Lisp Idioms

Gene Michael Stover

created 2004 January 23
updated Tuesday, 2005 November 22

Contents

1 Introduction

This is a catalogue of Lisp expressions that are elegant, simple, or terse. These expressions might not be the most efficient way to do something. If efficiency is necessary for utility, then I'll claim that though some of these expressions might not be efficient at run-time, they make efficient use of programmer time.

1.1 Why?

Why would I make a catalogue of Lisp idioms? Many reasons. From least important to most:

- When I'm programming an application, I'm not in the same mental mode that I am when I read a book about Lisp. I often come to situations in which I know something can be done tersely in Lisp, but because I'm trying to get an application out the door, I don't stop to think of how to do it that way. With a catalogue of elegant Lisp expressions, I hope I'll take a moment to find the right expression without taking all the time to switch mental modes.
- Seeing a lot of Lisp idioms in a single collection will hopefully make it easier to see similarities & differences between them, understand them better, & make more.
- Because I enjoy seeing Lisp idioms, so I wanted a large collection of them so I can get lots of enjoyment in a single place.

2 Other Sources

A source of what could be considered idioms for FORMAT is "Advanced Use of Lisp's FORMAT Function"[Sto04].

3 Caveat Emptor

These expressions are not always the most efficient way to do something, so beware of using them in production or for large data sets. A common inefficiency among these expressions is that they CONS a lot.

4 Accumulate a List

If you have some function which returns items one at a time, & you need to process & accumulate the items in order, it is often efficient to push the items onto a list, like a stack, then to reverse the list before returning it.

Here's an example. First, let's make a function which returns items one at a time. Our example function just returns integers to be simple. In a real world case, you might be reading items from a file or something like that.

Here's our simple item-returning function:

```
> (defvar *item* 0)
*ITEM*
> (defun next () (incf *item*))
NEXT
```

Here is the recommended technique for accumulating those items into a list, in order:

```
> (defun accumulate-a (n)
    (do ((lst () (push (next) lst))
         (i 0 (1+ i)))
        ((>= i n) (nreverse lst))))
ACCUMULATE-A
> (accumulate-a 5)
(1 2 3 4 5)
```

This technique is fast because it pushes onto a list (which is fast), then uses NREVERSE to reverse the list. The NREVERSE function is possibly the fastest way to reverse the elements in a list because it permits the Lisp system to re-use the CONS cells.

Let's compare this technique to some alternatives.

The first alternative is to use REVERSE to reverse the list, like this:

```
> (defun accumulate-b (n)
    (do ((lst () (push (next) lst))
         (i 0 (1+ i)))
        ((>= i n) (reverse lst))))
ACCUMULATE-B
> (accumulate-b 5)
(6 7 8 9 10)
```

The second alternative, possibly the most naïve technique shown here, APPEND s items to the list we're accumulating:

```
> (defun accumulate-c (n)
    (do ((lst () (append lst (list (next))))
         (i 0 (1+ i)))
        ((>= i n) lst)))
ACCUMULATE-C
> (accumulate-c 5)
(11 12 13 14 15)
```

The third alternative pushes the items onto a resizable vector. The cost of this technique should have the same order as that of the CONS / NREVERSE technique.

```
(defun accumulate-d (n)
  (let ((v (make-array 1 :adjustable t :fill-pointer 0)))
  (do ((i 0 (1+ i)))
      ((>= i n) (coerce v 'list))
    (vector-push-extend (next) v))))
```

The fourth technique is like the third except that it allocates an array of the exactly requested size. *This is unrealistic if you don't know the number of items ahead of time, which is what I assume is the case when you are considering any of these techniques.* I'm including it here out of curiousity.

```
(defun accumulate-e (n)
  (let ((v (make-array n :adjustable nil :fill-pointer 0)))
  (do ((i 0 (1+ i)))
      ((>= i n) (coerce v 'list))
    (vector-push (next) v))))
```

