Winter 2024
Victoria Sakhnini

Assignment: 04

Due:	Tuesday, February 13, 2024 9:00 pm
Coverage:	Slide 09 of Module 7
Language level:	Beginning Student
Allowed recursion:	Simple recursion
Files to submit:	`range.rkt`, `hot-dog.rkt`, `div-by-3.rkt`, `bonus-a04.rkt`

- **Make sure you read the OFFICIAL A04 post on Piazza** for the answers to frequently asked questions.
- Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above (Slide 09 of Module 07).
- Unless otherwise specified, you may use any function (or its helper) that you wrote in a previous part of a question as a helper function for that question.
- It is likely that your functions will not be very *efficient*, and may be slow on long lists. There is no need to test your functions with excessively long lists.
- The names of functions we tell you to write, and symbols and strings we specify must match the descriptions in the assignment questions exactly. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic test results will catch many, but not necessarily all of these types of errors.
- Policies from Assignment A03 carry forward.

Here are the assignment questions you need to solve and submit.

1. **(10%)**: In this question you will perform step-by-step evaluations of Racket programs, as you did in assignment one. Please review the instructions on stepping in A01.

   To begin, visit this web page:

   `https://www.student.cs.uwaterloo.ca/~cs135/stepping`

   Complete the two required questions under the "Module 5b: Lists with Recursion" category and the two required questions under the "Module 6: Natural Numbers" category.

2. **(35%)**: Place your solution for the following two parts in a file named `range.rkt`.

   (a) Write a function `in-range` that consumes two numbers (`a` and `b`) and a list of numbers and produces the number of elements in the list that are between `a` and `b` (inclusive). The range is inclusive, so the numbers `3`, `pi` and `4` are all between `4` and `3`.

   Note1: if a > b, the function should produce the number of elements in the list that are in the range [b,a].

   (b) Write a function `spread` that consumes a non-empty list of numbers and produces the non-negative difference between the maximum element in the list and minimum element of the list. If the maximum element is the same as the minimum element, `spread` produces zero.

3. **(40%)**: You may already be familiar with the amazing and entertaining app that can detect if an image contains a *hot dog*, or if it does *not* contain a hot dog. [Android] [iPhone].

   Place your solution for the following two parts in a file named `hot-dog.rkt`.

   Important! For this question you may not use the built-in `member?` function.

   (a) Write a predicate `contains-hot-dog?` that consumes a list of `Any` and produces `true` if the list contains the symbol `'hot-dog` and `false` otherwise. The symbol must be exactly `'hot-dog` (all in lower/down case with a hyphen in the middle).

   ```
   (contains-hot-dog? (cons 'pizza (cons 'hot-dog (cons 'hamburger
      empty)))) => true
   ```

   (b) Write a predicate `spells-hot-dog?` that consumes a string and produces `true` if it contains the letters required to spell `"hot dog"` and `false` otherwise. In other words, it contains at least one of each of the following: {h, t, d, g, space} and at least two o's. The letters may appear in the string as either upper or lower/down case letters.

   The following two string arguments produce `true`: `"Hot Dog!"`, `"abcdefgh too"`
   The following two string arguments produce `false`: `"hot-dog"`, `"hotdg"`

   Pro Tip: There is an example from the slides that you may find very useful. As always, you may use examples from the slides as part of your solution.

4. **(15%)**: An alternative definition for natural numbers is that 0, 1, and 2 are all natural numbers and any number that is three larger than a natural number is also a natural number.

   You may not use any form of multiplication or division (including remainder, modulo, etc.) for this question.

(a) First, write a data definition for natural numbers using the above observation. Name this data definition `Nat3` to distinguish it from `Nat`.

Next, write a function template for `Nat3`. Call it `nat3-template`.

The only design recipe components required for templates are the contract and the function definition. For the function definition, use `...` as placeholders, as in the templates shown in the course notes

Your code must still be syntactically correct, so it can "run", but do not try to apply your `nat3-template` function. Note that this question will be marked by hand, and there will be no feedback on this question from the basic tests.

(b) 0 is divisible by three. So is $0+3$, $0+3+3$, $0+3+3+3$, and so on.

Write a predicate, `div-by-3?` based on `nat3-template` which consumes a `Nat3` and produces `true` if it is divisible by 3 and `false` otherwise.

Submit your data definition, template, and code in a file named `div-by-3.rkt`.

This concludes the list of questions for you to submit solutions. Don't forget to always check the basic test results after making a submission.

Assignments will sometimes have additional questions that you may submit for bonus marks.

5. **(Bonus 5%):** Recall `div-by-3?` from the problems above. Another way to to determine whether a natural number is divisible by three is to sum the digits. If the sum of the digits mod 3 is 0 then the number itself is divisible by three. For example, `1+2+3+4+5 mod 3 = 0` and so we know that 12345 is divisible by 3.

   Write a function `div-by-3-alt?` that consumes a `Nat` and produces `true` if it is divisible by 3 and `false` otherwise.

   Download (that's not the same as cutting and pasting!) the file `a04bonuslib.rkt` and save it to the same directory you'll be saving your code for Assignment 04 in. This file contains two functions - `last-digit` and `other-digits` you may find useful as helpers for this question. Add a line `(require "a04bonuslib.rkt")` at the top of `bonus-a04.rkt` The `require` line is all that needs to be done to use the two functions in `a04bonuslib.rkt` in the code you write in `bonus-a04.rkt`.

   Do not copy the two functions into `bonus-a04.rkt` - you will receive zero marks for this question if you do so.

   Restrictions: Your solution must use the observations above about summing digits. You may use `last-digit` and `other-digits`, boolean expressions, numerical comparison functions like `<`, `cond`, `define`, `+`, and `-` (and nothing else).

   A data definition for decimal integers may be helpful:

```
;; A Digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
;; A DecInt is one of:
;;   ○ a Digit
;;   ○ (+ (* b 10) d) where b is a DecInt and d is a Digit
```

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.