First Midterm

Please read all the problems carefully. Do everything we ask for, neither more nor less. If you have any questions on what a problem means, don't hesitate to ask. Don't get bogged down on any one problem; there are 50 points possible, plus some extra credit, so allow approximately one minute per point. If you have trouble on a problem, go on to the next one.

In answering these questions, you may use any Scheme function we have covered in class, in the text, in the homework, or earlier on this exam, unless a problem says otherwise. You may use functions defined in earlier problems regardless of whether your answer to the earlier problem was correct; in fact, if you rebuild the same code a second time instead of using a previously defined function, you won't get full credit. If you have a question about whether we covered a function or precisely what it does, ask us. Unless a problem says otherwise, you may write auxiliary (helper) functions as part of any answer. Unless a problem specifically asks you to consider errors, you should assume that each problem is correct and solvable, and ask us if you believe otherwise.

Please write your answers clearly—we can't give you credit if we can't decipher what you've written. We'll give partial credit for partially correct answers, so writing something is better than writing nothing. Good luck!

Problem 1 (3 points)

Evaluate each of the following expressions. That is, what does DrScheme display in the interactions window when you enter the expression(s) in the definitions window and click Run?

```
(define UC-CAMPUSES 10)
(define CALSTATE-CAMPUSES 23)
```

(a)
```
(* (- 20 16) (+ 16 4))
```

(b)
```
(>= (/ 66 6) 12)
```

(c)
```
(and   (>= UC-CAMPUSES 10)
       (< UC-CAMPUSES CALSTATE-CAMPUSES))
```

(d)
```
(* 5
   (cond
      ((< CALSTATE-CAMPUSES (* UC-CAMPUSES 2)) 30)
      (else 20)))
```

Problem 2 (5 points)

Suppose you have two programs that produce exactly the same results (the same outputs for the same inputs). On the back of this page, briefly list two ways in which one of the programs could still be better than the other.

Problem 1 (3 points)
Problem 2 (5 points)
Problem 3 (6 points)
Problem 4 (19 points)
Problem 5 (27 points)
Problem 6 (10 points xc)
Total (60points+10xc)

Problem 3 (6 points)

Evaluate each of the following expressions. You may show lists in any of three ways: `(cons "A" (cons "B" empty))`, `(list "A" "B")`, or `'("A" "B")`. Use this definition independently for each of the five parts:

```
(define L (list "assam" "jasmine" "earl grey" "hibiscus"))
```

(a) `(first L)`

(b) `(rest L)`

(c) `(first (rest L))`

(d) `(cons (first L) (rest (rest L)))`

(e)
```
(cond
   ((equal? (cons "rooibois" empty) "rooibois") "Tea")
   ((cons? empty)                               "With")
   ((empty? (rest (rest (rest L))))             "Milk")
   ((string=? (first (list "Mint")) "Mint")     "And")
   (else                                        "Honey"))
```

Problem 4 (19 points)

With the state billions of dollars in debt, UCI's registrar has turned to student labor and asked you to implement some software. You decide to represent each course in a structure with five fields:

- dept (the department or school that offers the course, e.g., `"I&CSCI"` or `"IN4MATX"`)
- number (a string containing the course number, e.g., `"H21"` or `"139W"`)
- title (a string containing the course title, e.g., `"HNRS INTR COM SCI I"`)
- desc (a string containing the catalogue description)
- enroll (the total number of students enrolled in this course so far this academic year)

(a) (2 points) Define the structure `course` with the fields given above.

(b) (2 points) Write a Scheme expression to create this course, with 125 students enrolled: Math 6B, Boolean Algebra/Logic, with a course description of "Important Stuff" (just to save you some writing). Note that we're not asking you to define a function here.

(c) (3 points) Complete the following definition for the predicate function `valid-course?`, which takes an expression and checks whether it appears to be a valid course.

```
(define valid-course?
  (lambda (X)
    (and                                  ; and, like +, can take more than 2 arguments
      (__________________ X)                  ; is it a course structure at all?
      (dept-code-valid? (__________________ X)); is the department code valid?
      (number-valid? (course-number X))
      (title-valid? (__________________ X))    ; is the title valid?
      (desc-valid? (course-desc X))
      (and (number? (enroll X)) (>= (enroll X) 0)))))
```

(d) (4 points) A course description is valid if it contains at least one word and at most 40 words. Write a definition for the predicate `desc-valid?` that takes a course structure and returns true if and only if its description is valid as just described. You may assume that a function `word-count`, which takes a string and returns the number of words in that string, is already defined; using `word-count` in your definition of `desc-valid?` should make the task pretty easy.

```
;; desc-valid?:  course -> boolean
;; Return true if the course's description is valid and false otherwise
```

(e) (4 points) Assume that you have already defined `DEPT-CODE-LIST`, which is a list of all the valid school and department abbreviations (e.g., `"I&CSCI"` or `"MATH"` or `"ECON"`). Write a definition for the predicate function `dept-code-valid?`, which takes a course structure and a list of strings and returns true if its department code is an element of the list of strings. (Hint: You can do this without recursion if you use a function we defined in class.)

```
;; dept-code-valid?:  course  list-of-strings  ->  boolean
;; Return true if the course's department code is on the list
```

(f) (4 points) We'll say that two courses are the same if they have the same department and the same course number. Define the function `courses-match?` according to this contract and purpose statement:

```
;; courses-match?:  course  course  ->  boolean
;; Return true if both inputs have the same department and the same course number
```

Problem 5 (27 points)

The registrar stores all of the courses in a list of course structures; essentially, that is the catalogue.

