Ricorsione

Si dice **ricorsiva** una funzione che chiama se stessa

```
(defun fibonacci (n)
  (if (or (= n 0) (= n 1))
    1
    (+ (fibonacci (- n 1))
       (fibonacci (- n 2)))))
```

Una funzione ricorsiva si dice **tail-recursive** se non rimane nulla da fare dopo la chiamata ricorsiva. La fibonacci di sopra non è tail-recursive

```
(defun my-nth (lista n)
  (if (= n 0)
    (car lista)
    (my-nth (rest lista) (- n 1))))

my-nth è tail-recursive
```

Es. Fattoriale

```
(defun fac (n)
  (cond ((= n 0) 1)
        (t (* n (fac (- n 1))))))
```

Per otternere una versione tail recursive del fattoriale serve una funzione ausiliaria

```
(defun fact (n)
  (fact-aux n 1))

(defun fact-aux (n res)
  (if (= n 0)
    res
    (fact-aux (- n 1) (* n res))))

? (time (fact 50))
(FACT 50) took 1 ticks (0.017 seconds) to run.
30414093201713378043612608166064768844377641568960512000000000000
? (time (fac 50))
(FAC 50) took 1 ticks (0.017 seconds) to run.
30414093201713378043612608166064768844377641568960512000000000000
```

Esempi

Una funzione che controlla se un elemento appartiene ad una lista

```
(defun my-member (obj lista)
  (if (null lista)
    nil
    (if (eql obj (car lista))
      lista
      (my-member obj (cdr lista))))) ; “else” di if
```

Una funzione che conta le occorrenze di un elemento in una lista

```
(defun conta (obj lista)
  (if (null lista)
    0
    (if (eql obj (car lista))
      (+ 1 (conta obj (cdr lista)))
      (conta obj (cdr lista))))))    ; “else” di if
```

Esempi

Cosa ritornano ?

```
(defun enigma (x)
  (and (not (null x))
       (or (null (car x))
           (enigma (cdr x)))))

(defun mistero (x y)
  (if (null y)
    nil
    (if (eql (car y) x)
      0
      (let ((z (mistero x (cdr y))))
        (and z (+ z 1))))))
```

Esempio funzioni mutuamente ricorsive:

```
(defun a (x) (if (= x 0) t (b (- x))))
(defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
? (a 5)
T
```

&optional - &rest

`&optional` permette di specificari parametri che non devono essere necessariamente passati

```
(defun root (x &optional n)
  (if n
    (expt x (/ 1 n))
    (sqrt x)))
```

`&rest` accumula tutti i parametri passati in ua lista

```
(defun my+ (n &rest other)
  (my+aux n other))

(defun my+aux (n other)
  (if (endp other)
    n
    (my+aux (+ n (first other)) (rest other))))
```

&key - &aux

Se un parametro viene definito `&key` l'argomento può essere passato senza riferimento posizionale

```
(defun cut-list (l &key direction (how 1))
  (if (eq direction 'left)
    (nthcdr how l)
    (butlast l how)))

? (cut-list '(a b c) :direction 'left)
(B C)
? (cut-list '(a b c) :direction 'left :how 2)
(C)
? (cut-list '(a b c) :direction 'sinistra :how 2)
(A)
```

Se un parametro viene definito `&aux` non gli deve essere passato alcun valore al momento della chiamata

```
(defun primsec (l &aux (f (pop l)))
  (cons f (first l)))
? (primsec '(a b c))
(A . B)
```

Data Abstraction

Nascondere l'implementazione e fornire **accessori** (procedure)

– **costruttori.** Procedure che ritornano un nuove oggetto del tipo desiderato

– **lettori.** Procedure che fanno accesso all'oggetto e ritornano informazioni legate all'oggetto

– **scrittori.** Procedure che permettono di modificare le informazioni legate all'oggetto

