CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 4

Due:	**Tuesday**, October 8, 2019 at 9:00 pm
Language level:	Beginning Student
Files to submit:	`vowels.rkt`, `duplicates.rkt`, `change.rkt`, `prime.rkt`
Warmup exercises:	HtDP 10.1.4, 10.1.5, 11.2.1, 11.4.3, 11.5.1, 11.5.2
Practice exercises:	HtDP 10.1.6, 10.1.8, 11.4.7, 11.5.3

- **Make sure you read the OFFICIAL A04 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 03 carry forward.
- This assignment covers material up to Slide 09 of Module 08.
- Of the built-in list functions, you may use only `cons`, `first`, `second`, `third`, `rest`, `empty?`, `cons?`, `member?`, `length`, and `string->list`. You may also use `equal?` to compare two lists. List abbreviations may not be used.
- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
- Unless the question specifically says otherwise, you are always permitted to write helper functions. You may use any constants or functions from any part of a question in any other part.
- Solutions will be marked for both correctness **[80%]** and style **[20%]**. Follow the guidelines in the Style Guide, pages 1–15 (skipping sections 3.7.2 and 3.7.4 for now).

1. **[10% Correctness]** A vowel in English is one of (a, e, i, o, u). And sometimes y, but not for the purposes of this question. Write a function `count-vowels` that counts the vowels in a string. Both upper and lower case letters count. Place your solution in the file `vowels.rkt`.

```
(check-expect
  (count-vowels "the quick brown fox jumped over the lazy dog") 12)
```

2. **[20% Correctness]** Write a function `remove-duplicates` that consumes a list of numbers, and produces the same list, with all but the last occurrence of each number removed. For example, if the input was the list $1, 3, 1, 2, 4, 2, 7, 2, 5$, the output would be the list $3, 1, 4, 7, 2, 5$. The order of the elements in the input list needs to be maintained in the output list, as shown in the example.

```
(check-expect (remove-duplicates (cons 1 (cons 3 (cons 1 (cons 2
                                  (cons 4 (cons 2 (cons 7 (cons 2
                                   (cons 5 empty)))))))))
              (cons 3 (cons 1 (cons 4 (cons 7 (cons 2 (cons 5 empty)))))))
```

   Place your solution in the file `duplicates.rkt`.

3. **[20% Correctness]** Some people still use coins for small purchases. Here in Canada we have the following five coins:

   - `'nickel`, worth 5 cents each
   - `'dime`, worth 10 cents each
   - `'quarter`, worth 25 cents each
   - `'loonie`, worth one dollar each
   - `'toonie`, worth two dollars each

   (a) Write the function `count-change`, which consumes a list of symbols and produces a number which is the total amount of change in cents (not dollars). If the list happens to contain some other symbol (e.g., a foreign coin) just ignore it; treat its value as zero.

   (b) Write the function `make-change`, which consumes a value in cents as a natural number and produces a list of symbols adding up to that value. Since Canada has no one-cent coin (no "penny"), values should be rounded to the nearest nickel. Round down for one or two cents; round up for three or four cents. Since nobody wants to carry around a lot of coins, your solution should not use any coin more than three times, except for the toonie. For some values, there are multiple ways to make correct change.

```
(check-expect (count-change (cons 'dime (cons 'dime empty))) 20)
(check-expect (make-change 11) (cons 'dime empty))
(check-expect (count-change (make-change 137)) 135)
```

   Place your solutions in `change.rkt`.

4. **[30% Correctness]**

   (a) Write a function `prime?` that consumes a natural number and produces `true` if that number is prime, and `false` otherwise. A prime number is a number divisible by only 1 and itself; 0 and 1 are not prime. Put another way, a number is prime if the largest factor of that number (other than itself) is 1.

   You may find it useful to observe that natural number `b > 1` is a factor of natural number `a` if and only if `(remainder a b)` is 0 (zero).

```
(check-expect (prime? 17) true)
(check-expect (prime? 100) false)
```

   (b) Write a function `next-prime` that consumes a natural number and produces the next prime strictly greater than that number. The consumed number can be any natural number. That is, it does not need to be prime itself.

```
(check-expect (next-prime 7) 11)
(check-expect (next-prime 15) 17)
```

   Note: This function requires a slight change from the count-up template because it does not know what it is counting up to until it gets there.

