SICP section 3.3.1

September 29, 2007 at 11:20 **Tags** SICP

Until now all the code I've written for chapter 3 was in Scheme. Now I want to switch to Common Lisp, mainly in order to explore its facilities for state and assignment.

It is actually quite interesting to compare Scheme with Common Lisp in this respect, because they're a bit different. Scheme has a *setter* function for most types it supports, including pairs (`set-car!`, `set-cdr!`). On the other hand, CL has a common `setf` macro which is useful for all[1] types. It can be used to set variables, structure elements, array cells, pairs, and so on.

In fact, as a remnant from the past CL has special functions for setting the `car` and `cdr` of pairs: `rplaca` and `rplacd`. Apart from the ugly names (`rplaca` is "replace car"), these functions are no longer recommended for use by the CL community. Instead of writing `(rplaca pair val)`, it is a better style to write `(setf (car pair) val)`. Although a bit longer, it is better in the sense that one doesn't have to employ a special function, and uses the common `setf` which is very familiar[2].

Exercise 3.12

Let's translate the code to CL and see the results[3]:

```
(defun my-append (x y)
  (if (null x)
    y
    (cons (car x) (my-append (cdr x) y))))

(defun my-append! (x y)
  (setf (cdr (last x)) y)
  x)

(defvar x (list 'a 'b))
(defvar y (list 'c 'd))
(defvar z (my-append x y))

z
=> (a b c d)
(cdr x)
=> (b)

(defvar w (my-append! x y))

w
=> (a b c d)
(cdr x)
=> (b c d)
```

Note that the call to `my-append!` attaches `y` onto the tail of `x` and returns `x`. Therefore, `w` and `x` point to the same location.

Exercise 3.13

```
(defun make-cycle (x)
  (setf (cdr (last x)) x)
  x)
```

This creates a `circular list`. The `cdr` of the last cell in the list, instead of pointing to `nil`, points to the first cell of the list. Now any attempt to walk the list or print it will result in an infinite loop.

Circular lists are useful in some situations, but one has to know one's dealing with them and write code accordingly. Printing out a circular list is just one of those things you don't do[4].

Exercise 3.14

`mystery` reverses the list `x`. Here's the code rewritten in CL:

```
(defun mystery (x)
  (labels (
    (my-loop (x y)
      (if (null x)
        y
        (let ((temp (cdr x)))
          (setf (cdr x) y)
          (my-loop temp x)))))

    (my-loop x '())))
```

Running it:

```
(defvar v '(a b c d))
(defvar w (mystery v))

w
=> (d c b a)

v
=> (a)
```

Exercise 3.15

I'll skip this. I think I understand box-and-pointer diagrams well enough (coming from a C background, I must) and they're too tiresome to draw.

Exercise 3.16

Again, I will not draw the diagrams themselves, but I will present the data structures that cause this output. You can use a paper and a pencil to draw these simple diagrams according with the code. First, here's the code in CL:

```
(defun bad-count-pairs (x)
  (if (not (consp x))
    0
    (+  (bad-count-pairs (car x))
        (bad-count-pairs (cdr x))
        1)))
```

Now, let's define a simple list:

```
(defvar z '(a b c))
(bad-count-pairs z)
=> 3
```

If we set the `car` of the second element of the list to point to the third (instead of the symbo`b`), we'll get a count of 4:

```
(setf (car (cdr z)) (cddr z))
(bad-count-pairs z)
=> 4
```

If we also set the `car` of the first element of the list to point to the second:

```
(setf (car z) (cdr z))
(bad-count-pairs z)
=> 7
```

If this is not clear, draw the diagrams for these operations on paper and convince yourself!

To get an endless loop, any loop in the list will suffice, such as setting the `car` of an element to itself:

```
(setf (car z) z)
(bad-count-pairs z)
=> *** - Program stack overflow. RESET
```

Exercise 3.17

Since each object in CL is `eq` to itself, we can just keep a table of the pairs we've already seen. I'll use CL's hash table facility for this purpose:

```
(defun good-count-pairs (x)
  (let ((pairs-table (make-hash-table :test #'eq)))
    (labels (
      (traverse-count (x)
        (cond
          ((not (consp x)) 0)
          ((gethash x pairs-table) 0)
          (t
            (setf (gethash x pairs-table) 1)
            (+  (traverse-count (car x))
                (traverse-count (cdr x))
                1)))))

      (traverse-count x))))
```

Now, the count for all the structures of exercise 3.16 return 3, as expected.

Exercise 3.18

We'll employ a technique similar to the `good-count-pairs` function – remember which pairs were already seen. If we run into one we've seen before, the list has loops.

```
(defun has-loop? (x)
  (let ((pairs-table (make-hash-table :test #'eq)))
    (labels (
      (traverse-list (x)
        (cond
          ((null x) nil)
          ((gethash x pairs-table) t)
          (t
            (setf (gethash x pairs-table) 1)
            (traverse-list (cdr x))))))

      (traverse-list x))))
```

Exercise 3.19

The algorithm outlined in the solution of 3.18 uses linear space, of course[5]. It is possible to do it in constant space, and in fact this is a common interview question. The algorithm for this is a bit clever, but simple to understand.

To find out if a list has a loop, we'll traverse it using two pointers. One will walk the list normally, from one element to the next. Another will advance 2 elements at a time. If, and only if, the list has a loop, the double-speed pointer will meet the normal pointer again after the beginning. Think about it for a moment – it actually makes a lot of sense.

To do this I'll employ an iterative technique, using CL's `do` form. It has a built-in ability of advancing several iterators, which is useful in this case:

```
(defun has-loop-O1space? (x)
  (do ( (iter-1 (cdr x) (cdr iter-1))
        (iter-2 (cddr x) (cddr iter-2)))
      ((null iter-2) nil)
    (when (eq iter-1 iter-2)
      (return t))))
```

The code follows the algorithm I outlined exactly. One small thing to note is the lack of boundary tests. I rely on the very convenient fact that in CL, `(cdr nil)` is just `nil`. Therefore, there will be no errors generated in the `do` loop even if an empty list is passed in[6].

Exercise 3.20

Skipping.

---

[1] I'm not 100% sure on this and would love some constructive comments for this claim. CL has so many dusty corners, one has to be an expert to know.

[2] I tried asking in the `#lisp` IRC room, but couldn't fathom a deeper reason for `setf`'s superiority in this case.

[3] I'm attaching `my-` to the names of the functions because CL doesn't allow redefining built-ins.

[4] Trying to do it for this exercise almost killed my Windows session.

[5] Since it's a hash table, in most likeness it uses even more memory than the amount of elements in the list, but asymptotically it's still O(n).

[6] In Scheme, on the other hand, `(cdr '())` generates an error – which forces the programmer to do more manual tests.

For comments, please send me an email.

Back to top