Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

# Assignment: 04

| | |
|---|---|
| Due: | Tuesday, October 18, 2022 9:00 pm |
| Coverage: | Slide 12 of Module 08 |
| Language level: | Beginning Student with List Abbreviations |
| Allowed recursion: | simple recursion (see the note in question 4) |
| Files to submit: | `range.rkt, hot-dog.rkt, sarcasm.rkt, fizz-buzz.rkt, nat-list.rkt` |

- **Make sure you read the OFFICIAL A04 post on Piazza** for the answers to frequently asked questions.
- Because of Reading Week, **no early examples** are required for this assignment. There is **no friday deadline**, only the Tuesday deadline (after reading week).
- Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above (Slide 12 of Module 08). You may also use the builtin functions `char-upcase` and `char-downcase`.
- 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.

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

1. 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 6b: Lists with Recursion" category and the two required questions under the "Module 7: Natural Numbers" category.

2. 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`.

   (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. 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? (list 'pizza 'hot-dog 'hamburger)) => 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. Place your solution for the following two parts in a file named `sarcasm.rkt`.

   (a) As described in Module 06, a recursive function application is considered *simple recursion* when it moves *one step* closer to a base case *according to a data definition*.

   You may wish to write a function that can process two subsequent elements of a list at once, but that does not follow the data definition for a `(listof X)`.

   However, we can create a new data definition:

   ```
   ;; A (pair-listof X) is one of:
   ;; * empty
   ;; * (cons X empty)
   ;; * (cons X (cons X (pair-listof X)))
   ```

   Write the function template for a function named `pair-listof-X-template` that processes a `(pair-listof X)`. The only design recipe components required are the contract and the function definition.

   For the function definition, use `...` as placeholders, as in the templates shown in the notes and the example below.

   ```
   ;; listof-X-template: (listof X) -> Any
   (define (listof-X-template lox)
     (cond [(empty? lox) ...]
           [(cons? lox) (... (first lox)
                             (listof-X-template (rest lox)))]))
   ```

   Your code must still be syntactically correct, so it can "run", but do not try to apply your `pair-listof-X-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) As discussed in Module 05, understanding the semantics of English can often be quite difficult. In particular, sarcasm can be very difficult to understand, especially when in written form (as opposed to spoken form). A popular method of indicating sarcasm when communicating online is to use *sarcasm case*.

   `FoR ExAmPlE, tHiS SeNtEnCe iS WrItTeN In sArCaSm cAsE.`

   Starting with the first character, which is in position zero, all characters in an even position are in upper case, and all characters that are in an odd position are in lower/down case.

   Write a function `sarcastic` that consumes a string and produces that string in *sarcasm case*. The above example illustrates a string produced by `sarcastic`. Note that the presence of spaces or punctuation does not affect the behaviour of the function. Only the position of a character determines whether a character is in upper or lower case.

   ```
   (sarcastic "GOOD LUCK... you'll need it!")
          => "GoOd lUcK... yOu'lL NeEd iT!"
   ```

*Note:* We strongly recommend you use the template function you developed in part (a) and use simple recursion. If you choose to ignore our recommendation, we will allow alternate solutions and you are not required to use simple recursion for this question.

5. The children's game "Fizz Buzz" has become an infamous interview question for introductory programmers. If you are not familiar with the children's game, you might want to review the Wikipedia Entry.

   For the Waterloo version of the game, four parameters are required:

   - `start`, the first integer in the produced sequence
   - `end`, an integer greater than or equal to `start`, the last integer in the produced sequence
   - `fizz`, a positive integer (3 in the Children's game)
   - `buzz`, a positive integer different from `fizz` (5 in the Children's game)

   In the Waterloo version, a sequence is produced consisting of integers that increment from `start` to `end` inclusively. If a number is divisible by `fizz` then it is replaced by the symbol `'fizz` and if it is divisible by `buzz` then it is replaced by `'buzz`. The additional Waterloo twist is that if a number is divisible by both `fizz` and `buzz` it is replaced by the symbol `'honk` (as a tribute to the Canadian Goose).

   Write a function `fizz-buzz` that consumes four parameters: `start end fizz buzz`, and produces the Waterloo version of the "Fizz Buzz" sequence as a list. You might find the `remainder` function quite helpful.

   ```
   (fizz-buzz 8 15 3 5) => (list 8 'fizz 'buzz 11 'fizz 13 14 'honk)
   ```

   In your contract, use `(listof (anyof Int Sym))` for the type produced by `fizz-buzz` (this will be introduced in Module 08).

   Place your solution in a file named `fizz-buzz.rkt`.

6. We have already seen how Racket can handle large natural numbers (for example, try `(expt 2 10000)`).

   As a gross over-simplification of how Racket internally stores large natural numbers in memory, we can imagine that Racket stores them as a non-empty list of single `Digit`s, where:

   ```
   ;; A Digit is one of: 0 1 2 3 4 5 6 7 8 9
   ```

   For example, the number `8675309` could be represented as:

   ```
   (list 9 0 3 5 7 6 8)
   ```

Note that it might seem counter-intuitive to store the rightmost `Digit` (the "ones `Digit`") as the first element in the list, but this makes performing any arithmetic operation on the representation much more straightforward.

For this question, ignore the fact that you can write superfluous leading zeros at the beginning of a natural number (*e.g.,* 007).

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

(a) Write a function `nat->list` that converts a natural number `n` into a list of `Digit`s.

```
(nat->list 8675309) => (list 9 0 3 5 7 6 8)
(nat->list 0) => (list 0)
```

To solve this problem with simple recursion, use the following data definition:

```
;; A Nat is one of:
;; - Digit
;; - Digit + 10 * Nat
```

You are not required to write a template for this data definition, but it is recommended. To write such a template function, consider what the "opposite" of multiplication is (hint: remember `quotient` and `remainder` from A00).

(b) Write a function `list->nat` that converts a non-empty list of `Digit`s back into a natural number.

```
(list->nat (list 9 0 3 5 7 6 8)) => 8675309
(list->nat (list 0)) => 0
```