Common LISP (autore: Vittorio Albertoni)

Premessa

Il linguaggio di programmazione LISP nacque verso la fine degli anni cinquanta del secolo scorso, quando i computer si programmavano per lo più in linguaggio assembly o, per le applicazioni scientifiche, in linguaggio Fortran.

Erano anche gli anni in cui si stava ormai superando la fase iniziale dello sviluppo dell'intelligenza artificiale, campo nel quale con il linguaggio assembly non si sarebbe andati da nessuna parte e con il linguaggio Fortran, orientato esclusivamente al calcolo numerico, non si riusciva ad elaborare, oltre ai numeri, anche i simboli, come richiesto dall'intelligenza artificiale.

Il linguaggio LISP è nato per rimediare a questo stato di cose.

Verso la metà degli anni cinquanta del secolo scorso Allen Newell, J. C. Shaw e Herbert Simon svilupparono presso la RAND Corporation una tecnica, da loro denominata «list processing», creando le basi per un linguaggio adatto alla manipolazione di liste e simboli, due importanti tipi di dato nel campo dell'intelligenza artificiale, denominato IPL (Information Processing Language).

E' da queste basi che John McCarthy creò il linguaggio LISP, chiamato così per LISt Processing, in omaggio al nome dato alla tecnica dai tre pionieri, che vide la luce alla fine del 1958 e divenne diffusamente utilizzato con la versione 1.5 del 1962.

Già nei primi anni di vita il LISP si dimostrò molto versatile per arrangiamenti di vario tipo: a partire dagli anni sessanta del secolo scorso, al LISP 1.5 di McCarthy si affiancarono lo Standard LISP, il MacLISP sviluppato presso il MIT, il LISP 1100 sviluppato presso la UNIVAC, l'InterLISP sviluppato presso la Xerox e lo StanfordLISP sviluppato presso l'Università di Stanford.

Nel 1970 vede la luce una versione del LISP con l'obiettivo di unificare un po' tutto: si chiama Scheme.

Ma prosegue comunque lo sviluppo di dialetti vari, tanto che attorno al 1980 se ne contano una dozzina.

Ulteriore e forse definitivo tentativo unificante avviene nel 1984 con la nascita del Common LISP, tuttora prevalentemente usato, al quale è dedicato questo manualetto.

Degno di nota, ad opera di Lutz Műller nel 1991, lo sviluppo di un fork semplificante del LISP originario, il NewLISP, sintesi di tutto il LISP fino allora noto con selezione di ciò che serve in maniera più ricorrente per fare cose non troppo complicate, comunque potente ma con modesto utilizzo di risorse[1].

Alla selezione si aggiunge l'arricchimento con funzioni preconfezionate, non presenti nel linguaggio originario, soprattutto nel campo del calcolo numerico (statistico, finanziario, matriciale).

Secondo alcuni il NewLISP non si può nemmeno considerare una derivata del LISP ma un nuovo linguaggio che eredita lo stile del LISP.

[1] Al NewLISP ho dedicato il manualetto «newlisp» allegato al post «Newlisp» pubblicato nell'ottobre 2019 sul mio blog all'indirizzo *www.vittal.it*.

Indice

1 Predisposizione del computer

In genere i linguaggi di programmazione dispongono di interpreti o compilatori in un unico esemplare, come accade per Python, Julia, Perl, Go, Java, Raku, oppure in più di un esemplare, comunque nel limite di poche unità, come avviene per C, C++ e Pascal.

Per Common LISP, in particolare per chi lavora su Linux, abbiamo invece l'imbarazzo della scelta.

Caratteristica comune è la presenza di un interprete che agisce su REPL (Reed-Eval-Print-Loop), cioè in una shell dove si scrivono i comandi per vederli immediatamente eseguiti, e di un compilatore, generalmente dedicato alla conversione dei comandi in linguaggio macchina in modo da velocizzarne l'esecuzione.

Per lavorare con il LISP dobbiamo installare sul computer almeno un interprete/compilatore. In gergo si dice che dobbiamo avere sul computer una implementazione del Common LISP.

Qui descrivo le implementazioni più diffuse e di più facile acquisizione nel mondo del software libero, tutte gratuite.

1.1 CMUCL

E' la più anziana ed esiste per il sistema UNIX dal 1980, cioè da prima ancora che esistesse il Common LISP, col nome CMU e venne adattata al Common LISP nel 1985 assumendo il nome CMUCL, acronimo di Carnegie Mellon University Common Lisp.

Il suo compilatore si chiama Python, omonimia assolutamente casuale e che non ha nulla a che vedere con il linguaggio di programmazione Python che, ai tempi, nemmeno c'era.

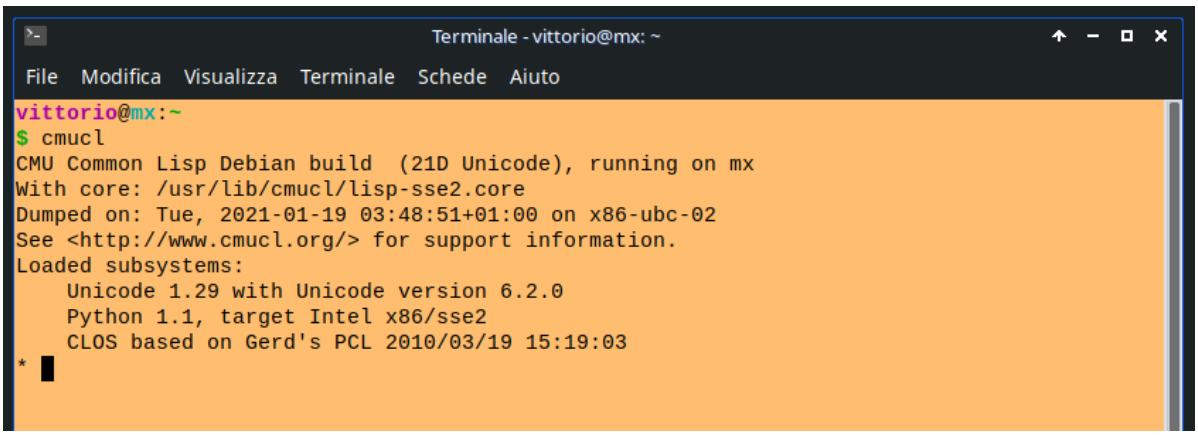
La troviamo all'indirizzo *https://www.cons.org/cmucl/*. Nella pagina del DOWNLOAD, aprendo la directory /RELEASE, troviamo i file di installazione per Linux e Mac. L'ultima versione è la 21e del 14 maggio 2023.

