Linguaggi di Programmazione 2022-2023
Lisp e programmazione funzionale III

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Indice

- Operatore **QUOTE**
- Espressioni autovalutanti
- Tipi di dati (Common) Lisp
  - Equivalenza dati e programmi
- Valutazione di funzioni ricorsive
  - Esempi
  - Ricorsione su liste: *coda*, *testa*, *testa-coda*

Dati simbolici e operazione `quote`

- Tutti gli esempi che abbiamo visto finora hanno utilizzato dei numeri (soprattutto interi) o stringhe

- Il LISP ci permette anche di manipolare dati simbolici, ovvero ci permette di costruire delle liste come

```
(a b c d e f g h I)
((gino 10) (ugo 10) (maria 2) (ettore 20))
```

od anche (*se ancora non l'avete notato*)

```
(+ quaranta 2)
(defun media (x y) (/ (+ x y) 2.0))
```

Dati simbolici e operazione `quote`

- Come possiamo costruire una lista che contiene i due simboli **`A`** e **`B`**?

```
prompt> (list A B)
Errore!  A non ha un valore associato.
```

- Date le regole di valutazione degli argomenti dell'espressione **`(list A B)`**, il sistema LISP cerca di trovare il valore associato all'identificatore **`A`** (nell'ambiente globale), non lo trova e segnala un errore

- Abbiamo bisogno di un operatore che dica al LISP di non procedere alla valutazione di una (sotto)espressione

- L'operatore in questione è **`quote`**

Dati simbolici e operazione `quote`

- **`quote`** è un altro operatore speciale (come **`cond`**, **`if`** etc.) la cui sintassi è

  `(quote <e>)`

- L'espressione *<e>* non viene valutata e viene ritornata letteralmente
- Esempi

```
prompt> (quote 42)
42

prompt> (quote A)
A

prompt> (quote (1 B 3))
(1 B 3)

prompt> (quote (1 2 (3 4) (5 6 "foo")))
(1 2 (3 4) (5 6 "foo"))
```

Dati simbolici e operazione `quote`

- **`quote`** è talmente importante che il LISP fornisce una comoda abbreviazione che utilizza un singolo carattere

  `'<e>` equivale a **`(quote <e>)`**

- Esempi

```
prompt> '42 ; Notate che il quote è ridondante.
42

prompt> 'A
A

prompt> '(1 B 3)
(1 B 3)

prompt> '(1 2 (3 4) (5 6 "foo"))
(1 2 (3 4) (5 6 "foo"))
```

Dati simbolici e operazione `quote`

- L'operatore **`quote`** non sembra agire su numeri e stringhe
- Ciò accade perchè numeri e stringhe sono *autovalutanti*, ovvero essi *sono espressioni il cui valore ha la stessa rappresentazione tipografica dell'espressione tipografica che li denota*
- I simboli (o identificatori) denotano invece I valori che sono loro associati (ad esempio tramite **`defparameter`**)
- Le liste, se non quotate, rappresentano invece delle espressioni da valutare, ergo la necessità di avere l'operatore di **`quote`**
- Esempi

```
prompt> 42
42

prompt> '42
42

prompt> '"io sono una stringa"
"io sono una stringa"

prompt> "io sono una stringa"
"io sono una stringa"
```

Dati simbolici e operazione `quote`

- Tutto ciò ha diverse importanti conseguenze
- Esempi

```
prompt> (list 'A 'B)
(A B)

prompt> '(A B)
(A B)

prompt> '(defun media (x y) (/ (+ x y) 2.0))
(DEFUN MEDIA (X Y) (/ (+ X Y) 2.0))

prompt> (defun media (x y) (/ (+ x y) 2.0))
MEDIA

prompt> (defparameter m
            '(defun media (x y) (/ (+ x y) 2.0))
M

prompt> (third m) ; (car (cdr (cdr m)))
(X Y)
```

Dati simbolici e operazione `quote`

Conseguenza numero 1

Il programma Hello World!, in LISP è cortissimo

```
prompt> "Hello world!"
"Hello world!"
```

Conseguenza numero 2 !!!!!

In LISP *i programmi ed i dati sono esattamente la stessa cosa!*

Simboli, numeri, liste e “atomi”

- In LISP abbiamo una ripartizione degli oggetti principali in due categorie: *atomi* e *cons-cells*

- In prima battuta gli atomi sono simboli e numeri (ma non solo; ad esempio le stringhe)
- I non-atomi sono le *cons-cells* (quindi le liste)

- Esiste un predicato che controlla se il suo argomento è un atomo o meno: **`atom`**

```
prompt> (atom 3)
T

prompt> (atom 'un-simbolo)
T

prompt> (atom "una stringa")
T

prompt> (atom (cons -42 42))
NIL

prompt> (atom (list 1 2 3))
NIL
```

Simboli e liste (e altri elementi), ovvero le Symbolic Expressions

- Dunque in Lisp abbiamo
  - Numeri
  - Simboli
  - Stringhe (notare la differenza)
  - Cons-cells
    - quindi liste
  - Altri oggetti di base

