Introduzione al Common Lisp

Autore: Matteo Lucarelli
ultima versione su: www.matteolucarelli.net
Documento in costruzione

Il Lisp è uno dei linguaggi più antichi (John McCarthy,1960) utilizzato diffusamente ancora oggi. Il nome deriva da LISt Processor, ma qualcuno dice Long Irritating Series of Parentheses (!), infatti si basa su poche e chiare regole sintattiche, prima delle quali è il concetto di lista. Questo porta a due caratteristiche interessanti: è particolarmente portato per il trattamento di liste ed, essendo lo stesso codice lisp espresso in forma di lista, è facile scrivere codice che a sua volta tratti, generando o modificando, codice lisp. Questo è uno dei motivi che lo rendono molto usato nelle ricerche sull'intelligenza artificiale (l'altro è che McCarthy è stato uno dei padri della AI !). Il lisp è comunque utilizzato in molti ambiti, tra i quali come linguaggio di estensione all'interno di Emacs e dei più diffusi sistemi CAD (AutoCAD, IntelliCAD, ecc.).

Alcune caratteristiche peculiari sono:

- Dynamic redefinition: la possibilità di ridefinizione dinamica del codice, ovvero lo stesso può essere modificato senza interromperne l'esecuzione.
- Dynamic typing: le variabili non hanno tipo predefinito, e non richiedono dichiarazione.
- Garbage collection: gestione automatica dell' allocazione e della liberazione della memoria.

Il codice lisp inoltre può essere eseguito:

- in modo interattivo: attraverso l'interprete che si comporta come un terminale,
- come un linguaggio di scripting
- oppure può essere compilato per ottenere la massima velocità

A parte quanto detto, la caratteristica più evidente del lisp è di non assomigliare a nessun altro linguaggio. Per questo motivo lo studio del lisp, nonostante le iniziali difficoltà, è indicato anche solo a scopo didattico.

Si apprenderà gradualmente come i noti concetti di variabile, funzione, ecc. assumano nel lisp un significato sfumato e differente. In realtà l'uso di questi termini non sarebbe neppure corretto, anche se verranno comunque utilizzati per una più graduale comprensione.

Il programma lisp può essere scritto con qualsiasi editor. E' comunque consigliabile disporre di un editor che faciliti la gestione delle parentesi.

Del lisp esistono vari dialetti (scheme è il piú noto), mentre oggi la lingua "principale" si indica propriamente come ""Common Lisp" (abbreviato CL). CLtL1, CLtL2 ed ANSI sono le varie versioni standard del Common Lisp. In questo documento ci si riferisce all'implementazione gcl (GNU Common Lisp), aderente a CLtL2 e parzialmente allo standard ANSI.

Il primo approccio con LISP sarà attraverso una shell interattiva, quindi lanciando il comando "gcl" (oppure "clisp", "alisp", oppure con una voce di menu) ci si troverà di fronte ad un prompt pronto ad eseguire comandi LISP. Nel seguito utilizzeremo la seguente convenzione:

```
>>(lista di simboli separati da spazi)
risultato
```

Dove la prima linea rappresenta un'espressione, tipica nel lisp, digitata al prompt seguita da <invio>. La seconda riga rappresenta la sua valutazione, cioè quanto l'interprete lisp restituisce dopo aver "eseguito" l'espressione. La sencoda linea è riportata solo quando necessario, anche se l'interprete in realtà restituisce sempre un valore.

Sintassi fondamentale

- Il Lisp utilizza la notazione prefissa, valutando le espressioni da sinistra a destra.
- Una espressione è una lista, cioè una sequenza di di simboli o valori (detti atomi) separati da spazi, e delimitata da parentesi tonde.
- Il programma lisp è costituito solo da espressioni, ovvero non contiene direttive (goto, break, ecc.), ma solo espressioni, variamente annidate, che restituiscono un valore (o più).

Ad esempio un'operazione di somma è espressa da:

```
>> (+ 2 2)
4
```

cioè da una lista composta dagli atomi: "+" (operatore, posto per primo vista la notazione prefissa),"2" e "2".

