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

Assignment LispA

Due Date: Tuesday, 2 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 basic functions of Lisp: car (first), cdr (rest), cons, eq (and the other equality predicates), null, atom, listp, cond, defun, lambda and (if necessary) helping functions defined in terms of these. 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.

For example, suppose you are asked to implement a Lisp function that returns the square of its argument, which must be a number. Here is the commented Lisp code:

```
; SQUARE: return the square of obj.
; PRECONDITIONS: obj is a number.

(defun square (obj)
  (* obj obj))
```

Note that in the comment we don't specify how we get the result, since that may change. Writing clear comments saves work later, since the comment is correct for different implementations.

Here's another example:

```
; TEST-NUM: return true if obj is a number, false otherwise.  If obj is
; not a number, print an error message.
; PRECONDITIONS: none

(defun test-num (obj)
  (if (numberp obj) t
    (print "Argument is not a number.") ))
```

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 `lispA`. You should also hand in a printout of your functions, along with your answers to questions 1 and 6B.

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 use any print statements. Simply return the values we ask for. Otherwise our automated testing will mark your functions as incorrect.

The Actual Assignment

```
1.  Consider the Lisp forms

        (list e1 (car e2))
        (cons (list e1 e2) e3)
        (cons (cdr e1) (car e2))
        (cons (car e1) (cdr e2))
        (cons (car (car e1)) (car (cdr e2)))

    where e's are forms.  For each of them, state what restriction is
    required on the value of the e's, and what the value of the form is in
    terms of the value of its parameters. For example,

        (car e)    e must be a list, returns 1st element of val


2.  Write a function second-item that has one list parameter and returns the
   second item in the list.

    Examples:

        (second-item '(a b c))    should return b
        (second-item '(a (b c)))  should return (b c)
```

3. Write a function count-items that has one list parameter and returns the number of items in the list.

   Examples:

```
(count-items '(a b c))    should return 3
(count-items '(a (b)))    should return 2
(count-items '((a b)) )   should return 1
```

4. Write a function count-atoms that returns the number of top-level atoms.

   Examples:

```
(count-atoms '(a b c) )    should return 3
(count-atoms '(a (b)) )    should return 1 -- (b) is not an atom
(count-atoms '((a) (b)) ) should return 0 -- the only top-level items
                                             are lists.
```

5. Write a function sum-list that has one list parameter and returns the sum of the numbers in the list. (You may assume that only numbers will be in the list.)

6. A. Write a function sum-lists that has one parameter that is a list of lists of numbers, and returns a list that contains the sum of the corresponding sublist. Use sum-list as a helper function.

   Example:

```
(sum-lists '((1) (2 3 4) (4 5 3)) )  -- returns (1 9 12)
```

   B. Can you replace the call to sum-list with a lambda expression, and leave the rest of the function sum-lists the same? If so, write the function (call it sum-list2); if not, explain why not.

7. Write a function combine-lists that takes two parameters that are lists, and returns the first list mushed together with the second.

   Example:

```
(combine-lists '(a (b) c) '(d e))  -- returns (a (b) c d e)
```

8. Write a function second-item-atom that has one list parameter and returns true if the second item is an atom.

   Examples:

```
(second-item-atom '(a b c))    should return t
(second-item-atom '(a (b c)) ) should return nil
```

```
9.  To 'flatten' a list means to remove all parentheses within.  For example,
    if we were to flatten (a ((b) c)), we would end up with (a b c).

    Write a function flatten-prepend that takes two lists as arguments,
    flattens them, and prepends them.  For this function, you may NOT use
    helper functions or lambda expressions, including any functions that you
    have written previously.  (Note: this one is hard.  Work it out on paper
    with several examples before you start typing, or you'll type forever.)

    Example:

        (flatten-prepend '(a (b) c) '(d ((e) f)) )  -- returns (a b c d e f)
```