Programmazione Funzionale e Lisp

Antonio Vivace, revision 4

Introduzione

La **programmazione funzionale** è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche.

Il punto di forza di questo paradigma è la mancanza di *side-effects* delle funzioni.

Viene posto un maggiore accento sulla *definizione* di funzioni, rispetto ai paradigmi procedurali e imperativi, che invece prediligono la specifica di una *sequenza* di comandi da eseguire.

Un programma funzionale è **immutabile**: i valori non vengono trovati cambiando lo stato del programma, ma costruendo nuovi stati a partire dai precedenti.

Valutazione di funzioni

La valutazione avviene scrivendo sullo stack di sistema gli **activation frames**. I parametri formali delle funzioni vengono associati ai valori (**passaggio per valore**, nessun effetto collaterale) degli argomenti.

Fasi della **valutazione di funzione**:

1) **Caricamento dei parametri** in un'area di memoria raggiungibile dalla procedura;
2) **Trasferimento** del controllo alla procedura;
3) **Allocazione** delle risorse (memoria) necessarie alla procedura;
4) **Computazione** della procedura;
5) **Caricamento dei risultati** in un'area di memoria raggiungibile dal chiamante;
6) **Restituzione** del controllo al chiamante.

Queste fasi agiscono sui registri e sullo stack utilizzato dall'esecutore (*runtime*) del linguaggio.

Activation frames

Chiamati anche *stack frames* o *activation records*, costituiscono il **call stack**. Ogni activation frame corrisponde alla chiamata di una subroutine che non è ancora terminata con un valore di ritorno. L'activation frame in cima allo stack è quello della routine attualmente in esecuzione.

Usualmente, contengono **almeno** i seguenti oggetti (in ordine di push):

- Argomenti (parametri) passati alla funzione (se presenti);
- L'indirizzo di ritorno del chiamante;
- Spazio per le variabili locali della funzione (se presenti).

In generale contengono:

1) Registri da salvare prima della chiamata di una subroutine;
2) Indirizzo di ritorno;
3) Variabili, definizioni locali e valori di ritorno;
4) Spazio per i valori degli argomenti;
5) *Static link*;
6) *Dynamic link*;
7) Ulteriori dati dipendentemente dal linguaggio e/o politiche di allocazione del compilatore.

Il **Dynamic Link** (chiamato anche *control link*) punta all'activation frame del chiamante, ed è utilizzato durante la terminazione della funzione chiamata.

Lo **Static Link** (chiamato anche *access link*) punta all'activation frame associato al più vicino scope che contiene la definizione della subroutine. È utilizzato per implementare lo static scope nei linguaggi in cui la definizione delle subroutine può essere innestata. In altre parole, una funzione definita all'interno di un'altra deve poter accedere allo stack frame della funzione che la "contiene".

Inoltre, la gestione dello *static link* in Common Lisp è più complessa del solito, dato che il linguaggio ammette la creazione *a runtime* di funzioni (che si devono "ricordare" il valore delle loro variabili libere al momento della creazione). Queste funzioni sono implementate con particolari strutture dati chiamate *closures*.

Questa caratteristica di Common Lisp permette la creazione di **funzioni anonime** con l'operatore **lambda**. Una *lambda expression* può essere utilizzata ovunque possiamo usare un nome di una funzione.

Stack pointer e Frame pointer

Quando la dimensione degli activation frame può cambiare, tra differenti funzioni o tra invocazioni di particolari funzioni, **fare la pop di un frame dallo stack non costituisce un decremento fisso dello stack pointer**.

Alla terminazione della funzione chiamata, invece, lo stack pointer è ripristinato al frame pointer che equivale al valore dello stack pointer appena prima che la funzione sia stata chiamata.

Ogni activation frame contiene quindi uno stack pointer alla cima del frame che gli sta immediatamente sotto.

Lo stack pointer è un registro mutabile condiviso da tutte le invocazioni, un frame pointer di una data chiamata di una funzione, non è altro che la copia dello stack pointer così com'era prima che la funzione fosse chiamata.

