MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1989

Problem Set 4

Issued: October 3, 1989

Due: Friday, October 13, 1989 (all sections). Note: this is an exception to the usual rule of Wednesday due dates because of the Columbus day long weekend.

Reading:

- Text: Sections 2.2, 3.4 and 3.4.1.
  Note: Sections 3.4 and 3.4.1 are explained in terms of streams. For this assignment, you can treat "streams" as identical to "lists".
- A listing of the code for this problem set is attached. You may load it with the usual Edwin commands.

Quiz I Reminder

Logistics: Quiz I is on Thursday, October 19, in 50-340 (Walker Memorial Gymnasium), from 5-7 PM xor 7-9 PM. You may take the quiz during either of the two sessions, but students taking it in the first session must stay until 7 PM. The quiz is open book.

Coverage: The quiz covers everything up to and including Friday, October 6 (symbols and quotation).

Late-breaking story: Intelligence services report that there is likely to be a major leak at the lecture on Thursday, October 12. A clandestine handbill having "sympathetic vibrations" with the quiz is likely to be circulated. Don't miss it!

Part 1: Homework Exercises

Do the following exercises before you come to the lab. They provide practice with list structure and introduce ideas that you will need in order to work on lab programs.

Exercise 1: Finger exercises. Suppose we define `a` and `b` as follows:

```
(define a (cons 1 2))
(define b (list 3 4 5))
```

What is the response of the interpreter to each of the following:

(a) `(cons a b)`
(b) `(append a b)`
(c) `(append b a)`
(d) `(list a b)`

Exercise 2: Suppose we define `a b c` and `d` to have the values `4 3 2` and `1` respectively. What is the response of the interpreter to each of the following?

(a) `(list a 'b c 'd)`
(b) `(list (list a) (list b))`
(c) `(cons (list d) 'b)`
(d) `(cadr '((a b) (c d) (a d) (b c)))`
(e) `(cdr '(a))`
(f) `(atom? (caddr '(Welcome to MIT)))`
(g) `(memq 'sleep '(where getting enough sleep))`
(h) `(memq 'time '((requires good) (management of time)))`

Exercise 3: Do exercise 2.29 on page 103 of the text.

Exercise 4: The procedure `member` is similar to `memq` (see the text, page 102), except that it uses `equal?` rather than `eq?` to compare the item against each element of the list. What is the result of evaluating each of the following expressions?

(a) `(member '(great fun) '(spelunking is great fun))`
(b) `(member '(great fun) '((spelunking is) (great fun) (so they say)))`
(c) `(member '(great fun) '((spelunking is) ((great fun) (so they say))))`

Exercise 5: The procedure `flatten2` takes a list of lists (also called a 2-level list) and flattens out the top-level. For example,

`(flatten2 '((1 2) (3) (4 (5) 6)))` ⇒ `(1 2 3 4 (5) 6)`

Give a definition for `flatten2`.

Exercise 6: The function `accumulate` is a higher-order function that takes a binary combining operator `f`, an "initial" element `z` and a list `(`x_1 `...` x_n`)`, and produces a result equivalent to:

`(f` x_1 `(f` x_2 `(f ... (f` x_n `z))))`

This is how it does it:

```
(define accumulate
  (lambda (f z l)
    (if (null? l)
        z
        (f (car l) (accumulate f z (cdr l))))))
```

Now, consider:

```
(define flatten2
  (lambda (lst)
     (accumulate f? z? l?)))
```

What should the arguments *f*?, *z*? and *l*? be?

Part 2: Background for programming assignment: The All-Nighter Sweepstakes

(Just read this part in preparation for Part 3; there is nothing to be handed in.)

It is barely three weeks into the season and already it is becoming extremely difficult to keep track of the relative standings of the contestants in the Fall 1989 PANA[1] Tour. We decide that before things get completely out of control, we're going to build a database system to organize all the results. A popular way to structure databases these days is to use the *Relational Model*, where a database is represented as a collection of relations, or *tables*.