Chi usa Linux quasi sicuramente trova ciò che serve nel repository della sua distro.

Probabilmente è l'implementazione di Common LISP che fornisce la performance più elevata.

Non è disponibile per Windows.

Una volta installata viene avviata con il comando a terminale `cmucl` e si presenta così



Per uscire si scrive il comando `(quit)`.

1.2 SBCL

E' un fork di CMUCL e, dal momento che il Carnegie e il Mellon che danno il nome all'Università in cui questo è nato erano, rispettivamente, un re dell'acciaio e un grande banchiere, il suo nome è l'acronimo di Steel Bank Common Lisp.

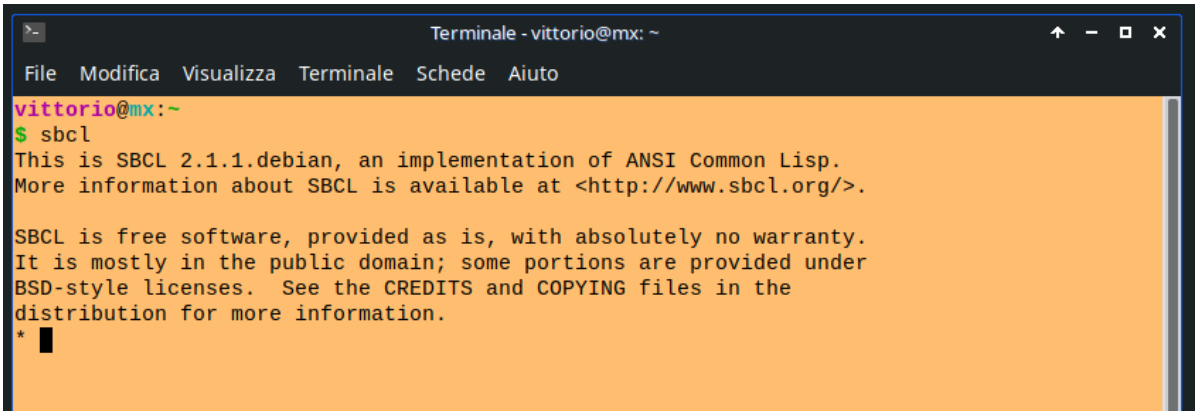
Lo troviamo all'indirizzo *https://www.sbcl.org/*. Nella pagina del DOWNLOAD troviamo una comoda platform table con caselle corrispondenti alle varie versioni disponibili per vari sistemi operativi (oltre a Linux, anche Mac e Windows). Cliccando sulla casella corrispondente alla

nostra situazione ci procuriamo quanto necessario per installare l'implementazione sul nostro computer.

Al solito, chi usa Linux molto probabilmente trova il tutto nel repository della sua distro.

Chi usa Windows, soprattutto se si tratta di un Windows datato, può trovare installer adatti anche all'indirizzo *https://github.com/akovalenko/sbcl-win32-threads/wiki*.

Una volta installata viene avviata con il comando a terminale `sbcl` e si presenta così



Per uscire si scrive il comando `(quit)`.

1.3 GCL

Sta per GNU Common Lisp ed è l'implementazione di Common LISP nel progetto GNU.

Il suo compilatore ha la particolarità di produrre codice oggetto nativo generando innanzi tutto codice C e attivando poi un compilatore C.

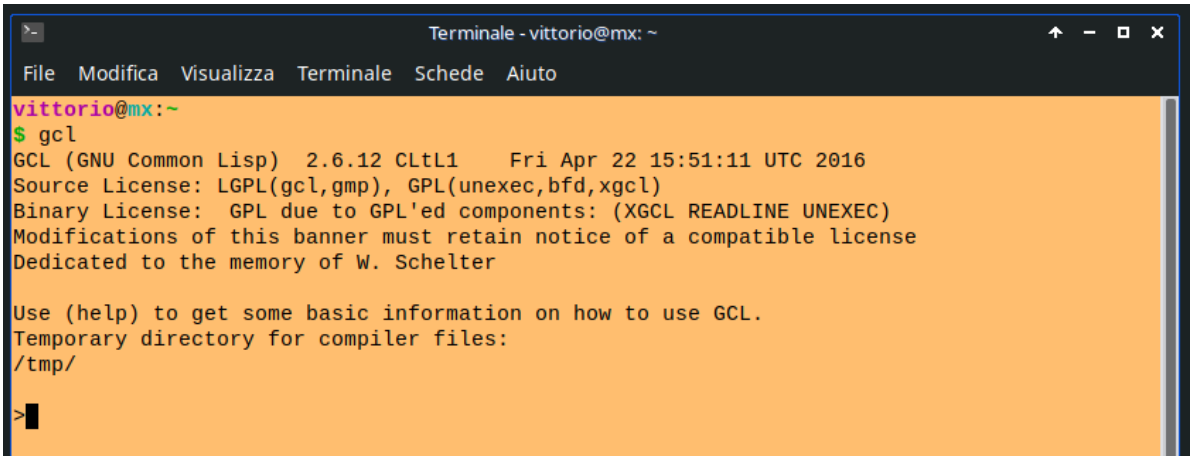
Questa particolarità lo rende adatto per lo sviluppo di grandi progetti impegnativi.