Ricorsione

```
(defun fattoriale (n)
    (if (= n 0)
        1
        (* n (fattoriale (- n 1)))))

(defun fibonacci (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (t (+ (fib (- n 2)) (fib (- n 1))))
           ))
```

Possiamo riscrivere `fattoriale` in questa maniera:

```
(defun fatt-ciclo (n acc)
    (if (= n 0)
        acc
        (fatt-ciclo (- n 1) (* n acc))))

(defun fattoriale (n)
    (fatt-ciclo n 1))
```

Abbiamo riscritto `fattoriale` utilizzando uno degli argomenti come *accumulatore*. `fatt-ciclo` non rappresenta altro che cicli in "incognito".

Funzioni Tail-ricorsive

La chiamata ricorsiva di una funzione tail-ricorsiva può essere ottimizzata dal compilatore con un operazione di `jump`, senza dover creare un nuovo record di attivazione.

Abbiamo riscritto `fattoriale` come tail-ricorsiva, ma lo non possiamo fare per `fib`, perchè richiede la **combinazione** di due chiamate ricorsive.

Tutti gli algoritmi che richiedono **iterazioni** (cicli) possono essere trasformati in un insieme di funzioni **ricorsive**. Il contrario **non** è vero.

In particolare, la traduzione da tail-ricursivo a iterativo può essere automatizzata.

Non tutti i programmi ricorsivi possono essere trasformati in tail-ricorsivi.

Funzioni mutualmente ricorsive

Un insieme di funzioni **mutualmente ricorsive** può rappresentare una Macchina di Turing (*Cfr. Tesi di Churc e Kleene*). Tutti i linguaggi funzionali puri sono Turing-completi.

Cdr-recursion o rest-recursion

```
(defun appendi (l1 l2)
    (if (null l1)
        l2
        (cons (car l1)  (appendi (cdr l1) l2)))
```

Questa funzione è ricorsiva e presenta una tipica forma di ricorsione *strutturale* sul resto di una lista.

Ricorsioni semplici e doppie

A volte una ricorsione sul resto della lista (cdr) non è però sufficente. Si potrebbe, ad esempio, aver bisogno di navigare su liste innestate, e fare ricorsione anche sul primo elemento (car) per "sbucciarlo". Vedi `count-atoms`.

Cons-cells e `cons`

Una delle strutture dati più importanti in Lisp è la **cons-cell**: una coppia di puntatori a due elementi.

Le cons-cells sono create dalla funzione `cons`, che **alloca la memoria** necessaria al mantenimento della struttura: vengono generati in memoria dei **grafi di puntatori** arbitrariamente complessi.

Una cons-cell può essere pensata come una struttura che si riferisce a due altre cose. Le "altre cose" possono essere un atomo od un altra cons-cell.

```
cl-prompt> (cons 1 2)
(1 . 2)
```

La notazione *dotted pair* è l'unica **irregolarità sintattica infissa** in Lisp.

`cons` può essere quindi usata per rappresentare sequenze(liste) di oggetti.

Operatore `quote`

È uno degli operatori speciali, dice all'interprete di non procedere alla valutazione della (sotto)espressione ma di passarla letteralemente.

Abbreviazione: `'<e>` equivale a `(quote <e>)`

Questo operatore sembra non agire su numeri e stringhe perchè essi sono **autovalutanti**: *sono espressioni il cui valore ha la stessa rappresentazione tipografica dell'espressione tipografica che li denota.*

Le liste, invece, se non quotate, rappresentano delle espressioni da valutare.

Conseguenza:

In Lisp, i programmi ed i dati sono esattamente la stessa cosa.

Atomi, cons-cells e Simbolic Expressions

In Lisp, gli oggetti principali si dividono in due categorie:

- Gli **atomi** sono simboli e numeri (e stringhe);
- I **non-atomi** sono le *cons-cells* (le liste).

Le cons-cells, i numeri, i simboli e le stringhe, costituiscono le **Simbolic Expressions**, dette anche *Sexps*.

Programmi e sexps in Lisp sono equivalenti.

Funzioni condizionali

`if`

```
(if statement expr else_expr)
```

Con il terzo argomento (`else_expr`) facoltativo.

`cond`

```
(cond (cond1 expr1)
      (cond2 expr2)
      (cond3 expr3)
      (t expr4))
```

expr n viene valutata se *cond n* è vera. Se tutte risultano false, *expr4* viene valutata visto che `t` è sempre vera.