VANTAGGI

• Non dovete ricordare **come** avete memorizzato i dati

•Cambiando l'implementazione dovete cambiare solo gli accessori

• Non si devono ricordare complicate funzioni di basso livello

Esempio

Un data base di CD usando liste di associazione

```
((compositore '(L. V. Beethoven)) (opera '(sinfonia n.
5)) (esecutore '(H. V. Karajan)))

(defun make-cd (&key compositore opera esecutore)
  (list (list 'compositore compositore)
        (list 'opera opera)
        (list 'esecutore esecutore)))
(defun get-compositore (cd)
  (second (assoc 'compositore cd)))

? (make-cd :compositore '(l. v. beethoven)
           :esecutore '(h. v. karajan)
           :opera '(sinfonia n 5))
((COMPOSITORE (L. V. BEETHOVEN)) (OPERA (SINFONIA N 5))
(ESECUTORE (H. V. KARAJAN)))
? (get-compositore *)
(L. V. BEETHOVEN)
```

Esempio

```
(defun set-compositore (cd comp)
  (list (list 'compositore comp) (rest cd)))

? (set-compositore (make-cd :compositore '(l. v. beethoven)
                            :esecutore '(h. v. karajan)
                            :opera '(sinfonia n 5))
                   '(W. A. Mozart))
((COMPOSITORE  (W.  A.  MOZART))  ((OPERA  (SINFONIA  N  5))
(ESECUTORE (H. V. KARAJAN))))

(defun trova-esecutore-sinfonie (cd)
  (if (member 'sinfonia (get-opera cd))
    (get-esecutore cd)))
```

Cosa succede se vogliamo aggiungere un nuovo campo alla nostra rappresentazione di CD ?

Esempio

Dobbiamo cambiare solo `get-... , make-cd, set-...`

```
(defun make-cd (&key compositore opera esecutore
                     data-incisione)
  (list (list 'compositore compositore)
        (list 'opera opera)
        (list 'esecutore esecutore)
        (list 'data-incisione data-incisione)))

(defun get-data-incisione (cd)
  (second (assoc 'data-incisione cd)))

(defun set-data-incisione (cd data)
  (append (butlast cd) (list (list 'data-incisione data))))
```

MEGLIO LA SEGUENTE FUNZIONE !

```
(defun set-data-incisione (cd data)
  (if (eq 'data-incisione (first (first cd)))
    (cons (list 'data-incisione data) (rest cd))
    (cons (first cd) (set-data-incisione (rest cd) data))))
```

MAPPING

La special form MAPCAR serve ad applicare una funzione ad una lista (o più liste) e a raccogliere i risultati in un'altra lista

```
? (mapcar #'abs '(1 -4 -7))
(1 4 7)

? (mapcar #'+ '(1 -4 -7) '(0 5 6))
(1 1 -1)

? (mapcar #'not '(t nil t nil t nil))
(NIL T NIL T NIL T)
```

REMOVE-IF serve a rimuovere tutti gli elementi di una lista che soddisfano una condizone

```
? (remove-if #'numberp '(a b 5 c))
(A B C)
? (remove-if-not #'numberp '(a b 5 c))
(5)
```

FUNCALL - APPLY

FUNCALL e APPLY sono special forms che servono ad applicare una funzione passata come argomento ai restanti parametri

`(FUNCALL #'nome-funzione &rest args)` - applica la funzione `#'nome-funzione` agli argomenti in `args`

`(APPLY #'nome-funzione arg &rest more-args)`- applica `#'nome-funzione` alla lista di argomenti ottenuti appendendo l'ultimo argomento alla lista formata da gli altri argomenti

L'argomento finale di APPLY deve essere una lista.

```
? (funcall #'+ 1 2 3 4)
10
? (funcall (function +) 1 2 3 4)
10
```

FUNCALL- APPLY

```
? (apply #'+ '(1 2 3))
6

? (apply #'+ 1 '(2 3))
6

? (apply #'+ 1 2 '(3))
6

? (apply #'+ 1 2 3 nil)
6

? (apply #'append '((a b) (c d)))
(A B C D)

? (apply #'append '(a b) '((c d)))
(A B C D)
```

FUNCALL- APPLY

```
? (setf una-funzione #'mod)
#<Compiled-function MOD #x689F3E>

? (una-funzione 7 3)
> Error: Undefined function UNA-FUNZIONE called with
arguments (7 3) .

? (funcall #'una-funzione 7 3)
> Error: Undefined function: UNA-FUNZIONE .

? (funcall una-funzione 7 3)
1
```