Ogni simbolo viene valutato. Per non far valutare una lista si utilizza l'operatore quote o, più spesso, la sua abbreviaziazione 'apice.

```
>>(quote (+ 2 2))
(+ 2 2)
```

restituisce l'espressione letterale, senza valutarla, ed è equivalente a:

```
>>'(+ 2 2)
(+ 2 2)
```

L'apostrofo (backtick in inglese : <AltGr><apice> sulla tastiera italiana) ha la stessa funzione, con la differenza che sarà possibile rientrare in modo "codice" tramite l'uso della virgola:

```
>>`( 5 + 7 = ,(+ 5 7))
(5 + 7 = 12)
```

I simboli che iniziano con il carattere : (duepunti), sono keyword (etichette), e sono utilizzati nelle strutture, negli array associativi, ecc.

Il simbolo NIL, equivalente della lista vuota '(), ha anche il significato di falso. Sono quindi "true" tutti i valori non-NIL, rappresentati dal carattere T (maiuscolo o minuscolo).

Il lisp è case insensitive, quindi utilizzare caratteri maiuscoli o minuscoli è indifferente.

Il carattere delimitatore dei commenti è il punto-e-virgola.

Una caratteristica particolare è che lo stesso simbolo può rappresentare contemporanemente sia una variabile che una funzione (per setq, defun e lamba si veda più avanti):

```
>>(defun double (x) (* x 2))   ; definisce la funzione double
>>(setq double 2)              ; definisce la variabile double
>>(double double)              ; chiama la funz. double sulla var double
4
```

Per riferirsi esplicitamente alla funzione piuttosto che alla variabile sono disponibli due funzioni:

```
>>(symbol-value 'double)
2

>>(symbol-function 'double)
(LAMBDA-BLOCK DOUBLE (X) (* X 2))
```

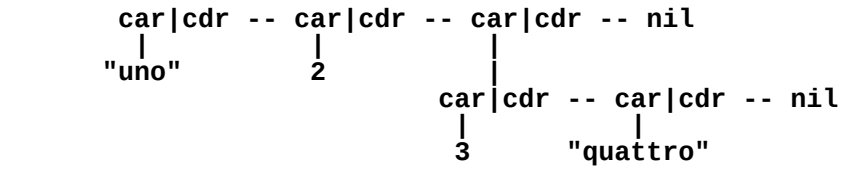
Alcuni concetti di base

Per capire il lisp, oltre a quanto giÃ visto sulla sintassi di base, è sono necessari solo alcuni concetti, primo fra i quali l'idea di lista.

Per le liste è comodo riferirsi mentalmente alla classica lista dinamica (in realtà è la rappresentazione classica a venire dal lisp). Liste sono rappresentabili da una sequenza di coppie di celle dette CONS. Ogni cons contiene due puntatori: CAR e CDR, che indirizzano il contenuto ed il cons successivo. Nel caso di liste annidate il car può puntare ad un ulteriore cons. Il cdr dell'ultimo elemento contiene NIL (è vuoto). Ad esempio la lista:

```
("uno" 2 (3 "quattro"))
```

può essere pensata come:



Il lisp ammette anche una rappresentazione più esplicita (e basica) dell'oggetto CONS, come coppia puntata:

```
(car . cdr)
```

Ad esempio:

```
>>'("uno" . nil )
("uno")
```

Con questa rappresentazione la lista precedente si scrive:

```
("uno" . ( 2 . ( ( 3 . ("quattro" . nil)  ) . nil  ) ) )
```

Ognuno dei mattoni fondamentali (CONS, car e cdr) ammette un'operatore corrispondente:

```
>>(cons 'a 'b)  ; forma un cons
(A . B)

>>(cons 1 (cons 2 (cons 3 nil))) ; forma una lista
(1 2 3)

>>(car '(1 2 3)) ; estrae il primo elemento dalla lista
1

>>(cdr '(1 2 3)) ; estrae la restante parte della lista
(2 3)
```

Da notare come la formazione di liste sia possibile semplicemente reiterando l'operatore cons.

Per completezza citiamo anche l'operatore atom:

```
>>(atom 'a)
```

```
T

>>(atom '(1 2))
NIL
```

che ritorna true se l'oggetto è un'atomo, cioè un singolo simbolo, NIL (cioè false) se è una lista.

L'originale lavoro di McCarthy "Recursive Functions of Symbolic Expressions and Their Computation by Machine", dimostra che questi pochi operatori (quote, atom, cons, car, cdr), a cui si aggiungono l'operazione di uguaglianza (eq), un operatore condizionale (cond) e il concetto di funzione (la lambda-expression) chevedremo in seguito, sono sufficienti a definire completamente il linguaggio lisp.