(a) (5 points) Write a definition for the function `total-enrollment`, which takes a catalogue (i.e., a list of courses) and returns the sum of all the enrollments of all the courses on the list.

```
;; total-enrollment:  list-of-course  ->  number
;; Return the total enrollment in all the courses
```

(b) (3 points) Define the function `average-enrollment` that takes a catalogue (i.e., a list of courses) and returns the average enrollment in the courses in the catalogue. (Hint: Use what you've already defined. You may use an auxiliary function if you need it.)

```
;; average-enrollment:  list-of-course  ->  number
;; Return the average (mean) enrollment of all courses in the input list
```

(c) (6 points) Define the function `course-in-catalogue?`, which takes a course and a catalogue and returns true if the catalog contains a course that matches the argument's department code and course number. (Hint: Make use of what you've already defined.)

```
;; course-in-catalogue?:  course  list-of-course  ->  boolean
;; Return true if the course matches some course in the list (in dept and number)
```

(d) (6 points) In keeping with the dire budget situation, the registrar may need to plan for the disestablishment of certain programs or departments, which would mean removing all of that department's courses from the catalogue.

Write a definition for the function `disestablish`, which takes a catalogue (i.e., a list of courses) and a string (representing a department code code) and returns a catalogue (i.e., a course list) with all the courses matching the specified department removed.

```
;; disestablish:  list-of-course  string  ->  list-of-course
;; Return the course list with all the courses whose dept matches the string removed.
```

(e) (7 points) A somewhat less drastic way of saving money is to reduce administrative costs by merging departments; in that case, all of a department's courses are re-listed in a new department. Write a definition of `merge-departments`, which takes three arguments—a catalogue (i.e., a list of courses), an original department code and a new department code (both strings)—and returns the catalogue with the code of all the original department's courses changed to the new department. You may assume that there will be no duplicate course numbers. (Hint: First write `change-dept`, which takes a single course and a new department code and returns a course with the new code substituted for the original one.)

```
;; merge-departments:  list-of-course  string  string  ->  list-of-course
;; Return the course list with all course departments matching the first string
;; changed to the second string
```

Problem 6 (10 points extra credit)

(a) (4 points extra credit) Because strings may contain white space and punctuation, not to mention numbers or symbols that don't count as words, we often speak of strings as containing "tokens," where a token is a string of characters that are separated from the next token by white space or punctuation. (The precise characters that count as white space or punctuation we won't worry about here.)

Define the function `word-count`, which takes a string and returns the number of valid words in the string. Note that some tokens may not be valid words. In your definition, use these functions:

- `first-token`, which takes a string and returns the first token in that string
- `rest-of-tokens`, which takes a string and returns a copy of that string with the first token removed
- `string-empty?`, which takes a string and returns true if it's empty or contains only white space and punctuation
- `word?`, which takes a string and returns true if the string is a valid word.

```
;; word-count: string -> number
;; Return the number of valid words in the string
```

(b) (6 points extra credit) Write a new definition for `merge-departments`: Before adding a course with a changed department name to the catalogue, this version checks to see if a course with that department/number is already there. If it is, it changes the number (using a function called `mutate-number` that you don't have to write) before adding the course.

Take this approach: First, write functions to create two lists—the courses that aren't being changed and the courses that are. Then, write the function `add-one-course` to take one changed course and the list of unchanged courses, either adding the course to the list (if it's not a duplicate—use the function `course-in-catalog?`) or changing its number to something unique and then adding it. Next, write a function `add-courses` that takes the two lists and calls `add-one-course` on each of the changed courses, adding them one by one to the list of unchanged courses. Finally, `merge-departments` calls `add-courses`.

```
;; get-unchanged-courses:  list-of-course  string  ->  list-of-course
;; Return a list of courses on the input list whose dept does not match the string
```

```
;; get-changed-courses:  list-of-course  string  string  ->  list-of-course
;; Return a list of courses on the input list whose dept matches the first input
;; string, with the dept changed to the second string (using change-dept, above)
```

```
;; add-one-course:  course  list-of-course  ->  list-of-course
;; Add course to list (changing its course number if necessary to avoid duplication)
```

```
;; add-courses:  list-of-course  list-of-course  ->  list-of-course
;; Add each course on the first list to the second list (avoiding duplication)
```

```
;; merge-departments:  list-of-course  string  string  ->  list-of-course
;; Return the course list with all course departments matching the first string
;; changed to the second string (and course numbers changed to avoid duplication)
```