LISP

Manuale di riferimento

Common Lisp (second edition), G.L. Steele, Digital Press, 1990.
`www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.htm`

Alcuni testi di introduzione al Lisp + programmi didattici:

LISP (second Edition), P. Winston & P. Horn, Addison-Wesley, 1984

Artificial Intelligence Programming (second edition), Charniack, Riesbeck, McDermott, Meehan, Lawrence Erlbaum Associates Publishers (LEA), 1987

Paradigms of Artificial Intelligence Programming: case studies in Common Lisp, P. Norvig, Morgan Kaufmann, 1992

On Lisp, P. Graham, Prentiche Hall, 1994 (AVANZATO)

Object-oriented programming in Common Lisp, S. Keene, Addison-Wesley, 1989

Introduzione

Lisp è stato inventato da John McCarthy nel 1958 (MIT AI Lab Memo N.1)

E' per molti versi il linguaggio dell'IA ma...

Gnu Emacs, il migliore editor in Unix, è in LISP
Autocad, un sw molto diffuso per lo sviluppo di applicazioni CAD è in LISP
Interleaf, un sistema di publishing è in LISP

Lisp significa **Lis**t **P**rocessing, ma Lisp non è solo questo

Il Lisp permette metodologie di sviluppo del software avanzate

Gli attuali compilatori permettono lo sviluppo di programmi efficienti al pari di altri linguaggi

Dati e procedure si esprimono nello stesso modo

CLOS è una estensione del Lisp per lo sviluppo di programmi object-oriented

Interazione con LISP

```
? (+ 5 9)
14                                  <-- valore ritornato

? (setq colori '(blu marrone verde))
(BLU MARRONE VERDE)        <-- fissare variabili globali

? colori
(BLU MARRONE VERDE)     <-- ottenere il valore di variabili

? (setq colori (remove 'blu colori))
(MARRONE VERDE)         <-- modificare il valore di variabili

? (setq colori (cons 'giallo colori))
(GIALLO MARRONE VERDE)

? (defun agg (col lista)
    (cons col lista))
AGG                                 <-- definire nuove funzioni

? (agg 'rosso colori)
(ROSSO GIALLO MARRONE VERDE)
```

Strutture dati

Le strutture dati del Lisp sono chiamate **S-expression**

Number (atom) es. `6, 6.0, 6.0e1 (integer float)`

Simbol - (atom) un carattere seguito eventualmente da altre lettere o numeri. es. `FOO, Y12, 39A`

T - simbol speciale che rappresenta `true`, qualunque S-expression è T

NIL - simbol speciale, rappresenta `false`

Character - (atom) es. #\c

List - es. `(a b), (“il” numero` #\è 10), (), NIL

Cons - celle di cons, es. `(A . B)`

Function - es. `#’+, #’CAR`

String - (atom, sequence) es. `“questo è il mio nome”, “”, “\””, “\\”`

Vector- (sequence) es. #(1 3 4 7), #*101101110, le stringhe sono vettori

Structure; Hash-Table; Stream; Pathname;Keyword

Programmi

Sintassi: ogni S-expression è un programma sintatticamente corretto !

Semantica: le S-expression sono eseguite per mezzo della funzione EVAL. EVAL prende una S-expression e ritorna un'altra S-expression (valore).

1) se l'espressione è un numero, una stringa, un carattere, T, o NIL allora il valore è l'espressione stessa

2) se l'espressione è una lista del tipo `(function arg1 ... argn)` allora il valore è ottenuto valutando gli argomenti, da `arg1` a `argn` e chiamando la funzione con questi valori

3) se l'espressione è una lista del tipo `(reserved-word arg1 ... argn)` allora il valore è ottenuto dipende completamente dalla definizione di `reserved-word` (macro e special-form)

4) se l'espressione è un simbolo il valore è l'ultimo valore associato al simbolo (temporalmente)

LOOP READ-EVAL-PRINT

Top-level dell'inteprete LISP (in linguaggio tipo ALGOL)

```
BEGIN
LOOP: exp := READ(input);
      val := EVAL(exp);
      PRINT(val,output);
      GOTO LOOP
END
```

- `EVAL` applica le regole di valutazione di una S-expression
- `input` e `output` sono oggetti "streams"
- `exp` è una S-expression

QUOTE

Una lista viene valutata dal lisp applicando la funzione o la parola-riservata posta come primo elemento della lista al resto della lista (argomenti), dopo che questo è stato "valutato"

```
? (+ 5 6)
11
```

Come possiamo quindi passare come argomento una lista ad una funzione che opera sulle liste?

```
? (first (a b))
> Error: Attempt to call 'A'which is an undefined
function

? (first (quote (a b)))
A
? (first '(a b))
A
```

(QUOTE expression) ==> expression

SETQ

E' il costrutto di base con cui si fanno assegnamenti di valore a variabili in LISP.

