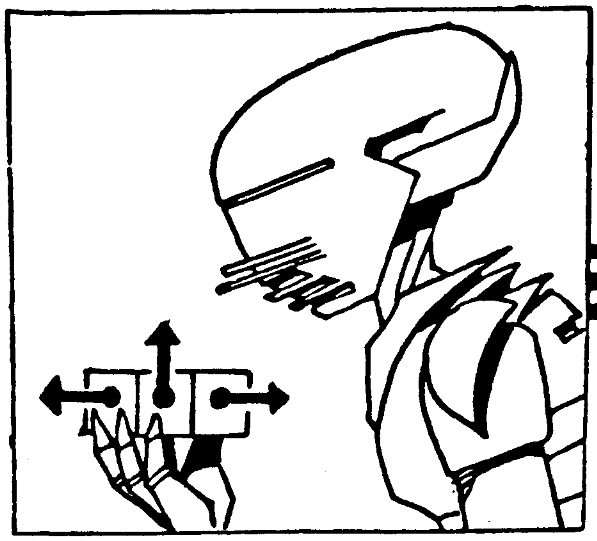

Query-io:

Patrick,
I was wondering how someone can build a list from left to right efficiently in Lisp without using a loop macro.

Good question! Let's take a simple function to illustrate the problem. This function has to build the list of the n first integers, starting from 0.

Here is a typical right to left (backward) consing algorithm

```
(defun right-to-left (n)
   (let ((ls ()))
      (dotimes (i n ls)
         (setf ls (cons i ls)))))
```

Its speed order is O(n).

Now let's convert it to left to right (forward) consing. One way is to replace CONS by NCONC:

```
(defun left-to-right-nconc (n)
   (let ((ls ()))
     (dotimes (i n ls)
       (setf ls (nconc ls (cons i nil))))))
```

The problem is that the speed order is n square. NCONC has to go to the last cons of the growing list to add the next element. Another way is to keep the original algorithm and reverse the list at the end:

```
(defun left-to-right-nreverse (n)
  (let ((ls ()))
    (dotimes (i n)
      (setf ls (cons i ls))
    (nreverse ls)))
```

Note that we can use NREVERSE since all the cons cells of ls are freshly consed. The speed order is O(n). Yet if this function is used often for small values of n, it can be significantly slower than the previous ones because of the overhead of NREVERSE. Each cons cell has to be accessed twice. The next algorithm is a variant of the NCONC one. The idea is to keep a reference to the last cons cell of the list being built and add the next element to it:

```
(defun left-to-right-tail (n)
  (let* ((ls ())
         (tail ls))
    (dotimes (i n)
      (if ls
          (setf tail (setf (rest tail) (cons i nil)))
          (setf tail (setf ls (cons i nil)))))
    ls))
```

This code is probably the least readable of all but its speed order is O(n) and each list cons is accessed only once. On most implementations, its speed is close to the right to left consing one.

The above algorithms apply to the very general case of building lists from left to right. In special situations, other types of algorithms can be better. An important special case is filtering. Starting from an input list, some elements are selected to construct the output list. Using a map function is just the right thing to do here:

```
(defun filter (l)
  (mapcan #'(lambda (n) (if (evenp n) (cons n nil) nil)) l))
```