LAMBDA

E' una special form che viene usata per definire funzione (temporanea) senza legare ad essa un nome. Ritorna una "lexical closure" (una funzione che, quando invocata, esegue il corpo della esspressione-lambda osservando le regole dello scope lessicale).

```
(LAMBDA lambda-list {forms}*)
```

```
? #'(lambda (x) (+ x 3))
(LAMBDA (X) (+ X 3)
? (funcall * 5)
8
? (mapcar #'(lambda (x) (- (sqrt x))) '(2 3))
(-1.4142135623730951 -1.7320508075688772)
? (funcall #'(lambda (x) (- (sqrt x))) 3)
-1.7320508075688772
```

Con mapcar (o remove-if) + lambda posso creare cicli:

```
(mapcar #'(labmbda (x) (+ x 2)) '(1 2 3 4 5))
(3 4 5 6 7)
? (remove-if #'(lambda (x) (= 0 (mod x 3))) '(3 4 5 6))
(4 5)
```

DOTIMES

Permette di eseguire un insieme di forme un numero predefinito di volte, per `cont` che varia da 0 al valore di `numiter` meno `1`.
Il valore di `forma-risultato` è quello ritornato. Se `forma-risultato` è omesso, DOTIMES restituisce `NIL`

```
(dotimes (cont numiter [forma-risultato])
       {form}*)
```

```
(defun fattoriale (n)
  (let ((res 1))
    (dotimes (x n res)
         (setq res (* (+ 1 x) res)))))
```

DOLIST

```
(DOLIST (par lista forma-risultato) {form}*)
```

Per ogni elemento x di `lista` esegue le `{form}*` con il parametro `par` legato all'elemento x

```
(defun lung (lista)
  (let ((res 0))
    (dolist (x lista res)
      (incf res 1))))

? (lung '(a b c))
3

? (dolist (x '((a b) (c d) (e d))) (car x))
A
C
E
```

DO

E' la special form più generale per eseguire iterazioni. Ritorna il valore dell'ultima delle `result-form`, dopo aver eseguito le `{body-form}*` ripetutamente finchè `end-test` non risulta vero. Le variabili `var` sono inizializzate a `initial-value` (in parallelo) ed ad ogni ripetizione successiva sono aggiornate al valore ritornato da `step`

```
(DO ({(var [initial-value [step]])}*)
      (end-test {result-form}*)
   {body-form}*)


(defun fattoriale (n)
  (do ((x 1 (+ 1 x))
       (res 1))
      ((> x n) res)
    (setq res (* res x))))
```

DO*

`DO*` si comporta esattamente come `DO` ma le variabili sono valutate in sequenza.

```
(defun factor (n)
  (do* ((x 1 (1+ x))
        (res 1 (* res x)))
       ((>= x n) res)))

(defun my-expt (m n)
  (do* ((result 1 (* m result))
        (exponent n (- exponent 1)))
       ((zerop exponent) result)))
```

LOOP

`(LOOP {form}*)`

E' il costrutto più semplice per eseguire iterazioni, esegue il "body" ripetutamente senza fare alcun controllo. Quando viene eseguita una espressione del tipo `(return form)` si esce dal loop ed il valore ritornato è quello di `form`

```
(defun lunghezza (lista)
  (let ((res 0))
    (loop
      (when (endp lista)
        (return res))
      (setq lista (rest lista))
      (incf res))))
```

Da non fare!

Nella funzione precedente abbiamo agito su una variabile legata ad un valore passato come parametro. Era SAFE perchè non abbiamo alterato in maniera permanente il parametro

```
? (defun foo (x)
    (setf (cdr x) '(5)))
FOO
? x
(1 2)
? (foo x)
(5)
? x
(1 5)
```

In questo esempio con la `setf` abbiamo agito in maniera distruttiva sulla lista `x.` Si capirà meglio questo esempio quando vedremo come il LISP gestisce la memoria