Si trova all'indirizzo *https://www.gnu.org/software/gcl/*.

Cliccando sulla voce di menu GET LATEST RELEASE apriamo una pagina in cui troviamo gli installer per Linux e Windows.

Se lavoriamo su Linux possiamo installare la versione presente nel repository della nostra distro, che viene in genere proposta nella sua intera vastità, comprendente anche il tool Tk per creare programmi dotati di interfaccia grafica utente.

Una volta installata viene avviata con il comando a terminale `gcl` e si presenta così



Per uscire si scrive il comando `(quit)`.

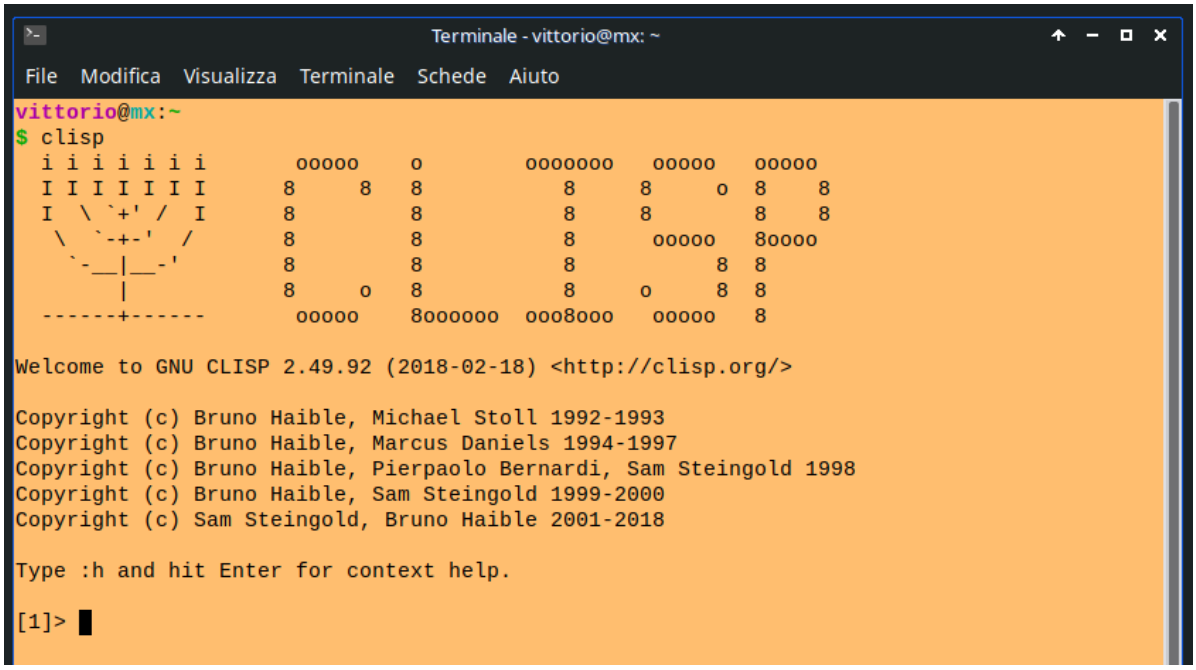
1.4 CLISP

Altra implementazione che ci offre il progetto GNU, è il parente povero di GCL. Lo possiamo ritenere una versione di GNU Common Lisp per dilettanti.

Lo troviamo all'indirizzo *https://www.gnu.org/software/clisp/*.

E' presente nei repository delle distro Linux e possiamo usarla su Windows attraverso Cygwin o MinGW/MSYS e installarlo su Mac attraverso MacPorts.

Una volta installata viene avviata con il comando a terminale `clisp` e si presenta così



Per uscire si scrive il comando `(quit)`.

* * *

Direi che la carrellata può finire qui.

Esistono molte altre implementazioni non altrettanto facili da trovare come le quattro che ho citato o non altrettanto alla portata anche di principianti (come alcune il cui compilatore traduce semplicemente il codice LISP in codice C per poi essere utilizzato in programmi compilati in C).

Delle quattro implementazioni che ho presentato l'unica che possiamo avere senza complicazioni su tutti i tre sistemi operativi più diffusi (Windows, Mac OS e Linux) è SBCL, che diventa quella consigliata per utilizzare questo manualetto.

Ottima, forse migliore, alternativa per chi usa Linux ritengo sia CLISP.

2 Come funziona

Nella REPL, la shell interattiva, possiamo vedere eseguite le istruzioni nel momento in cui le immettiamo premendo Invio dopo averle compiutamente inserite.

Tutto ciò che facciamo nella REPL, tuttavia, nel momento in cui usciamo con il comando `(quit)` viene completamente dimenticato.

Se vogliamo creare un programma per riutilizzarlo quando serve dobbiamo scrivere i comandi che lo compongono in un file di testo e salvarlo. Non necessariamente, ma bene farlo per qualificare il file, diamo al file l'estensione `.lsp` (alcuni preferiscono usare l'estensione `.lisp`)[2].

E' buona norma non inserire comandi nella prima riga del file. Questa riga va lasciata vuota o può essere utilizzata per un commento (scritta preceduta dal punto e virgola (;)) o, in Linux, per la shebang, di cui parleremo tra poco.

[2]L'abitudine all'uso di estensioni di tre lettere risale ai tempi in cui le prime edizioni di Windows non supportavano estensioni di file di più di tre lettere.

Se usiamo l'implementazione SBCL, l'esecuzione del programma può avvenire, essendo posizionati nella directory dove abbiamo salvato il file del programma, scrivendo a terminale (quello che in Windows si chiama prompt dei comandi) il comando
`sbcl -- script` <nome_file_programma>[3].