A tal proposito si veda anche "The Root of Lisp" di Paul Graham.

Le funzioni di calcolo

I valori numerici vengono trattati come interi, frazioni, reali o complessi in funzione del contesto. Ad esempio una operazione tra frazioni è valutata in una frazione:

```
>>(/ 3/5 3)
1/5
```

Esistono quindi delle funzioni di conversione tra le differenti notazioni:

```
>>(rational 1.5)
3/2
```

I numeri complessi si esprimono nella forma:

```
#C(partereale parteimmaginaria)
```

Per comporre un numero complesso esiste la funzione:

```
>>(complex 1 2)
#C(1 2)
```

Mentre per estrarne le componenti:

```
>>(imagpart #C(1 2))
2

>>(realpart #C(1 2))
1
```

La notazione esponenziale si esprime con il carattere S oppure con il consueto carattere E:

```
>>1234S10
1.234S13
>>1234e-3
1.234
```

Oltre alle operazioni di base sono disponibili tutte le funzioni di calcolo più comuni:

```
mod   ; resto
sqrt  ; radice quadrata
(expt base esponente) ; elevamento a potenza
(log num base) ; logaritmo (la base di default è e)
sin   ; analogamente a cos, tan, atan, ecc. l'argomento è in radianti
abs   ; valore assoluto
```

```
round ; arrotondamento intero
min   ; accetta argomenti multipli come anche max
sinh  ; similmente esistono cosh, tanh, atanh ()
lcm   ; minimo comune multiplo , accetta argomenti multipli
gdc   ; massimo comun divisore
```

Sono inoltre predefinite alcune costanti utili nel calcolo, tra cui:

```
PI                  ; pigreco
MOST-POSITIVE-FIXNUM  ; il maggior numero intero disponibile
```

similmente esistono anche most-positive-long-float e most-positive-short-float, least-positive-..., most-negative-..., ecc.

Stringhe

Il carattere delimitatore delle stringhe è il doppio apice. Le stringhe, come le altre costanti, vengono valutate in sè stesse:

```
>>"ciao"
"ciao"
```

Come carattere di escape si usa \ (backslash) quindi per inserire un doppio apice si usa

```
\"
```

mentre per inserire un backslash si usa

```
\\
```

I singoli caratteri sono rappresentati preceduti dalla sequenza #\:

```
#\A
```

Alcune funzioni utili sulle stringhe:

```
>>(char "abc" 1)                    ; estrae il secondo carattere
#\b
>>(concatenate 'string "com" "mon") ; concatena due stringhe
"common"
```

Inoltre le stringhe sono array di caratteri quindi si possono usare tutte le funzioni relative agli array (si veda più avanti)

Variabili

Le variabili globali vengono definite dalle due macro defvar e defparameter. La differenza è che defvar non sovrascriverà eventuali simboli già definiti in precedenza. Per convenzione i simboli associati a variabili globali cominciano e finiscono con un asterisco:

```
(defvar *varglobale* 50.5)
```

Per definire variabili locali si usa l'operatore let, seguito da due liste, la prima di definizione delle variabili e la seconda è l'ambito di validità delle stesse (cioè il codice nel quale le variabili esistono):

```
>>(let ((a 20)(b 15)) (+ a b))
35
```

Per modificare il valore di una variabile si utilizzano gli operatori setq (solo variabili) e setf (più generale):

```
(setq *varglobale* '3/8)
```

che può effettuare anche assegnamenti multipli:

```
(setq pigreco 3.14 radicedue 1.41)
```

Le variabili non sono tipizzate, quindi è possibile assegnare successivamente valori di tipi diversi (ad esempio prima una frazione e poi una stringa). Per verificare che una variabile contenga, in un dato momento, un tipo preciso esiste tutta una serie di funzioni booleane, il cui nome, per convenzione finisce con il carattere "p": floatp, complexp, listp, alpha-char-p, ecc.