PROGN - PROG1

`(PROGN {form}*)` - esegue le form e ritorna il valore dell'ultima
`(PROG1 {form}*)` - esegue le form e ritorna il valore della prima

```
? (let ((l '(a b c)))
    (prog1 (car l) (setq l (cdr l))))
A
? (let ((l '(a b c)))
    (progn (car l) (setq l (cdr l))))
(B C)
```

Il tipico uso di `PROGN` è quando in un particolare costrutto una sola forma viene valutata ma ne vorremmo valutare più d'una

```
? (let ((res 0))
    (dolist (x '(a b) (progn (print 'lunghezza=)
                             (princ res)
                             nil))
      (incf res)))
LUNGHEZZA= 2
NIL
```

Compilazione

Si possono compilare singole funzioni

`(COMPILE nome-funzione)`

o compilare un file

`(COMPILE-FILE nomefile)`

Il compilato va poi ricaricato con `(LOAD nomefile-compilato)`

? **`(compile-file “home/users/pippo/lisp/lezione.lisp”)`**

? **`(load “home/users/pippo/lisp/lezione.fasl”)`**

Il suffisso del file compilato varia da compilatore a compilatore. Se si omette il suffisso solitamente il Lisp carica il file compilato (se esiste).

Stampa

(PRINT `x &optional (os *standard-output*)` - stampa `x` e uno spazio (su `os`) dopo aver iniziato una nuova linea e ritorna `x`

(PRINC `x &optional (os *standard-output*)` - ritorna `x` dopo averlo stampato (su `os`). Omette i doppi apici delle stringhe ed i caratteri di controllo

(PRIN1 `x &optional (os *standard-output*)` - ritorna `x` dopo averlo stampato (su `os`).

(PPRINT `x &optional (os *standard-output*)` - non ritorna nulla, setta temporaneamente `*print-pretty*` a `T` e stampa una nuova linea ed `x`

```
? (progn (print "a") (prin1 'b) (princ "c") (print '(a
b c)))

"a" Bc
(A B C)
(A B C)
? (pprint "Hasta Luego")

"Hasta Luego"
```

Stampa

`(FORMAT dest format-string &rest args)` - ritorna NIL dopo aver mandato args su dest, formattati secondo le indicazioni di `format-string`. Se `dest` è `NIL` allora ritorna gli `args` formattti come una stringa.

```
? (let ((x '("abc" "def")))
    (format t "1) ~s~%2) ~a~%3) ~{~a ~}~%" x x x))
1) ("abc" "def")
2) (abc def)
3) abc def
NIL
```

`~s` passa l'espressione a `PRIN1`, **`~a`** passa l'espressione a `PRINC`, **`~%`** inizia una nuova linea, **`~{`** inizia una sottoformato iterativo da applicare a tutti gli elementi di una lista e **`~}`** lo termina.

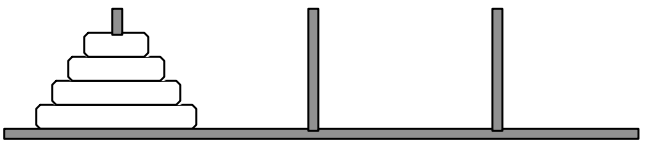
```
? (let ((x "ha male")) (format nil "pippo ~s" x))
"pippo \"ha male\""
? (let ((x "ha male")) (format nil "pippo ~a" x))
"pippo ha male"
? (let ((x "pippo")(y "ha male")) (format nil "~a ~a"
x y))
"pippo ha male"
```

Es. Torre di Hanoi

Vincoli.

– un disco piccolo non può mai stare sotto uno grande
– si può spostare solo un disco alla volta

Obiettivo: spostare tutti i dischi da un piolo ad un altro usando un terzo piolo di servizio



Il problema si risolve per una pila di n dischi se: 1) lo si risolve per una pila di n-1 dischi, 2) si sposta poi l'n-esimo disco e, 3) si rispostano di nuovo gli n-1 dischi.