Now compare the performaces of the three techniques:

```
> (defun time-test (moo n)
    (declare (type symbol moo) (type integer n))
    (format t "~&~A is ~A.  ~A is ~A." 'moo moo 'n n)
    (let ((start-time (get-internal-real-time)))
      (funcall moo n)
      (/ (- (get-internal-real-time) start-time)
         internal-time-units-per-second)))
TIME-TEST

> (defvar *moo* '(accumulate-a accumulate-b accumulate-c
                  accumulate-d accumulate-e))
*MOO*

> (mapcar #'(lambda (n)
              (format t "~&~A is ~A" 'n n)
              (cons n (mapcar #'(lambda (moo)
                                  (time-test moo n))
                              *moo*)))
          (loop for i in '(0 1 10 12 14 16)
                collect (expt 2 i)))
((1 0 0 1/1000 0 7/500) (2 0 0 0 0 0)
 (1024 1/1000 1/1000 101/1000 1/250 1/500)
 (4096 3/1000 3/1000 2703/1000 83/1000 3/500)
 (16384 13/1000 7/500 29743/1000 49/1000 27/1000)
 (65536 53/1000 171/1000 546013/1000 361/1000 21/250))
```

Let's convert that to a LaTeX table.

```
> (dolist (pair *)
    (format t "~&~D & ~{~,2E~^ & ~} \\\\ \\hline" (car pair)
            (cdr pair)))
```

Table 1 shows those results.

The first column shows the number of items we asked the technique to accumulate. The other columns show the number of seconds the technique required

n	accumulate-a	accumulate-b	accumulate-c	accumulate-d	accumulate-e
1	0.00e+0	0.00e+0	1.00e-3	0.00e+0	1.40e-2
2	0.00e+0	0.00e+0	0.00e+0	0.00e+0	0.00e+0
1024	1.00e-3	1.00e-3	1.01e-1	4.00e-3	2.00e-3
4096	3.00e-3	3.00e-3	2.70e+0	8.30e-2	6.00e-3
16384	1.30e-2	1.40e-2	2.97e+1	4.90e-2	2.70e-2
65536	5.30e-2	1.71e-1	5.46e+2	3.61e-1	8.40e-2
262144	2.13e+0	1.01e+0	forever	7.66e-1	7.37e-1
1048576	1.21e+0	2.58e+0	forever	3.48e+0	2.04e+0

Table 1: Performance comparison of the different techniques for accumulating a list

to accumulate that many items. Lower numbers indicate better performance. (The ACCUMULATE-C technique, which uses APPEND , is so slow that I didn't bother to test it for more than 2^{16} items.)

Except for the ACCUMULATE-C technique, the performances of the techniques are so close that their differences could reflect the inaccuracy of the timer or inequalities in how I ran the tests rather than the time required for the techniques. Nevertheless, if we insisted on believing the times, the CONS / NREVERSE is fastest, with the pre-allocated, non-resizable vector second. I'm surprised at that; I thought the pre-allocated vector would be faster, though you don't always know the number of items you'll accumulate, so it's not always appropriate. I presume the pre-allocated vector techinque took a hit because it must allocate the vector & then all the CONS cells, whereas the CONS / NREVERSE technique allocates only the CONS if NREVERSE takes advantage of its permission to re-use the CONS cells.

In case it matters, I used SBCL 0.8.8.

5 Pairs to Lists

If you are given a list of pairs (CONSes), but you need a list of lists, how do you convert?

For example, you have a list of pairs, like this:

```
(("red" . 1) ("green" . 2) ("blue" . 3))
```

but you need a list of two-element lists (not pairs), maybe so you can print them like this:

```
> (setq x '(("red" 1) ("green" 2) ("blue" 3)))
(("red" 1) ("green" 2) ("blue" 3))
> (format t "~:{~&~A is ~A~}" x)
red is 1
```

```
green is 2
blue is 3
NIL
```

How do you convert

```
(("red" .  1) ("green" .  2) ("blue" .  3))
```

to

```
(("red" 1) ("green" 2) ("blue" 3))
```

?

Here's one way.

```
(defun pairs-to-lists-explicit (pairs)
  (mapcar #'(lambda (pr)
              (list (car pr) (cdr pr)))
          pairs))
```

I called the function PAIRS-TO-LISTS-EXPLICIT because it's fairly explicit in how it works.