- Le cons-cell (che abbiamo visto) più numeri, simboli e stringhe (ma non solo) costituiscono le

  Symbolic Expressions

  dette anche

  Sexp's

Valutazione di espressioni e funzioni ricorsive: dettagli e funzione `eval`

- Dato che programmi e sexp's in Lisp sono equivalenti, possiamo dare le seguenti regole di valutazione (ed implementarle nella funzione **`eval`**!)
- Data una sexp
  - Se è un atomo (ovvero, se non è una cons-cell)
    - Se è un numero ritorna il suo valore
    - Se è una stringa ritornala così com'è
    - Se è un simbolo
      - Estrai il suo valore dall'*ambiente corrente* e ritornalo
      - Se non esiste un valore associato allora segnala un errore
  - Se è una cons-cell $(O\ A_1\ A_2\ \ldots\ A_n)$ allora si procede nel seguente modo
    - Se O è un operatore speciale, allora la lista $(O\ A_1\ A_2\ \ldots\ A_n)$ viene valutata in modo speciale
    - Se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata (**`apply`**) alla lista $(VA_1\ VA_2\ ..\ VA_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1, A_2, \ldots, A_n$.
    - Se O è una Lambda Expression la si applica alla lista che $(VA_1\ VA_2\ ..\ VA_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1, A_2, \ldots, A_n$
    - Altrimenti si segnala un errore

Valutazione di funzioni ricorsive: esempi

- Fattoriale

```
(defun fatt (n)
      (if (zerop n) 1 (* n (fatt (- n 1)))))
```

- Al prompt:

```
prompt> (fatt 3)
```

n diventa associato a 3 (“*bound to*” 3):

```
(eval '(fatt 3))
→ (eval '(if (zerop 3) 1 (* 3 (fatt (- 3 1)))))
      … → (eval '(* 3 (fatt 2))))
                … → (eval '(*3 (* 2 (fatt 1))))
                … → (* 3 (* 2 (* 1 (fatt 0))))
      … → (* 3 (* 2 (* 1 1)))
… → … 6
```

- **eval** applica la funzione al suo argomento, definendo i legami di N “in cascata” e costruendo una espressione che alla fine verra’ valutata, a partire dalla sotto-espressione piu’ annidata

Valutazione di funzioni ricorsive: esempi con liste

- La struttura ricorsiva delle liste si presta molto bene alla programmazione ricorsiva

- Metodo:
  - scrivere il valore della funzione nel caso banale (usualmente la lista vuota)
  - ridursi ricorsivamente al caso base operando su un argomento ridotto

Valutazione di funzioni ricorsive: esempi con liste

- Specificare la funzione
  - sommatoria degli elementi di una lista (si assume che gli atomi siano numeri interi)
  - `Sum: List → Integer`

- Specificare la soluzione in modo ricorsivo
  - se la lista è vuota: Lista = () ⇒ 0
  - altrimenti: Lista = L ⇒ first(L) + sum(rest(L))

- Implementazione

```
(defun sum (l)
     (if (null l)
         0
         (+ (first l) (sum (rest l)))))
```

Valutazione di funzioni ricorsive: esempi con liste

- Esempio: lunghezza di una lista (al primo livello)

```
(lung '(a b c))          ⇒ 3
(lung '(((a) b) c)       ⇒ 2
(lung nil)               ⇒ 0
```

- Se la lista è vuota allora la lunghezza è 0, altrimenti è 1 + la lunghezza del resto della lista

```
(defun lung (l)
   (if (null l)
       0
       (+ 1 (lung (rest l)))))
```

Valutazione di funzioni ricorsive: esempi con liste

- **`last`**: funzione che ritorna l'ultimo elemento di una lista (la funzione **`last`** è predefinita il Common Lisp, ma con una semantica leggermente diversa)

```
(defun last-l (l)
     (cond ((null l) nil)
           ((atom l)          ; Notare questo caso.
            (error "L'argomento non e` una lista"))
           ((null (rest l)) (first l))
           (t (last-l (rest l)))))
```

Valutazione di funzioni ricorsive: ricorsioni semplici e doppie

- Ricorsione semplice ("ricorsione `cdr`"): la ricorsione e' sempre definita sul resto di una lista
- Non è sempre sufficiente.
- Esempio: contare gli atomi di una sexp

```
(count-atoms '((a b c) 1 (xyz d))) → 6
```

Valutazione di funzioni ricorsive: `count-atoms`

- Base:
  `()` → `0`,
  atomo → `1`
  due casi, perché *in questo caso* sexp può essere sia lista sia atomo

- Ipotesi Ricorsiva:
  `(count-atoms(rest L))` → numero atomi di **`rest`** di **`L`**

- Passo:
  se **`(first L)`** è un atomo →
  `(+ 1 (count-atoms(rest L))`
  altrimenti...????

