UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

**Prof. R. Fateman**

**Fall, 2002**

**Sketch of solutions to Assignment 1: Learn Common Lisp Now!**

**Due:** Sept 5, 2001, 11:59PM

Your solution set should have tests and more comments.

**1.** `pairup`

**Solution 1.** [5 points] Of course, several solutions are possible. Here's the most straightforward:

```
(defun pairup (keys values)
  "Pair successive elements of two proper lists of the same length. If one
is shorter, the pairs are terminated at that point"
  (cond ((or (endp keys) (endp values)) nil)
        (t (cons (cons (first keys) (first values))
                 (pairup (rest keys) (rest values))))))
```

Then again, there are shorter ways. Here are two:

```
(defun pairup (keys values) (mapcar #'cons keys values))

(defun pairup (keys values)  (pairlis keys values))
```

**2.** Write a (recursive) version of the LISP function `assoc`. Call it `myassoc`.

**Solution 2.** [5 points] Here is the straightforward solution:

```
(defun myassoc (key a-list)
  "Returns the first sub-list in a-list whose first element is eql to key."
  (cond ((endp a-list) nil)
        ((and (consp (first a-list))
              (eql key (car (first a-list))))
         (first a-list))
        (t (myassoc key (rest a-list)))))
```

Some people prefer `first` and `rest` when referring to elements of a list, but `car` `cdr` when referring to elements of a pair.

**3.** `change-binding`

**Solution 3.** [5 points]

```
(defun change-binding (key value a-list)
  "Alter the a-list so that the binding of key becomes value. No entry results in
an error: failure to find a binding is probably a bad thing."
  (let ((binding (myassoc key a-list)))
    (if binding (setf (cdr binding) value)
      (error "change-binding:  key ~s does not have a binding in a-list" key))))
```

In this problem, you must make sure and check that `myassoc` finds the `key` in the `a-list` before changing its binding. Giving a format list to `error` is a neat trick. You could write a more elaborate program that sets up a binding if it doesn't find one to change.

**4.** `hash-assoc` and `hash-change-binding`.

**Solution 4.** [10 points]

```
(defun hash-pairs (keys values)
  "Pair sucessive elements of two lists using a hash table."
  (let ((table (make-hash-table)))
    (mapc #'(lambda (key value) (setf (gethash key table) (cons key value)))
            keys values)
    table))

(defun hash-assoc (key table)
  "If key has a value in table, return (key . value)."
  (let ((value (gethash key table)))
    (if value  value
      (error "hash-assoc:  key does not have a binding in table"))))

(defun hash-change-binding (key value table)
  "Alter the table so that the binding of key becomes value."
  (setf (gethash key table) value))
```

`gethash` returns only the value found in the table, while `assoc` returns a `(key . value)` pair. If you write the program `hash-pairs` so that it stores only the value under the key, there is a problem if the value is `nil`. If `nil` is returned, was that the value, or did `assoc` fail to find any value? An elaborate version might use the multiple-value return of `gethash`, but this is unnecessary.

**5.** The `whois` program.

**Solution 5.** [15 points] The function `whois` will require several supporting facilities:

- a global variable to hold the database
- a function to add an entry to the database
- a function to print an entry from the database
- a function to match simple queries against entries of the database
- a function to look up an advisee in the database

Although the problem only asks for the implementation of some of these functions, the solution is easier to follow if all are implemented:

```
(defvar *advisees* nil
  "An entry for an advisee will look like (first last visit).")

(defun add-advisee (first last visit)
  "Add the advisee whose name is (first last) and who last visited on visit
to the list of advisees."
  (setq *advisees* (cons (list first last visit) *advisees*)))

(defun print-advisee (entry)
  "Print the record of one of Professor Coder's advisees."
  (format t "~a ~a last visited you on ~a.~%"
          (first entry) (second entry) (third entry)))

(defun match-query (query name)
  "Return t if the query is satisfied by the name."
  (and (or (eql (first query) '?) (eql (first query) (first name)))
       (or (eql (second query) '?) (eql (second query) (second name)))))

(defun lookup-advisees (query table)
  "Find all entries in the table that match the query."
  (remove query table :test-not #'match-query))

(defun whois (advisee)
  "Remind Professor Coder who his advisees are."
  (let ((entries (lookup-advisee advisee *advisees*)))
    (cond ((null entries)
           (format t "You have no advisee with name ~a on file.~%" advisee))
          ((rest entries)
           (format t "You have more than one advisee with name ~a: ~%")
           (mapc #'print-advisee entries)) ;mapcar here would waste storage.
          (t (print-advisee (first entries))))))
```

Note that if Professor Coder tries `(whois '(? ?))`, then he gets all of the entries in his database, which is just what he deserves, in my opinion. An alternative would be to give an error message.

Since `lookup-advisees` removes all entries that do not match the query, this version of `whois` will work correctly even if two students have the same first names and the same last names.

**6.** `showfile`

**Solution 6.** [5 points] Actually I combined this with the answer to the next question. I define `showfile2` which takes a second argument, the stream on which to print the result. (If you want the standard output, c should be `t`.)

```
(defun showfile2(filename c)
  (with-open-file
   (d filename :direction :input)
   (do* ((count 1 (1+ (mod count 66)))  ;do* is like let*. binds in sequence.
         (page 1 (cond ((= count 1)(format c "~c" #\page)
                        (incf page))
                       (t page)))
         (line  (read-line d nil)  (read-line d nil)))
        ((null line) (values))           ;test, return value
     (format c "~%~3d ~3d  ~a" page count line))))
```

**7.** Change `showfile` again so that another file can be used for output (other than standard output).

**Solution 7.** [5 points] Here is what I wrote..

```
(defun showfile(filename &optional (outfile nil))
  (if outfile
      (with-open-file
       (c outfile :direction :output :if-exists :overwrite)
       (showfile2 filename c) )
    (showfile2 filename t)              ;output to standard output
    ))
```