Per il sistema operativo Linux possiamo scrivere nella prima riga del file di programma la shebang, in modo che il programma stesso possa essere eseguito semplicemente richiamando a terminale il file che lo contiene, previa abilitazione all'esecuzione con `chmod 555`.
La shebang per l'implementazione SBCL è
`#! /usr/bin/sbcl --script` [4] .

Altra possibilità di eseguire i nostri programmi è quella di caricarli nella REPL con la funzione `load`, usando questa sintassi
`(load` <file_programma>`)`
dove <file_programma> è, tra doppi apici, il percorso al file del programma.

Nel caso dell'implementazione GCL ciò funziona solo previa compilazione del file di programma.

Compilazione

Il primo, spesso unico, obiettivo della compilazione è quello di tradurre il nostro script di programma in linguaggio macchina in modo da poterlo eseguire più rapidamente.

Con la velocità che sviluppano i processori che troviamo oggi installati sui computer si tratta di una facilitazione di cui nemmeno ci accorgiamo, soprattutto se abbiamo a che fare con programmini da dilettanti.

La compilazione si può riferire ad una parte del programma, come una funzione o una macro che abbiamo nella REPL, con la sintassi
`(compile` <identificativo_funzione>`)`.
In questo modo la compilazione avviene senza produrre alcun file e viene semplicemente ritenuta nella REPL in modo che la versione compilata sia disponibile per il successivo richiamo della funzione nella REPL stessa.

Possiamo compilare un intero file di programma con la sintassi
`(compile-file` <file_programma>`)`
dove <file_programma> è, tra doppi apici, il percorso al file del programma.
In questo caso, nella stessa directory dove si trova il file del programma viene prodotto il programma compilato, consistente, nel caso dell'implementazione SBCL, in un file con estensione `.fasl` [5].

Il file compilato viene eseguito con le stesse modalità che abbiamo visto per il file di programma non compilato.

* * *

Abbiamo visto che tra le varie possibili implementazioni esistono alcune particolarità di funzionamento.

Tutte riconoscono tuttavia allo stesso modo il linguaggio Common LISP, le cui basi cerco di descrivere nelle pagine seguenti, in modo che anche un dilettante qualsiasi possa essere in grado di produrre qualche piccolo programma utilizzando questo linguaggio altamente professionale.

---

[3] Questa modalità non funziona nel caso dell'implementazione CMUCL. Nell'implementazione GCL il comando per eseguire un programma è `gcl -f`, seguito dal nome del file e nell'implementazione CLISP è `clisp`, seguito da nome del file.

[4] Questa modalità non funziona nel caso dell'implementazione CMUCL.
Nell'implementazione GCL la shebang è
`#! /usr/lib/gcl-2.6.12/unixport/saved_ansi_gcl -f`
e nel caso di CLISP è
`#! /usr/bin/clisp`

[5] Nell'implementazione CMUCL l'estensione del file è `.sse2f`, nell'implementazione GCL l'estensione del file è `.o`, trattandosi di un file oggetto utile per trattamento con linguaggio C e nell'implementazione CLISP l'estensione del file è `.fas`

3 Premessa al linguaggio

Il LISP è un linguaggio diverso da tutti gli altri.

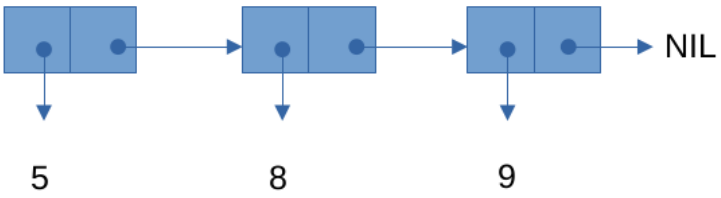
Questa diversità è dovuta al modo tutto originale con cui LISP utilizza la memoria del computer, attraverso il costrutto CONS.

Esso è composto da due metà.

Quella di sinistra, denominata CAR, contiene un puntatore al dato. Quella di destra, denominata CDR, contiene un puntatore al successivo CONS [6] .

Attraverso il puntatore al successivo CONS si crea la struttura di base del linguaggio LISP, la lista (ricordo che LISP sta per LISt Processing). Quando questo puntatore punta a nulla (NIL) la lista termina.

Graficamente possiamo esprimere tutto questo così



Questo costrutto nella memoria del computer viene espresso nel linguaggio LISP in questo modo

(5 8 9)

che è una lista che contiene gli elementi 5, 8 e 9.

Utilizzando il linguaggio LISP la possiamo costruire con l'istruzione

```
(cons 5 (cons 8 (cons 9 nil)))
```

o, più semplicemente ricorrendo alla funzione `list`, con l'istruzione

```
(list 5 8 9).
```

Il fatto che la lista sia costituita da un insieme di puntatori, tutto ciò che facciamo utilizzando la lista, che è la base di tutto ciò che facciamo in LISP, avviene con l'uso di puntatori.

La conseguenza più rilevante di questo è che il passaggio dei parametri alle funzioni del linguaggio avviene sempre per riferimento e mai per valore [7] .

La lista può contenere tra i suoi elementi altre liste. In tal modo si creano liste annidate e attraverso gli annidamenti si possono creare alberi.

Tutto questo strano impianto è finalizzato a poter elaborare con il linguaggio non solo dati numerici ma anche simboli, come richiesto da applicazioni che abbiano a che fare con l'intelligenza artificiale.

Ma si è così creato un linguaggio molto difficile da imparare e molto laborioso da utilizzare, soprattutto quando si voglia affrontare l'elaborazione simbolica.

Linguaggio difficile anche da leggere a causa della presenza di tutte le parentesi necessarie a definire le liste che compongono il programma, il che rende il linguaggio molto lontano dal modo umano di rappresentare operazioni, formule e funzioni, anche per l'uso della Reverse Polish Notation, che antepone gli operatori, in realtà funzioni, agli operandi. In compenso questo lo rende molto vicino al linguaggio macchina e semplifica la compilazione a vantaggio della velocità di elaborazione.

