MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 1997

**Problem Set 4**

Issued: Tuesday, October 7, 1997

Due: Wednesday, October 15, in recitation.

Tutorial preparation for: Week of October 13.

Reading for this Problem Set: *SICP* Sections 2–2.2.3, 2.3
Reading for Lectures: *SICP* Sections 2.4–5

**The Coffee Drinkers Delight**

# 1 Tutorial exercises:

*SICP* Exercises 2.20, 2.29, 2.32, 2.37, 2.57, and 2.64.

# 2 Background

Read this section of the Problem Set to prepare to do the exercises in Part 3. There is nothing to hand in for this section.

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 1997 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 Table's.

[1] Web Surfers/Coffee Drinkers International

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 interruption 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:

| EVENT | CAFEMEISTER | HOURS | RUNNERUP | 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-YR-WSCDI-RANK |
|---|---|---|
| Ben | Starbucks | 3 |
| Alyssa | seven-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 returns a table or query results. 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 evaluate

```
                  ;v-slice: (List(col-names) x Table) --> Table
(v-slice '(event cafemeister runnerup)
         events)
```

which produces the table:

| EVENT | CAFEMEISTER | RUNNERUP |
|---|---|---|
| Netscape-Pro-Am | Ben | Alyssa |
| Micrsft-Explrr-Invitatnl | 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 the named 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 evaluate

```
                    ;h-slice: (Expr x Table) --> Table
(h-slice '(eq? sponsor 'Starbucks)
         athletes)
```

which returns the table:

| NAME | SPONSOR | LAST-YR-WSCDI-RANK |
|---|---|---|
| Ben | Starbucks | 3 |
| Eric | Starbucks | 1 |

In general, the `h-slice` operator takes an expression $E$ and a table $T$. It treats each row of $T$ as an *environment* for evaluating $E$, with the column names regarded as variables and the row entry in a column taken as the “value” of the column name. For example, with the first row of the **ATHLETES** table as an environment, the variable NAME has the symbol `ben` as its value, the variable SPONSOR has the symbol `starbucks` as its value, and the variable LAST-YR-WSCDI-RANK has the number 3 as its value.

The result of (`h-slice` $E$ $T$) is a new table consisting of exactly those rows of $T$ in which the expression has the value `#t`.

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?).

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? cafemeister name)         ;;; (1)
                  (h-slice '(eq? sponsor 'starbucks)   ;;; (2)
                           (join events athletes))))
```

The `join` operator pairs every event with every athlete. The `h-slice` operator (2) retains only those rows where the sponsor is Starbucks. The `h-slice` operator (1) retains only those rows where the event cafemeister is the same as the name of the athlete. Finally, 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* expressing 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:

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

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

**Example Query Q5:**

```
(v-slice '(event)
         (h-slice '(and (eq? cafemeister name)
                        (eq? sponsor 'starbucks))
                  (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 8 hours non-stop, and for which events?":

```
(v-slice '(cafemeister event)
         (h-slice '(> hours 8)
                  (h-slice '(eq? cafemeister name)
                           (h-slice '(< last-yr-wscdi-rank 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.

**Lab Problem 1** We think of Tables as consisting of a list of column names and a list of rows, where the length of each row is the same as the length of the list of column names. No column name may appear more than once in the list of column names.

We could define a constructor

$$\texttt{make-table} : (\text{List(Col-name)} \times \text{List(List(Val))}) \to \text{Table}$$

and corresponding selectors with

```
(define make-table cons)
(define col-names-of car)
(define row-of cdr)
```

For example, this is how we construct the **EVENTS** table:

```
(define events
  (make-table
   '(event                    cafemeister  hours  runnerup   ru-hours)
   '((netscape-pro-am              ben       9      alyssa      6)
     ...
     (end-of-term-grand-slam       alyssa   30      louis       28))))
```

**1A.** Defining `make-table` simply to be `cons` is ok as long as we trust that it will only be applied to arguments which will result in well-formed tables. Absent such trust, it would be better software engineering to have `make-table` check its arguments and give an indication of error when the arguments don't fit together properly. Redefine `make-table` so it does this check. Try it on some "good" and "bad" sets of arguments to show how it works.

**1B.** Define an operator `rename-column` which takes a table and two column names $N_1$ and $N_2$. It returns a new table identical to the old one, except the old column named $N_1$ is now named $N_2$. For example

```
(define meister-athletes
 (rename-column athletes 'name 'cafemeister))
```

creates a new **MEISTER-ATHLETES** table:

| CAFEMEISTER | SPONSOR | LAST-YR-WSCDI-RANK |
|---|---|---|
| Ben | Starbucks | 3 |
| Alyssa | seven-11 | 4 |
| Cy-D | Coffee-Connection | 6 |
| Lem-E | Green-Mountain | 2 |
| Louis | LaVerdes | 5 |
| Eva-Lu | Coffee-Connection | 7 |
| Eric | Starbucks | 1 |

Make sure your `rename-column` gives an appropriate error indication if the renaming would not yield a well-formed table.

**Problem 2** We will implement environments as a list of two equal-length lists: a list of variables (without repeats) and a list of values.