(**setq** var1 form1 var2 form2 ...) => assegna a var1 il valore di form1, a var2 il valore di form2, ecc.

Ritorna il valore assegnato al suo ultimo argomento(var)

```
? (setq x (+ 3 2 1))
6
? (setq y 10 z (+ 20 10))
30
? (setq w (+ x y z)
46
```

Funzioni su liste

(**FIRST** list) ==> il primo elemento della lista list

(**REST** list) ==> la lista ottenuta da list eliminando il primo elemento della lista

(**LAST** list) ==> la lista formata dall'ultimo elemento di list

(**APPEND** &rest llist) ==> la lista ottenuta concatenando le list in llist

(**LIST** &rest args) ==> la lista i cui elementi sono in args

(**CONS** x y) ==> una lista il cui primo elemento è x ed il cui rest è y

? **(cons 'a ' b)** Se il secondo elemento non è una lista
(A . B) <--- una cella di cons

Le celle di cons sono le strutture fondamentali del LISP

Funzioni su liste

(**CAR** list) - è un altro nome per FIRST

(**CDR** list) - è un altro nome per REST

(**BUTLAST** list) - la lista ottenuta rimuovendo l'ultimo elemento

(**NTHCDR** n list) - applica la funzione CDR n volte

(**NTH** n list) ==> l' n-esimo elemento della lista list

Liste di associazione

Sono espressioni del tipo `((key1 val1) (key2 val2) ... (keyn valn))`

(**ASSOC** key association-list &key (test #'eql)) ==> ritorna la prima coppia la cui chiave è "uguale" a key, altrimenti NIL

```
? (assoc 'a  '((a b) (c d)))
(A B)
? (assoc 'a  '((a c) (a b) (c d)))
(A C)

? (rest (assoc 'a '((a . b) (c . d))))
B
```

(**ACONS** key datum a-list) ==> costruisce una nuova lista di associazione aggiungendo (key . datum) alla vecchia a-list

(acons x y a) = (cons (cons x y) a)

Numeri

integer - ration - float (single-float, double-float) - numeri complessi

La divisione di due **float** è un **float**
La divisione di due **integer** è un **ratio**
La divisione di due **ratio** è un **ratio**

```
? (/ 3.0 5)
0.6
? (/ 3 7)
3/7
? (/ 3/7 3/5)
5/7
```

(**FLOAT** number) converte in un float
(**ROUND** ratio-or-float) l'intero che approssima meglio ratio-or-float

```
? (round pi)
3
0.14159265358979312
```

ATTENZIONE RITORNA DUE VALORI

#c(1.3e-15 0.75) nu. complesso con parte reale 1.3e-15 e parte imm. 0.75

Funzioni

Le funzioni d'utente si possono definire con la forma speciale `DEFUN`

```
(DEFUN nomefunzione (arg1 ... argn)
      form1 ... formk)
```

`DEFUN` ritorna `NOMEFUNZIONE` dopo aver definito una funzione di nome `nomefunzione.`

Quando nomefunzione è invocata le variabili arg1 ... argn vengono legate ai valori che gli viengono passati come argomenti.

form1 ... formk sono poi valutati. Il valore ritornato è quello di formk.

```
? (defun areatriangolo (b a)
    (/ (* b a) 2))
AREATRIANGOLO
? (areatriangolo 1.0 3.0)
1.50
```

FUNZIONI

```
? (defun funzioneprova (x)
   (setq x (* x 2))
   (setq x (+ x 10)
   (+ x x))
FUNZIONEPROVA
? (funzioneprova 10)
60
```

SCOPING LESSICALE

I legami (binding) delle variabili, parametri di una procedura, stabiliti da una chiamata funzionale hanno (quasi sempre) uno **SCOPING lessicale**

Quando il Lisp ha valutato una funzione non rimane alcuna traccia dei legami precedentemente stabiliti