In questo manualetto si trova quanto serve per familiarizzare con le basi del linguaggio.

[6] Anche le più strane espressioni hanno una spiegazione. CONS sta per Constructor. CAR e CDR derivano dalla terminologia relativa al primo computer su cui fu utilizzato il LISP, un IBM 704, i cui registri avevano più componenti, due dei quali erano chiamati Address Part e Decrement Part: da qui deriva CAR, che sta per Contents of the Address part of Register, e CDR, che sta per Contents of the Decrement part of Register.

[7] Nel linguaggio C, dove normalmente il passaggio dei parametri avviene per valore, per poter passare i parametri per riferimento si sono dovuti inventare i puntatori, che in LISP sono connaturati.

4 Il linguaggio

Il Common LISP è un linguaggio funzionale che riconosce determinati tipi di dato e li elabora attraverso funzioni precostituite nel linguaggio o sviluppabili dall'utente.

Per sapere tutto su questo linguaggio andiamo all'indirizzo *https://lisp-lang.org/*.

Per sapere il minimo che può servire per fare qualche cosa possiamo fermarci qui.

4.1 Tipi di dato

4.1.1 Numero

I valori numerici vengono trattati dal Common LISP come interi, frazioni, reali o complessi in funzione del contesto.

Intero Si scrive con uno o più caratteri numerici e viene letto e ritornato come tale.

Frazione Si scrive con caratteri numerici separati da `/` e viene letta e ritornata come tale.

Reale Si scrive con caratteri numerici utilizzando il punto (`.`) come separatore decimale e viene letto e ritornato come tale.

Complesso Si scrive avvalendosi della funzione `complex` così utilizzata:
`(complex` <parte_reale> <parte_immaginaria>`)`
e viene letto e ritornato come
`#C(`<parte_reale> <parte_immaginaria>`)`.
Le singole parti sono estraibili rispettivamente con le funzioni `realpart` e `imagpart`.

4.1.2 Carattere

E' il carattere singolo. Si scrive con un carattere preceduto da `#\` e viene letto e ritornato come tale.

Sono caratteri validi `#\space` e `#\newline`, ideati apposta per rendere visibili i caratteri spazio e accapo.

4.1.3 Stringa

E' una successione di caratteri scritta includendo la successione stessa tra doppi apici e viene letta e ritornata come tale. In realtà si tratta di un array di caratteri.

4.1.4 Simbolo

E' composto da uno o più caratteri alfanumerici scritti dopo un apice singolo (`'`).

4.1.5 Lista

E' una successione di elementi separati da uno spazio racchiusa tra parentesi tonde e preceduta da apice singolo. Viene letta e ritornata come tale.

4.1.6 Array

Un array monodimensionale, cioè un vettore, si può creare utilizzando la funzione `vector` così:
`(vector` <primo_elemento> <secondo_elemento> `.....)`
Quello così costruito è un vettore a dimensione fissa e viene letto e ritornato tra parentesi tonde preceduto dal carattere #.

Più in generale l'array a dimensione fissa si predispone con la funzione `make-array` con la sintassi
`(make-array '(`<righe> <colonne> <altra_dimensione> `....))` e si può inizializzare l'array stesso attraverso l'opzione `:initial-contents` seguita da una lista contenente le liste rappresentanti ciascuna riga.

Con l'istruzione
```
(make-array '(2 2) :initial-contents '((3 4) (2 8)))
```
si crea la matrice

$$\begin{vmatrix} 3 & 4 \\ 2 & 8 \end{vmatrix}$$

L'array multidimensionale così creato viene letto e ritornato tra parentesi tonde preceduto dal carattere # e da un numero che indica la dimensione.
Nel caso dell'esempio viene ritornato
```
#2A((3 4) (2 8))
```
Per costruire un array a dimensione variabile occorre utilizzare l'opzione `:fill-pointer` seguita dall'indirizzo del punto da cui è possibile ampliare l'array.

4.1.7 Dizionario

E' una successione di elementi accoppiati a una parola chiave. E' detto anche array associativo o plist (property list).

E' racchiuso tra parentesi tonde precedute da apice singolo.

La parola chiave è preceduta da : e viene letta e ritornata come tale.

4.2 Variabili

La variabile è la posizione di memoria in cui viene allocato un dato di un tipo qualsiasi perché possa essere elaborato o cambiato, comunque tenuto a disposizione nel corso delle elaborazioni che deve fare il programma.

La variabile si definisce con la funzione `defvar` in questo modo
`(defvar` <nome_variabile>`)`
Se a <nome_variabile> si fa seguire un valore la variabile viene inizializzata.

Alla variabile si assegna un valore con la funzione `setq` in questo modo
`(setq` <nome_variabile> <valore>`)`
Se la variabile non è ancora stata definita essa viene qui definita e inizializzata. In certe implementazioni (come in SBCL) ciò genera un warning sul fatto che la variabile non era stata definita. In certe altre (come in CLISP) tutto fila senza problemi.

La variabile il cui nome è racchiuso tra i caratteri `*` è definita come variabile globale.

La variabile si tipizza automaticamente a seconda del tipo di valore che le viene assegnato.

4.3 Costanti

La costante è la posizione di memoria in cui viene allocato un dato di un tipo qualsiasi perché possa essere tenuto a disposizione, invariato, nel corso delle elaborazioni che deve fare il programma.

La costante si definisce e si inizializza con la funzione `defconstant` in questo modo
`(defconstant` <nome_costante> <valore>`)`

La costante il cui nome è racchiuso tra i caratteri `*` è definita come costante globale.

4.4 Funzioni precostituite

Qualsiasi cosa facciamo con il linguaggio LISP la facciamo utilizzando una funzione. Persino la somma tra due numeri, che altrove eseguiamo utilizzando l'operatore +, qui la facciamo utilizzando la funzione +.