```
                mk-env: (List(Var) x List(Val)) --> Env
(define mk-env list)
(define vars-of car)
(define vals-of cadr)
```

The selector for environments is the procedure

$$\texttt{lookup} : \text{Var} \times \text{Env} \to \text{Val}$$

which returns the value associated with a given variable in a given environment. For example:

```
(lookup 'cafemeister
        (mk-env
         '(event          cafemeister  hours  runnerup   ru-hours)
         '(netscape-pro-am    ben      9      alyssa     6)))
```

will return the symbol `ben`.

Complete the definition:

```
(define (lookup var env)
    ...))
```

Is your definition tail-recursive?

**Problem 3** The procedure `v-slice-row` takes a list of column names, a longer list of names including those in the first list, and a row, and returns the sub-row with only those entries named in the first list. Here is an example:

```
(v-slice-row
 '(event cafemeister)
 '(event              cafemeister  hours  runnerup   ru-hours)
 '(netscape-pro-am        ben      9      alyssa      6))
```

returns the list:

```
(netscape-pro-am ben)
```

Use `lookup` and `map` to complete a simple definition of:

```
(define (v-slice-row desired-col-names all-col-names row)
    ...  )
```

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

```
(define (v-slice 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 expressions in environments made from rows of a table. Let's start with the procedure `evaluate` that evaluates an expression in a single environment.

```
(evaluate '(< hours 10)
          (mk-env '(hours ru-hours) '(7 6)))
;Value: #t
```

Here are some examples of expressions:

```
name
23
(+ ru-hours 10)
(< hours (+ ru-hours 10))
'Alyssa
(eq? cafemeister 'Alyssa)
(and (eq? cafemeister 'Alyssa)
     (eq? name 'Alyssa))
```

In this setting, expressions are either variables (such as `name`), values such as numbers or Booleans, or (for simplicity) the application of an operator to exactly two other expressions. Symbols (such as `'Alyssa`) may also be values, they have to be explicitly quoted and then handled as a special case in order not to confuse them with variables. Here are some useful abstractions for expressions.

```
(define variable? symbol?)

(define (atomic-value? e)
  (or (number? e)
      (boolean? e)
      (string? e)))

(define (symbol-value? e)
  (and (pair? e)
       (eq? (car e) 'quote)
       (and (pair? (cdr e))
            (symbol? (cadr e))
            (null? (cddr e)))))

(define (quoted-val e)        ;e must be a symbol-value
        (cadr e))

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

Using these abstractions, we define `evaluate`:

```
(define (evaluate expr env)
    (cond
     ((variable? expr)
      (lookup expr env))
     ((atomic-value? expr) expr)
     ((symbol-value? expr) (quoted-val expr))
                       ;expr is of the form (op exp1 exp2):
     (else
      (let ((primitive-op (lookup (op-of expr) op-table))
            (val1 (evaluate (arg1-of expr) env))
            (val2 (evaluate (arg2-of expr) env)))
        (primitive-op val1 val2)))))
```

The only thing missing above is the definition of `op-table`. Supply it, and then test `evaluate` with several possible expressions using a couple of environments made from the the column names and rows of the **EVENTS** table.

**Problem 6** Using `filter` and `evaluate`, complete the following definition:

```
(define (h-slice expr 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** We will need a procedure that returns a list of "products" of every element of one list with every element of another list.

```
(define (tensor x-list y-list op)
    (map (lambda (x)
           (map (lambda (y) (op x y))
                y-list))
         x-list))

(define (all-pairs x-list y-list op)
  (apply append (tensor x-list y-list op)))
```

**Pre-Lab Problem**

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

```
(all-pairs '(1 2) '(a b c) list)
```

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

**In-Lab Problem:**

(c) Using `all-pairs`, complete the simple definition of the `join` relational operator:

```
(define (join table1 table2)
       ...)
```

(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 in most cases immediately use `h-slice` to keep only certain rows. As in several of our example queries, often we want just the rows where some pair of fields are equal.

Let's define a new relational operator `join+` which generalizes `join`. If `table1` and `table2` have no column names in common, then `join` and `join+` agree when applied to the two tables. But while `join` may only be applied to tables which have no column name in common (why?), `join+` will suffer no such restriction. If the tables do have column names in common, then `join+` returns only those concatenated rows from `table1` and `table2` which agree at the common names. It also keeps only one copy of the common entries of the row. For example,

```
(join+ events
       (rename-column                   ;meister-athletes
         athletes 'name 'cafemeister))
```

should produce the same result as:

```
(let ((all-but-name
       '(event cafemeister hours runnerup
         ru-hours sponsor last-yr-wscdi-rank)))
  (v-slice all-but-name
    (h-slice '(eq? cafemeister name)
             (join events athletes))))
```

Give a definition for `join+`. 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.

Finally, re-express Q4 using `join+` and `rename-column`, and run it again.

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.