Hanoi

```
(defun torre (dischi da a serv)
  (if (endp dischi)
    NIl
    (progn (torre (rest dischi) da serv a)
           (muovi (first dischi) da a)
           (torre (rest dischi) serv a da))))

(defun muovi (item da a)
  (format t "~%muovo ~a da ~a a ~a" item da a))

? (torre '(3 2 1) 'P1 'P2 'P3)

muovo 1 da P1 a P2
muovo 2 da P1 a P3
muovo 1 da P2 a P3
muovo 3 da P1 a P2
muovo 1 da P3 a P1
muovo 2 da P3 a P2
muovo 1 da P1 a P2
NIL
```

I/O

- Per leggere e scrivere il Lisp usa gli STREAM.
- Lo standard output e lo standard input sono stream già definiti
- Per leggere e scrivere da e su un file lo si deve aprire come uno stream
- WITH-OPEN-FILE è una special form che permette di creare uno stream e connettere un file allo stream

```
(WITH-OPEN-FILE (var pathname :direction
                 {:input | :output}) {form}*)
```

I/O (esempio)

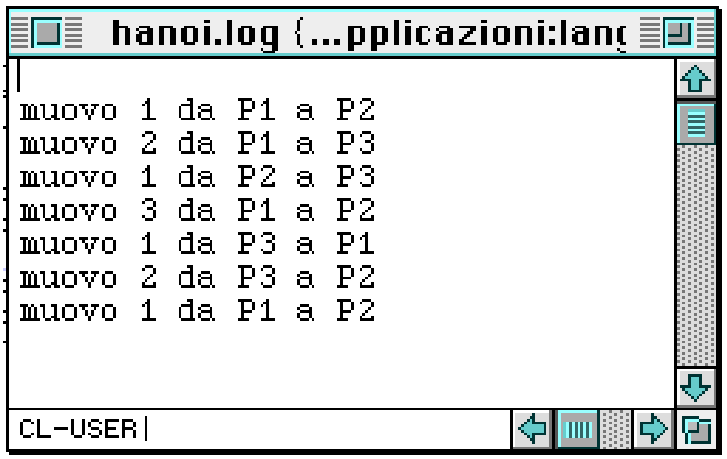
```
(defun hanoi-su-file (dischi da a serv file)
  (with-open-file (ofile "hanoi.log" :direction
                         :output)
    (torre-su-file dischi da a serv ofile)))

(defun torre-su-file (dischi da a serv stream)
  (if (endp dischi)
    NIl
    (progn (torre-su-file (rest dischi) da serv a
                                          stream)
           (muovi-su-file (first dischi) da a stream)
           (torre-su-file (rest dischi) serv a da
                                          stream))))

(defun muovi-su-file (item da a &optional (stream t))
  (format stream "~%muovo ~a da ~a a ~a" item da a))
```

IO

```
? (torre-file '(3 2 1) 'P1 'P2 'P3 "hanoi.log")
NIL
```



READ

```
(READ &optional (is *standard-input*)
                (eof-error-p T)
                eof-value
                (recursive-p NIL))
```

Ritorna la successiva espressione letta su `is`. Se `is` è `NIL`, l'input è preso da `*standard-input*`. Se `is` è `T` l'input è preso da `*terminal-io*`. Altrimenti `is` deve essere uno stream aperto in lettura (con `with-open-file`). Quando lo stream non contiene più oggetti viene sollevato un errore se `eof-error-p` è `T`. Il valore ritornato come end-of-file è `eof-value` (se `eof-error-p` è `NIL`).

```
(READ-LINE &optional (is *standard-input*)
                     (eof-error-p T)
                     eof-value
                     (recursive-p NIL))
```

Ritorna due valori: una stringa contenente la successiva linea da `is`, senza il carattere di ritorno a capo; e `T` se e solo se la linea terminava in `is` con un `end-of-file`

READ (cont)

`*terminal-io*`, `*standard-input*` e `*standard-output*` sono variabili globali. Si sconsiglia di non cambiare il valore di `*terminal-io*`.

```
(defun lfac ()
  (princ "immetti un numero: ")
  (format t "il suo fattoriale vale ~d" (factor
(read))))

? (lfac)
immetti un numero: 7
il suo fattoriale vale 5040
NIL

? (read) (+ 5 6)
(+ 5 6)
? (read-line) (* (+ 5 6)
"(* (+ 5 6)"
NIL
```