Liste

Citiamo solo alcune delle moltissime funzioni disponibili per il trattamento delle liste. Si noti che in generale tali funzioni non hanno effetti collaterali, quindi non modificano il contenuto dei simboli implicati. Ad esempio:

```
>>(setq a '(1 2 3 4 5)) ; assegna ad a la lista (1 2 3..)
(1 2 3 4 5)

>>(cons 0 a)            ; aggiunge alla lista l'elemento 0
(0 1 2 3 4 5)

>>a                     ; ma non modifica il valore di a
(1 2 3 4 5)

>>(setq a (cons 0 a))   ; a meno che non venga utilizzato setq
(0 1 2 3 4 5)
```

Per estrarre elementi (esistono anche first, second, third, ecc.):

```
>>(nth 1 '(a b c d)
B

>>(subseq '(a b c d) 1 3)
(B C)
```

Per concatenare, intersecare, unire, ecc:

```
>> (union '(a b c) '(b c d))
(a b c d)

>> (intersection '(a b c) '(b c d))
(b c)

>> (append '(a) '(b c d) '(e f))
(a b c d e f)
```

Per ottenere la dimensione di una lista:

```
>>(length '(a b c d))
4
```

Per rimuovere un elemento:

```
>>(remove c '(a b c d))
(a b d)
```

Per estrarre il primo elemento o i restanti (analoghe ai già visti car e cdr):

```
>>(first '(a b c d))
A

>>(rest '(a b c d))
(b c d)
```

Inoltre, essendo car e cdr molto utilizzati, esistono anche una serie di operatori combinati, ad esempio cddr equivale a (cdr (cdr ...)), cadr equivale a (car (cdr ...)), ecc.

Per estrarre elementi da una lista:

```
>>(member 'c '(a b c d))
(c d)
>>(find 'a (a b c d))
A
```

Gli operatori push e pop permettono di lavorare su una lista con la logica dello stack:

```
>>(setq a nil)
NIL

>>(push 4 a)
(4)

>>(push 5 a)
(5 4)

>>(pop a)
5
```

L'operatore sort, infine, è molto potente, infatti permette di utilizzare come condizione di ordinamento una funzione booleana arbitraria:

```
>>(sort '(2 3 7 1) '>)
(7 3 2 1)
```

Strutture

Le strutture sono simili a quanto disponibile in C o in Pascal. La macro defstruct definisce una struttura. Ad esempio:

```
(defstruct login nome password)
```

definisce una struttura (login) composta da due campi (nome e password). La macro definisce inoltre la funzione di creazione make-nomestruct :

```
>>(make-login)
#S(LOGIN :NAME NIL :PASSWORD NIL)

>>(setq udata (make-login :name "utente"))
#S(LOGIN :NAME "utente" :PASSWORD NIL)
```

e tutte le funzioni di estrazione dati dalla struttura nella forma nomestruct-nomefield:

```
>>(login-name udata)
"utente"
```

Array & C

Per trattare coppie del tipo chiave-valore (array associativi) esiste la semplice struttura plist (property list) che è trattabile come una normale lista di coppie di elementi:

```
>>(setq capitale '(:italia "roma" :francia "parigi"))      ; crea una plist
(:ITALIA "roma" :FRANCIA "parigi")

>>(getf capitale :francia)                                 ; estrae un valore
"parigi"

>>(setq capitale (append capitale '(:germania "berlino"))) ; aggiunge una cop
(:ITALIA "roma" :FRANCIA "parigi" :GERMANIA "berlino")
```

Molto simile, ma più raffinata, è la struttura chiama hash table, ottimizzata per offrire un tempo di accesso indipendente dalla dimensione.

```
>>(setq capitali (make-hash-table))   ;istanza di una nuova hash table

>>(gethash :francia capitali)         ;richiesta di un valore non definito
NIL
NIL

>>(setf (gethash :francia capitali) "parigi") ;assegnazione di un valore

>>(gethash :francia capitali)                 ;richiesta di un valore definit
"parigi"
T
```

Si noti come per l'assegnazione vada usata setf in luogo di setq, e come la funzione gethash restituisca due valori: il primo è il valore cercato (nil se non definito), mentre il secondo è un booleano, vero se il valore è definito (questo perchè il valore può essere definito NIL).

La sruttura array prevede una rappresentazione in forma di lista annidata preceduta dal simbolo #

```
#((a00 a01 ...a0n) (a10 a11 ...a1n) (am0 am1 ...amn))
```

Anche gli array sono ottimizzati in modo simile alle hash table:

```
>>(setq ary (make-array '(3 3)))    ; istanza di un array 3x3

>>(setf (aref ary 0 0) "valore00" ) ; assegnazione dell'elemento 0 0

>>(aref ary 0 0)                    ; richiesta dell'elemento 0 0
"valore00"

>>(array-rank ary)                  ; ritorna l'ordine
2
```

Sono inoltre disponibili delle funzioni booleane (NOT,OR,ecc.) che operano su array binari.