Il Common LISP ci offre decine e decine di funzioni preconfezionate per fare tantissime cose e ci offre pure il modo di creare noi altre funzioni, e vedremo come.

Qui richiamo le funzioni preconfezionate di più ricorrente uso.

4.4.1 Calcolo numerico

`+`	addizione
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione (tra interi resto 0 o tra reali)
`mod`	resto di divisione
`truncate`	parte intera di un numero reale
`round`	arrotondamento di un reale all'intero più vicino
`floor`	arrotondamento di un reale per difetto
`ceiling`	arrotondamento di un reale per eccesso
`abs`	valore assoluto
`sqrt`	radice quadrata
`expt <base> <esponente>`	elevamento a potenza
`log <numero> <base>`	logaritmo (se la base non è indicata logaritmo naturale)
`sin cos tan`	funzioni trigonometriche per angolo in radianti
`asin acos atan`	funzioni trigonometriche inverse
`sinh cosh tanh`	funzioni iperboliche
`asinh acosh atanh`	funzioni iperboliche inverse
`min`	valore minimo di una successione di numeri
`max`	valore massimo di una successione di numeri
`lcm`	minimo comune multiplo di una successione di numeri
`gcd`	massimo comun divisore di una successione di numeri

Per agevolare determinati calcoli il numero pi greco è memorizzato nella costante di sistema `PI`.

4.4.2 Comparazione

`=`	uguale
`/=`	diverso
`<`	minore
`>`	maggiore
`<=`	minore o uguale
`>=`	maggiore o uguale

I simboli indicati valgono per la comparazione tra valori numerici.

Se al simbolo si fa precedere char o string senza spazi intermedi (esempio char= o string<) la comparazione vale, rispettivamente, per caratteri e stringhe.

Esclusivi per la comparazione tra valori numerici sono

`zerop`	uguale a zero
`minusp`	minore di zero
`plusp`	maggiore di zero

I valori booleani restituiti dalle funzioni di comparazione sono `T` per vero e `NIL` per falso.

4.4.3 Manipolazione di stringhe

`concatenate 'string` <stringa> <stringa> `....`	concatena stringhe.

`char` <stringa> <posizione>
estrae il carattere che si trova in una certa posizione nella stringa, con indice che parte da zero sul primo carattere a sinistra.

4.4.4 Manipolazione di array

Per lavorare su un array è bene innanzi tutto crearne un'istanza ed assegnarla ad una variabile con
`(setq` <nome_variabile> `(make-array '(`<righe> <colonne> `...)))`
Come abbiamo visto nel paragrafo 4.1.6, attraverso l'opzione `:initial-contents` possiamo anche inizializzare l'array che andiamo a creare. Se non utilizziamo questa opzione creiamo un array delle dimensioni indicate e con elementi uguali a 0.
Dopo di che
`setf (aref` <nome_variabile> <riga> <colonna> `...)` <valore>
assegna i valori nella posizione indicata
`aref` <nome_variabile> <riga> <colonna> `...`
estrae il valore contenuto nella posizione indicata
`array-rank` <nome_variabile>
indica l'ordine dell'array
Nel caso di array monodimensionali (vettori) abbiamo le semplificazioni
`(setq` <nome_variabile `(vector` <elemento> <elemento>`...))`
per creare l'istanza del vettore, inizializzarlo ed assegnarlo ad una variabile
svref <nome_variabile> <posizione>
per estrarre il valore contenuto nella posizione indicata
Rammento che tutti gli indici per indicare la posizione partono da 0

4.4.5 Manipolazione di liste

`cons` <elemento> <lista>
crea una lista aggiungendo un primo elemento a quelli di un'altra lista
`nth` <indice> <lista>
estrae da una lista l'elemento corrispondente all'indice (partendo da zero)
in luogo di nth <indice> si può usare first, second, ecc.
`append` <lista> <lista> `...`
concatena liste
`union` <lista> <lista>
determina l'unione di due liste (in senso insiemistico)
`intersection` <lista> <lista>
determina l'intersezione di due liste (in senso insiemistico)
`length` <lista> ritorna il numero degli elementi di una lista
`remove` <elemento> <lista>
crea una lista con gli elementi di un'altra lista rimuovendo quello indicato

4.4.6 Manipolazione di dizionari

Serve bene un esempio.
Creiamo il dizionario chiamato Capitali:
(setq Capitali '(:Italia Roma :Francia Parigi))
Con
(getf Capitali :Francia)
chiediamo e ci viene indicata la capitale della Francia
Con
(setq Capitali (append Capitali '(:Germania Berlino)))
arricchiamo il dizionario aggiungendo la capitale della Germania.

4.4.7 Input/Output

Per l'input di dati dalla tastiera abbiamo a disposizione due funzioni:
`read` legge un numero o un carattere dalla tastiera,
`read-line` legge una stringa, anche con spaziature, dalla tastiera.

Anche per l'output abbiamo due funzioni:
`print` per scrivere sullo schermo in maniera grezza,
`format` per scrivere sullo schermo in modo formattato, con la sintassi

`format t "<stringa> <segnaposto>" <valore_per_segnaposto>`

dove il <valore_per_segnaposto> può essere indicato richiamando il nome della variabile che lo contiene, essere scritto nella forma prevista dal suo tipo o anche attraverso una espressione per calcolarlo.

Il segnaposto multiuso, se non si hanno particolari esigenze di formattazione di dati numerici è
`~a` [8].

Altro segnaposto molto utile è
`~%` che si usa, senza poi indicarne il valore, per andare a capo (sta per `newline`).

