Fall 2024
Abouelkhair, Becker, Jung, Petrick, Reetz, Roh

Assignment: 5

Examples Due:	Thurs, October 10, 2024 9:00 pm
Remainder Due:	Tuesday, October 22, 2024 9:00 pm
Coverage:	Slide 29 of Module 08
Language level:	Beginning Student with List Abbreviations
Files to submit:	`examples-a05.rkt`, `course-selection.rkt`, `symbol-lists.rkt`, `pizza-party.rkt`,

General assignment policies:

- General policies from previous assignments carry forward. **Policies** from the course web site apply.

A05-specific policies and advice:

- Consult the **official A05 post on Ed** for the answers to frequently asked questions.
- For all functions named in the assignment, examples must be submitted by the (earlier) deadline above.
- It is possible 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.

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

1. **(10%):** Complete all the required stepping problems in **Module 8: Nested Lists** at

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

   You should refer to the instructions from A01 Question 1 for the stepper question instructions.

2. **(30%)**: Before the start of every term, UWaterloo asks students to choose the courses they intend to take in the following term during the Course Selection period. In this question, we will be representing students and the courses they have selected as a `DesiredCourses`.

```
;; A DesiredCourses is one of:
;; * empty
;; * (cons (list Str (listof Sym)) DesiredCourses)
```

   An example `DesiredCourses`:

```
(define selections
(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347))))
```

Each key-value pair in a `DesiredCourses` represents an enrolled student. The first element in the pair is a student's alphanumeric Quest username (e.g. `"mpines"`) and the second element is a list of the codes of the courses they wish to take, represented as symbols (e.g. `'CS135`).

Implement the following functions:

(a) `missed-deadline-add` consumes a `DesiredCourses` and a student's Quest username. If a student's Quest username already appears in the consumed `DesiredCourses`, the consumed `DesiredCourses` is simply produced. If the student is not found, a new key-value pair for the consumed student is added to the **end** of the `DesiredCourses`. Since the student has sadly missed the course selection deadline, they should have no courses associated with them in the `DesiredCourses`. For example, `(missed-deadline-add selections "w2cordur")` produces:

```
(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347))
  (list "w2cordur" empty))
```

(b) `taking-course?` consumes a `DesiredCourses`, a student's Quest username, and a course code, and produces `true` if the student has selected the course (and `false` otherwise). For example, `(taking-course? selections "d32pines" 'CS115)` produces `true`, but `(taking-course? selections "d32pines" 'CS135)` produces `false`. If the consumed Quest username does not appear in the `DesiredCourses`, produce `false`.

(c) `add-course` consumes a `DesiredCourses`, a student's Quest username, and a course code, and adds the given code to the **end** of the list of courses the student wishes to take. If the student has already selected the course, do not change the `DesiredCourses`. If the Quest id does not appear in the `DesiredCourses`, add a new key-value pair for the student, along with the desired course, to the **end** of the `DesiredCourses`.
For example, `(add-course selections "gnclstan" 'CS246)` produces:

```
(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347 'CS246)))
```

and `(add-course empty "mdluffy" 'CS246)` produces:

```
(list (list "mdluffy" (list 'CS246)))
```

(d) `create-classlist` consumes a `DesiredCourses` and a course code, and produces a list of all the students that want to take the consumed course. The students' Quest username must appear in the same order as in the consumed `DesiredCourses`. For example, `(create-classlist selections 'MATH135)` produces `(list "mpines" "d32pines")`.

Place your solutions to the above problem in `course-selection.rkt`.

3. **(10%)**: Write a function `make-symbol-lists` that consumes a list of natural numbers and a symbol and produces a nested list where inner list k has length equal to the k^{th} natural number from the consumed list and is composed only of the consumed symbol.
For example:

```
(check-expect (make-symbol-lists (list 2 1 3) 'X)
              (list (list 'X 'X) (list 'X) (list 'X 'X 'X)))
```

Place your solutions to the above problems in `symbol-lists.rkt`.

4. **(50%)**: Suppose we want to organize a class pizza party (perhaps to celebrate Halloween). Students may choose any number of slices from one of three pizza types, represented by symbols: `'Hawaiian`, `'meaty` or `'veggie`. We will assume that each of the above are also available in various forms to meet any other dietary restrictions so no one is left out.

A `StudentChoice` is a fixed-length list containing three items (in the following order): a student's name (string), their pizza type (symbol from the list above) and the number of desired slices (natural number).

To help organize the pizza requests, students are grouped by their respective lecture section where each `Section` has an instructor (string), section number (positive integer) and a list of `StudentChoice`, in that order. Sections are then grouped together as a `Course`; i.e., a `Course` is a `(listof Section)`.

(a) Write full data definitions for `StudentChoice`, `Section`, `Course` and corresponding templates `studentchoice-template`, `section-template`, `course-template` for functions that consumes them.
Remember that a template also always includes a contract.
Hint: you may find it useful to also write a template for a function that consumes a `(listof StudentChoice)`.

(b) A curious instructor wants to know what the most popular pizza type choice is in their section. Write a function popular-pizza that consumes a Section and produces the symbol of the most popular type (based on the number of students who choose that type not slice count). For simplicity, you may assume that the count for each pizza type will be distinct. Also, only provide examples and tests where the counts are distinct.

(c) To help organize orders, write a sort function sort-choices that consumes a Section and produces the same Section where the list of StudentChoice is ordered by pizza type (i.e., all 'Hawaiian choices first, then 'meaty, followed by 'veggie). Also, for a given pizza type, the corresponding StudentChoices are ordered by name in alphabetical order; i.e., the list of StudentChoice in the resulting Section will have three groups, one for each pizza type, and each group will be sorted alphabetically.

    To perform the sort, write a function choices<= that consumes two StudentChoices and produces true if the the first StudentChoice should come before the second StudentChoice in the ordering; false otherwise. Then modify insertion sort from Module 06 to use choices<= to complete the sort.

(d) Between providing their choice and the pizza arriving, some students may not remember their choice. Write a function pizza-lookup that consumes a Course, a section number and a student name and produces a fixed-length list containing the pizza type and slice count for the student (in that order). You may assume that a StudentChoice for the consumed student name will be found within the matching section.

(e) Write a function count-slices that consumes a Course and produces a fixed-length list of three items corresponding to the total slice count for each type of pizza (count for 'Hawaiian first, then 'meaty, followed by 'veggie) for everyone in the course.

Place your solutions to the above problems in pizza-party.rkt.

This concludes the list of questions for you to submit solutions, but see the following pages as well, especially the bonus. Don't forget to always check 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:

```
(sum-first 0) ; =>
(cond [(zero? 0) 0][else ...]) ; =>
(cond [true 0][else ...]) ; =>
0
```

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, *(sum-first (sub1 n))* evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of *(sum-first n)* proceeds as follows:

```
(sum-first n) ; =>
(cond [(zero? n) 0][else ...]) ;(we know $n > 0$) =>
(cond [false 0][else ...]) ; =>
(cond [else (+ n (sum-first (sub1 n)))]) ; =>
(+ n (sum-first (sub1 n)))
```

Now we use the inductive hypothesis to assert that *(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, *(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).