Per gli array monodimensionali (vettori) sono disponibili alcune funzioi semplificate:

```
>>(setq v (vector 1 2 3))   ; assegna a v  un vettore
#(1 2 3)
>>(svref v 0)               ; estrae il primo componente
1
```

Le funzioni

Le funzioni sono definite dalla macro defun seguita dal nome, dalla lista degli argomenti e dal

corpo della funzione (si noti che il nome della funzione non viene valutato come invece dovrebbe, essendo appunto defun una macro e non un semplice operatore):

```
(defun incrementa (a) (+ a 1))
```

Quindi:

```
>>(incrementa 5/4)
9/4
```

La lista che definisce il prototipo di una funzione è chiamata lambda-expression:

```
(lambda (parametri) funzione)
```

e può essere utlizzata esplicitamente per definire temporaneamente una funzione:

```
>>( (lambda (x) (* 2 x)) 2)
4
```

oppure (espressione più complessa) nella creazione di una funzione che restituisca una funzione:

```
>>(defun make-adder (n) #'(lambda (x) (+ n x)))
>>(setq add2 (make-adder 2))
>>(funcall add2 3)
5
```

La definizione della funzione incrementa, usando una lambda-expression invece della macro defun, andrebbe scritta:

```
(setf (symbol-function 'incrementa) '(lambda (a) (+ a 1)))
```

Per definire una lista di argomenti di dimensione variabile si usa l'operatore &rest. Ad esempio:

```
(defun funzione (a &rest b) ...)
```

assegna il primo elemento al simbolo a ed i restanti alla lista b.

Per definire dei parametri opzionali con i rispettivi valori di default si usa l'operatore &optional. Ad esempio:

```
(defun funzione(&optional(simbolo val-default)) ...)
```

assegna a simbolo il valore val-default in caso di mancanza di argomenti.

Una caratteristica molto interessante è la possibilità di definire degli argomenti opzionali e dotati di nome, che prendono il nome di argomenti keyword:

```
(defun funzione (&key (nome1 var1)(nome2 var2)...) (...codice))
```

In questo modo la funzione potrà essere chiamata definendo gli argomenti, opzionali, in ordine casuale:

```
(funzione :nome2 valore2)
(funzione :nome2 valore2 :nome1 valore1)
```

Se non viene specificato il nome corrisponderà al nome della variabile:

```
>>(defun foo (&key x y) (cons x y))
>>(foo :X 3)
(3)
```

L'operatore funcall serve a richiamare una funzione in modo esplicito. Ad esempio:

```
(funcall 'sqrt 4)
```

è perfettamente equivalente a:

```
(sqrt 4)
```

In questo caso ovviamente l'uso è superfluo, funcall è però necessario nel caso di funzioni utilizzate come argomenti di funzioni. Ad esempio per definire una funzione che richiami due volte una seconda funzione argomento:

```
(defun tt (f x) (funcall f (funcall f x)))
>>(tt 'sqrt 16)
2.0
```

L'operatore apply esegue una funzione utilizzando come argomenti gli elementi una lista:

```
>>(setq a '(1 2 3))
>>(apply '+ a)
6
```

L'operatore mapcar itera una funzione sugli elementi di una lista

```
>>(mapcar 'sqrt a)
1.0 1.4142135623730951 1.7320508075688772)
```

Evidentemente la funzione usata da apply deve ammettere argomenti multipli, mentre mapcar utilizza funzioni con argomenti singoli. Alcuni lisp (Allegro CL) richiedono anche il carattere cancelletto, #, per identificare una funzione utilizzata come argomento (cioè per estrarre il puntatore alla funzione dal simbolo associato):

```
(mapcar #'sqrt a)
```

Iterazione

Il più semplice operatore di loop è appunto loop, che ripete le istruzioni che lo seguono (in nuumero arbitrario). Il loop termina con l'istruzione return.

```
>>(loop (setq a (+ a 1)) (when (> a 7) (return a)))
8
```

Più complessa è la struttura do:

```
(do ((var init step)(var init step)...) (endtest result) (statement)(statemen
```

dove la prima lista identifica le vaiabili locali, il loro valore iniziale e la modalità di incremento, la seconda lista definisce la condizione di termine e il valore di ritorno, seguono le istruzioni eseguite nel loop. Ad esempio:

```
>>(do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 3) y)
    (print y)
    (print 'working))
1
WORKING
2
WORKING
4
WORKING
8
```

Per effettuare cicli su liste è disponibile l'operatore dolist:

```
(dolist (var lista ret) (...codice))
```

che reitera il codice assegnando a var tutti i valori della lista. L'argomento ret, opzionale, è restituito come valutazione dell'intera espressione.

