MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996-97

Problem Set 5

Issued: Tuesday, October 1, 1996

Due: Friday, October 11, in recitation.

Tutorial preparation for: Week of October 7.

Reading Assignment: Sections 2.2, 2.3

1. Tutorial exercise

Please prepare Exercises 2.28, 2.37, 2.53 and 2.54 from the text.

2. Background: The Coffee Drinkers Delight

You just need to read this section to prepare for Part 3; there is nothing specific to be handed in.

Even though it is just a few weeks into the season, it's getting hard to keep track of the standings of the contestants in the Fall 1996 WSCDI[1] Tour. Before things get further out of control, we are going to build a database system to organize the results of this competition. A popular way to structure databases 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 surfing the Web and imbibing copious amounts of caffeine without the need to head to the "head" is declared the Cafemeister. We also record the runners-up and the number of consecutive hours they spent on in the event. All this is shown in the **EVENTS** table:

[1]Web Surfers/Coffee Drinkers International

EVENT	CAFEMEISTER	HOURS	RUNNER-UP	RU-HOURS
Netscape-Pro-Am	Ben	9	Alyssa	6
Microsoft-Explorer-Invitational	Lem-E	10	Ben	9
Capuccino-Cup	Eva-Lu	7	Cy-D	6
Juan-Valdez-Challenge	Louis	15	Eva-Lu	9
Seattle-Sleepless-Masters	Louis	19	Cy-D	15
Tour-de-Caffeine	Alyssa	28	Lem-E	24
Espresso-Express	Eva-Lu	15	Ben	10
Coupe-Mondial	Alyssa	17	Eric	12
End-of-Term-Grand-Slam	Alyssa	30	Louis	28

Here is the **ATHLETES** table:

NAME	SPONSOR	LAST-YEAR-WSCD-RANKING
Ben	Starbucks	3
Alyssa	7-11	4
Cy-D	Coffee-Connection	6
Lem-E	Green-Mountain	2
Louis	LaVerdes	5
Eva-Lu	Coffee-Connection	7
Eric	Starbucks	1

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 operations: `v-slice`, `h-slice` and `join`.

The `v-slice` operator takes "vertical slices" of a table.

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

```
(v-slice '(event cafemeister runner-up)
         EVENTS)
```

which produces the table:

EVENT	CAFEMEISTER	RUNNER-UP
Netscape-Pro-Am	Ben	Alyssa
Microsoft-Explorer-Invitational	Lem-E	Ben
Capuccino-Cup	Eva-Lu	Cy-D
Juan-Valdez-Challenge	Louis	Eva-Lu
Seattle-Sleepless-Masters	Louis	Cy-D
Tour-de-Caffeine	Alyssa	Lem-E
Espresso-Express	Eva-Lu	Ben
Coupe-Mondial	Alyssa	Eric
End-of-Term-Grand-Slam	Alyssa	Louis

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

The `h-slice` operator takes "horizontal slices" of a table.

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

```
(h-slice '(eq? sponsor 'Starbucks)
         ATHLETES)
```

which produces the table:

NAME	SPONSOR	LAST-YEAR-WSCD-RANKING
Ben	Starbucks	3
Eric	Starbucks	1

In general, the `h-slice` 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. Note that in specifying a predicate, we will distinguish between symbols (by using quotation) and variables, whose values we need. For these variables, we will using an "environment" to find the associated value, that is we will use the value in a row corresponding to the slot in the list of column names specified by the variable.

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

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

```
(v-slice '(event cafemeister)
  (h-slice '(= hours (+ ru-hours 3))
           EVENTS))
```

producing the table:

EVENT	CAFEMEISTER
Netscape-Pro-Am	Ben

Both `h-slice` and `v-slice` 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	ER
Lem-E	This-Old-House

and let T2 be:

SHOW	CHANNEL
ER	7
This-Old-House	2

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

PERSON	WATCHES	SHOW	CHANNEL
Louis	ER	ER	7
Louis	ER	This-Old-House	2
Lem-E	This-Old-House	ER	7
Lem-E	This-Old-House	This-Old-House	2

Now you 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 (How many rows do you think there would be??).

