 1 \land x > 0 \land z \ge 0 \land b^z \le x) \rightarrow (z \le (\texttt{myLog}\ b\ x)) \tag{2}$$

   Property (1) states that the return value of **myLog** is no bigger than $\log_b x$. Property (2) asserts that **myLog** returns the biggest non-negative integer to satisfy Property (1). Both properties are conditional.

   You are recommended to use QuickCheck to confirm that your code satisfies the two properties above. You do not need to submit your Haskell encoding of these properties, nor are we going to grade any such encodings if they were submitted. The correctness of your implementation of `myLog` will be evaluated based on the extent to which it satisfies the TA's QuickCheck encoding of the two properties above.

   Please read the Appendix below for some information on QuickCheck that you will need for this assignment question.

2. **[Thompson]** Exercise 5.32. Contrary to the statement of the exercise in the textbook (which asks only for a "discussion"), you are required to provide the Haskell implementation of all the database functions, namely:

   ```
   books       :: Database -> Person -> [Book]
   borrowers   :: Database -> Book -> [Person]
   borrowed    :: Database -> Book -> Bool
   numBorrowed :: Database -> Person -> Int
   makeLoan    :: Database -> Person -> Book -> Database
   returnLoan  :: Database -> Person -> Book -> Database
   ```

   Your work shall honor the following requirements:

   (a) If a person has not checked out any book, her name shall not appear in the database at all.

   (b) A person may check out a book at most once. When a person requests to have the same book checked out for the second time, the operation is considered a NOP (no operation), and the contents of the database remain the same.

   Full marks will only be granted to solutions that honor all of the requirements above. **Hint:** Start with a simple implementation without worrying about the requirements above. Once the code is working, then enrich your implementation to satisfy one requirement at a time, until your code meets all the requirements. In short, don't be overly ambitious in the beginning. You are only human. And that is okay.

3. **[Thompson]** Exercise 6.10.

   Hint: You need the following type declaration:

   ```
       type Picture = [[Char]]
   ```

4. A ***social graph*** is represented as a list of edges.

   ```
   type Graph = [(Int, Int)]
   ```

   More specifically, a person is represented by an integer label. There is an *undirected* edge between two persons if they are friends of one another. Such an edge is encoded by a pair of type `(Int, Int)`. Note that if there is an edge (x, y) from person x to person y, it means x considers y a friend, and y also considers x a friend. A social graph is represented by a list of such pairs.

   Develop a Haskell function that returns the list of all common friends of two given persons.

   ```
   commonFriends :: Graph -> Int -> Int -> [Int]
   ```

   More specifically, `(commonFriends` G x y`)` returns the list of all common friends shared by persons x and y in social graph G. As an example, suppose we are given the following social graph:

   ```
   G :: Graph
   G = [(1,3), (4, 2), (4, 1), (2, 3)]
   ```

   Then `(commonFriends G 1 2)` evaluates to `[3, 4]` (or `[4, 3]`, depending on implementation details). Your solution shall **NOT** involve explicit recursion. Use list comprehension instead. A solution involving explicit recusion will receive zero mark. **Hint:** Introduce helper functions to help you break the problem into manageable subproblems.

Appendix: Useful Information on QuickCheck

- Install QuickCheck using `cabal`. For example, type the following command on UNIX:

  ```
  cabal install QuickCheck
  ```

- To use QuickCheck in your Haskell code, import the module `Test.QuickCheck`.

- Newer versions of QuickCheck use types that you are not familiar with. To get around this, there is no need for you to specify the types of your properties before defining them. For example, the textbook suggests you to define a property as follows:

  ```
  prop_ComAdd :: Integer -> Integer -> Bool
  prop_ComAdd a b = (a+b) == (b+a)
  ```

  Instead, you can omit the type declaration, as in the following:

  ```
  prop_ComAdd a b = (a+b) == (b+a)
  ```

- You will need to use ***conditional properties***. A conditional property is one that holds only under some condition. For example, under the condition that `(a>0 && b>0)`, we want to test the property `((rem a b) >= 0 && (rem a b) < b)`. Such a conditional property can be formulated as follows:

  ```
  prop_rem_range a b =
    (a>0 && b>0) ==>
      ((rem a b) >= 0 && (rem a b) < b)
  ```

  The operator "==>" separates the condition from the property. When a conditional property is tested, QuickCheck will discard any test cases that fail the condition, and check the property only against the test cases that satisfy the condition. For example, if you invoke the following:

  ```
  quickCheck prop_rem_range
  ```

  then QuickCheck responds with a message such as the following:

  ```
  +++ OK, passed 100 tests; 380 discarded.
  ```

  Since test cases are generated randomly, the actual number of test cases discarded by QuickCheck may be different on each invocation.

- If most of the test cases generated by QuickCheck fail the condition, then after a while QuickCheck will give up. To prevent QuickCheck from giving up too early, you can tweak the parameter `maxDiscardRatio`, which controls when QuickCheck will give up. For example, you can invoke QuickCheck by supplying a bigger-than-usual `maxDiscardRatio`:

  ```
  quickCheckWith stdArgs { maxDiscardRatio = 1000 } prop_rem_range
  ```