Linguaggi di Programmazione 2022-2023
Lisp e programmazione funzionale IV

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Funzioni di uguaglianza

- Un problema preliminare che si pone in Lisp è quello di controllare se due oggetti sono uguali; il Common Lisp mette a disposizione una pletora di predicati di uguaglianza con semantica diversa

- Vediamo i due più importanti: **`eql`** e **`equal`**

Simboli, liste e funzioni di uguaglianza

- Il predicato **eql** viene usato per controllare l'uguaglianza di simboli e numeri interi (e, ma non bisogna dirlo, di puntatori)

```
prompt> (eql 42 42)
T

prompt> (eql 42 3)
NIL

prompt> (eql 'quarantadue 'quarantadue)
T

prompt> (eql 'quarantadue quarantadue)
NIL

prompt> (eql 'quarantadue 'fattoriale)
NIL

prompt> (eql '(42) '(42))
NIL
```

Simboli, liste e funzioni di uguaglianza

- Il predicato **`equal`** si comporta come **`eql`**, ma è in grado di controllare se due liste sono uguali; in pratica non fa altro che applicare **`eql`** ricorsivamente a tutti gli atomi di una lista: se un'applicazione di **`eql`** ritorna il valore `NIL` allora **`equal`** fa lo stesso, altrimenti viene ritornato il valore `T`

```
prompt> (equal 42 42)
T

prompt> (equal 42 3)
NIL

prompt> (equal 'quarantadue 'quarantadue)
T

prompt> (equal '(1 2 3 (a s d) 4) '(1 2 3 (a s d) 4))
T

prompt> (equal '(1 due tre (a s d) 4) '(1 2 3 (a s d) 4))
NIL
```

Liste e funzioni

- Ora che abbiamo modo di costruire quello che è un *contenitore* basilare (le liste) ed abbiamo l'operazione di **`quote`**, possiamo cominciare a vedere veramente quali sono I vantaggi di un paradigma funzionale
- Consideriamo la seguente lista

```
(defparameter pari (list 2 4 6 8 10))
```

- Supponiamo di voler moltiplicare tutti gli elementi della lista per un certo valore e di ritornare una nuova lista con I nuovi elementi; questa funzione è semplicemente

```
(defun scala-lista (l fattore)
    (if (null l)
       nil
       (cons (* fattore (car l))
             (scala-lista (cdr l) fattore))))
```

Liste e funzioni

- Si noti che la funzione **`scala-lista`** ripete la struttura di appendi
- L'operazione fatta dalla funzione **`scala-lista`** si può astrarre se astraiamo il concetto di valore funzionale
- L'astrazione "applica la funzione *f* a tutti gli elementi della lista *L* e ritorna una lista dei valori" è nota come "map"; in Common Lisp la funzione **`mapcar`** svolge questo compito
- La funzione **`mapcar`** è predefinita, ma essa può essere scritta come

```
(defun mapcar* (funzione lista)
    (if (null lista)
        nil
        (cons (funcall funzione (car lista))
              (mapcar* funzione (cdr lista)))))
```

  Dove il nome **`mapcar*`** viene usato per evitare errori nell'ambiente Common Lisp
- La funzione ***`funcall`*** serve invece chiamare una funzione con un certo argomento

Liste e funzioni

- Supponiamo di avere una serie di funzioni chiamate **scala-4**, **scala-10**, **scala-pi** etc.

```
(defun scala-4  (x) (* x 4))
(defun scala-10 (x) (* x 10))
(defun scala-pi (x) (* x pi))
```

dove **pi** è la costante **3.14....**

- La funzione **scala-lista-10** può essere scritta come

```
prompt> (defun scala-lista-10 (lista)
           (mapcar 'scala-10 lista))
SCALA-LISTA-10

prompt> (scala-lista-10 '(1 2 3 4 5))
(10 20 30 40 50)
```

Liste e funzioni

- Ancora, la funzione **scala-lista-100** può essere scritta come

```
prompt> (defun scala-lista-100 (lista)
            (mapcar 'scala-100 lista))
SCALA-LISTA-10

prompt> (scala-lista-100 '(1 2 3 4 5))
(100 200 300 400 500)
```

- Eccetera eccetera
- Ovviamente questo ci dice che possiamo astrarre la funzione ***scala-lista-x*** in modo abbastanza semplice

```
(defun scala-lista (lista funzione-scalante)
   (mapcar funzione-scalante lista)
```

- Il parametro ***funzione-scalante*** è associato alla funzione che ci interessa; ad esempio per scalare di un fattore 10 ora possiamo scrivere

```
prompt> (scala-lista (list 1 2 3) 'scala-10)
(10 20 30)
```

