Assignment: 5
Due: Tuesday, February 7th, 9:00 pm
Language level: Beginning Student with list abbreviations
Allowed recursion: Pure structural recursion
Files to submit: lists.rkt, numbers.rkt, fmembers.rkt
Warmup exercises: HtDP 10.1.4, 10.1.5, 11.2.1, 11.2.2, 11.4.3, 11.5.1, 11.5.2, 11.5.3
Practise exercises: HtDP 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7

• Coverage: until M6-37
• You should note a heading at the top of the assignment: “Allowed recursion.” In this case the heading restricts you to pure structural recursion. That is, the recursion that follows the data definition of the data it consumes, and parameters of the recursive calls are either unchanged or one step closer to the base case.
• Provide your own examples that are distinct from the examples given in the question description of each function.
• Of the built-in list functions, you may only use cons, first, second, rest, empty?, cons?, (list ...), member?, and append.
• Do not use quoted lists for this assignment. Use (list...) instead.
• Your solutions must be entirely your own work.
• For this and all subsequent assignments, you should include the design recipe for all functions, including helper functions, as discussed in class.
• Solutions will be marked for both correctness and good style as outlined in the Style Guide.
• In your function contracts, follow the (listof X) notation that is given on Slide 12 of Module 05 (e.g., include a list of numbers as (listof Num)).
• You must use the cond special form, and are not allowed to use if in any of your solutions.
• It is very important that the function names and number of parameters match ours. You must use the basic tests to be sure. 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.
Here are the assignment questions you need to submit.

1. (a) Write a function `between-list` which consumes a list of integers (`nlst`), and additional two integers (`low` and `high` where `low < high`) and produces a list of all values from `nlst` that are between `low` and `high` (inclusive for both). The values of the produced list must be in the same order as they were in the consumed list.
   For example, `(between-list (list -1 4 10 -5 0 3 -7) -1 7) => (list -1 4 0 3)`

(b) Write a function `create-divisible` which consumes three non-zero natural numbers (`low`, `high`, and `n`, where `low < high`), and produces a list of all natural numbers (in ascending order) that are between `low` and `high` (inclusive for both) that are divisible by `n` (meaning the remainder when dividing by `n` is zero).
   For example, `(create-divisible 3 17 5) => (list 5 10 15)`

(c) Write a function `avg-list` which consumes a non-empty list of numbers and produces the average.
   For example: `(avg-list (list 3 0 -5.5 6.82)) => 1.08`

(d) Write a function `my-substring` which consumes a string (`st`) and two natural numbers (`start`, and `end`) and produces a string (`end-start`) long that contains the same characters as `st` from `start` inclusive to `end` exclusive. Remember that the first position in a string corresponds to 0. Assume that `start` and `end` are less than or equal to the length of `st`, and `end` is greater than or equal to `start`.
   For example: `(my-substring "abcdefgh" 4 8) => "efgh"
   Note: you must NOT use any built-in string functions other than `string->list` and `list->string`.

Place your solutions in the file `lists.rkt`

2. (a) Write a function `multiply` which consumes two natural numbers (`a` and `b`) and produces `a * b`.
   Note: you may NOT use multiplication nor division in your solution. Your solution should use structural recursion.

(b) Write a function `calc` which consumes a non-zero natural number (`n`) and produces \[1 - 2^2 + 3^3 \ldots \pm n^n.\]

Place your solutions in the file `numbers.rkt`
3. In this question we will use the following data definitions:

```
(define-struct lecturer (name salary faculty))
;; A Lecturer is a (make-lecturer Sym Num Sym)
;; requires: salary is a non-negative number
;;
(define-struct professor (name salary faculty research-area))
;; A Professor is a (make-professor Sym Num Sym Sym)
;; requires: salary is a non-negative number
;;
;; A FacultyMember is one of:
;; * a Lecturer
;; * a Professor
```

(a) Write a function `same-faculty?` which consumes a list of `FacultyMember` structures and produces `true` if all members in the consumed list are from the same faculty and `false` otherwise. For empty list the function should produce `true`.

(b) Write a function `keep-faculty-list` which consumes a list of `FacultyMember` structures (`flist`) and a symbol (`afaculty`), and produces a list of all members from the consumed list who work in (`afaculty`). The values in the produced list must be in the same order as they were in the consumed list.

Place your solutions in the file `fmembers.rkt`

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

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

There is a strong connection between recursion and induction. Mathematical induction is the proof technique often used to prove the correctness of programs that use recursion; the structure of the induction parallels the structure of the function. As an example, consider the following function, which computes the sum of the first `n` natural numbers.

```
(define (sum-first n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum-first (sub1 n)))]))
```

To prove this program correct, we need to show that, for all natural numbers `n`, the result of evaluating `(sum-first n)` is \(\sum_{i=0}^{n} i\). We prove this by induction on `n`.

**Base case:** `n = 0`. When `n = 0`, we can use the semantics of Racket to evaluate `(sum-first 0)` as follows:
Since $0 = \sum_{i=0}^{0} i$, we have proved the base case.

**Inductive step:** Given $n > 0$, we assume that the program is correct for the input $n - 1$, that is, $(\text{sum-first (sub1 n)})$ evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of $(\text{sum-first n})$ proceeds as follows:

\[
\begin{align*}
(\text{sum-first n}) &\quad \text{yields (cond [(zero? n) 0][else \ldots]) (we know n > 0)} \\
&\quad \text{yields (cond [false 0][else \ldots])} \\
&\quad \text{yields (cond [else (+ n (sum-first (sub1 n)))]}) \\
&\quad \text{yields (+ n (sum-first (sub1 n)))}
\end{align*}
\]

Now we use the inductive hypothesis to assert that $(\text{sum-first (sub1 n)})$ evaluates to $s = \sum_{i=0}^{n-1} i$. Then $(+ n s)$ evaluates to $n + \sum_{i=0}^{n-1} i$, or $\sum_{i=0}^{n} i$, as required. This completes the proof by induction.

Use a similar proof to show that, for all natural numbers $n$, $(\text{sum-first n})$ evaluates to $(n^2 + n)/2$.

**Note:** Summing the first $n$ natural numbers in imperative languages such as C++ or Java would be done using a for or while loop. But proving such a loop correct, even such a simple loop, is considerably more complicated, because typically some variable is accumulating the sum, and its value keeps changing. Thus the induction needs to be done over time, or number of statements executed, or number of iterations of the loop, and it is messier because the semantic model in these languages is so far-removed from the language itself. Special temporal logics have been developed to deal with the problem of proving larger imperative programs correct.

The general problem of being confident, whether through a mathematical proof or some other formal process, that the specification of a program matches its implementation is of great importance in safety-critical software, where the consequences of a mismatch might be quite severe (for instance, when it occurs with software to control an airplane, or a nuclear power plant).