Esempi di funzioni ricorsive

Somma

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

Lunghezza di una lista (al primo livello)

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

Ultimo elemento di una lista

```
(defun last-e (l)
    (cond ((null l) nil)
    ((atom l)
     (error "l'argomento non è una lista"))
    ((null (rest l)) (first l))
    (t (last-l (rest l)))))
```

Contare gli atomi di una lista

```
(defun count-atoms(x)
    (cond ((null x) 0)
          ((atom x) 1)
          (t (+ (count-atoms (first x)
                (count-atoms (rest x)))))))
```

Questo è un caso in cui la ricorsione viene effettuata sia sulla testa che sulla coda di una lista.

Profondità massima di annidamento

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

Se è un atomo la profondità è 1, se è vuota è 0, altrimenti è il massimo tra la profondità del primo elemento aumentata di 1 e la profondità del resto.

Appiattire una lista

```
(defun flatten (x)
    (cond ((null x) x)
          ((atom x) (list x))
          (t (append (flatten (first x))
                     (flatten (rest x))))))
```

Immagine speculare di una sexp

```
(defun mirror (x)
    (if (atom x)
        x
        (append (mirror (rest x))
                (list (mirror (first x))))))
```

Si fa l'`append` del mirror del resto della lista e del mirror della **lista** costituita dal mirror del primo elemento perchè altrimenti toglierebbe la parentesi se il primo elemento è una lista, o darebbe errore di parametro se fosse un atomo (infatti `(append '(1 2) 1')` risulta in `The value 1 is not type LIST. [Condition of type TYPE ERROR]`).

Inversione di una lista

```
(defun inverti (x)
    (cond ((null x) x)
          ((atom x) x)
          (t (cons-end (first x)
                       (inverti (rest x))))))
```

Dove `cons-end` deve fare questa operazione `(cons-end S e1 ... en) -> (e1 ... en S`:

```
(defun cons-end (x y)
    (if (null x) (list y)
        (cons (first x) (cons-end y (rest x)))))
```

Slittamento di una lista

`Circulate` deve comportarsi in questo modo:

```
(circulate) '(1 2 3 4) 'left) -> (2 3 4 1)
(circulate) '(1 2 3 4) 'right) -> (4 1 2 3)
```

Quindi:

```
(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))))
```

Dove `(last-l list)` restituisce l'ultimo elemento di liste e `(but-last list)` restituisce `list` senza l'ultimo elemento.

Funzioni di uguaglianza

`eql` viene usato per controllare l'uguaglianza di simboli e numeri interi.

`equal` (o `eq`) si comporta come `eql`, ma è in grado di controllare se due liste sono uguali: applica `eql` ricorsivamente a tutti gli atomi di una lista.

scala-lista

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

Funzioni di Ordine Superiore

Le funzioni sono oggetti di *prima classe*.

Le funzioni che prendono una (o più) funzioni come argomenti sono dette di *ordine superiore*. La loro esistenza è al cuore del paradigma funzionale.

`mapcar`

L'astrazione *applica la funzione **f** a tutti gli elementi di una lista L e ritorna una lista di valori* è nota come **map**.

In Common Lisp, `mapcar` (`mapcar 'funzione lista`) svolge questo compito, ma si potrebbe ridefinire come:

```
(defun mapcar2 (funzione lista)
    (if (null lista) nil)
    (cons (funcall funzione (car lista))
          (mapcar2 funzione (cdr lista))))
```

Abbiamo utilizzato `funcall` che opera in questo modo: `(funcall 'funzione arg)` applica `funzione` su `arg`.

Definire funzioni come `scala-lista` diventa triviale facendo uso di `mapcar`:

`scala-lista` con `mapcar`

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

Funzioni Anonime ed operatore `lambda`

Sarebbe bene poter costruire delle funzioni *ausiliarie* ogniqualvolta ce ne fosse bisogno. A questo scopo è possibile utilizzare l'operatore *lambda*.

Ad esempio:

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

`scala-lista` con `lambda`

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

Operatore `let`

Consideriamo la seguente funzione:

$f(x,y) = x\ (1 + x\ y) + y\ (\ 1 - y)$

Usando l'operatore lambda possiamo costruire una serie di valori intermedi da riutilizzare:

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

```
    (+ 1 (* x y))
    (- 1 y)))
```

In pratica, costruiamo una funzione che opera sui valori intermedi a e b, a loro volta passati immediatamente come risultati di valutazione di funzione.