Valutazione di funzioni ricorsive: `conta-atomi`

- Ricorsione "`car-cdr`" ("doppia"): ricorsione anche sul `car` (`first`) di una lista

- **`(first L)`** e **`(rest L)`** sono sottostrutture di **`L`**: possiamo usare l'Ipotesi Ricorsiva:

  - **`(count-atoms(first L))`** e **`(count-atoms(rest L))`** contano correttamente i loro atomi

  - Passo:

```
(+ (count-atoms(first L)) (count-atoms(rest L)))
```

Valutazione di funzioni ricorsive: `count-atoms`

- Composizione: ricorsione anche sul `car` (`first`) di una lista

```
(defun count-atoms(x) ; 'x' e` una sexp.
   (cond ((null x) 0)
         ((atom x) 1)
         (T (+ (count-atoms (first x))
               (count-atoms (rest x))))))
```

Ancora valutazione di funzioni ricorsive: `profondità`

- Profondità massima di annidamento di una Sexp (n. max parentesi “sospese”):
  - atomo → `0`, **`() → 1`**, **`(a (b ((c)) (d))) → 4`**

1. Base: ovvia (caso banale atomo o lista vuota)
2. Ip. Ric: **`(prof (first x))`**, **`(prof (rest x))`**
3. Passo:
   - `prof` di `x` e’ max fra

     `(prof (first x)) + 1`

     e

     `(prof (rest x))`

Valutazione di funzioni ricorsive: profondità

- Composizione

```
(defun prof (x)
     (cond ((null x) 1)
           ((atom x) 0)
           (t (max (+ 1 (prof (first x)))
                   (prof (rest x))))))
```

- La funzione **max** è predefinita in (Common) Lisp

Valutazione di funzioni ricorsive: `flatten`

- Appiattire una lista: **`flatten`**

```
(flatten '((((a (b (c d)) e) f)) → (a b c d e f)
(flatten 'a)                     → (a) ; Perche`?
```

- Base:
  - lista vuota, non cambia
  - Atomo → `(list` Atomo`)`

- Ip. Ric.:

```
(flatten (first x)), (flatten (rest x))
```

- Passo:

`append` di **`(flatten (first x))`**
con **`(flatten (rest x))`**

Valutazione di funzioni ricorsive: `flatten`

- Appiattire una lista: **`flatten`**

```
(defun flatten (x)                      ; x e` s-espressione
     (cond ((null x) x)                 ; NB: (atom nil) = T
           ((atom x) (list x))          ; serve per  usare
                                        ; append
           (T (append (flatten (first x))
                      (flatten (rest x))))))
```

Valutazione di funzioni ricorsive: `mirror`

- Immagine speculare di una sexp

```
(mirror '(a (b (c d)) e) f)) → (f (e ((d c) b) a)
```

- Base:
  - lista vuota o atomo, non cambia

- Ip. Ric.: `L = (e1 … en)`:

```
(mirror (first L)), (mirror (rest L))
```

- Passo:
  `append` di
  - `(mirror (rest l))` con
  - `(list (mirror (first x)))`

Valutazione di funzioni ricorsive: `mirror`

- Immagine speculare di una sexp

```
(defun mirror (x) ; x e` una sexp
     (if (atom x)
         x
         (append (mirror (rest x))
                 (list (mirror (first x))))))
```

- NB: Senza **`list`**, **`append`** toglierebbe una parentesi al **`mirror`** del **`first`** se questo è una lista; darebbe errore di parametro se fosse un atomo

Valutazione di funzioni ricorsive: `inverti`

- Inversione di una lista

1. Data `L = (e1 … en)`, `(inverti L)` → `(en … e1)`
2. `(inverti NIL)` → `NIL`
3. `(inverti (rest L))` → `(en … e2)`
4. `e1` va posto in coda a `(en … e2)`

- Supponiamo che ci sia funzione `cons-end` che faccia quest'ultima operazione
  - `(cons-end S L)` → `(e1 … en S)`
    - Poi definiremo `cons-end` (procediamo top-down)

Valutazione di funzioni ricorsive: `inverti`

- Inversione di una lista

```
(defun inverti (x)
    (cond ((null x) x)
          ((atom x) x) ; anche Sexp...
          (T (cons-end (first x)
                       (inverti (rest x))))))
```

Valutazione di funzioni ricorsive: `circulate`

- Scrivere la funzione **`circulate`** in modo che operi come segue al primo livello

```
prompt> (circulate '(1 2 3 4) 'left)
(2 3 4 1)

prompt> (circulate '(1 2 3 4) 'right)
(4 1 2 3)
```

Valutazione di funzioni: `circulate`

- La funzione **`circulate`**

```
(defun circulate (lst direction)
     (cond ((atom lst) lst)
           ((null lst) nil)
           ((eq direction 'left)
            (append (cdr lst) (list (car lst))))
           ((eq direction 'right)
            (cons (last-l lst) (but-last lst)))
           (T lst) ))
```

Sommario

- Abbiamo appena visto altre caratteristiche del (Common) Lisp
  - Operatore **`quote`**
  - Espressioni *autovalutanti*, simboli (variabili) et al.
  - Introduzione alla funzione **`eval`**
  - Esempi di varie funzioni ricorsive
    - Funzioni ricorsive su liste: ricorsione in testa e ricorsione in testa-coda