Per la formattazione di numeri interi abbiamo a disposizione le funzioni:

`~d`	per esprimere l'intero in base decimale
`~x`	per esprimere l'intero in base esadecimale
`~o`	per esprimere l'intero in base ottale
`~b`	per esprimere l'intero in formato binario
`~nr`	per esprimere l'intero in base qualsiasi indicata da n
`~@r`	per esprimere l'intero in numerazione romana
`~r`	per esprimere l'intero in lettere inglesi
`~:r`	per esprimere l'intero, come ordinale, in lettere inglesi

Nei primi 5 casi, possiamo inserire tra la ~ e la sigla del segnaposto l'ampiezza, in numero di caratteri, dello spazio in cui posizionare il numero con allineamento a destra.
Questo è un esempio di istruzione con relativo risultato

```
* (format t "~15d~%~15d" 16 1472)
             16
           1472
```

Per la formattazione di numeri a virgola mobile usiamo il segnaposto
`~f`
e, appena dopo la ~, possiamo inserire, in numero di caratteri, l'ampiezza dello spazio in cui posizionare il numero e, separato da virgola, il numero di cifre decimali da evidenziare.
In questo esempio viene stampato il valore della costante di sistema π con tre cifre decimali

```
* (format t "~,3f" PI)
3.142
```

In quest'altro vengono posizionati i numeri con allineamento a destra e due cifre decimali

```
* (format t "~10,2f~%~10,2f" 12.75648 1654.762435)
     12.76
   1654.76
```

[8] Rammento che il carattere ~ (detto tilde), con la tastiera Linux si ottiene premendo insieme i tasti Alt Gr e ì, con la tastiera Mac premendo insieme i tasti Alt e 5 e con la tastiera Windows premendo insieme Alt e 126 della tastierina numerica

4.4.8 Istruzioni condizionali

Se le istruzioni stanno in una sola espressione:
`if (<condizione>) <istruzione>`
viene eseguita <istruzione> se <condizione> è vera
o anche
`if (<condizione>) <istruzione_per_vero> <istruzione_per_falso>`
Se le istruzioni sono complesse e richiedono l'uso di più espressioni:
`if (<condizione>) (progn <istruzione> <istruzione> ...)`
o, meglio:
`when (<condizione>) <istruzione> <istruzione> ...`
In presenza della necessità di elaborare condizioni multiple si può evitare di annidare gli if usando
`cond ((<condizione>) <istruzione> (<condizione>) <istruzione> ...)`
Nel caso <condizione> richieda la contemporanea verifica di più situazioni si utilizzano gli operatori logici booleani
`and` (e logico)
`or` (o logico)
`not` (negazione logica)

4.4.9 Cicli

Per eseguire un'istruzione un numero definito di volte abbiamo a disposizione, tra le tante, queste due funzioni semplici semplici:
`loop repeat <n> do <istruzione>`
`dotimes (<contatore> <n>) <istruzione>`
Per esempio, per scrivere 4 volte la stringa "Ciao" possiamo utilizzare
`(loop repeat 4 do (print "Ciao"))`
oppure
`(dotimes (i 4) (print "Ciao"))`
Per reiterare operazioni sugli elementi di una lista, tra le tante disponibili, abbiamo la facile funzione
`dolist (<variabile> <lista>) <istruzione>`
Per esempio
`(dolist (x '(2 3 4)) (print (* x x)))`
scrive i numeri `4 9` e `16`.

4.5 Funzioni definite dall'utente

Possiamo noi stessi definire una funzione con la sintassi
`(defun <nome_funzione> (<parametro> <parametro> ...)  (<espressione>))`
In questo modo, per esempio, possiamo definire una funzione per il calcolo del fattoriale di un numero
`(defun fattoriale (n) (if (zerop n) 1 (* n (fattoriale (- n 1)))))`
La definizione di una funzione può servire quando in un programma si debba fare più volte uno stesso calcolo e si voglia evitare di riscrivere ogni volta la formula per eseguirlo. Una volta definita la funzione, infatti, basta richiamarla con il suo nome fornendo i parametri richiesti.

Possiamo anche salvare la nostra funzione in un file con estensione .lsp e caricarla nella REPL con
`(load <percorso_al_file>)`
in modo da poterla utilizzare nella REPL stessa richiamandola con il nome e fornendo i parametri richiesti.

Possiamo anche definire temporaneamente una funzione per utilizzarla al volo attraverso la così detta espressione lambda, con la seguente sintassi

`(lambda (<parametro> <parametro> ...) (<espressione>))`

Per esempio, se nella REPL scriviamo
`((lambda (x) (* x x x)) 3)`
otteniamo il risultato `27`.

4.6 Scrivere il programma

Le istruzioni che compongono il programma nel linguaggio LISP vengono eseguite attraverso la così detta valutazione di liste.

Ciascuna istruzione che debba essere eseguita deve essere contenuta in una lista.

Si tratta però di una lista diversa da quella che abbiamo visto nel paragrafo 4.1.5. Quella è un tipo contenitore di dati e consiste in una serie di elementi contenuti tra parentesi tonde preceduta da apice singolo.

L'interprete LISP non valuta una lista preceduta da apice singolo ma la ritorna tale e quale.

La lista non preceduta da apice viene valutata e perché la valutazione possa avvenire occorre che il primo elemento sia una funzione. Gli altri elementi che seguono sono i parametri che vengono passati alla funzione stessa.

La semplice somma di due numeri avviene attraverso questo meccanismo.

Se scriviamo nella REPL
`'(+ 4 5)` ci viene restituita la lista `(+ 4 5)`
Abbiamo, infatti, una lista che, essendo preceduta da apice non viene valutata.

Se scriviamo nella REPL
`(+ 4 5)` ci viene restituito il valore `9`
Abbiamo, infatti, una lista che, non essendo preceduta da apice viene valutata: essa contiene come primo elemento la funzione + seguita dai parametri 4 e 5 e la valutazione fornisce il risultato 9.

Se scriviamo nella REPL
`(3 4 5)` viene segnalato errore
Abbiamo, infatti, una lista che, non essendo preceduta da apice viene valutata ma l'interprete, non trovando la funzione oggetto di valutazione, segnala errore.