The Tour consists of a series of events. For each event, the contestant who spent the most consecutive hours in last-minute work is declared Zombie.[2] We also record the runners-up and the number of consecutive last-minute hours they spent on it. All this is shown in the `EVENTS` table:

EVENT	ZOMBIE	HOURS	RUNNER-UP	RU-HOURS
6.001-PS2-Pro-Am	Ben	9	Alyssa	6
6.004-Lab-6-Invitational	Lem-E	10	Alyssa	9
French-IV-Open	Cy-D	8	Alonzo	6
18.06-HW3-All-Stars	Alonzo	11	Eva-Lu	8
6.035-Masters	Louis	19	Cy-D	15
Tour-de-6.170	Alyssa	25	Lem-E	22
German-III-Weltmeisterschaft	Eva-Lu	12	Lem-E	10
6.003-Coupe-le-Monde	Alyssa	14	Ben	11
End-of-Term-Grand-Slam	Alyssa	30	Louis	25

Here is the `ATHLETES` table:

NAME	SPONSOR	LAST-YEAR-PANA-RANKING
Ben	Mocha-Java-of-Mexico	2
Alyssa	Juan-Valdez-of-Colombia	3
Lem-E	Santos-of-Guatemala	7
Louis	Yrgacheffe-of-Ethiopia	6
Alonzo	Celibe-de-Haiti	4

[1] Professional All-Nighters Association.

[2] We hasten to add that this is a dubious distinction. Zombie-hood is not to be confused with quality performance in an event.

```
Eva-Lu      11-7                        1
Cy-D        Juan-Valdez-of-Colombia     5
```

To query the database, we use operations of the *Relational Algebra*, each of which is a function from tables to tables. There are three main operators, `project`, `select` and `join`.

The `project` operator takes "vertical slices" of a table.

Example Query Q1: "List all events, their zombies and runners-up". Formally, we'd say:

```
(project '(event zombie runner-up)
         EVENTS)
```

which produces the table:

```
EVENT                           ZOMBIE    RUNNER-UP
-----                           ------    ---------
6.001-PS4-Pro-Am                Ben       Alyssa
6.002-Lab-6-Invitational        Lem-E     Alyssa
French-IV-Open                  Cy-D      Alonzo
18.06-HW3-All-Stars             Alonzo    Eva-Lu
6.035-Masters                   Louis     Cy-D
Tour-de-6.170                   Alyssa    Lem-E
German-III-Weltmeisterschaft    Eva-Lu    Lem-E
6.003-Coupe-le-Monde            Alyssa    Ben
End-of-term-Grand-Slam          Alyssa    Louis
```

In general, the `project` operator takes a list of column names and a table T, and produces a new table with just those columns from T.

The `select` operator takes "horizontal slices" of a table.

Example Q2: "List all athletes sponsored by Juan Valdez". Formally, we'd say:

```
(select '(eq? sponsor 'Juan-Valdez-of-Colombia)
        ATHLETES)
```

which produces the table:

```
NAME        SPONSOR                     LAST-YEAR-PANA-RANKING
----        -------                     ----------------------
Alyssa      Juan-Valdez-of-Colombia     3
Cy-D        Juan-Valdez-of-Colombia     5
```

In general, the `select` operator takes a predicate and a table T, and produces a new table with the same columns, but with only those rows of T that satisfy the predicate.

Of course, since the result of a relational operator is itself a table, we can compose relational expressions.

Example Q3: "List events and zombies who won by 3 hours exactly".

```
(project '(event zombie)
    (select '(= hours (+ ru-hours 3))
        EVENTS))
```

producing the table:

```
EVENT                          ZOMBIE
-----                          ------
6.001-ps2-Pro-Am               Ben
18.06-HW3-All-Stars            Alonzo
Tour-de-6.170                  Alyssa
6.003-Coupe-le-Monde           Alyssa
```

Both `select` and `project` operate only on a single table. The `join` operator, on the other hand, takes two tables T1 and T2 and produces a new table each of whose rows is a concatenation of a row from T1 and a row from T2. We illustrate it with a very small example. Let T1 be

```
PERSON  WATCHES
------  -------
Louis   Gong-Show
Lem-E   This-Old-House
```

and let T2 be:

```
SHOW            CHANNEL
----            -------
Gong-Show       4
This-Old-House  2
```