Esempio - IO

```
(defun map-feature (feature fn-transf in-file out-file)
  (with-open-file (ifile in-file :direction :input)
     (with-open-file (ofile out-file :direction :output)
       (let (line lst lstout counter llst)
       (setq counter 1)
       (loop
        (unless (setq line (read-line ifile nil nil))
          (return nil))
        (setq lst (read-from-string
                    (concatenate 'string "(" line ")")))
        (setq lstout nil)
        (dolist (x lst)
          (cond ((= feature counter)
                  (push (funcall fn-transf x) lstout))
                (t (push  x lstout)))
          (incf counter))
        (setq counter 1)
        (format ofile "~{~S ~}~%" (reverse lstout)))))))
```

Trasforma l'elemento in posizione "feature" di ogni riga in in-file scrivendola in out-file.

Stringhe

Le stringhe insieme alle liste fanno parte di un tipo di dato più generale chiamato **sequence**

```
? (length "a b c")
5
? (length '(a b c))
3

(STRING= str1 str2 &key start1 end1 start2 end2)

(STRING-EQUAL str1 str2 &key start1 end1 start2 end2))
```

non è case sensitive

```
? (string= "abc" "fbabd" :start1 0 :end1 3 :start2 2
                                           :end2 5)
NIL
? (string= "abc" "fbabd" :start1 0 :end1 2 :start2 2
                                           :end2 4)
T
```

Stringhe: SEARCH

La funzione SEARCH consente di identificare la posizione di una stringa all'interno di un'altra.

```
? (search "ab" "akgbhs tabft")
8
? (search "ab" "akgbhs tabft" :start2 5)
8
? (search "ab" "akgbhs tabft" :start1 1)
3
```

Property List

- Ogni simbolo può avere numerose proprietà, ovvero dati legati al simbolo
- Il valore di un simbolo è una particolare proprietà
- La collezione di tutte le proprietà di un simbolo è contenuta nella property list

Accessori delle proprietà

(**GET** simbolo proprietà) - é il lettore
(**SETF** (GET simbolo proprietà) valore) - è lo scrittore

```
? (get 'mia-automobile 'tipo)
NIL
? (setf (get 'mia-automobile 'tipo) "Tipo 1600 i.e.")
"Tipo 1600 i.e."
? (get 'mia-automobile 'tipo)
"Tipo 1600 i.e."
? (setf mia-automobile 'Ok)
OK
? (setf (get 'mia-automobile 'valore) 13000)
13000
```

DESCRIBE

```
? (describe 'mia-automobile)
Symbol: MIA-AUTOMOBILE
Non-special Variable
INTERNAL in package: #<Package "COMMON-LISP-USER">
Print name: "MIA-AUTOMOBILE"
Value: OK
Function: #<Unbound>
Plist: (VALORE 13000 TIPO "Tipo 1600 i.e.")
? (setf (symbol-function 'mia-automobile) #'(lambda () (print "Ha i soliti problemini ... Fiat") NIL))
#<Anonymous Function #x8E35BE>
? (mia-automobile)
"Ha i soliti problemini ... Fiat"
NIL
? (describe 'mia-automobile)
Symbol: MIA-AUTOMOBILE
Non-special Variable, Function
INTERNAL in package: #<Package "COMMON-LISP-USER">
Print name: "MIA-AUTOMOBILE"
Value: OK
Function: #<Anonymous Function #x8E35BE>
Arglist: NIL
Plist: (VALORE 13000 TIPO "Tipo 1600 i.e.")
```

Array

(**MAKE-ARRAY** dim &key element-type initial-contents initial-element) - ritorna un array di dimensione dim

(**AREF** array &rest subscript) - ritorna l'elemento in posizione subscript di array

```
? (setf a (make-array '(5) :element-type 'integer))
#(NIL NIL NIL NIL NIL)
? (setf (aref a 0) 7)
7
? a
#(7 NIL NIL NIL NIL)
? (aref a 0)
7
? (setf aa (make-array '(2 2) :element-type 'float
:initial-contents '((1.0 2.2) (0.0 1.5))))
#2a((1.0 2.2) (0.0 1.5))
? (aref aa 0 1)
2.2
```