   (c) Write a function `prime-range` that consumes two natural numbers and produces the list of all prime numbers in the interval that starts with the first number and ends with the second number (inclusive). In other words, if given the natural numbers a and b (in that order) it produces all primes p such that $a \leq p$ and $p \leq b$. The list produced is in ascending order (the first value is the smallest value).

```
(check-expect (prime-range 1 10)
              (cons 2 (cons 3 (cons 5 (cons 7 empty)))))
(check-expect (prime-range 10 1) empty)
```

   Place your solutions in the file `prime.rkt`

---

5. **Bonus Question (5%)**

   We can represent a polynomial in a variable x,

$$a_0 + a_1 x + \cdots + a_n x^n$$

   as a non-empty list `(cons a-0 (cons a-1 ... (cons a-n empty) ...))`. We will also add the restriction that either the entire polynomial is zero (represented as `(cons 0 empty)`) or the highest degree coefficient `a-n` is not zero.

   Write a function `eval-poly` that consumes a list of numbers (representing the coefficients of a polynomial) and a value for x, and produces the result of evaluating the given polynomial at the given value of x.

For example,

```
(eval-poly (cons 1.4 (cons 4 (cons 0 (cons 2 empty)))) 3)
```

should produce the value 67.4 (i.e., $2(3^3)+0(3^2)+4(3)+1.4$). To earn marks for this question, you may not use the built-in `expt` operator, or any other exponentiation operation, and you may not use more than n multiplications and n additions, where n is the degree of the polynomial.

Submit your code in the file `bonus-a04.rkt`.

Enhancements: Reminder—enhancements are for your interest and are not to be handed in.

Racket supports unbounded integers; if you wish to compute 2^{10000}, just type `(expt 2 10000)` into the REPL and see what happens. The standard integer data type in most other computer languages can only hold integers up to a certain fixed size. This is based on the fact that, at the hardware level, modern computers manipulate information in 32-bit or 64-bit chunks. If you want to do extended-precision arithmetic in these languages, you have to use a special data type for that purpose, which often involves installing an external library.

You might think that this is of use only to a handful of mathematicians, but in fact computation with large numbers is at the heart of modern cryptography (as you will learn if you take Math 135). Writing such code is also a useful exercise, so let's pretend that Racket cannot handle integers bigger than 100 or so, and use lists of small integers to represent larger integers. This is, after all, basically what we do when we compute by hand: the integer 65,536 is simply a list of five digits (with a comma added just for human readability; we'll ignore that in our representation).

For reasons which will become clear when you start writing functions, we will represent a number by a list of its digits starting from the one's position, or the rightmost digit, and proceeding left. So 65,536 will be represented by the list containing 6, 3, 5, 5, 6, in that order. The empty list will represent 0, and we will enforce the rule that the last item of a list must not be 0 (because we don't generally put leading zeroes on our integers). (You might want to write out a data definition for an extended-precision integer, or EPI, at this point.)

With this representation, and the ability to write Racket functions which process lists, we can create functions that perform extended-precision arithmetic. For a warm-up, try the function `long-add-without-carry`, which consumes two EPIs and produces one EPI representing their sum, but without doing any carrying. The result of adding the lists representing 134 and 25 would be the list representing 159, but the result of the lists representing 134 and 97 would be the list 11, 12, 1, which is what you get when you add the lists 4, 3, 1 and 7, 9. That result is not very useful, which is why you should proceed to write `long-add`, which handles carries properly to get, in this example, the result 1, 3, 2 representing the integer 231. (You can use the warmup function or not, as you wish.)

Then write `long-mult`, which implements the multiplication algorithm that you learned in grade school. You can see that you can proceed as far as you wish. What about subtraction? You need

to figure out a representation for negative numbers, and probably rewrite your earlier functions to deal with it. What about integer division, with separate quotient and remainder functions? What about rational numbers? You should probably stop before you start thinking about numbers like 3.141592653589…

Though the basic idea and motivation for this challenge goes back decades, we are indebted to Professor Philip Klein of Brown University for providing the structure here.