Then, `(join T1 T2)` will produce this table:

```
PERSON  WATCHES            SHOW            CHANNEL
------  -------            ----            -------
Louis   Gong-Show          Gong-Show       4
Louis   Gong-Show          This-Old-House  2
Lem-E   This-Old-House     Gong-Show       4
Lem-E   This-Old-House     This-Old-House  2
```

Now we can see why we didn't attempt to show the result of joining the `EVENTS` and `ATHLETES` tables— the total number of rows would be quite large (Question to ponder: how many rows would there be?).

Using the three relational operators `project`, `select` and `join`, we can express quite complex and interesting queries on the database.

Example Q4: "List the events where the zombies were sponsored by Juan Valdez":

```
(project '(event)
         (select '(eq? sponsor 'Juan-Valdez-of-Colombia)   ;;; (1)
                 (select '(eq? zombie name)                ;;; (2)
                         (join EVENTS ATHLETES))))
```

The `join` operator pairs every event with every athlete. The `select` operator (2) retains only those rows where the event zombie is the same as the concatenated athlete. The `select` operator (1) retains only those rows where the sponsor is Juan Valdez. Finally, the `project` operator gets rid of all the extra columns, keeping only the event names. Thus, the result is the (1-column) table:

```
EVENT
-----
French-IV-Open
Tour-de-6.170
6.003-Coupe-le-mond
```

```
End-of-term-Grand-Slam
```

Most algebras have *laws* expressing equivalences between expressions. For example, in ordinary algebra, we are familar with the following law (called *distributivity*):

$a \cdot b + a \cdot c = a \cdot (b + c)$

Similarly, relational algebra also has laws. One such law is:

`(select` p_1 `(select` p_2 R`))` = `(select (and` p_1 p_2`)` R`)`

Using this law, we can re-express Q4 as follows:

Example Q5:

```
(project '(event)
         (select '(and (eq? sponsor 'Juan-Valdez-of-Colombia)
                       (eq? zombie name))
                 (join EVENTS ATHLETES)))
```

Another law is:

`(select` p `(join` R_1 R_2`))` = `(join` R_1 `(select` p R_2`))`

whenever p refers only to columns in $R2$. Using this law, we can again re-express Q4 as follows:

Example Q6:

```
(project '(event)
         (select '(eq? zombie name)
                 (join EVENTS
                       (select '(eq? sponsor 'Juan-Valdez-of-Colombia)
                                ATHLETES))))
```

This is a significant optimization, because the number of rows going into the `join` operator has been drastically reduced.

Example Q7: "Which zombies with PANA ranking less than 3 spent more than 10 hours non-stop, and for which events?":

```
(project '(zombie event)
         (select '(> hours 10)
                 (select '(eq? zombie name)
                         (select '(< last-year-pana-ranking 3)
                                 (join EVENTS
                                       ATHLETES)))))
```

Part 3: To do in lab

We are now going to develop implementations of the relational operators presented in Part 2. Load the problem set into your editor buffer, and peruse it. It contains:

- Some definitions to be evaluated immediately, including the `EVENTS` and `ATHLETES` tables;

- Some incomplete definitions corresponding to the problems below, and
- The example queries Q1 through Q7 for testing.

The following problems involve filling in the missing parts of the definitions, and testing your code. As always, you should include printed transcripts of your Scheme interactions which test the code.

Problem 1 The constructor for tables is defined as:

```
(define (make-table col-names rows)
  (cons col-names rows))
```

where `col-names` is a list of symbols representing column names, and `rows` is a list of rows, where each row is itself a list of data. For example, this is how we construct the `EVENTS` table:

```
(define EVENTS
  (make-table
    '(EVENT                          ZOMBIE     HOURS  RUNNER-UP  RU-HOURS)
    '((6.001-PS2-Pro-Am               Ben        9      Alyssa     6)
      ...
      (End-of-Term-Grand-Slam         Alyssa     30     Louis      25))))
```

Define the corresponding selectors `col-names-of` and `rows-of`. Apply it to `ATHLETES` to see that it works.