Using the three relational operators: `v-slice`, `h-slice` and `join`, we can express quite complex and interesting queries on the database.

Example Query Q4: "List the events where the cafemeisters were sponsored by Starbucks":

```
(v-slice '(event)
         (h-slice '(eq? sponsor 'Starbucks)         ;;; (1)
                  (h-slice '(eq? cafemeister name)   ;;; (2)
                           (join EVENTS ATHLETES))))
```

The `join` operator pairs every event with every athlete. The `h-slice` operator (2) retains only those rows where the event cafemeister is the same as the concatenated athlete. The `h-slice` operator (1) retains only those rows whre the sponsor is Starbucks. Finaly, the `v-slice` operator gets rid of all the extra columns, keeping only the event names. Thus, the results is a (1-column) table:

EVENT
Netscape-Pro-Am

Most algebras have *laws* exprssing equivalences between expressions. For example, in ordinary algebra, we are familiar with the distributive law:

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

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

$$\texttt{(h-slice } p_1 \texttt{ (h-slice } p_2\ R)) = \texttt{(h-slice (and } p_1\ p_2)\ R)$$

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

Example Query Q5:

```
(v-slice '(event)
         (h-slice '(and (eq? sponsor 'Starbucks)
                        (eq? cafemeister name))
                  (join EVENTS ATHLETES)))
```

Another law is

$$\texttt{(h-slice } p \texttt{ (join } R_1\ R_2)) = \texttt{(join } R_1 \texttt{ (h-slice } p\ R_2))$$

whenever p refers only to columns in R_2. Using this law, we can again re-express Q4 as follows:

Example Query Q6:

```
(v-slice '(event)
         (h-slice '(eq? cafemeister name)
                  (join EVENTS
                        (h-slice '(eq? sponsor 'Starbucks)
                                 ATHLETES))))
```

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

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

```
(v-slice '(cafemeister event)
         (h-slice '(> hours 8)
                  (h-slice '(eq? cafemeister name)
                           (h-slice '(< last-year-wscd-ranking 3)
                                    (join EVENTS
                                          ATHLETES)))))
```

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 look it over. 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
  (lambda (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                         CAFEMEISTER  HOURS  RUNNER-UP   RU-HOURS)
   '((Netscape-Pro-Am                   Ben        9     Alyssa      6)
     ...
     (End-of-Term-Grand-Slam            Alyssa     30    Louis       28))))
```

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`. The idea behind `lookup` is to find values associated with variables in an environment, that is value in the `row` in the same spot as the symbol `col` in the list `col-names`.

For example:

```
(lookup 'cafemeister
        '(EVENT            CAFEMEISTER  HOURS  RUNNER-UP   RU-HOURS)
        '(Netscape-Pro-Am       Ben       9      Alyssa      6))
```

will return the symbol `Ben`. The pair of lists `col-names` and `row` 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`, 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)
        '()
        (cons (proc (car lst))
              (map proc (cdr lst))))))
```

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

```
(define v-slice-row
  (lambda (cols col-names row)
    ...  ))
```

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

```
(v-slice-row '(event cafemeister)
             '(EVENT            CAFEMEISTER  HOURS  RUNNER-UP   RU-HOURS)
             '(Netscape-Pro-Am       Ben       9      Alyssa      6))
```

should return the list:

```
(Netscape-Pro-Am Ben)
```

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

```
(define v-slice
  (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 `h-slice` operator, we need to be able to evaluate a predicate on each row of a table. Let's 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? cafemeister (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 h-slice
  (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 queries 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 `h-slice` to keep only those rows where some pair of fields are equal. Let's define a new relational operator `equi-join` that makes this more convenient. For example

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

should produce the same results as:

```
(h-slice '(eq? cafemeister 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.