Linguaggi di Programmazione 2024-2025
Lisp e programmazione funzionale V

Marco Antoniotti

Fabio Sartori

Riassunto Operazioni dell'Ambiente Lisp

- L'ambiente Lisp, o meglio la sua *command-line*, esegue tre operazioni fondamentali

- **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)

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

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

Valutazione di espressioni e funzioni: `apply` ed `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

Le funzioni `apply` e `eval`

- La funzione **`apply`** è definita come

  `apply : funzione list` $\rightarrow$ `sexp`

  ovvero prende un designatore di funzione (ovvero un symbolo, una lambda-espressione od una funzione) e ritorna un valore

- La funzione **`eval`** costruisce il valore denotato da una sexp

  `eval : sexp env` $\rightarrow$ `sexp`

Le funzioni `apply` e `eval`

- Le funzioni `apply` e `eval` possono essere scritte direttamente in Lisp (o Scheme)

- Ovvero, dato che in Lisp i dati ed i programmi sono la stessa cosa è possibile scrivere facilmente un **interprete** Lisp (o Scheme) in Lisp (o Scheme)
  - Questi interpreti sono detti **meta-circolari**
  - La costruzioni di varianti di interpreti meta-circolari è uno dei metodi con cui si procede ad esplorare nuove modalità di programmazione

Le funzioni `apply` e `eval`

- Costruiamo la funzione **`valuta`** (**`eval`** è standard) a partire dalle regole di valutazione definite precedentemente (*)
- La funzione valuta prende una S-expression `sexp` ed un ambiente `env`
  - Vedremo poi che cosa è un “ambiente”
- La funzione **`valuta`** procede nel seguente modo
- **Caso 1**: `sexp` è una espressione autovalutante?

```
(self-evaluating-p sexp)
```

  - Se si allora ritorna il suo valore, cioè sexp stessa
  - Se no, allora…

(*) la funzione **`valuta`** *si comporta quasi come in Scheme; il Lisp ha regole di valutazione leggermente diverse.*

Le funzioni `apply` e `eval`

- **Caso 2**: `sexp` è una variabile?

```
(variable-p sexp)
```

  - Se si, allora recuperane il valore associato
    - Dove?
    - In `env`

```
(var-value sexp env)
```

  - Se no allora …

Le funzioni `apply` e `eval`

- **Caso 3**: `sexp` è una espressione quotata della forma `(quote <e>)`?

  `(quoted-exp-p sexp)`

  - Se si, allora ritorna `<e>` così com'è
  - Se no allora …

Le funzioni `apply` e `eval`

- **Caso 5**: `sexp` è una lambda-expression? Ovvero una lista della forma `(lambda (…) …)`?

```
(lambda-exp-p sexp)
```

  - Se si, allora crea una *chiusura* ricordando l'ambiente in cui questa espressione viene valutata (ovvero: ricordando *static* link)

```
(make-fun (lambda-exp-vars sexp)
          (lambda-exp-body sexp)
          env)
```

  - Se no allora …

Le funzioni `apply` e `eval`

- **Caso …**: `sexp` è una applicazione di una funzione a degli argomenti?

```
(application-exp-p sexp)
```

  - Se si, allora applica (`apply`) l'operatore alla lista dei valori ottenuti valutando ogni argomento

```
(apply (eval (operator sexp) env)
       (list-of-values (operands sexp) env))
```

  - Se no allora …

Digressione: sequenze di valutazioni in Lisp (o Scheme)

- Ricordiamo che **`let`** non è nient'altro che zucchero sintattico per un'applicazione di una funzione (anonima)

- Anche le sequenze di valutazioni sono rappresentabili come applicazioni successive di funzioni

- Esempio

Ordine d'esecuzione

```
(defun foo (x)
   ((lambda (y)
       (bar (1+ x)))                          ; 2
    (format t "Calling FOO (~S)~%" x)))       ; 1
```

L'esecuzione di **`foo`** prima esegue la chiamata a **`format`**, il cui valore viene passato in **`y`**, che viene **ignorata** durante l'esecuzione di `(bar (1+ x))`

Digressione: sequenze di valutazioni in Lisp (o Scheme)