Here's a less explicit & less efficient function to do the same thing:

```
(defun pairs-to-lists-idiomatic (pairs)
  (mapcar #'list
    (mapcar #'car pairs)
    (mapcar #'cdr pairs)))
```

This section function has the same result, but it's not as explicit. It's also probably less efficient than PAIRS-TO-LISTS-EXPLICIT because it traverses the PAIRS list twice, CONSing each time. The first function, PAIRS-TO-LISTS-EXPLICIT, traverses the list only once, though it CONSes via LIST & the outer MAPCAR.

6 Reverse Lists within a List

What if I have a list of lists, & I want to reverse the elements within the inner lists? In other words, I have this:

```
((a 1) (b 2) (c 3))
```

and I want this:

```
((1 a) (2 b) (c 3))
```

This one is too easy to quality as an idiom, methinks. Here's a function to do it:

```
(defun reverse-inner (lsts)
  (mapcar #'reverse lsts))
```

7 Apply an Operator Longitudinally

I have a list of lists, & I want to apply some function to all of the n^{th} elements in the inner lists.

For example, maybe I have this:

```
((pear 3) (potatoe 7) (leek 5))
```

and I want to sum all of those numbers. (Maybe the numbers are the quantities of each item that I have.) So I want to apply some function to that list & have the function return 15.

This is a good use for REDUCE . Here it is in action:

```
> (setq x '((pear 3) (potatoe 7) (leek 5)))
((PEAR 3) (POTATOE 7) (LEEK 5))
> (reduce #'+ x :key #'second)
15
```

Tip: When you write your own functions to be used as the first argument to REDUCE , be sure that they behave properly whether given two, one, or zero arguments.

In theory, you can also APPLY an operator to a list. The next chunk of code uses that technique to accomplish the same feat we did with REDUCE :

```
> (apply #'+ (mapcar #'second x))
15
```

This "apply" technique works for many cases, but if the number of items is large, you can get a stack overflow. The technique using REDUCE avoids the stack overflow in my experience.

8 Find the Min or Max Element in a Sequence

It turns out to be really, really simple. Just use REDUCE to apply MIN (or MAX) to the sequence. You'll get the minimum (or maximum) element in return.

Let's get the minimum element from a list of integers:

```
> (reduce #'min '(1 2 3 4 5))
1

;; Works if the list is out of order
> (reduce #'min (reverse '(1 2 3 4 5)))
1
```

Finding the maximum element is almost the same:

```
> (reduce #'max '(1 2 3 4 5))
5
```

To get the *position* of the minimum (or maximum) element, use POSITION around those REDUCE expressions, like this:

```
> (let ((seq '(1 2 3 4 5)))
    (list
      ;; The position of the minimum element in SEQ
      (position (reduce #'min seq) seq)
      ;; The position of the maximum element in SEQ
      (position (reduce #'max seq) seq)))
(0 4)
```

Warning: While I'm pretty sure the REDUCE technique for finding the minimum or maximum is efficient at run-time, I'm not so sure about the POSITION / REDUCE technique for finding the position of the minimum or maximum element. It's short & sweet at programming time.

9 Transpose a Matrix

Let's say you have a matrix represented as a list of lists.[1] How do you transpose the matrix? In other words, if you have this datum:

```
((1 2 3)
 (4 5 6))
```

how do you convert it to this datum:

```
((1 4)
 (2 5)
 (3 6))
```

?

It's easy. Check it out:

```
(defun transpose (x)
   (apply #'mapcar (cons #'list x)))
```

Here is the TRANSPOSE function in action:

```
> (transpose '((1 2 3) (4 5 6)))
((1 4) (2 5) (3 6))
> (transpose *)
((1 2 3) (4 5 6))
```

A Other File Formats

This document is available in multi-file HTML format at http://cybertiggyr.com/gene/lid/.

This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/lid/lid.pdf

[1] This probably is not the most efficient way to store a mathematical matrix, but it happens sometimes.

References

[Sto04] Gene Michael Stover. Advanced use of lisp's format function. *personal web site*, January 2004. http://cybertiggyr.com/gene/fmt/.