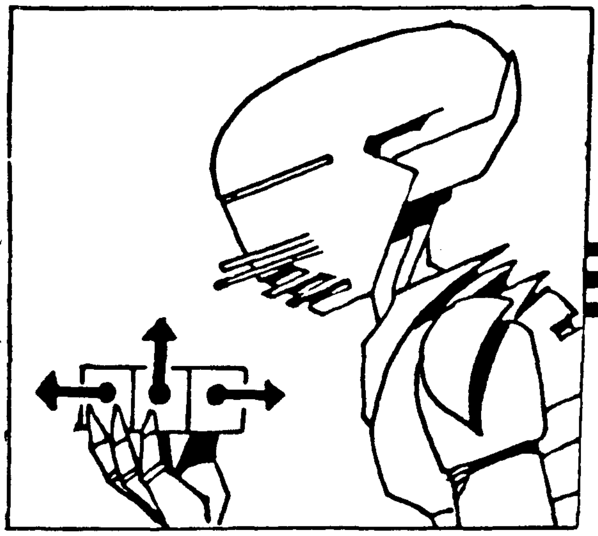

Query-io

So far, I haven't been inundated with questions from you, dear readers. But that's all right, I still have some in stock. Just make sure it doesn't dry up. Oh, by the way, I got a very interesting comment on the last issue. I decided to share it with you:

Patrick;

While I agree with the principle text of your query-io column, your choice of example is encouraging users to shoot themselves in the foot. I don't know what you are planning for next issue, but here is some grist for your mill.

It is extremely ill-advised to attempt to micro-optimize code in the manner suggested by Query-io, as the performance improvement obtainable is microscopically small, while the peril of introducing subtle bugs is large.

Consider the original code (here, made into a function)

```
(defun add-del-1 (new old)
   (push new list-of-items)
   (delete old list-of-items)
   list-of-items)
```

What does this code do if `(EQL new old) -> T` ? In all probability, this original code is subtly buggy; to be concrete, suppose `LIST-OF-ITEMS` is `(A B C)` and we call `(ADD-DEL-1 'FOO 'FOO)`.

Interpretation #1: At first glance would suggest that the result should be `(A B C)`, that is, `FOO` would be added and then deleted.

Interpretation #2: At second glance, keeping in mind the "bug" of depending on the side effect behavior of DELETE, the correct result will be (FOO A B C), because PUSH puts foo at the beginning of the list, and DELETE cannot remove it there, so the original list remains. Under some interpretations, the user was depending on a subtle feature, and the absence of the (SETQ list-of-items --) is not a bug but an obscure use of the known behavior of delete.

Interpertation #3: At third glance, suppose we "expect" result #2 (FOO A B C), and call (ADD-DEL-1 'FOO 'FOO) again. What result? Surprise! The result is (FOO FOO A B C). Now the credibility of interpretation #2 is strained to the breaking point; Its hard to concieve of the contract of ADD-DEL-1 permitting a result with two copies of FOO.

So now lets backtrack a bit. We've demonstrated that the original definition of ADD-DEL-1 is probably buggy, but remember that we weren't trying to fix a bug, we were trying to bum performance, and incidentally in the process of bumming performance, uncovered a "bug" in the absence of a SETQ. Two possible functions come out of this:

```
;; Fix the bug, but don't change the algorithm
(defun add-del-2 (new old)
  (push new list-of-items)
  (setq list-of-items (delete old list-of-items))
  list-of-items)

;; Fix the bug, and make a minor speed improvement
(defun add-del-3 (new old)
  (setq list-of-items (delete old list-of-items))
  (push new list-of-items)
  list-of-items)
```

Now for the crunch! Reading the comments in ADD-DEL-2 and ADD-DEL-3, and looking at the code, one might believe that the comments and code for both functions are correct. In fact, neither of them is! Both ADD-DEL-2 and ADD-DEL-3 give different results than ADD-DEL-1 in the case that OLD and NEW are EQL.

Moral: Changing code is a subtle business. Tweaking code as suggested in Query-io, for the sake of saving one CDR operation, is not worth the danger of introducing subtle bugs. Even *fixing* subtle bugs in existing code can only be done in conjunction with careful consideration of what callers may have been depending on buggy behavior.

