Assignment: 03  
Due: Tuesday, October 6 at noon (Waterloo time)  
Language level: Beginning Student  
Files to submit: warmup.rkt, vowels.rkt, relative.rkt, alternate.rkt, division.rkt, and factors.rkt  
Practice exercises: HtDP 10.1.6, 10.1.8, and 11.4.7, and 11.5.3

- Make sure you read the OFFICIAL A3 post on Piazza for the answers to frequently asked questions.
- This assignment covers concepts up to the end of Module 07. Unless otherwise specified, you may only use Racket language features we have covered up to that point.
- Policies from Assignments 1 and 2 carry forward.
- Your solutions must be entirely your own work.
- It is very important that your function names match ours. You must use the basic tests to be sure. In most cases, solutions that do not pass the basic tests will not receive any correctness marks. The names of the functions must be written exactly. The names of the parameters are up to you, but should be meaningful. The order and meaning of the parameters are carefully specified in each problem.
- You may use examples from the problem description in your own tests (but you should have many more than that).
- All of the questions can be solved with simple recursion. However, simple recursion is not mandatory.
1. [5% Correctness]

For this question you will be solving several different problems.

(a) Write the function replace-word that consumes two strings and a list of strings, and produces a new list where all occurrences of the first string have been replaced by the second string.

Example:

(check-expect
  (replace-word "exam" "assessment"
    (cons "content"
      (cons "exam"
        (cons "assignment" empty)))))

(b) Write the Racket function add that consumes two natural numbers and produces their sum (just like the + function). You may not use the + function (or the - function). Instead use add1, sub1, and recursion on natural numbers.

Example:

(check-expect (add 2 3) 5)

(c) Write the Racket function mult that consumes two natural numbers and produces their product (just like the * function). You may not use the * function (or the +, -, or / function). Instead use add1, sub1, and recursion on natural numbers. (You can also use add from the previous part of this question as a helper function, since it follows the same restrictions).

Example:

(check-expect (mult 2 3) 6)

Place your solutions in warmup.rkt

2. [10% Correctness]

Write the Racket function total-vowels that consumes a list of strings, and produces the total number of lower-case vowels in all of the strings. For the purposes of this question, the vowels are "a", "e", "i", "o", and "u".

Example:

(check-expect (total-vowels (cons "test" (cons "look" empty))) 3)

Place your solutions in vowels.rkt

3. [10% Correctness]

It is sometimes useful to express numbers relative to previous numbers, rather than in absolute terms.
Write the function `differences` that consumes a non-empty list of numbers, and produces a list of the differences between each number and the number before it. The produced list will always have one fewer number than the consumed list (since the first number doesn’t have a number before it). If there is only one number, then there are no differences (so the list should be empty).

Example:

```racket
(check-expect
 (differences (cons 4 (cons 7 (cons 1 empty))))
 (cons 3 (cons -6 empty)))
```

Place your solutions in `relative.rkt`

4. Consider the following alternate definition for a natural number:

A Natural Number (Nat) is one of the following

- `0`
- `(* 2 Nat)`
- `(add1 (* 2 Nat))`

(a) [5% Correctness]

Come up with an alternate version of the natural number template that follows the structure of the data definition given. (You do not need to copy the data definition to your file).

Call your template `alt-nat-template`. (Note: The template should be an actual definition, not a comment.)

Hint: you should use `even?` and `odd?` to determine which of the two recursive cases applies. (Of course, 0 is also even, so make sure your conditions are in the correct order).

(b) [5% Correctness]

Using your template, write a function `powers-of-2` that consumes a natural number and determines how many times it can be divided by 2 before either reaching 0 or an odd number. If it starts at either 0 or an odd number, then it can be divided no (0) times.

Examples:

```racket
(check-expect (powers-of-2 12) 2)
(check-expect (powers-of-2 0) 0)
```

Explanations: 12/2 = 6, 6/2 = 3, 3 is not evenly divisible by 2. 0 is already 0, so no steps are taken.

Place your solutions in `alternate.rkt`
5. [20% Correctness]

Building on Question 1, parts b. and c., you will be implementing division without using Racket’s built-in division functions. More specifically:

Write the Racket function `divide` that consumes two natural numbers `a` and `b`, and produces a list of exactly two numbers. The first element of the list is the quotient when `a` is divided by `b`, and the second element is the remainder. Remember: division by 0 is undefined, so your function should include appropriate documentation to communicate this restriction.

To make things interesting, for this question you may only use the following mathematical functions: `+`, `-`, `*`, `add1`, and `sub1`. You may also use any numerical predicates such as (but not limited to) `<` and `zero?`.

Example: (check-expect (divide 17 5) (cons 3 (cons 2 empty)))

Hint: When computing \(a/b\) for natural numbers `a` and `b`, you are trying to find a solution to \(a = q \times b + r\) where the remainder (r) satisfies \(0 \leq r < b\)

Place your solutions in `division.rkt`

6. [15% Correctness]

(a) Write the function `all-factors` that consumes a natural number `n`, and produces a list of all natural numbers \(x\) where \(0 < x < n\) and \(x\) divides `n` evenly. In other words, the factors of `n` that are less than `n`. The factors must be in ascending order (smallest to largest).

Example:

(check-expect (all-factors 30) (cons 1 (cons 2 (cons 3 (cons 5 (cons 6 (cons 10 (cons 15 empty))))))))

(b) Write the function `is-prime?` that consumes a natural number and produces true if it is prime, and false otherwise. A prime number is a positive integer that has exactly two distinct factors: 1 and itself. (1 is not prime because it has only one factor, itself).

Examples:

(check-expect (is-prime? 30) false)
(check-expect (is-prime? 7) true)

(c) Write the function `is-composite?` that consumes a natural number and produces true if it is a composite number, and false otherwise. A composite number is a positive integer that has at least one factor other than 1 and itself. While 0 is evenly divisible by all positive natural numbers, it is not considered to be composite.

Examples:
Place your answers in `factors.rkt`.

**Challenges and 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, like in question 5? What about rational numbers? You should probably stop before you start thinking about numbers like $3.141592653589\ldots$
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.