Funzioni anonime ed operatore `lambda`

- L'esempio precedente è semplice ma ci stuzzica: sarebbe bene poter costruire delle funzioni ausiliarie ogniqualvolta ce ne fosse bisogno
- Come abbiamo già visto, in LISP è possibile definire delle funzioni anonime a questo scopo utilizzando l'operatore speciale `lambda`, risale ai tempi precedenti ai calcolatori

Funzioni anonime ed operatore `lambda`

- Con l'operatore lambda possiamo creare tutte le funzioni che vogliamo senza assegnare loro un nome

```
prompt> (lambda (x) (+ x 42))
#<funzione>

prompt> ((lambda (x) (+ x 42)) 42)
84

prompt> (scala-lista '(1 2 3)
                     (lambda (x) (* x 3)))
(3 6 9)
```

Funzioni anonime ed operatore `lambda`

- Possiamo anche creare funzioni che costruiscono delle funzioni e le ritornano come valori

```
prompt> (defun adder-x (x)
           (lambda (y) (+ x y)))
adder-x

prompt> (defparameter adder-42 (adder-x 42))
adder-42

prompt> adder-42
#<function>

prompt> (funcall adder-42 42)
84
```

Funzioni anonime ed operatore `lambda`

- Dato l'operatore **`lambda`**, possiamo riscrivere la funzione **`scala-lista`** in maniera più elegante

```
(defun scala-lista (lista fattore)
   (mapcar (lambda (e) (* e fattore)) lista))
```

```
prompt> (scala-lista (list 1 0 1 0 1) 42)
(42 0 42 0 42)
```

Operatore `lambda` ed operatore `let`

- Consideriamo la seguente funzione

$$f(x,y) = x(1+xy)^2 + y(1-x) + (1-x)(1+xy)$$

- Usando l'operatore `lambda`, possiamo costruire una serie di valori intermedi da riutilizzare

```
(defun f (x y)
   ((lambda (a b)
        (+ (* x (quadrato a))
           (* y b)
           (* a b)))
      (+ 1 (* x y))
      (- 1 x)))
```

- Ovvero, la funzione anonima viene chiamata con due argomenti che rappresentano I valori intermedi da riutilizzare

Operatore `lambda` ed operatore `let`

- Questo tipo di chiamate a funzioni anonime è così utile da essere stato ri-codificato con un nuovo operatore speciale: **`let`**
- Usando l'operatore **`let`**, la funzione precedente diventa

```
(defun f (x y)
   (let ((a (+ 1 (* x y)))
         (b (- 1 x))
         )
      (+ (* x (quadrato a))
         (* y b)
         (* a b))))
```

- Ovvero, l'operatore **`let`** ci permette di introdurre dei nuovi nomi (variabili) locali da poter riutilizzare all'interno di una procedura; la sua sintassi è la seguente

`(let` `((`n_1 e_1`) (`n_2 e_2`) … (`n_k e_k`))` *espressione*`)`

- Esempio

```
prompt> (let ((a 40) (b (+ 1 1))) (+ a b))
42
```

Tipiche funzioni di ordine superiore

- Le funzioni che prendono una (o più) funzioni come argomenti sono dette *funzioni di ordine superiore*
- La loro esistenza è al cuore del paradigma funzionale di programmazione
- Finora abbiamo visto la funzione **`mapcar`**, ve ne sono altre:
  - **`compose`**
  - **`filter`** (in Common Lisp varianti di **`remove`** e **`delete`**)
  - **`fold`** (in Common Lisp **`reduce`**)
  - **`complement`**

Funzione compose

- La funzione **compose** corrisponde alla nozione matematica di composizione di funzioni
- La semantica della funzione è la seguente: date due funzioni (di *un* solo argomento) *f* e *g* come argomenti, ritorna una nuova funzione che corrisponde alla composizione *f(g(x))*

```
(defun compose (f g)
   (lambda (x)
         (funcall f (funcall g x))))

prompt> (funcall (compose 'first 'rest)
                 '(1 2 3 4 5))
2
```

Funzione `filter`

- La funzione **`filter`** rimuove gli elementi della lista che non soddisfano il predicato

```
(defun filter (predicato lista)
   (cond ((null lista) nil)
         ((funcall predicato (car lista))
          (cons (car lista)
                (filter predicato (cdr lista)))
         (T (filter predicato (cdr lista)))))
```

```
prompt> (filter 'oddp '(1 2 3 4 5))
(1 3 5)
```