- Questo idioma è così utile che viene riscritto come

  (**progn** <e_1> <e_2> … <e_N>)

  o

  (**begin** <e_1> <e_2> … <e_N>) in Scheme

- Quindi `foo` può essere riscritta come

```
(defun foo (x)
   (progn
      (format t "Calling FOO (~S)~%" x)
      (bar (1+ x))))
```

Digressione: sequenze di valutazioni in Lisp (o Scheme)

- Quando `progn` è l'espressione principale di una `defun` (ed anche in altre forme Lisp) allora la si può elidere senza problemi; il corpo della `defun` si dice essere un `progn` *implicito*

- Quindi `foo` può essere infine riscritta come

```
(defun foo (x)
   ;; progn implicito.
   (format t "Calling FOO (~S)~%" x)
   (bar (1+ x))
```

Le funzioni `apply` e `eval` (ripresa)

- Vediamo alcune delle funzioni di cui abbiamo assunto l'esistenza

- **`self-evaluating-p`**

```
(defun self-evaluating-p (x)
   (and (atom p) (not (symbolp x))))
```

- **`quoted-exp-p`**

```
(defun quoted-exp-p (x)
   (and (consp x) (eq (first x) 'quote)))
```

- **`lambda-exp-p`**

```
(defun lambda-exp-p (x) (and (consp x) (eq (first x) 'lambda)))
```

- **`valuta-seq`** (e, similarmente, **`list-of-values`**)

```
(defun valuta-seq (seq env)
   (mapcar (lambda (s) (valuta s env)) seq))
```

Le funzioni `apply` e `eval` (mettiamo tutto assieme)

- La funzione **`valuta`** (**`eval`** è standard) si può quindi costruire a partire dalle regole di valutazione definite precedentemente (*)

```
(defun valuta (sexp &optional (env *the-global-environment*))
   (cond ((self-evaluating-p sexp) sexp)
         ((variable-p sexp) (var-value sexp env))
         ((quoted-exp-p sexp) (exp-of-quotation sexp))
         ((if-exp-p sexp) (valuta-if sexp env))
         ((lambda-exp-p sexp) (make-fun (lambda-exp-vars sexp)
                                        (lambda-exp-body sexp)
                                        env))
         ((sequence-exp-p sexp)
          (valuta-seq (sequence-expressions sexp) env))
         ((cond-exp-p sexp) (valuta (cond-to-if sexp) env)
         ((definition-exp-p sexp) (valuta-def sexp env))
         ((application-exp-p sexp)
          (applica (valuta (operator sexp) env)
                   (list-of-values (operands sexp) env)))
         (t (error ";;; Non so come valutare : ~S." sexp))))
```

(*) la funzione **`valuta`** *si comporta quasi come in Scheme; il Lisp ha regole di valutazione leggermente diverse.*

Le funzioni `apply` e `eval` (ripresa)

- Costruiamo la funzione **`applica`** (**`apply`** è standard) a partire dalle regole di valutazione definite precedentemente (*)

```
(defun applica (fun arguments)
   (cond ((primitive-fun-p fun)
          (apply-primitive-fun fun arguments))
         ((fun-p fun)
          (valuta-seq (fun-body fun)
                      (extend-environment (fun-parameters fun)
                                          arguments
                                          (fun-environment fun))))
         (t
          (error "Funzione ~S sconosciuta in APPLICA." fun))))
```

- La funzione **`applica`** richiama la funzione **`valuta-seq`**, la quale richiama **`valuta`**; ovvero **`valuta`** ed **`applica`** (**`eval`** e **`apply`**) sono mutualmente ricorsive.

(*) la funzione **`apply`** *si comporta quasi come in Scheme; il Lisp ha regole di valutazione leggermente diverse.*

La rappresentazione interna di “funzioni”

- La valutazione di una espressione `LAMBDA` genera una funzione che viene rappresentata nell’ambiente come una struttura particolare (cfr., la funzione **`make-fun`** chiamata da **`valuta`**) detta **chiusura**
- Questa struttura contiene il corpo dell’espressione `LAMBDA`, la lista dei parametri fomali e l’ambiente in cui l’espressione `LAMBDA` è stata costruita
  - Ovvero la struttura contiene lo ***static link*** all’ambiente di valutazione dove recuperare i valori delle variabili libere nel corpo dell’espressione `LAMBDA`
- La funzione **`applica`** usa lo *static link* contenuto nella chiusura

Ambienti (environments)