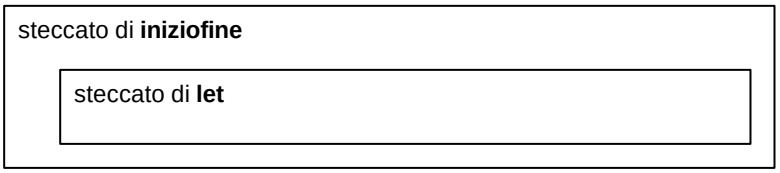
```
? (areatriangolo 1.0 3.0)
1.50
? a                          <-- un parametro usato nella def di areatriangolo
Error: Attempt to take the value of an unbound variable
'A'

?(defun f1 (x)  (setq x  (* x 2)))
F1
? (defun f2 (x) (f1 x) (setq (+ 1 x)))
F2
? (f2 10)
11
```

LET

Serve a definire variabili locali ad una procedura ed a creare uno "steccato" all'interno del quali la variabile è visibile

```
? (defun iniziofine (lista)
    (let ((primo (first lista))
          (ultimo (last lista)))
      (cons primo ultimo)))
INIZIOFINE
? (iniziofine '(a b c d))
(A D)
```



LET

```
(let ((var1   val-iniziale1)
      (var2  val-iniziale2)
       ...
      (varn   val-inizialen))
  form1
  form2
    ...
  formk)
```

LET valuta le espressioni che ritornano i valori iniziali dei prametri in parallelo

```
? (let ((a 3) (b 5)) (+ a b 2))
10
? (let ((x 1)
        (y (+ x 1)))
    y)
ERROR: Attempt to take the value of the unbound symbol X
```

LET*

LET* ha lo stesso formato di LET

LET* valuta le espressioni che ritornano i valori iniziali dei parametri in sequenza

```
? (let* ((x 10)
         (y (* x x)))
    y)
100
```

```
(let* ((x a)    è equivalente a  (let ((x a))
       (y b))                     (let ((y b))
  ...)                              ...))

(let ((a nil) (b nil) ...)  = (let (a b) ...)

(let* ((a nil) (b nil) ...) = (let* (a b) ...)
```

SCOPING DINAMICO

Esistono variabili che hanno uno SCOPING DINAMICO (*variabili speciali*) il cui valore è sempre l'ultimo assegnato (e visibile). Per convenzione le var speciali hanno nomi che iniziano e finiscono con *

Una variabile speciale può essere dichiarata con il costrutto **`defvar`**

```
? (defvar *var1* 10)
*VAR1*
? (defun check-special-var ()
       (let  ((localvar (+ *var1* 1)))
           (setq *var1* 1000)
           localvar))
CHECK-SPECIAL-VAR
? (check-special-var)
11
? *var1*
1000
```

Interazione tra variabili locali e globali

Cosa succede se si definisce una variabile locale con lo stesso nome di una variabile "globale" precedentemente definita ?

```
? (setq m 3)
3
? (let ((m 5))
    (/ m 2))
5/2
? m
3

? (setq x 7)
7
? (let ((x 1)
        (y (+ x 1)))
    y)
8
```

LET effettua i legami (binding) parallelamente.

VARIABILI STATICHE E DINAMICHE

Visibilita' (scope) lessicale: una variabile lessicale (normale, statica) puo` essere riferita solamente da "forme" poste testualmente all'interno del costrutto che lega (bind) la variabile.

Visibilita' (scope) dinamico: una variabile speciale (dinamica) puo` essere riferita con il valore a cui e` legata da un certo costrutto solamente durante la valutazione di tale costrutto.

VARIABILI STATICHE E DINAMICHE

```
? (setq normale 5)
5
? (defun check-normale () normale)
CHECK-NORMALE
? (check-normale)
5
? (let ((normale 6)) (check-normale))
5

? (defvar *speciale* 5)
*SPECIALE*
? (defun check-speciale () *speciale*)
CHECK-SPECIALE
? (check-speciale)
5
? (let ((*speciale* 6)) (check-speciale))
6
? *speciale*
5
```

Predicati: uguaglianze

Sono procedure che ritornano un valore di verità. In LISP tutte le espressioni hanno un valore di verità VERO (T) eccetto NIL.

```
? (TRUE '(> 1 2))
T
```

(**=** x y) - controlla se gli argomenti sono numeri uguali anche se non dello stesso tipo.

(**EQ** x y) - controlla se gli argomenti sono esattamente gli stessi oggetti nella memoria.

