Do

Forma repetitiva geral

(DO ((*variável v-inicial próximo) ...)*
(teste-saída resultado)
corpo ....)

- Cada variável é inicialmente ligada ao valor inicial.
- Se o teste de saída é verdadeiro, resultado é retornado.
- Senão, o corpo é executado e cada variável é atualizada para o próximo valor.
- Se o próximo valor é omitido, a variável não é alterada.

```
(defun tira_elem (lista e)
   (do ((l-aux lista (cdr lista)) (res ( )))
         ((null l-aux) res)
          (if (not (equal (car l-aux) e))
             (setq res (cons (car l-aux) res)))))
```

```
> (tira_elem ´(1 4 6 V (d 4) 4 (1 3)) 4)
  (1 6 V (d 4) (1 3))
```

Funções de ordem superior

- Funções que recebem outras funções como argumento
- MAPCAR – aplica uma dada função repetidamente aos argumentos dados

(mapcar <nome da função> <lista de argumentos 1> <lista de argumentos 2> ...)

```
(defun soma-um (x)   (+ x 1))
SOMA-UM
> (mapcar 'soma-um '(1 4 7 3))
(2 5 8 4)
```

- Se a função tem mais de um argumento, mapcar deve ter tantas listas quantos forem os argumentos da função

```
> (mapcar '+ '(1 2 3) '(3 4 1))
(4 6 4)

> (mapcar 'equal  '(1 2 3) '(3 2 1))
(NIL T NIL)

> (defun par-dobro (x) (list x (+ x x)))
PAR-DOBRO

> (mapcar 'par-dobro '(2 5 3))
((2 4) (5 25) (3 9))
```

- Apply – aplica uma função a uma lista de argumentos

(apply <nome da função) <lista de argumentos>)

```
> (apply '+ (1 2 3 4))
10

> (defun CONTA-ATOMOS (L)
    (cond ((null L) 0)
          (( atom L) 1)
          (T (apply ´PLUS (mapcar 'CONTA-ATOMOS L))))))
```

Recursão X Iteração

- As repetições em um programa Lisp podem ocorrer por meio de recursão ou iteração.
- Alguns problemas são mais naturalmente resolvidos usando recursão, outros por iteração.
- As principais construções de Lisp para recursão são:
  - Do
  - Dolist
  - Dotimes
  - Mapcar

Exemplo:

- Construir uma função Lisp que, calcula a função de Fibonacci, definida por:

  f(0) = 1
  f(1) = 1
  f(n) = f(n-1) + f(n-2)

```
(defun FIBONACCI (N)
   (cond ((zerop N) 1)
         ((equal N 1) 1)
         (t (plus (FIBONACCI (- N 1))
                  (FIBONACCI (- N 2))))))
```

Essa função é naturalmente recursiva, mas a solução é bastante ineficiente, por exigir cálculos repetidos.

O uso da recursão ou da interação deve considerar o balanceamento entre a facilidade da construção da solução e a eficiência.

Recursividade na cauda

- Recursividade na cauda: uma função é recursiva na cauda se os valores retornados por essa função não são alterados no nível anterior
- Conta-atomos e Fibonacci NÃO são recursivas na cauda

```
(defun fibonacci-iter (N)
   (cond ((equal N 0) 1)
         ((equal N 1) 1)
         (t (do ((N1 2 (+ N1 1)) (R-ant 1) (R 2))
                ((equal N1 N) R)
                (setq AUX (+ R-ant  R))
                (setq R-ant  R)
                (setq R AUX)))))
```

Escopo léxico X escopo dinâmico

- Common Lisp tem dois tipos de variáveis:
  léxicas e especiais
- As variáveis léxicas são definidas:
  - Por alguma construção sintática (let ou defun) e podem ser referenciadas pelo código que aparece dentro dessa construção.
  - Por setq e podem ser referenciadas em qualquer lugar, desde que não tenham sido redefinidas (dependendo da implementação)
- Variáveis léxicas tem escopo léxico – definido de acordo com o texto do código do programa

Exemplo – variável léxica definida por let

```
(let ((x 1))          ; Define uma ligação para x
    (print x)
    (let ((x 2))      ; Define uma segunda ligação para x
        (print x)
        (setq x 3) ; Muda o valor da segunda ligação
        (print x))
     (print x))       ; O valor da primeira ligação não se altera
```

A saída será: 1 2 3 1

O SETQ muda o valor da segunda ligação (interna) de x, sem alterar a primeira (externa)

Exemplo – variável léxica definida por setq

```
> (setq var-lex 5)
5
> (defun verifica-lex ( ) var-lex)
VERIFICA-LEX

> (verifica-lex)
5
> (let ((var-lex 6)) (verifica-lex))
5
```

```
 >(defun foo( )
(let ((x 1) (y 2)) ; define ligações para x e y
      (frotz x y) ;    referências as ligações
      (let ((y 2) (z 3)) ; define outra ligação para y e uma para z
            (baz x y z)) ; referências a x (primeira) y (segunda) e z.
                         ; a primeira ligação de y náo pode ser
                           ; acessada aqui.
                           ; ela foi "sombreada" pela segunda ligação
         z)) ;       referência a variável livre z
```

Na última linha, z é uma variável livre

```
> (setq z  5)
5
> (foo)
5
```

Variáveis especiais

- As variáveis especiais são definidas por DEFVAR
- Por convenção, são escritas entre *

```
> (defvar *x*)
```

- Variáveis especiais tem escopo dinâmico – definido pela sequência de chamadas, ou da execução do programa

Exemplo – variável especial definida por defvar

```
> (defvar var-lex 5)
5
> (defun verifica-lex ( ) var-lex)
VERIFICA-LEX

> (verifica-lex)
5
> (let ((var-lex 6)) (verifica-lex))
6
> (verifica-lex)
5
```

```
>(defvar x 1)
x

>(defun baz () x)
baz

> (defun foo ()
        (let ((x 2))
             (baz)))
foo

> (foo)
2
```

Função symbol-value

- A função symbol-value permite acessar o valor da ligação dinâmica de uma variável

```
> (setq a `global-a)
A
> (defvar *b* `global-b)
*B*
(defun fn ( ) *b*)
FN
> (let ((a `local-a) (*b* `local-b))
   (list a *b* (fn) (symbol-value `a) (symbol-value `*b*)))
(local-a local-b local-b global-a local-b)
```