Problem 2 The following function:

```
(define lookup
  (lambda (col col-names row)
    ...))
```

takes a symbol (`col`), a list of column-names (`col-names`) and a list of corresponding data (`row`), and returns the datum from row corresponding to `col`. For example:

```
(lookup 'zombie
        '(event             zombie  hours  runner-up  ru-hours)
        '(6.001-PS4-Pro-Am  Ben     26     Alyssa     22      ))
```

will return the symbol `Ben`. The pair of lists `col-names` and `rows` is also called an *environment*, i.e., an association of names and values.

Complete the definition of `lookup`, and run it on a few examples to show that it works.

Problem 3 The function `map` (also called `mapcar`), which applies a procedure to every member of a list and returns all the results in a new list, was introduced in class:

```
(define map
  (lambda (proc lst)
    (if (null? lst)                   ; if lst is empty
        '()                           ; then return an empty list
        (cons (proc (car lst))        ; otherwise apply proc to first element
              (map proc (cdr lst))))) ; and attach it to front of list
                                      ; obtained by processing rest of list
```

Use it, along with the `lookup` function, to complete the definition of:

```
(define project-row
  (lambda (cols col-names row)
    ... ))
```

Here, `cols` is a list of symbols, and `col-names` and `row` form an environment, as in Problem 2. It returns a list of data corresponding to the columns named by `cols`. For example:

```
(project-row '(event zombie)
             '(event             zombie  hours  runner-up  ru-hours)
             '(6.001-PS4-Pro-Am  Ben     26     Alyssa     22      ))
```

should produce the list:

```
(6.001-PS4-Pro-Am Ben)
```

Problem 4 Use `map` and `project-row` to complete the definition of:

```
(define project
  (lambda (cols table)
    ... ))
```

Run query Q1, and another query of your own invention to demonstrate that it works correctly. Be sure also to explain your query in English, as we did in the examples.

Problem 5 In order to implement the `select` operator, we need to be able to evaluate a predicate on each row of a table. Let us start with a function that evaluates a predicate on a single row, i.e., we will use it as follows:

```
(evaluate '(< hours 10)
          col-names
          row)
```

i.e., we will evaluate the predicate `(< hours 10)` in the environment specified by `col-names` and `row`.

A predicate is an expression that is either atomic or not. If not atomic, it is the application of an operator to one or more other expressions. If it is atomic, then it is either a symbol (in which case it represents a datum in `row`) or it is a number (in which case it represents itself). Here are some examples of expressions:

```
23
(+ ru-hours 10)
(< hours (+ ru-hours 10))
(quote Alyssa)
(eq? zombie (quote Alyssa))
```

Here are some useful abstractions to extract the operator and arguments of an expression:

```
(define op-of   (lambda (e) (car e)))
(define arg1-of (lambda (e) (cadr e)))
(define arg2-of (lambda (e) (caddr e)))
```

We assume that no operator takes more than two arguments, i.e., for our relational language, unlike Scheme, operators like `+`, `and`, etc., take exactly two arguments.

Here is the skeleton of the `evaluate` function:

```
(define evaluate
  (lambda (expr col-names row)
    (cond
      ((symbol? expr)      (lookup expr col-names row))
      ((number? expr)      expr)
      ((eq? (op-of expr) '=) (= (evaluate (arg1-of expr) col-names row )
                                (evaluate (arg2-of expr) col-names row)))
      ((eq? (op-of expr) '<) (< (evaluate (arg1-of expr) col-names row )
                                (evaluate (arg2-of expr) col-names row)))

      ...  and so on for other operators  ...

      (else (error "EVALUATE: expression not well-formed" expr))
    )))
```

Fill in the clauses of the conditional for the operators `quote`, `>`, `eq?`, `+`, `and`, `or`, and `not`. Feel free to include more operators if you wish.

Test your function `evaluate` using the column names and first row of the `events` table, and several possible predicate expressions.