Dave,

I agree completely with your comments. However I don't think my intent was to encourage or suggest to people to do micro optimization on code they don't understand, but merely to explain the problems you can get into relying on side effects of destructive functions.

I tried to make my point using a classic situation where a piece of code works "by chance" and showing that a seemingly small change, done by somebody with good intent but limited knowledge, uncovers (one of) the bugs in the code.

To answer the last philosophical point in your conclusion: if I would come across a piece of code like `Add-del-1`, I would change it. Not because I could save one `CDR`, but because this code is unmaintainable. I completely agree with your last point though: "Even fixing subtle bugs in existing code can only be done in conjunction with careful consideration of what callers may have been depending on buggy behavior."

One message I want to get across is: "Try to understand the code you are working with, otherwise you are likely to screw up sooner or later".

And now, on with some questions,

> *What is a cdr-coded list?*

Cdr-coding is a technique used to store lists efficiently on some Lisp implementations. The usual and obvious representation of a list is as several cons-cells. The first element of the cons-cell point to an element of the list and the second element of the cons-cell points to the next cons-cell. For an n elements list, we need n cons-cells, using up n words for indirection pointers linking the cons-cells together. Moreover, the cons-cells are not necessarily close together, increasing chances for page faults while going through the elements of the list.

When a list is cdr-coded, it is stored as one block of storage of n words, each pointing to their corresponding element of the list. Functions looking up lists like get, assoc, member... will be faster on these lists and are likely to generate less page faults. Applying functions that change the CDR of a sublist like `Rplacd`, `Nconc`, `Delete`, `Nreverse` on a cdr-coded list is less easy, because the CDRs are implict and not allocated. `Rplacd` has to allocate one before changing it and the original list is repaired using an invisible indirection pointer. This process is transparent to the user. Lists produced by `List`, `List*` `Copy-list` are cdr-coded, lists built with `Cons`, `Push`, `Nreverse`, `Mapcar` are not. Even if only few implementations use cdr-coding, keeping this in mind while designing a program will help get an extra speed up on those implementations and will avoid surprisingly low performances on certain operations, because the program produces cdr-coded lists and modifies their implicit CDRs.

> (read-from-string my-string :start n) *does not seem to work. What's wrong?*

The problem is that read-from-string takes two optional arguments: EOF-ERROR-P and EOF-VALUE before the key arguments. They need to be supplied before one can supply any key, otherwise the first keyword (key name) is taken as a non null value for EOF-ERROR-P and its value is taken as EOF-VALUE. It produces the strange effect that read-from-string starts reading at the beginning of my-string instead of starting at n.

There is a lesson to be learned from this. When writing an interface function using key arguments, don't use optional arguments at the same time. It works but it is very confusing.

The following is a Common Lisp program that computes a relatively familiar function in a relatively unfamiliar way. It has been written with uninformative variable names. The idea is to figure out what the program does without running it.

This one is hard. It is a logarithmic algorithm for computing a well-known function. Like many logarithmic algorithms, it was derived by mapping the original problem into another domain. In this case the program maps the original problem, which manipulates numbers, into a problem that involves manipulating something else. We solve the mapped problem logarithmically, and finally map back. The final twist is that the object being manipulated in the program is represented in a diabolical manner!

```
(defun f (i)
 (labels ((g (n n-1 n-2 m m-1 m-2)
          (let ((k (* n-1 m-1)))
        (values (+ (* n m) k)
            (+ (* n m-1) (* n-1 m-2))
            (+ k (* n-2 m-2)))))
      (h (i)
          (cond ((zerop i) (values 1 0 0))
         ((= i 1) (values 1 1 0))
         ((evenp i)
          (multiple-value-bind (n n-1 n-2)
              (h (truncate i 2))
              (g n n-1 n-2 n n-1 n-2)))
         (t
          (multiple-value-bind (n n-1 n-2)
              (h (1- i))
              (g 1 1 0 n n-1 n-2))))))
  (values (h i))))
```

Dick Gabriel