L'operatore dotimes invece serve a realizzare cicli su numeri interi. La seguente espressione equivale ad un ciclo for per n da 0 a 9 (compresi):

```
(dotimes (n 10) (...codice))
```

Anche per dotimes la lista di inizializzazione può contenere un terzo simbolo, che sarà il valore di ritorno della intera espressione.

Per concatenare più operazioni è disponibile l'operatore progn, che valuta tutte le espressioni che lo seguono, e restituisce il valore dell'ultima:

```
>>(progn (setq a 5)(setq b 6)(+ a b))
11
```

Istruzioni condizionali

Esistono diversi operatori di confronto:

```
>>(eq 'a 'A)                ; confronto tra simboli
T
>>(= 3 4)                   ; confronto tra numeri
NIL
>>(equal '(1 2 3) '(1 2 3)) ; confronto tra liste
T
```

Inoltre l'operatore eql si comporta come = per i numeri e come eq per i simboli.

L'operatore if necessita sempre di entrambe le condizioni (true e false):

```
(if test val-true val-false)
```

dove val-true/false possono ovviamente essere ogni simbolo o lista, quindi anche altro codice.

Se invece non serve una delle due condizioni si usano gli operatori when e unless:

```
(when test val-true)
(unless test val-false)
```

A differenza di if gli operatori when e unless ammettono un numero arbitrario di istruzioni:

```
>>(when t (setq a 5) (+ a 6))
11
```

quindi non è necessario ricorrere all'operatore progn.

Se invece le condizioni sono molte esiste l'operatore cond:

```
(cond ((test1 val1)
       (test1 val2)
       ...)))
```

Ad esempio si veda la classica funzione ricorsiva di calcolo del fattoriale (la condizione "t" posta alla fine si utilizza come default):

```
(defun fat (n)
    (cond ((= n 1) 1)
          (t (* n (fat (- n 1)))
          )
    )
)
```

Per un confronto tra molti valori è disponibile l'operatore case:

```
(case var (val1 ret1)(val2 ret2)...)
```

che valuta il ret corrispondente al val corrente di var.

```
(case a
        (0 "zero")
        (1 "uno")
        (2 "due")
)
```

Gli operatori and e or, molto utilizzati nel controllo di flusso, accettano un numero arbitrario di argomenti, ed effettuano il confronto in modo sequenziale tra di essi.

Input/output

Per la gestione degli stream sono disponibili i consueti comandi open, close, read, ecc.

```
>>(setq filep (open "nomefile")) ; apre uno stream in lettura da file
>>(read-line filep)              ; legge una linea come stringa
"prima linea del file"
>>(close filep)                  ; chiude lo stream
```

Naturalmente la funzione open ammette alcuni parametri tra i quali:

```
DIRECTION   :INPUT, :OUTPUT, :IO, :PROBE
IF-EXISTS   :ERROR, :NEW-VERSION, :RENAME, :RENAME-AND-DELETE,
            :OVERWRITE, :APPEND, :SUPERSEDE, NIL
IF-DOES-NOT-EXIST  :ERROR, :CREATE, or NIL
```

Gli stream *standard-input* e *standard-output* sono predefiniti come variabili globali.

La più semplice funzione di output disponibile è print, che stampa qualsiasi oggetto o sequenza di oggetti nella sua reppresentazione lisp:

```
(print object stream)
```

se non specificato stream si assume *standard-output*. Si noti che eseguendo una print l'interprete restituisce una stampa doppia, cioè la stampa e la valutazione del comando.

Esistono inoltre una serie di varianti della print: prin1, princ, pprint, oltre alla più generale funzione write.