Questo tipo di chiamate a funzioni anonime è così utile da essere stato ri-codificato con un nuovo operatore speciale: **let**.

La funzione precedente diventa:

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

La sintassi di `let` è quindi `(let (lista_di_variabili) (expr))`.

Possiamo dunque introdurre dei nuovi nomi (variabili) *locali* da poter riutilizzare all'interno della procedura.

Tipiche funzioni di ordine superiore

Alcune delle tipiche funzioni di ordine superiore sono:

- `mapcar`
- `compose` (in Common Lisp varianti di `remove` e `delete`)
- `filter`
- `fold` (in Common Lisp `reduce`)
- `complement`

`compose`

Corrisponde alla nozione matematica di *composizione di funzioni*. Date due funzioni(di *un* solo argomento) **f** e **g** come argomenti, ritorna una nuova funzoine che corrisponde a $\mathbf{f(g(x))}$.

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

Nota: non dimenticare di usare `quote` quando si passano funzioni come argomenti, o Lisp tenterà di valutarle immediatamente:

```
cl-prompt> (funcall (compose 'function1 'function2)
                    `(1 2 3 4 5 6))
```

Il risultato sarà quindi la valutazione di *function1* con argomento *function2* valutata in `(1 2 3 4 5 6)`.

`filter`

Data una lista ed un predicato, 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)))))

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

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)))))
```

In altre parole, se la lista è vuota torna il valore iniziale, altrimenti chiama la funzione data con il primo elemento della lista come primo argomento, e il risultato della chiamata ricorsiva sul resto della lista come secondo argomento.

Indicatori di parametro

E' possibile anteporre un indicatore di parametro all'argomento, durante la definizione di funzioni, che ne modifica il carattere:

- `&rest` rende il parametro variabile
- `&optional` rende il parametro opzionale
- `&key` rende il parametro a chiave

Una lista di argomenti variabili è chiamata **lambda lists** e si definisce con l'indicatore `&rest`.

I parametri opzionali, a chiave e variabili vanno sempre dichiarati **dopo** quelli obbligatori.

I simboli i cui nomi iniziano con `:` sono detti *keywords* ed hanno sè stessi come valore.

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

cl-prompt> (make-point :x 10 :y 122)
(10 122)
```

Input/Output

`read` legge un **intero** oggetto Lisp riconoscendone la sintassi. `print` stampa un oggetto Lisp rispettandone la sintassi, in particolare il valore ritornato è il valore dell'oggetto la cui rappresentazione tipografica è appena stata stampata. `format` è un po' più complesso: ritorna `NIL` e permette l'uso di direttive, introdotte dal carattere `~`:

- `~D` stampa numeri interi;
- `~%` va a capo;
- `~S` stampa un oggetto Lisp secondo la sua sintassi standard;
- `~A` stampa un oggetto Lisp secondo una sintassi esteticamente "piacevole".

`format` può stampare in ognuno dei tre **streams** che Common Lisp ha a disposizione:

- `*stantard-input*`;
- `*stantard-output*` (abbreviabile con `t`);
- `*error-output*`.

Ad esempio:

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

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

I/O su file

Per scrivere e leggere su file si usa la macro `with-open-file`

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

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

Questa macro si preoccupa di chiudere sempre e counque lo stream associato a `<var>`, anche in presenza di errori.

Macro

Le Macro in Lisp sono un modo di **trasformare codice Lisp**. Durante una fase di **macroespansione** l'espressione Lisp viene passata alla funzione macro. Questa funzione macro può fare della computazione arbitraria e poi restituisce altro codice Lisp. Il codice risultante è quello che puoi l'interprete/compilatore vedrà e compilerà.

Un esempio sul particolare comportamento delle macro:

```
(defmacro eight () (+ 3 5))
```

Utilizzare questa macro in un'espressione, `(eight)`, è equivalente ad utilizzare `8`. Tuttavia:

```
(defmacro eight () '(+ 3 5))
```

L'espressione è **quotata**, quindi invocare `(eight)` ritorna `(+ 3 5)` che a sua volta è valutato a `8` a runtime.

