Efficiency Issues

Pitfall 1:
Building a list and traversing it again.

Example:

```
(defun rev (list)
   (cond ( (null list) nil )
         ( t (append (rev (cdr list))
                     (list (car list)))
         )
   )
)
```


Efficient version:

```
(defun rev2 (remainder sofar)
   (cond ( (null remainder) sofar )
         ( t (rev2 (cdr remainder)
                   (cons (car remainder) sofar)
             )
         )
   )
)

; A nicer interface.
(defun my-reverse (list)
    (rev2 list nil)
)
```

Pitfall 2:
Evaluating the same expression twice

Example:

```
(defun mostatoms (list)
   (cond ( (null list) 0 )
         ( (> (countatoms (car list))
              (mostatoms (cdr list)))
                    (countatoms (car list))
         )
         ( t (mostatoms (cdr list)) )
   )
)
```


Efficient version:

```
(defun most2 (list)
   (cond ( (null list) 0 )
         ( (biggest (countatoms (car list))
                    (most2 (cdr list))) )
   )
)

(defun biggest (x y)
   (cond ( (> x y) x )
         ( t y )
   )
)
```

In fact, Lisp has a built-in `max` function, so in this case we don't need to write our own helper.

Let

When a helper function is not a natural solution to Pitfall 2, use `let`.

```
(let ( ( var1 value1 )
       ( var2 value2 )
       ...
       ( varn valuen )
     )
     ; Can now use var1 through varn
)
; var1 through varn now revert to having
; no value, or to their previous values if
; they had any.
```


More Practise Writing Functions

```
; Returns the sum of all elements of ‘list’.
; Preconditions:




(defun sumlist (list)
    (cond ( (null list) 0 )
          ( t (+ (car list)
                 (sumlist (cdr list))) )
    )
)
```

```
; Returns a list containing the sum for each
; list in ‘lists’.
; Preconditions:




(defun sums (lists)
    (cond ( (null lists) nil )
          ( t (cons (sumlist (car lists))
                    (sums (cdr lists))
              )
          )
    )
)
```



```
; Returns a list containing the length of each
; list in ‘lists’.
; Preconditions:




(defun lengths (lists)
    (cond ( (null lists) nil )
          ( t (cons (length (car lists))
                    (lengths (cdr lists))
              )
          )
    )
)
```

```
; Returns a list containing the cdr of each
; list in 'lists'.
; Preconditions:
```

```
(defun cdrs (lists)
    (cond ( (null lists) nil )
          ( t (cons (cdr (car lists))
                    (cdrs (cdr lists))
              )
          )
    )
)
```


Passing Functions as Arguments

Example 1: consall

We could have different functions `sums`, `lengths`, and `cdrs`. Each one does a certain thing to every element of a list.

The code is very repetitive. And if we want to do something else to every element of a list, we need to write yet another function.

Instead, we can write one general function.

```
; Returns a list containing the result of applying 'fcn'
; to each element of 'list'.
; Preconditions:
(defun consall (list fcn)
    (cond ( (null list) nil )
          ( t (cons (funcall fcn (car list))
                    (consall (cdr list) fcn)
              )
          )
    )
)
```

Sample run

```
gardiner.cs 1% cl
Allegro CL 4.3 [SPARC; R1] (5/10/96 17:51)
Copyright (C) 1985-1996, Franz Inc., Berkeley, ...

USER(1): :ld fcnpars.cl
; Loading ./fcnpars.cl

USER(2): (sums `((3 1) (6 29 4) (5) ()))
(4 39 5 0)

USER(3): (lengths `((3 1) (6 29 4) (5) ()))
(2 3 1 0)

USER(4): (cdrs `((3 1) (6 29 4) (5) ()))
((1) (29 4) NIL NIL)

USER(7): (consall`((3 1) (6 29 4) (5) ()) `sumlist)
(4 39 5 0)

USER(8): (consall`((3 1) (6 29 4) (5) ()) `length)
(2 3 1 0)

USER(9): (consall`((3 1) (6 29 4) (5) ()) `cdr)
((1) (29 4) NIL NIL)
```