(**EQL** x y) - prima controlla se gli argomenti sono EQ. In caso contrario controlla se sono numeri dello stesso tipo e valore.

(**EQUAL** x y) - prima controlla se i due argomenti sono EQL. In caso contrario, se gli argomenti sono oggetti con componenti o liste, controlla se i loro elementi sono EQUAL. Stringhe e vettori di bit sono controllati elemento per elemento.

Esempi

```
? (equal '(a b) '(a b))
T
? (eq '(a b) '(a b))
NIL
? (eq (cons 'a 'b) (cons 'a 'b))
NIL
? (eql 7 (let ((x 7)) x))
T
? (eql 8/2 4.0)
NIL
? (= 8/2 4.0)
T
? (eq "pippo" "pippo")
NIL
? (equal "pippo" "pippo")
T
? (setq x '(a b))
(A B)
? (setq y x)
(A B)
? (eq y x)
T
```

Esempi

```
? (eq 1 1)
dipende dalla particolare implementazione del LISP

? (eq 'a 'a)
T
stessi simboli vengono automaticamente memorizzati come
oggetti uguali

? (eq 1 1.0)
NIL
? (equal '(a (b c) ((d))) '(a (b c) ((d))))
T
? (equal '(a (b c) ((d))) '(a (c b) ((d))))
NIL
```

MEMBER

(**MEMBER** item lista) - se item è un elemento di lista allora ritorna la sottolista di lista che inizia da item altrimenti ritorna NIL

```
? (member 'b '(a b c b))
(B C B)
? (member '(a b) '((a b)))
NIL
```

Nel controllare l'uguaglianza MEMBER usa EQL. Questo comportamento può essere cambiato usando le keywords

```
? (member '(a b) '((a b)) :test #'equal)
((A B))
```

Le keywords consentono di specificare parametri opzionali indipendentemente dalla loro posizione (ordine) nella definizione della funzione.

FUNCTION

Ogni simbolo può avere un valore ed un valore funzionale; viene ritornato dalla special form `FUNCTION`

```
? (function car)
#<Compiled-function CAR #x590B76>

? #'car
#<Compiled-function CAR #x590B76>

? (symbol-function 'car)
#<Compiled-function CAR #x590B76>
```

I valori funzionali possono essere trattati come valori normali: le funzioni possono ritornare valori funzionali e i simboli possono avere anche come valore un valore funzionale

```
? (setq num-equal #'=)
#<Compiled-function = #x68AFBE>
? (member 2 '(1 2.0 3) :test num-equal)
(2.0 3)
```

FUNCTION

Il modo usuale per stabilire un legame funzionale è quello di usare `DEFUN`

Si può anche definire esplicitamente il valore funzionale di un simbolo. Questo valore viene ritornato da `symbol-function`

```
? (defun num-eql (x y)
    (and (numberp x) (numberp y) (eql x y)))
NUM-EQL
? (setf (symbol-function 'num-eql1) #'num-eql)
#<Compiled-function NUM-EQL #x7ADF7E>
? (num-eql1 5 5)
T
?(num-eql 5 5)
T
```

`Formato SETF:`

(setf <locazione-memoria> <valore>)

Predicati su tipi di dati

Servono a verificare se un oggetto appartiene o no ad un particolare tipo di dato

`(ATOM x)` - è un atomo ?

`(NUMBERP x)` - è un numero?

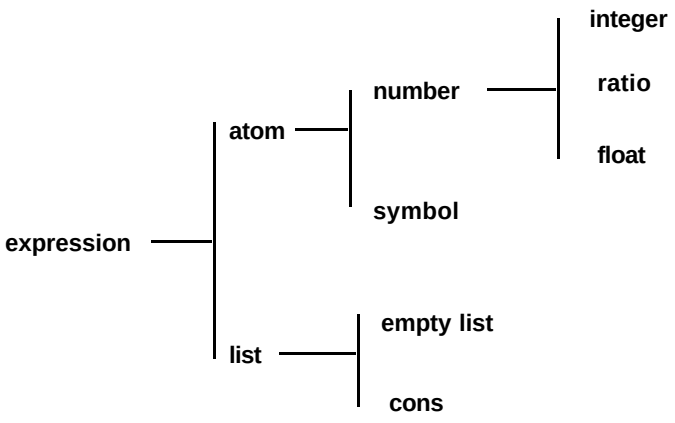
`(SYMBOLP x)` - è un simbolo?

