Computer Science CSC 324S: Programming Languages
University of Toronto – St. George Campus

Assignment LispB

Due Date: Tuesday, 16 March, 6:10pm exactly

Introduction and Requirements

All programs should be written in Lisp and should be well documented and thoroughly tested. Make sure you use good functional programming style; marks will be deducted for programs that look like translated procedural or object-oriented programming.

You are only allowed to use the following functions of Lisp: and, or, if, car (first), cdr (rest), cons, eq (and the other equality predicates), null, atom, listp, cond, defun, lambda, append, mapcar, length, apply, let, let* and (if necessary) helping functions defined in terms of these. setq and setf are **not** allowed. For questions which require the definition of one or more functions, verify your answers by running them before handing them in.

You must use the function names we ask you to. We will be marking part of this electronically. You must also indent your code to make it readable. (Emacs, available in cdf, provides good built-in indentation for Lisp.) You must comment your code clearly. The purpose of every variable must be stated. Every function must have a comment describing its parameters and what the method does.

We will mark you not only on the clarity of your comments, but grammar, punctuation, and spelling as well.

You must run your functions on test cases and generate a terminal session containing the results.

If you don't know how to generate a record of a terminal session, type the following command at a UNIX prompt: `man script`

A suite of test cases should encompass:

Trivial solutions: for example, stop condition for recursion;

Borderline cases: for example, empty lists;

More elaborate cases: for example, those that are "almost" stop conditions;

Complex cases, for example, nested lists, or more elaborate data structures.

Note that each terminal session will contain the responses Lisp returned when you tested your functions with your test cases.

Very roughly, 10% of the marks will be for documentation and formatting, 10% for functional programming style, 10% for your testing, 10% for our testing, and 60% for correctness.

Place all functions that are well-formed in a file `wellFormedA.lisp`, and all other functions in a file `notWellFormedA.lisp`. Capitalization is important. We are assuming that file `wellFormedA.lisp` contains functions that don't have any syntax errors; we must be able to evaluate the whole file without problems. Electronically submit both files, plus the output from your terminal session, to the directory `lispB`. You should also hand in a printout of your functions, along with your answer to question 1.

For your hardcopy:

Organize the items sequentially, in the order of the handout

Highlight the number (letter) of the item (using a highlighter)

If you are printing your assignment double-sided, make sure both sides are properly oriented

The terminal sessions should be printed and stapled separately

Put everything in an envelope and print your name and number on it.

Warning

Please do not leave any print statements in your final version. Simply return the values we ask for. Otherwise our automated testing will mark your functions as incorrect.

The Actual Assignment

```
1.  Given a sorted list of numbers, if you can write an efficient binary
    search, do so.  If not, explain why not.


2.  Write a function have-common-element that takes two list arguments l1 and
    l2 and returns true if there is an i such that the ith element of l1 and
    the ith element of l2 are equal.  Precondition: l1 and l2 have the same
    number of elements.

    Example: (have-common-element '(1 2 3) '(4 2 4)) should return true,
    because the second element of each is 2.


3.  Write a function pair-up that takes two list arguments and returns a list
    of pairs made up of corresponding elements in list1 and list2.

    Example: (pair-up '(a b) '(c d)) should return ((A C) (B D)).


4.  A marks file has the following form (we'll call it a "marks form"), for n
    marks and m students:

    (((mark-1-name  out-of)
      (mark-2-name  out-of)
      ...
      (mark-n-name  out-of))

     ((student-1-name  mark-1 mark-2 ... mark-n)
      (student-2-name  mark-1 mark-2 ... mark-n)
      ...
      (student-m-name  mark-1 mark-2 ... mark-n)))


    Write a function calculate-marks that takes a marks form and returns a
    list of pairs of student name and final marks.

    Details:

        All the marks have the same weight in the final mark; if there are
        three marks, they all contribute 1/3 toward the final mark.

        You can use '/' to divide a mark by its corresponding out-of; the
        result will be a rational number like 3/4.  You should leave them in
        that form rather than trying to produce a decimal.

    Example: (calculate-marks '(((a1 10)
```

```
                              (a2 20))
                             ((stud-1 5 8)
                              (stud-2 9 15))))
```

should return ((STUD-1 9/20) (STUD-2 33/40))

We STRONGLY suggest that you write several helper functions. Part of your mark will be on clear design, and helper functions can help tremendously with that.

5. Naturally, the next step when designing a marks program is to detect cheating. We're going to define "cheating" as getting the same mark as another student on an assignment.

To find all the cheaters in a marks form, we can produce a graph where each vertex represents a student. For this assignment, you are to produce an adjacency list representation. For example, the form

```
((a b c)
 (b)
 (c d)
 (d))
```

represents a graph with edges a->b, a->c, and c->d. As an example of how this works, in order to find the neighbours of a, simply take the cdr of the list that starts with a and you get the list of neighbours (b c). What this represents is that a cheated with b and c, and c cheated with d.

For the same marks form as in question 4, write a function make-suspicious that returns a graph of students who have the same mark on at least one assignment. For example, if the input is

```
(make-suspicious '(((a1 10)
                    (a2 20)
                    (a3 10))
                   ((stud-5  1  1  1)
                    (stud-1 10 15 15)
                    (stud-2  8 15 20)
                    (stud-3  5  8 10)
                    (stud-4 10  3 10))))
```

Then the graph would be

```
((stud-5)
 (stud-1 stud-2 stud-4)
 (stud-2)
 (stud-3 stud-4)
 (stud-4))
```

This is because stud-1 and stud-2 have the same mark on a2, stud-1 and stud-4 have the same mark on a1, and stud-3 and stud-4 have the same mark on a3.

```
Note that we do not require an edge to be listed twice -- we don't expect
both (stud-1 stud-4) and (stud-4 stud-1) to be listed.  Just having one
edge is fine.  Thought questions: Why do we have this clarification?  Why
is your life easier this way?

Last, note that even if there are no edges for a vertex, it should still
be listed, but it will have no adjacent vertices.
```