Assignment #1

Posted on September 17 – due by September 25, 2:00pm

For all questions in this assignment, write Haskell code that computes the specified function. Use good Haskell style by including in your code the signature of each top-level function above its definition (most signatures are actually provided in the questions). You can include as many other (helper) functions as you like.

Please use the apply process for CS470 or CS670 to receive a directory in the UNIX system into which you can put your answers (see here). For this assignment, please put the answers to all questions into a single Haskell file named “A1.hs”. Please note that Question 5 is a bonus question for extra points and Question 6 is only required for students enrolled in CS670. CS470 students may also submit an answer to Question 6 and get one extra point if it is correct.

Question 1: Warm-Up

Here are some easy programming tasks to get started:

(a) Write a function `palindrome :: [Char] -> Bool` that tells you whether a given string is a palindrome, i.e., whether it is identical when read left-to-right and right-to-left.

(b) Write a function `oddEvenCount :: [Int] -> (Int, Int)` that receives a list of integers and returns a pair (a, b), where a and b stand for the number of odd and even integers, respectively, that are contained in the list.

(c) Write a function `caesar :: [Char] -> [Char]` that receives a string of small letters and returns its encoding in Caesar’s code. This means that every ‘a’ turns into a ‘b’, every ‘b’ into a ‘c’, every ‘c’ into a ‘d’, and so on, and finally, every ‘z’ into an ‘a’. Hint: Look at the successor function `succ` to make your life easier.

Question 2: Relative Primes

If two integers do not share any positive divisors except 1, they are called relatively prime. For example, 10 and 21 are relatively prime, because 10 has positive divisors 1, 2,
5, and 10, whereas 21 has positive divisors 1, 3, 7, and 21. The integers 15 and 27, on the other hand, are not relatively prime, because they both have the divisor 3.

(a) Write a function \( \text{relPrimes} :: \text{Integral} \ a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \) that determines for given two integers whether they are relatively prime or not.

(b) Write a function \( \text{listRelPrimes} :: \text{Integral} \ a \Rightarrow a \rightarrow [a] \) so that \( \text{listRelPrimes} \ n \) returns a list of all positive integers, in ascending order, that are relatively prime to integer \( n \). Note that this list is always infinite, and therefore, you should never use it by itself, such as by typing \( \text{listRelPrimes} 5 \) at the GHCi command line. Instead, use it in expressions such as \( \text{take 10 (listRelPrimes 5)} \) for obtaining the first 10 elements of the resulting list. Of course your function definition should make use of the \( \text{relPrimes} \) function that you already wrote.

(c) Write a function \( \text{sumRelPrimes} :: \text{Integral} \ a \Rightarrow a \rightarrow a \) so that \( \text{sumRelPrimes} \ n \) returns the sum of all positive integers less than or equal to \( n \) that are relatively prime to \( n \). Again, please make use of your previously written functions.

**Question 3: Fibonacci**

Write a function \( \text{fibo} :: \text{Int} \rightarrow [\text{Integer}] \) that for a given non-negative integer \( n \) returns the first \( n \) elements of the Fibonacci sequence. For example, \( \text{fibo} 6 \) should output \( [0, 1, 1, 2, 3, 5] \).

**Question 4: Going Backwards**

You already know the function \( \text{reverse} \) that receives a list and returns it with its elements in reversed order. Now write your own version of this function, named \( \text{foldReverse} \), that does the same thing but uses a fold function (\( \text{foldl} \) or \( \text{foldr} \)) and a lambda expression to accomplish it.

**Question 5 (Bonus): Finite Automata**

You also already know everything about deterministic finite automata (DFA). Let us look at the following variant: Our DFA has a finite number of states that are identified by integers. It reads a string of characters and processes it according to its list of state transitions of the type \( [(\text{Integer}, \text{Char}, \text{Integer})] \). Each transition consists of the current state, a character, and the target state. It indicates that if the DFA is in that current state and the current symbol in the string matches the character in the transition, then the machine switches into the given target state. If there is no such matching transition, the machine halts and the string is rejected. Once the machine has processed the entire string and the final state is in the set of accepting states, then the string is accepted, otherwise it is rejected.
Now write a function

\[
\text{accept :: } [(\text{Integer, Char, Integer})] \rightarrow [\text{Integer}] \rightarrow [\text{Char}] \rightarrow \text{Bool}
\]

that receives a list of state transitions, a list of accepting states, and an input string and tells us whether the given DFA accepts the input string. By default, the starting state is always state 1. For example, consider the following DFA with accepting state 4:

\[
\text{abaaMachine} = [(1, 'a', 2), (2, 'a', 3), (2, 'b', 2), (3, 'a', 4)]
\]

This machine accepts any string that starts with an ‘a’ followed by any number (zero or more) of ‘b’s and finally the substring ‘aa’. Consequently, your program should give the following outputs:

*Main> accept abaaMachine [4] "abbbaa"
True
*Main> accept abaaMachine [4] "abbba"
False
*Main> accept abaaMachine [4] "aabaa"
False
*Main> accept abaaMachine [4] "aaa"
True

**Question 6 (Required only for CS670):**

Write a function \( \text{addBinary :: [Int]} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \) that performs addition of two binary numbers that are represented by lists of integers 0 and 1. For example, your function should yield the following results:

*Main> addBinary [1, 0, 1] [1]
[1,1,0]
*Main> addBinary [1, 1, 1] [1]
[1,0,0,0]
*Main> addBinary [1, 1, 1] [0]
[1,1,1]
*Main> addBinary [1, 1, 1] [1, 1]
[1,0,1,0]
*Main> addBinary [1, 1, 1] [1, 1, 0, 0]
[1,0,0,1,1]