- Le funzioni **`eval`** ed **apply** si appoggiano sull'implementazione degli **ambienti** (**environments**), ovvero sulla manipolazione di mappe (!) di associazione tra simboli e valori; un **environment** è una sequenza di **frame**s

- Le funzione **`var-value`** non è nient'altro che una 'get' di una chiave in una mappa

- Come possiamo implementare le funzioni di manipolazione di un ambiente in (Common) Lisp?
  - **`make-frame`**
  - **`extend-env`**
  - **`var-value`**
  - **`var-value-in-frame`**

Ambienti (environments)

- Cominciamo dalla manipolazione di un frame
- Un frame viene rappresentato come una lista di coppie prefissa dal simbolo **`frame`**

```
(defun make-frame (vars values)
   (cons 'frame (mapcar 'cons vars values)))

;;; mapcar agisce su più liste!
```

- Esempi

```
cl-prompt> (make-frame '(x y z) '(0 1 0))
(FRAME (X . 0) (Y . 1) (Z . 0))

cl-prompt> (make-frame '(q w) '(il-simbolo-q "W"))
(FRAME (Q . IL-SIMBOLO-Q) (W . "W"))
```

Ambienti (environments)

- Un ambiente viene viene quindi esteso nella seguente maniera

```
(defun extend-env (vars vals &optional (base-env *the-empty-env*))
      (if (= (length vars) (length vals))
         (cons (make-frame vars vals) base-env)
         (if (< (length vars) (length vals))
            (error "Too many arguments supplied")
            (error "Too few arguments supplied"))))
```

- Come base definiamo anche

```
(defparameter *the-empty-env* '((frame (nil . nil) (t . t))))
```

- Esempi

```
cl-prompt> (defparameter env1
                (extend-env '(x y z) '(0 1 0) ()))
ENV1

cl-prompt> (extend-env '(q w) '(il-simbolo-q "W") env1)
((FRAME (Q . IL-SIMBOLO-Q) (W . "W")) (FRAME (X . 0) (Y . 1) (Z . 0)))
```

Riscrittura di espressioni

- Una delle operazioni più importanti che un interprete/compilatore fa è di **riscrivere** un'espression in un'altra (più semplice) al fine di riutilizzare del codice già scritto
- Le strutture dati Lisp usate per le espressioni da valuta ed applica rendono questa operazione particolarmente semplice
  - **`COND`** viene riscritta in **`IF`**
  - **`LET`** viene riscritta nella corrispondente operazione **`LAMBDA`**
- L'espressione riscritta viene poi ripassata al valutatore per completarne l'esecuzione

Interprete Meta Circolare

- Le funzioni valuta ed applica sono al centro dell' interprete meta-circolare (IMC) scritto in Common Lisp

- L'IMC è distribuito sul sito Moodle del corso e consta di tre files
  - `env.lisp`
    che contiene le operazioni riguardanti la manipolazione di frames ed environements
  - `imc.lisp`
    che contiene l'interprete vero e proprio; ovvero la coppia **`valuta/applica`** e varie operazioni di manipolazione di espressioni e di riscrittura
  - `repl.lisp`
    che contiene un semplice **read-eval-print** loop (senza controllo di errori)

- Tutte le operazioni descritte nelle pagine precedenti sono completamente sviluppate nel codice distribuito

Conclusioni

- Il Lisp è uno degli esempi più importanti di linguaggi funzionali
- Lo stile di programmazione funzionale, basato sulla composizione di funzioni e sul trattamento di funzioni alla stregua di oggetti primitivi, è estremamente importante
- Il linguaggio Common Lisp è molto più esteso di quanto non abbiamo visto; in particolare non abbiamo visto
  - Caratteristiche imperative
    - Assegnamenti: **`setf`**
    - Costrutti di iterazione: **`dolist`**, **`dotimes`**, **`do`**, **`loop`**
  - Caratteristiche Object Oriented
    - CLOS
    - Multimethods (solo Common Lisp, Dylan ,Chill e pochi altri linguaggi hanno questa caratteristica)
  - I/O
    - **`open`**, **`close`**, **`write`**
  - Macros
  - Gestioni eccezioni
    - **`error`**, **`handler-case`**, **`invoke-restart`**
- Il linguaggio è estremamente flessibile e può essere usato per moltissimi compiti
  - "*Practical Common Lisp*" di Seibel contiene numerosissimi esempi