Permutation Utilities in Lisp

gene m. stover

created Sunday, 2006 February 5
updated Tuesday, 2006 February 7

Contents

1 What is this?

Sometimes, you have a bunch of variables, each of which has a defined set of values it can take, & you want to generate all the permutations of the variables. Here are some utility functions in Lisp that do this task.

For example, maybe you have three variables, like this:

- *color*, which may be red, green, or blue;
- *size*, which may be small, medium or large; &
- *temperature*, which may be hot or cold.

You want a function to generate (*color*, *size*, *temperature*) for each combination of color, size, & temperature. Some of the values it might produce are:

1. (RED SMALL HOT)
2. (RED SMALL COLD)
3. (RED MEDIUM HOT)
4. (RED MEDIUM COLD)
5. (RED LARGE HOT)
6. (RED LARGE COLD)
7. (GREEN SMALL HOT)
8. (GREEN SMALL COLD)
9. (GREEN MEDIUM HOT)
10. (GREEN MEDIUM COLD)
11. (GREEN LARGE HOT)
12. (GREEN LARGE COLD)
13. (BLUE SMALL HOT)
14. (BLUE SMALL COLD)
15. (BLUE MEDIUM HOT)
16. (BLUE MEDIUM COLD)
17. (BLUE LARGE HOT)
18. (BLUE LARGE COLD)

The need for these functions shows up *all the time* in programming contests & sometimes at work.

2 License

Here is a copy of the license agreement that is on the source code. The same agreement appears in each source code file.

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name Gene Michael Stover nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

In particular, take note that:

1. The license agreement does not apply to this essay. This essay is covered by a different copyright notice which appears at the top of the essay.
2. Non-commercial & commercial use & distribution of the source code & binary code is specifically permitted.
3. You are not required to redistribute the source code, but if you do redistribute it, the license agreement must remain intact.

3 Lisp implementation

I implemented two functions, MAPERM & MAKE-PERMUTATOR.

3.1 maperm

MAPERM stands for "map permutator". It s analogous to MAPCAR.

To use MAPERM, you give it a function to call for each permutation & then lists of the values for each variable. It will call your function for each combination of values.

For example, the color, size, temperature example from Section 1 "What is this" works with MAPERM like this:

```
> (maperm #'(lambda (a b c) (print (list a b c)))
  '(red green blue) '(small medium large)
          '(hot cold))
 (RED SMALL HOT)
 (RED SMALL COLD)
 ...
 ...
 ...
 (BLUE LARGE HOT)
 (BLUE LARGE COLD)
18
```

Tfunction I gave to MAPERM prints each tuple. When it's done, MAPERM returns the number of tuples.

Here is the source code for MAPERM:

```
(defun maperm (fn &rest lists)
  (cond ((endp lists) nil)
        ((endp (rest lists)) (mapc fn (first lists)))
        (t (mapc #'(lambda (x)
                     (apply #'maperm
                            #'(lambda (&rest args)
                                (apply fn x args))
                            (rest lists)))
                 (first lists))))
  (reduce #'* (mapcar #'length lists)))
```

3.2 make-permutator

MAKE-PERMUTATOR returns a function which you can call repeatedly. Each time you call it, the function returns the next combination of values. When there are no more combinations, it returns NIL.

For example, the color, size, temperature example from Section 1 "What is this" works with MAKE-PERMUTATOR like this:

```
> (let ((prm (make-permutator '(red green blue)
                              '(small medium large)
                              '(hot cold))))
    (do ((x (funcall prm) (funcall prm)))
        ((null x))
        (print x)))
```

```
(RED SMALL HOT)
(RED SMALL COLD)
...
...
...
(BLUE LARGE HOT)
(BLUE LARGE COLD)
```

This example with MAKE-PERMUTATOR prints the same lines that the MAPERM example does. Notice that with MAKE-PERMUTATOR, you do not get the combination count unless you keep it or figure it out yourself.

Here is the source code for MAKE-PERMUTATOR:

```
(defun make-permutator (&rest lists)
  (cond ((endp lists) #'(lambda () nil))
        ((endp (rest lists)) (let ((lst (first lists)))
                               #'(lambda ()
                                   (and lst
                                        (let ((x (first lst)))
                                          (setq lst (rest lst))
                                          (list x))))))
        (t (let ((lst (first lists))
                 (prm (apply #'make-permutator (rest lists))))
             #'(lambda ()
                 (let ((y (funcall prm)))
                   (cond (y (cons (first lst) y))
                         ;; PRM is done, so we increment LST.
                         ((null (setq lst (rest lst)))
                          ;; We incremented LST, & it's NIL, so we
                          ;; are done.
                          nil)
                         (t
                          (setq prm (apply #'make-permutator (rest lists)))
                          (cons (first lst) (funcall prm)))))))))))
```

3.3 Performance

I suspect the deciding factor when it comes to choosing MAPERM or MAKE-PERMUTATOR is how you want to use it, not performance. Nevertheless, I was curious about their relative performances.

I wrote three functions to help use the two generator functions on the same inputs. Here are those three functions:

```
(defun make-test-lists (count)
  (declare (type integer count))
  (assert (plusp count))
  (let ((lst (loop for i from 1 to 3 collect i)))
    (loop for j from 1 to count collect lst)))
```

```
(defun time-maperm (count)
  (let ((calls 0))
    (time (apply #'maperm
                 #'(lambda (&rest args)
                     (declare (ignore args))
                     (incf calls))
                 (make-test-lists count))))
  calls))

(defun time-permutator (count)
  (let ((calls 0))
    (time (let ((prm (apply #'make-permutator (make-test-lists count))))
            (do ((x (funcall prm) (funcall prm)))
                ((null x))
                (incf calls))))
    calls))
```

Now let's use those helper functions & TIME to compare MAPERM & MAKE-PERMUTATOR:

```
> (let ((count 12))
    (list (time-permutator count)
          (time-maperm count)))

Evaluation took:
    5.214 seconds of wall time
    283421816 bytes consed.
Evaluation took:
    9.272 seconds of wall time
    517234960 bytes consed.
(531441 531441)
>
```

I am surprised that the MAKE-PERMUTATOR technique is noticeably faster than MAPERM. I guessed that using MAKE-PERMUTATOR would be slightly slower because it has to do the same work *and* has one extra level of function calls because each combination takes a function-call trip all the way back out to the client.

Since that's not the case, I'm guessing that the performance difference is due to the extra CONSing that MAPERM does implicitly, via the &REST arguments. In fact, the ratio between the amounts CONSed is just about equal to the ratio between the run times. Table 1 shows those ratios.

	consed	**seconds**
maperm	517234960	9.272
make-permutator	283421816	5.214
relative	1.825	1.778

Table 1: The CONS ratios & run time ratios

A Another way to implement it

Another Lisp implementation could use a bignum integer counter. To get the next combination, it would increment the integer, then extract the position in each list of attributes with MOD & ASH.

B C implementation

I sometimes think of making a permutation generator in C, but don't need it very often. I'll probably make one the next time I need one. (Or, more likely, the next time I need one, I'll wish I had made one now.)

C Other File Formats

- This document & its source code are available in multi-file HTML format at
  ./[1] .
- This document is available in Pointless Document Format (PDF) at prm.pdf[2] .

[1] `http://cybertiggyr.com/gene/prm/`
[2] `http://cybertiggyr.com/gene/prm/prm.pdf`