Funzione `accumula`

- La funzione **`accumula`** (detta anche `fold` o `reduce`) applica una funzione ad un elemento di una lista ed al risultato (ricorsivo) dell'applicazione di accumula al resto della lista

```
(defun accumula (f iniziale lista)
     (if (null lista)
         iniziale
         (funcall f (car lista)
                  (accumula f iniziale (cdr lista)))))
```

- Esempi

```
prompt> (accumula '+ 0 '(1 2 3))
6

prompt> (accumula '* 1 '(1 2 3 4))
24

prompt> (accumula 'cons NIL '(1 2 3))
(1 2 3)
```

Funzione `accumula`

- Consideriamo la seguente funzione

```
(defun iota (n)
     (if (= n 0)
        nil
        (cons n (iota (- n 1)))))
```

- Quindi

```
prompt> (iota 4)
(4 3 2 1)
```

- Grazie alla funzione `accumula`, possiamo riscrivere la funzione fattoriale (quasi)

```
(defun fattoriale (n)
     (if (zerop n) 1 (accumula '* 1 (iota n))))
```

Utili variazioni sul tema

- Certi simboli in Common Lisp hanno un'interpretazione particolare

- I simboli i cui nomi iniziano con un due punti ':' sono detti keywords ed hanno se stessi come valore

- **Esempi**

```
cl-prompt> :foo
:foo

cl-prompt> :forty-two
:forty-two
```

Lambda lists e keywords

- Le keywords sono usate estensivamente in Common Lisp e ci servono essenzialmente per definire delle funzioni con una sintassi di chiamata più interessante di quella semplice

- Prima di vedere l'uso delle keywords consideriamo però una serie di utili estensioni alla definizione di funzioni Common Lisp

Lambda lists e keywords

- Come abbiamo visto esistono funzioni in Common Lisp che prendono una sequenza variabile di argomenti

- **Esempi**

```
(list 1 2 3 4)                ⇒ (1 2 3 4)
(+ 1 1 1 1 1 1)               ⇒ 6
(< 1 2 3 4 55 66 77 100)      ⇒ T
```

- Ovviamente si possono definire delle funzioni con questo comportamento: basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione
  - Questa lista di argomenti è detta *lambda-list*

Indicatore di lista variabile di argomenti

Parametro contente la lista degli argomenti

```
(defun foo (a b c &rest l) (append l (list a b c)))
```

Lambda lists e keywords

- In Common Lisp vi sono anche funzioni che prendono dei parametri opzionali

- **Esempi**

```
(subseq "qwerty" 2)   ⇒ "erty"
(subseq "qwerty" 1 4) ⇒ "wer"
```

- Ovviamente si possono definire delle funzioni con questo comportamento: basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione

Lambda lists e keywords

- I parametri opzionali possono essere inizializzati con un valore di default

- **Esempi**

```
(defun fattoriale (n &optional (acc 1))
  (if (zerop n) acc (fattoriale (- n 1) (* n acc))))
```

```
cl-prompt> (fattoriale 4)
24
```

```
cl-prompt> (fattoriale 4 2)
48
```

Lambda lists e keywords

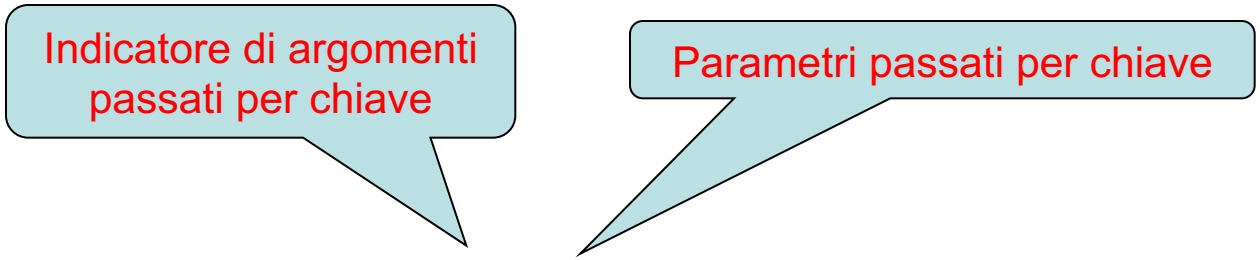
- In Common Lisp si possono definire delle funzioni che utilizzano i loro parametri associandoli a dei nomi, ovvero delle keywords
- Molte funzioni standard Common Lisp hanno questo comportamento

- **Esempi**

```
cl-prompt> (find 2 '(1 2 3 4 5 6))
2

cl-prompt> (find 2 '(1 2 3 4 5 6) :start 3)
NIL

cl-prompt> (find 2 '(1 2 3 4 5 6) :end 3 :start 1)
2

cl-prompt> (sort '((a 1) (q 2) (d 3) (f 4)) 'string>
                 :key 'first)
((Q 2) (F 4) (D 3) (A 1))
```