La funzione format formatta una stringa in modo simile ad una printf del linguaggio C:

```
(format t "hello world ~a~%" nome)
```

La stringa formattata verrà assegnata al simbolo che appare come primo parametro. Se questo è t l'output sarà diretto verso lo *standard-output*, se invece è nil la stringa verrà semplicemente restituita come valore dell'espressione. La stringa di controllo può contenere vari caratteri speciali:

```
~A ~nA       stampa l'argomento come farebbe il comando PRINC
~S ~nS       stampa l'argomento come farebbe il comando PRIN1
~D ~B ~O ~X  stampa un decimale, binario, ottale, o esadecimale
~C           stampa un carattere
~F           stampa un numero in virgola mobile
~&           ritorno a capo
```

L'eventuale n definisce il numero di caratteri da utilizzare.

Oltre alle funzioni di input/output da file e da terminale molte implementazioni includono librerie grafiche per rendere disponibili funzioni di gestione di finestre, bottoni, ecc. Ad esempio il gcl include gcl-tk per il noto TCL/TK.

L'uso in pratica

Questa sezione è riferita specificatamente all'implementazione gcl (GNU Common Lisp). Ciononostante buona parte di quanto riportato resta valido sostituendo al comando "gcl" quello dell'implementazione CL in uso (alisp, clisp, ecc.)

La variabile globale

```
si::*command-args*
```

conterrà la riga di comando, ovvero tutti gli eventuali parametri o argomenti con cui il programma è stato invocato, ovviamente nella normale forma di lista propria del linguaggio:

```
(comando parametri... argomenti...)
```

L'uso come linguaggio di scripting prevede di salvare il programma come un semplice file di testo, eventualmente con l'estensione convenzionale .lsp. Il file sarà eseguito dal comando:

```
gcl -f script.lsp
```

Nella maggior parte dei sistemi unix-like Ã¨ possibile rendere il file direttamente eseguibile (chmod 755) ed inserire una prima linea può che richiami l'interprete (attenzione al percorso: il normale comando "gcl" può essere uno shell-script oppure un link). Ad esempio per il gcl su linux:

```
#!/usr/lib/gcl-2.6.6/unixport/saved_ansi_gcl -f
```

In caso di dubbio la variabile si::*system-directory* contiene la directory in cui si trova l'eseguibile.

La prima linea verrà ignorata anche se non contiene il percorso per l'interprete. Va quindi in ogni caso lasciata vuota (o occupata da un commento).

La compilazione si può effettuare dall'interno dell'interprete (o da codice!), tramite il comando:

```
(compile-file "filesorgente.lsp" :output-file "fileoggetto")
```

Oppure da shell digitando il comando:

```
gcl -compile filesorgente.lsp
```

In entrambi i casi se non si specifica il file di output verrà utilizzato lo stesso nome del file di input con estensione .o

Specifica del gcl è la possibilità di mantenere i file intermedi in codice c tramite i flag (disponibili anche nell'analoga funzione già vista):

```
gcl -compile filesorgente.lsp -c-file -h-file
```

..il flag system-p e l'uso nel C

Sempre dall'interno dell'interprete è disponibile anche il comando:

```
(compile nomefunzione lambda-expr)
```

che compila una funzione temporaneamente, senza scrivere alcun file, allo scopo di velocizzarne l'esecuzione. Il parametro lambda-expr è opzionale e contiene la definizione della funzione, ovviamente se non presente la funzione deve già essere definita.

Il caricamento dinamico di codice si effettua tramite la funzione load. La funzione load permette di caricare il codice salvato in un file, indifferentemente sorgente o binario. Ad esempio se la definizione:

```
(defun double (a) (* 2 a))
```

è salvata nel file /usr/lib/clcommon/double.lsp, il comando

```
(load "/usr/lib/clcommon/double.lsp")
```

renderà disponibile la funzione double.

Durante l'interazione con l'interprete si rivela molto comoda la funzione describe:

```
>>(describe 'defun)
DEFUN - external symbol in LISP package
DEFUN [Special form and Macro]

>>(describe (symbol-function 'double))
(LAMBDA-BLOCK DOUBLE (A) (* 2 A)) - function

>>(describe (symbol-value 'a))
3 - fixnum (32 bits)
```

Naturalmente l'output esatto può cambiare in funzione dell'implementazione utilizzata

La funzione step invece valuta l'argomento passo-passo ed in modo interattivo:

```
>>(step (+ (* 3 4)(/ 6 7)))
Type ? and a newline for help.
  (+ (* 3 ...) ...)
(* 3 ...)
3
4
    = 12
    (/ 6 ...)
6
7
    = 6/7
  = 90/7
90/7
```