Problem 6 The function `filter`, which applies a predicate to every element of a list, returning a new list containing only those elements that satisfy the predicate, was introduced in class:

```
(define filter
  (lambda (pred lst)
    (cond
      ((null? lst)      '())                  ; if lst empty, return empty list
      ((pred (car lst))                       ; if car satisfies pred,
           (cons (car lst)                    ;    include it
                 (filter pred (cdr lst))))    ;            with  remainder
      (else                                   ; otherwise
           (filter pred (cdr lst))))))        ;    discard car, do remainder
```

Using `filter` and `evaluate`, complete the following definition:

```
(define select
  (lambda (pred table)
    ...))
```

Run queries Q2 and Q3 and two more queries of your own invention to demonstrate that it works correctly. Be sure also to explain your query in English, as we did in the examples.

Problem 7 Here is a function that computes the cross-product of two lists:

```
(define cross-product
    (lambda (x-list y-list)
        (flatten2 (map (lambda (x)
                         (map (lambda (y) (list x y))
                              y-list))
                       x-list))))
```

(a) What is the result of the following application?

```
(cross-product '(1 2) '(A B C))
```

(b) If the input lists for `cross-product` have m and n elements in them, respectively, how long is the output list?

(c) Using `map`, `flatten2` and `cross-product`, complete the definition for the `join` relational operator:

```
(define join
  (lambda (table-1 table-2)
    (let  ((N1 (col-names-of table-1))
           (N2 (col-names-of table-2))
           (R1 (rows-of table-1))
           (R2 (rows-of table-2)))
      (make-table
        ...
        ... ))))
```

(d) Run queries Q4, Q5, Q6, Q7 and one more query of your own invention to demonstrate that it works. Be sure also to explain your query in English, as we did in the examples.

Problem 8 Q6 was a significant optimization of Q4. Perform the same optimization on Q7 and run it again.

Problem 9 Since a `join` produces all pairings of the rows of its input tables, most rows in the output are meaningless. Thus, every time we do a `join`, we immediately use `select` to keep only those rows where some pair of fields are equal. Let us define a new relational operator `equi-join` that makes this more convenient. For example,

```
(equi-join 'zombie 'name EVENTS ATHLETES)
```

should produce the same result as:

```
(select '(eq? zombie name)
    (join EVENTS ATHLETES))
```

Give a definition for `equi-join` (Note: there are many ways of doing this, with varying levels of efficiency. Any solution will do, but you are welcome to try to make it efficient.) Re-express Q4 using `equi-join`, and run it again.