Lambda lists e keywords

- Ovviamente si possono definire delle funzioni che accettano parametri a “chiave”: basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione




```
(defun make-point (&key x y)
   (list x y))

cl-prompt> (make-point)
(nil nil)

cl-prompt> (make-point :y 42)
(nil 42)

cl-prompt> (make-point :y 42 :x -123)
(-123 42)
```

- Ovvero, ogni parametro passato a chiave diventa una keyword da poter utilizzare al momento della chiamata (anche i parametri passati “a chiave” possono avere un valore di default)

Lambda lists e keywords

- Parametri opzionali, a chiave e variabili vanno sempre dichiarati dopo quelli “obbligatori”
- Le regole per il loro uso sono molto più sofisticate di quelle che abbiamo visto
- Il loro uso è pervasivo in Common Lisp

Input/Output in Common Lisp

- Le due funzioni principale del Common Lisp per la gestione dell'I/O sono **`READ`** e **`PRINT`**

- Ad esse va associata la gestione di *files* e *streams* di I/O

Input/Output in Common Lisp

- La funzione **READ** fa molto di più di una semplice lettura
- L'esempio seguente dovrebbe essere convincente

```
prompt> (read)
(foo 41 (42) 43 45) ; READ aspetta un input.
(FOO 41 (42) 43 45) ; Valore ritornato da READ

prompt> (third (read))
(foo 41 (42) 43 45) ; READ aspetta un input.
(42)
```

- Ovvero **READ**, legge (da dove?) un **intero** oggetto Lisp, riconoscendone la sintassi

Input/Output in Common Lisp

- La funzione **PRINT** stampa (dove?) un oggetto Lisp rispettandone la sintassi
- Il valore ritornato da **PRINT** è il valore dell'oggetto la cui rappresentazione tipografica è appena stata stampata

```
prompt> (print 42)

42    ; PRINT stampa la rappresentazione di 42
      ; (preceduta da un a-capo).
42    ; L'ambiente Common Lisp stampa a video il
      ; valore dell'applicazione della funzione
      ; PRINT al valore 42.

promtp> (print "HELLO WORLD!")

"HELLO WORLD!"
"HELLO WORLD!"

prompt> (print (sqrt -1))

#C(0.0 1.0)
#C(0.0 1.0)
```

Output

- Ovviamente è bene avere a disposizione dei metodi per poter stampare un po' più agevolmente
- Il Common Lisp mette a disposizione la funzione **FORMAT** (simile alla `fprintf` C/C++, ed ai metodi `format` di varie classi Java)
- **FORMAT** è complessa, alcuni esempi semplici seguono

```
prompt> (format t "Il fattoriale di ~D  e` ~D~%" 3 (fact 3))
Il fattoriale di 3 e` 6
NIL
```

la prima linea è la stampa della stringa "formattata"; il **NIL** è il valore ritornato da **FORMAT**

```
prompt> (format t "~S + ~S = ~S~%" '(1) '(2) (append '(1) '(2)))
(1) + (2) = (1 2)
NIL
```

Output

- Le direttive nella stringa da formattare sono introdotte dal carattere **~** (il carattere tilde)
- La direttiva **~D** stampa numeri interi
- La direttiva **~%** va a capo
- La direttiva **~S** stampa un oggetto Lisp secondo la sua sintassi standard
- La direttiva **~A** stampa un oggetto Lisp secondo una sintassi esteticamente “piacevole”

```
prompt> (format t "~S e` una stringa!~%" "foo")
"foo" e` una stringa!
NIL

prompt> (format t "~A forse non e` una stringa!~%" "foo")
foo forse non e` una stringa!
NIL
```

Output formattato

- Che cosa è il `T` che appare come primo argomento a `format`?
- È l'indicazione di "dove" andare a stampare; nella fattispecie su "standard output"

Streams Common Lisp

- Il Common Lisp ha sempre a disposizione almeno tre **streams** standard
  - Standard input
  - Standard output
  - Standard error

- I tre stream sono i valori associati alle tre variabili

  `*standard-input*` (**`java.lang.System.in`** o C++ **`std::cin`**)
  `*standard-output*` (**`java.lang.System.out`** o C++ **`std::cout`**)
  `*error-output*` (**`java.lang.System.err`** o C++ **`std::cerr`**)

- Esempio

```
prompt> (format *standard-output* "~S e` non una stringa!~%" "foo")
"foo" non e` una stringa!
NIL
```