Possiamo anteporre ad una lista un apice rovesciato (il carattere `` ` ``) [9]ottenendo di evitare la valutazione della lista potendo tuttavia contrassegnare all'interno di questa una lista da valutare anteponendovi una virgola (il carattere ,).
Ad esempio, se scriviamo nella REPL
`` `(4 + 5 = ,(+ 4 5)) `` ci viene restituito `(4 + 5 = 9)`
in quanto l'interprete, fino alla virgola legge semplicemente `(4 + 5 =` e, dopo la virgola valorizza la lista `(+ 4 5)` e restituisce il risultato dell'addizione.

Un programma contiene una o più, tantissime nei programmi più complessi, liste da valorizzare e sono valorizzate dall'interprete una dopo l'altra, nell'ordine in cui si presentano.

Abbiamo già visto nel Capitolo 2 in che modo scrivere ed eseguire i programmi.

Quanto allo scrivere, rammento che LISP è case insensitive, cioè non distingue tra maiuscolo e minuscolo: la funzione `print` può essere indifferentemente essere richiamata con `print`, `PRINT`, `Print` o `PrInT`.

5 Piccoli programmi di esempio

Con il programma, meglio chiamarlo script, riportato all'inizio della pagina seguente viene chiesto il nome all'utente per rivolgergli un saluto.
La prima riga è la shebang per l'implementazione SBCL, che, nel sistema Linux, consente di eseguire il programma, una volta reso eseguibile, senza richiamare l'interprete ma richiamando

[9]L'apice rovesciato o apice inverso o backtick si ottiene su tastiera Linux premendo contemporaneamente i tasti AltGr e `'`, su tastiera Mac premendo contemporaneamente i tasti Alt e 9 e su tastiera Windows premendo contemporaneamente i tasti Alt e 96 della tastierina numerica.

semplicemente il nome del file che lo contiene. Fuori dal sistema operativo Linux questa riga viene semplicemente ignorata.

```
#!  /usr/bin/sbcl --script
(format t "Come ti chiami?~%")
(defvar nome)
(setq nome (read-line))
(format t "Ciao, ~a!~%" nome)
```

La seconda riga formula all'utente, in testo formattato, la richiesta del suo nome.
La terza riga inizializza la variabile in cui memorizzare il nome dell'utente. Potremmo evitare questa istruzione, ma gli interpreti di talune implementazioni, come la SBCL, eseguirebbero il programma propinandoci un warning al momento di eseguire l'istruzione successiva, in quanto la variabile `nome` è sconosciuta. Il programma però funzionerebbe comunque.
La quarta riga memorizza nella variabile nome quanto inserito dalla tastiera.
La quinta riga formula il saluto, in testo formattato, destinandolo al nome memorizzato nella variabile `nome`.
Memorizzato il programma nel file `saluto.lsp` può essere eseguito a terminale con questo risultato

```
vittorio@mx:~/Programmi
$ sbcl --script saluto.lsp
Come ti chiami?
Vittorio Albertoni
Ciao, Vittorio Albertoni!
```

Quest'altro script calcola il fattoriale del numero indicato dall'utente

```
#!  /usr/bin/sbcl --script
(defun fattoriale(n) (if(zerop n)1 (* n (fattoriale(- n 1)))))
(format t "Inserisci il numero di cui vuoi calcolare il fattoriale~%")
(defvar numero)
(setq numero (read))
(format t "Il fattoriale di ~a è~%~a~%" numero (fattoriale numero))
```

Dopo la riga dedicata alla shebang abbiamo la definizione di una funzione per calcolare il fattoriale di un numero. Come si vede si tratta di una bella funzione ricorsiva (funzione che utilizza sé stessa), a dimostrazione del fatto che LISP supporta la ricorsività.
Seguono le righe per richiedere all'utente il numero di cui calcolare il fattoriale, per definire la variabile in cui collocarlo e per collocarvelo leggendolo dalla tastiera.
Infine, nell'ultima riga, la formattazione del testo per evidenziare il risultato, calcolato richiamando la funzione prima creata e passandole il relativo parametro richiamando il nome della variabile in cui è memorizzato.
Memorizzato il programma nel file `fattoriale.lsp` può essere eseguito a terminale con questo risultato

```
vittorio@mx:~/Programmi
$ sbcl --script fattoriale.lsp
Inserisci il numero di cui vuoi calcolare il fattoriale
102
Il fattoriale di 102 è
9614466715035126609268655586972595484553559050596594643694447140485317151302545906033149618823644513849855959803620591575037100428655329280000000000000000000000000
```

Notiamo come, pur trattandosi di un numero enorme, esso venga scritto nella sua interezza, addirittura andando a capo dopo aver occupato i 90 caratteri ospitabili in una riga del terminale.
Ciò è dovuto al fatto che in LISP i numeri interi vengono trattati in precisione arbitraria, cioè senza alcuna limitazione imposta dal software ma con l'unica limitazione posta dall'harware, che, oggi come oggi, può reggere numeri interi fino ad alcune migliaia di caratteri.

Ora un piccolo script che traduce un numero intero in numero secondo la numerazione usata nell'antica Roma.

```
#!  /usr/bin/sbcl --script
(format t "Inserisci il numero da trascrivere in numerazione romana~%")
(defvar numero)
(setq numero (read))
(format t "Il numero ~a era scritto così dagli antichi romani~%~@r~%" numero numero)
```

Ormai penso che i commenti siano superflui.
Memorizzato il programma nel file `romano.lsp` può essere eseguito a terminale con questo risultato

```
vittorio@mx:~/Programmi
$ sbcl --script romano.lsp
Inserisci il numero da trascrivere in numerazione romana
1745
Il numero 1745 era scritto così dagli antichi romani
MDCCXLV
```