Listing of file to be loaded

```
;;; Code for 6.001 Fall 1989, PS 4
;;; Evaluate immediately everything up to the line of asterisks.
;;; Below that line are
;;;   - incomplete fragments to be filled out as part of the PS, and
;;;   - example queries for testing.

(define make-table
    (lambda (col-names rows)
      (cons col-names rows)))

(define EVENTS
  (make-table
    '(EVENT                          ZOMBIE     HOURS  RUNNER-UP  RU-HOURS)
    '((6.001-PS2-Pro-Am               Ben        9      Alyssa     6)
      (6.004-Lab-6-Invitational       Lem-E      10     Alyssa     9)
      (French-IV-Open                 Cy-D       8      Alonzo     6)
      (18.06-HW3-All-Stars            Alonzo     11     Eva-Lu     8)
      (6.035-Masters                  Louis      19     Cy-D       15)
      (Tour-de-6.170                  Alyssa     25     Lem-E      22)
      (German-III-Weltmeisterschaft   Eva-Lu     12     Lem-E      10)
      (6.003-Coupe-le-Monde           Alyssa     14     Ben        11)
      (End-of-Term-Grand-Slam         Alyssa     30     Louis      25))))

(define ATHLETES
  (make-table
    '(NAME       SPONSOR                   LAST-YEAR-PANA-RANKING)
    '((Ben        Mocha-Java-of-Mexico      2)
      (Alyssa     Juan-Valdez-of-Colombia   3)
      (Lem-E      Santos-of-Guatemala       7)
      (Louis      Yrgacheffe-of-Ethiopia    6)
      (Alonzo     Celibe-de-Haiti           4)
      (Eva-Lu     11-7                      1)
      (Cy-D       Juan-Valdez-of-Colombia   5))))

(define map
  (lambda (proc lst)
    (if (null? lst)                  ; if lst is empty
        '()                          ; then return an empty list
        (cons (proc (car lst))       ; otherwise apply proc to first element
              (map proc (cdr lst)))))) ; and attach it to front of list
                                     ; obtained by processing rest of list

(define filter
  (lambda (pred lst)
    (cond
      ((null? lst)      '())              ; if lst empty, return empty list
      ((pred (car lst))                   ; if car satisfies pred,
           (cons (car lst)                ;    include it
                 (filter pred (cdr lst)))) ;            with  remainder
      (else                               ; otherwise
           (filter pred (cdr lst))))))    ;    discard car, do remainder

(define cross-product
    (lambda (x-list y-list)
        (flatten2 (map (lambda (x)
                         (map (lambda (y) (list x y))
                              y-list))
```

```
                        x-list))))

(define op-of   (lambda (e) (car e)))
(define arg1-of (lambda (e) (cadr e)))
(define arg2-of (lambda (e) (caddr e)))

;;; ************************************************************
;;; Incomplete fragments to be filled out as part of the PS.

(define lookup
  (lambda (col col-names row)
    'TO-BE-COMPLETED    ))

(define project-row
  (lambda (cols col-names row)
    'TO-BE-COMPLETED    ))

(define project
  (lambda (cols table)
    'TO-BE-COMPLETED    ))

(define evaluate
  (lambda (expr col-names row)
    (cond
      ((symbol? expr)      (lookup expr col-names row))
      ((number? expr)      expr)
      ((eq? (op-of expr) '=) (= (evaluate (arg1-of expr) col-names row )
                                (evaluate (arg2-of expr) col-names row)))
      ((eq? (op-of expr) '<) (< (evaluate (arg1-of expr) col-names row )
                                (evaluate (arg2-of expr) col-names row)))

;;;   ...  and so on for other operators ...
;;;        TO BE COMPLETED

      (else (error "EVALUATE: expression not well-formed" expr))
    )))

(define select
  (lambda (pred table)
    'TO-BE-COMPLETED   ))

(define join
  (lambda (table-1 table-2)
    (let  ((N1 (col-names-of table-1))
           (N2 (col-names-of table-2))
           (R1 (rows-of table-1))
           (R2 (rows-of table-2)))
      (make-table
        'TO-BE-COMPLETED
        'TO-BE-COMPLETED    ))))

;;; ------------------------------------------------------------
;;; Example queries for testing.
;;;
(define Q1 (lambda ()
    (project '(event zombie runner-up)
             EVENTS)))

(define Q2 (lambda ()
```

```
    (select '(eq? sponsor 'Juan-Valdez-of-Colombia)
            ATHLETES)))

(define Q3 (lambda ()
    (project '(event zombie)
        (select '(= hours (+ ru-hours 3))
            EVENTS))))

(define Q4 (lambda ()
    (project '(event)
            (select '(eq? sponsor 'Juan-Valdez-of-Colombia)   ;;; (1)
                    (select '(eq? zombie name)                ;;; (2)
                            (join EVENTS ATHLETES))))))

(define Q5 (lambda ()
    (project '(event)
            (select '(and (eq? sponsor 'Juan-Valdez-of-Colombia)
                          (eq? zombie name))
                    (join EVENTS ATHLETES)))))

(define Q6 (lambda ()
    (project '(event)
            (select '(eq? zombie name)
                    (join EVENTS
                          (select '(eq? sponsor 'Juan-Valdez-of-Colombia)
                                  ATHLETES))))))

(define Q7 (lambda ()
    (project '(zombie event)
            (select '(> hours 10)
                    (select '(eq? zombie name)
                            (select '(< last-year-pana-ranking 3)
                                    (join EVENTS
                                          ATHLETES)))))))

;;; ---------------------------------------------------------------------
```