- Il **`T`** passato al posto di **`*standard-output*`** è una comodità

Streams Common Lisp

- Le funzioni **`READ`**, **`PRINT`** e **`FORMAT`** accettano un numero variabile di argomenti
  - Uno di questi è uno stream (di **output** per **`FORMAT`** e **`PRINT`** e di **input** per **`READ`**)

- Esempi

```
prompt> (read *standard-input*)
qwerty
QWERTY

prompt> (print '(42 + 2 = 44 gatti) *error-output*)

(42 + 2 = 44 gatti)
(42 + 2 = 44 gatti)
```

Streams e Files Common Lisp

- La manipolazione dei files in Common Lisp è relativamente complicata, ma simile a quella di Java (cfr., gli oggetti **`pathname`**)

- Per leggere e scrivere da e su un file si usa la **macro** **`with-open-file`**
  - Le macros in (Common) Lisp sono un utile strumento che permette di "estendere" il linguaggio; **`defun`** è solitamente implementata come una macro

- Sintassi

```
(with-open-file (<var> <file> :direction :input) <codice>)

(with-open-file (<var> <file> :direction :output) <codice>)
```

  La variabile `<var>` viene associata allo stream aperto sul `<file>` e può venire utilizzata all'interno di `<codice>`

- La macro **`with-open-file`** si preoccupa di chiudere sempre e comunque lo stream associato a `<var>` anche in presenza di errori (cfr., i meccanismi `try {…} catch () {…} finally {…}` in Java, C++, etc.

Streams e Files Common Lisp

- Esempio: scrittura e lettura sul file "foo.lisp" (nella cartella corrente)

```
(with-open-file (out "foo.lisp"
                     :direction :output
                     :if-exists :supersede
                     :if-does-not-exist :create)
    (mapcar (lambda (e)
                (format out "~S" e))
            '((1 . A) (2 . B) (42 . QD) (3 . D))))

(with-open-file (in "foo.lisp"
                    :direction :input
                    :if-does-not-exist :error)
    (read-list-from in))

(defun read-list-from (input-stream)
   (let ((e (read input-stream nil 'eof)))
      (unless (eq e 'eof)
         (cons e (read-list-from input-stream)))))
```

- Il secondo argomento a **READ** stabilisce che nessun errore debba essere generato quando si incontra la fine del file; in quel caso va invece ritornato il valore passato come terzo elemento (ovvero il simbolo **EOF**)

Streams e Files Common Lisp

```
CL-USER 13 > (defun read-list-from (input-stream)
               (let ((e (read input-stream nil 'eof)))
                 (unless (eq e 'eof)
                   (cons e (read-list-from input-stream)))))
READ-LIST-FROM

CL-USER 14 > (with-open-file (out "foo.lisp"
                                  :direction :output
                                  :if-exists :supersede
                                  :if-does-not-exist :create)
               (mapcar (lambda (e)
                         (format out "~S" e))
                       '((1 . A) (2 . B) (42 . QD) (3 . D))))
(NIL NIL NIL NIL)

CL-USER 15 > (with-open-file (in "foo.lisp"
                                 :direction :input
                                 :if-does-not-exist :error)
               (read-list-from in))
((1 . A) (2 . B) (42 . QD) (3 . D))
```

Interazione con l'ambiente Common Lisp

- L'ambiente Lisp, o meglio la sua *command-line*, esegue tre operazioni fondamentali, ed ora che sappiamo qualcosa in più su input ed output possiamo esplicitarle

- **Legge** (`READ`) ciò che viene presentato in input
  - Ciò che viene letto viene rappresentato internamente in strutture dati appropriate (numeri, caratteri, simboli, stringhe, cons-cells, ed altro ancora…)
- La rappresentazione interna viene **valutata** (`EVAL`) al fine di produrre un valore (o più valori)
  - Vedremo in seguito che cosa fa la funzione `EVAL`
  - Ovvero la scriveremo direttamente in Common Lisp

- Il valore così ottenuto viene **stampato** (`PRINT`)

- Questo è il **READ-EVAL-PRINT Loop** (**REPL**)

Sommario

- Uguaglianza
- Funzioni di ordine superiore
  - Le funzioni sono oggetti di *prima classe* (*first class*)
- Parametri opzionali (`&optional`), di lunghezza variabile (`&rest`) ed a chiave (`&key`)
- Alcune funzioni per l'Input/Ouput
- Il **REPL** (read-eval-print loop) è il ciclo principale dell'inteprete Lisp
  - Capire i suoi stadi è fondamentale per capire come l'ambiente Lisp opera