Assignment 2

due at 23:59 on Wed Sep 20 (80 points)

For this assignment, you will write a few small functions for generating or processing lists (including strings, which are lists of characters).

Save all your functions to a file called `a02.hs` and submit it to this dropbox. To ensure compatibility with my test procedures, you should use exactly the function names indicated. Also, for some exercises I will specify whether it should be written recursively, or whether it should use existing functions like `take` and `map`.

At the bottom of this document is a `main` program that I will use to test your code. You may paste it into your program and test that way too, by invoking `main` in GHCi, or using `stack runghc` on the filename.

1. We learned about built-in functions `take` and `drop`. Use them to write a function `slice i k` that extracts elements between positions i and k (including i, but excluding k). Here are some examples:

```
*Main> slice 3 6 [0,0,0,1,2,3,0,0,0]
[1,2,3]
*Main> slice 2 2 [1,2,3,4]
[]
*Main> slice 2 3 [1,2,3,4]
[3]
*Main> slice 2 10 [1,2,3,4]
[3,4]
*Main> slice 10 15 [1,2,3,4]
[]
```

2. Write a function that 'rotates' a list to the left by `n` places. My solutions uses `take`, `drop`, and the list concatenation operator, `++`. The following transcript demonstrates `++` and them gives some examples of `rotate`:

```
*Main> rotate 0 []
[]
*Main> rotate 2 [1,2,3,4,5]
[3,4,5,1,2]
*Main> rotate 5 "abracadabra"
"adabraabrac"
```

The behavior of the function is unspecified if the `n` is negative or larger than the size of the list.

3. Write a recursive function `everyOther` that takes a list and returns a list that retains every alternate element. The best way to understand is by example:

   ```
   *Main> everyOther [1..10]
   [1,3,5,7,9]
   *Main> everyOther ["Alice", "Bob", "Carl"]
   ["Alice","Carl"]
   *Main> everyOther "university"
   "uiest"
   *Main> everyOther "trail"
   "tal"
   *Main> everyOther []
   []
   *Main> everyOther [pi]
   [3.141592653589793]
   ```

   Hint: This function actually has *two* base cases: the empty list, and the list containing just a single element.

4. For this problem, you'll write a list-processing function that *consumes* a list, but just produces an integer. It should count how many times a particular element appears in a list. Here are some examples:

   ```
   *Main> countOccurences 5 [1..10]
   1
   *Main> countOccurences 5 [1,5,2,5,3,5,7,5]
   4
   *Main> countOccurences 'a' "abracadabra"
   5
   *Main> countOccurences 'o' "xylophone"
   2
   ```

   You can either write it recursively by pattern-matching on the list, or you can use a combination of existing functions.

5. For this problem, write a function `insertElem x k l` that inserts the element `x` at position `k` (zero-based indexing) in the list `l`. Here are some examples:

   ```
   *Main> insertElem 10 4 [2,4,6,8,12,14]
   [2,4,6,8,10,12,14]
   *Main> insertElem 'a' 3 "carmel"
   "caramel"
   *Main> insertElem 'H' 0 "askell"
   "Haskell"
   *Main> insertElem '!' 8 "FP rocks"
   "FP rocks!"
   ```

```
*Main> insertElem '!' 20 "FP rocks"
"FP rocks"
```

You should approach this recursively. Again, there are two base cases. One is that the index `k` is zero, so you insert the element `x` *right here* at the front of the list. The other base case is that the list is empty, so you don't insert it at all. (The given index was too big.)

6. In this problem and the next one, we're going to create functions that will help us generate prime numbers. (An integer *greater than one* is **prime** if its only divisors are one and itself.)

   Let's begin by creating a function `divides p q` which returns `True` if `p` is divisible by `q` with no remainder, or `false` otherwise. You can implement this using the `mod` operator which (like `div`) is often specified as an *infix* operator using back quotes. Here are some examples:

```
*Main> 7 `mod` 3
1
*Main> 18 `mod` 3
0
*Main> divides 7 3
False
*Main> divides 18 3
True
*Main> divides 29 2
False
*Main> divides 29 3
False
*Main> divides 27 3
True
```

7. Now you are ready to define `isPrime n`, which – as long as `n` is greater than 1 – will iterate through the range `[2..(n-1)]` and check that none of those values divide `n`. You should use `divides` from the previous question, and one or more of these built-in functions:

- `not` takes a Boolean and returns its opposite
- `any` takes a Boolean function and a list, and returns `True` if *any* of the list elements cause the function to produce `True`.
- `all` takes a Boolean function and a list, and returns `True` if *all* of the list elements cause the function to produce `True`.

You should also use a guard and `otherwise` to rule numbers out 1 (and any non-positive numbers), which by definition are not prime.