`(LISTP x)` - è una lista?

`(NULL x)` - è una lista vuota?

`(ENDP x)` - è una lista vuota? Se non è una lista da errore

Tipi di dati



Predicati su numeri

(**NUMBERP** x)- è un numero?

(**ZEROP** x) - è lo zero?

(**PLUSP** x) - è positivo?

(**MINUSP** x) - è negativo?

(**EVENP** x) - è pari?

(**ODDP** x) - è dispari?

(**>** n &rest numeri) - ritorna T s sono in ordine decrescente

(**<** n &rest numeri) - ritorna T s sono in ordine crescente

=, =/, >=, <=

AND - OR - NOT

(**AND** {forma}*) - gli argomenti sono valutati da sinistra a destra. Se uno degli argomenti viene valutato NIL, allora AND ritorna NIL. Se nessuno viene valutato NIL allora AND ritorna il valore dell'ultima forma.

(**OR** {forma}*) - gli argomenti sono valutati da sinistra a destra. Non appena uno degli argomenti viene valutato a qualcosa non NIL allora questo valore viene ritornato da OR. In caso contrario OR ritorna NIL.

```
? (and (integerp 7)  (null NIL))
T
? (and (> 7 6) 5)
5
? (or (> 5 6) (+ 10.0 3) (< 7 10))
13.0
```

(NOT oggetto) - ritorna NIL se oggetto è valutato *non* NIL e viceversa

IF - WHEN - UNLESS

(**IF** test then-form [else-form]) - valuta `test`; se questo ritorna non `NIL` allora valuta `then-form` altrimenti valuta `else-form`

(**WHEN** test {form}*) - valuta `test`; se questo ritorna non `NIL` allora valuta le forme {form}*

(**UNLESS** test {form}*) - valuta `test`; se questo ritorna `NIL` allora valuta le forme {form}*

```
? (defun my-abs (x)
    (when (numberp x)
      (if (>= x 0)
        x
        (- x))))
MY-ABS
? (defun my-if (test then else)
    (or (and test then) else))
MY-IF
? (my-if (> 2 5) 'si 'no)
NO
```

COND

(**COND** {(test {form}*)}*) - ritorna il valore dell'ultima forma della prima lista il cui `test` è non `NIL`

```
? (defun come-va? (temp)
    (cond ((> temp 30) 'caldo)      ; 30 < temp
          ((> temp 20) 'temperato)  ; 20 < temp <= 30
          ((> temp 0) 'freddo)      ;  0 < temp <= 20
          (t 'gelido)))             ;       temp < 0
COME-VA?
?(come-va? 7)
FREDDO

? (defun my-if (test then else)
    (cond (test then)
          (T else)))
MY-IF
? (my-if (> 5 4) 'ok (error "my-if"))
> Error: my-if
```

EQUAL e EQL

```
(defun EQL(x y)
   (cond ((eq x y)      ; sono EQ ?
          t)            ; allora sono anche EQL
         ((and (integerp x) (integerp y)) ;sono interi?
          (= x y))
         ( ... )        ; controlli per altri tipi di numeri
      ((and (characterp x) (characterp y))  ; sono caratteri?
       (char= x y)         ; confronto di caratteri
      ((t nil)))

(defun EQUAL (x y)
   (cond ((eql x y)      ; sono EQL ?
          t)             ; allora sono anche EQL
         ((and (consp x) (consp y)) ;sono cons?
          (and (equal (car x) (car y))   ;hanno CAR EQUAL?
               (equal (cdr x) (cdr y))    ;hanno CDR EQUAL?
          ((and (stringp x) (stringp y))  ; sono stringhe?
        (string= x y))          ; confronto di stringhe
          ( ... )))        ; controlli per altri tipi di numeri
```

CASE

(**CASE** keyform {({({key}*)|key}{form}*)}*) - ritorna il valore dell'ultima `form` della prima lista che ha o `key` uguale a `keyform` o una delle {key}* uguale a `keyform` o `T` o `OTHERWISE`. Valuta le forme da sinistra a destra, {key}* e `key` non sono mai valutate.

```
? (defun my-test (x)
    (case x (a 1) ((b c) 2) (otherwise 0)))
MY-TEST
? (my-test 'a)
1
? (my-test 'b)
2
? (my-test 'c)
2
? (my-test 'd)
0
```