`defun` è una macro.

Read-Eval-Print Loop (REPL)

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

- **Legge** (`read`) cioè che viene presentato in input, e viene rappresentato internamente con strutture dati appropriate (numeri, caratteri, simboli, stringhe, cons-cells, ...)
- La rappresentazione interna viene **valutata** (`eval`) al fine di produrre uno o più valori
- Il valore ottenuto viene **stampato** (`print`).

Valutazione di espressioni e funzioni

Abbiamo già detto che **programmi e sexps in Lisp sono equivalenti**, la valutazione di essi segue queste regole (funzione *eval*):

Data una sexp,

- Se è un atomo:
  - 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'ambiete corrente e ritornalo
    - se non esiste un valore associato segnala un errore
- Se è una cons-cell di tipo `(O A1 A2 A3 ... An)`
  - Se `O` è un operatore speciale, allora la lista viene valutata in modo speciale
  - Se `O` è un simbolo che denata una funzione nell'ambiente corrente, allora applica la funzione (`apply`) alla lista `(V_A1 V_A2 V_A3 ... V_An)` che raccoglie i valori delle valutazioni delle espressioni `A1 A2 A3 ... An`.
  - Se `O` è una *lambda expression* la si applica alla lista `(V_A1 V_A2 V_A3 ... V_An)` che raccoglie i valori delle valutazioni delle espressioni `A1 A2 A3 ... An`.
  - Altrimenti segnala un errore.

La funzione `apply` è definita come

```
apply : funzione lista -> sexp
```

Prende un designatore di funzione (un simbolo, una *lambda-expression* od una funzione) e ritorna un valore.

La funzione `eval` costruisce il valore denotato da una sexp:

```
eval : sexp env -> sexp
```

Queste due funzioni possono essere scritte direttamente in Lisp. Ovvero, dato che in Lisp i dati ed i programmi sono la stessa cosa, è possibile scrivere facilmente un **interprete** Lisp in Lisp:

- Questi interpreti sono detti **meta-circolari**;
- Le costruzioni di varianti di intepreti meta-circolari è uno dei metodi con cui si procede ad esplorare nuove modalità di programmazione.

`eval`

Dalle regole di valutazione definite precedentemente possiamo costruire una nostra funzione di valutazione:

```
(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))))
```

`apply`

```
(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 `valuta-seq`, la quale richiama `valuta`: `eval` ed `apply` sono mutalmente ricorsive.

Rappresentazione interna di funzioni

La valutazione di una espressione *lambda* genera una funzione che viene rappresentata nell'ambiente come una struttura particolare, detta **chiusura**.

Questa struttura contiene:

- il corpo dell'espressione *lambda*;
- la lista dei parametri formali;
- l'ambiente in cui l'espressione è stata costruita.

Ovvero: contiene lo **static link** dell'ambiente di valutazione dove recuperare i valori delle variabili libere nel corspo dell'espressione *lambda*.

`apply` usa lo **static link** contenuto nella chiusura.

Environments

`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 **frames**.

Possiamo implementare le seguenti funzioni di manipolazione di un ambiente Common Lisp:

- `make-frame`
- `extend-env`
- `var-value`
- `var-value-in-frame`

Manipolazione di un frame

Un frame viene rappresetato come una lista di coppie prefissa dal simbolo `frame`.

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

```
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"))
```

Estensione di un ambiente

```
(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"))))

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

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. Ad esempio:

- `cond` viene riscritta in `if`
- `let` viene riscritta nella corrispondente operazione `lambda`

Le espressioni riscritte vengono poi ripassate al valutatore per completarne l'esecuzione.

Fonti e bibliografia

- Programming Language Structures - *T. Wahls*
- Activation Records/Stack Frames - *D. August*
- Introduction to compilers, Stack frames - *T. Teitelbaum*
- What is the purpose of the EBP frame pointer register?, What is the Static Link used for - *Stackoverflow*
- Static Link and Dynamic Link - *Fischer*
- Lisp e Programmazione Funzionale - *M. Antoniotti, G. Pasi*
- Paradigma Funzionale, Call Stack - *Wikipedia*
- What makes Lisp macros so special?
- c2 wiki
- Common Lisp cookbook