Here are some examples using `isPrime`:

```
*Main> isPrime 27
False
*Main> isPrime 29
True
*Main> isPrime 1
False
*Main> isPrime 2
True
*Main> isPrime 101
True
*Main> isPrime 105
False
```

Here are all the primes less than 200:

```
*Main> filter isPrime [1..200]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,
 79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,
 157,163,167,173,179,181,191,193,197,199]
```

And here are the first 200 primes! (We're cleverly using an infinite list here, but limiting its evaluation by applying `take`.)

```
*Main> take 200 (filter isPrime [1..])
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
 73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,
 157,163,167,173,179,181,191,193,197,199,211,223,227,229,
 233,239,241,251,257,263,269,271,277,281,283,293,307,311,
 313,317,331,337,347,349,353,359,367,373,379,383,389,397,
 401,409,419,421,431,433,439,443,449,457,461,463,467,479,
 487,491,499,503,509,521,523,541,547,557,563,569,571,577,
 587,593,599,601,607,613,617,619,631,641,643,647,653,659,
 661,673,677,683,691,701,709,719,727,733,739,743,751,757,
 761,769,773,787,797,809,811,821,823,827,829,839,853,857,
 859,863,877,881,883,887,907,911,919,929,937,941,947,953,
 967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,
 1049,1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,
 1117,1123,1129,1151,1153,1163,1171,1181,1187,1193,1201,
 1213,1217,1223]
```

8. Finally, let's define a Boolean function `twinPrime`. A number `n` is the first member of a twin prime if `n` is prime *and* `n+2` is also prime. For example, 11 and 13 are twin primes. But your function should only produce `True` for 11, not for 13.

```
*Main> twinPrime 11
```

```
True
*Main> twinPrime 13
False
*Main> filter twinPrime [1..100]
[3,5,11,17,29,41,59,71]
```

When is the next time that the year will be a twin prime?

```
*Main> take 1 $ filter twinPrime [2017..]
[2027]
```

Test code

Note: This uses an external module called `Control.Monad.State`. If that gives you any problems running this code, you can execute this in your terminal:

```
stack install mtl
```

Once that completes, the test program should work.

```
-- Imports must appear before other functions
import Control.Monad.State

-- Test driver
main = flip execStateT (0,0) $ do
  -- Ex 1
  verify "slice1" [1,2,3] $ slice 3 6 [0,0,0,1,2,3,0,0,0]
  verify "slice2" []      $ slice 2 2 [1..4]
  verify "slice3" [3]     $ slice 2 3 [1..4]
  verify "slice4" [3,4]   $ slice 2 10 [1..4]
  verify "slice5" []      $ slice 10 15 [1..4]
  -- Ex 2
  verify "rotate1" []            $ rotate 0 ([] :: [Int])
  verify "rotate2" [3,4,5,1,2]   $ rotate 2 [1..5]
  verify "rotate3" "adabraabrac" $ rotate 5 "abracadabra"
  -- Ex 3
  verify "everyOther1" [1,3,5,7,9] $ everyOther [1..10]
  verify "everyOther2" ["Al","Ca"] $ everyOther ["Al", "Bo", "Ca"]
  verify "everyOther3" "uiest"     $ everyOther "university"
  verify "everyOther4" "tal"       $ everyOther "trail"
  verify "everyOther5" []          $ everyOther ([] :: [Int])
  verify "everyOther6" "a"         $ everyOther "a"
  -- Ex 4
  verify "countOccurs1" 1 $ countOccurences 5 [1..10]
  verify "countOccurs2" 4 $ countOccurences 5 [1,5,2,5,3,5,7,5]
```

```haskell
  verify "countOccurs3" 5 $ countOccurences 'a' "abracadabra"
  verify "countOccurs4" 2 $ countOccurences 'o' "xylophone"
  -- Ex 5
  verify "insert1" [2,4,6,8,10,12,14] $ insertElem 10 4 [2,4,6,8,12,14]
  verify "insert2" "caramel"          $ insertElem 'a' 3 "carmel"
  verify "insert3" "Haskell"          $ insertElem 'H' 0 "askell"
  verify "insert4" "FP rocks!"        $ insertElem '!' 8 "FP rocks"
  verify "insert5" "FP rocks"         $ insertElem '!' 20 "FP rocks"
  -- Ex 6
  verify "divides1" False $ divides  7 3
  verify "divides2" True  $ divides 18 3
  verify "divides3" False $ divides 29 2
  verify "divides4" False $ divides 29 3
  verify "divides5" True  $ divides 27 3
  -- Ex 7
  verify "isPrime1"  False $ isPrime 27
  verify "isPrime2"  True  $ isPrime 29
  verify "isPrime3"  False $ isPrime 1
  verify "isPrime4"  True  $ isPrime 2
  verify "isPrime5"  True  $ isPrime 101
  verify "isPrime6"  False $ isPrime 105
  -- Ex 8
  verify "twinPrime1" True $ twinPrime 11
  verify "twinPrime2" False $ twinPrime 13
  verify "twinPrime3" [3,5,11,17,29,41,59,71] $ filter twinPrime [1..100]
  -- Done
  get >>= say . show

  where
    say = liftIO . putStrLn
    correct (k, n) = (k+1, n+1)
    incorrect (k, n) = (k, n+1)
    verify :: (Show a, Eq a) => String -> a -> a -> StateT (Int,Int) IO ()
    verify = verify' (==)
    verifyF = verify' (\x y -> abs(x-y) < 0.00001)
    verify' :: (Show a) => (a -> a -> Bool) -> String -> a -> a ->
               StateT (Int,Int) IO ()
    verify' eq tag expected actual
      | eq expected actual = do
          modify correct
          say $ " OK " ++ tag
      | otherwise = do
          modify incorrect
          say $ "ERR " ++ tag ++ ": expected " ++ show expected
            ++ " got " ++ show actual
```

```
-- End of test driver
```