NewLISP (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 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: già nel 1960, al LISP di McCarthy si affiancarono il MacLISP sviluppato presso il MIT, l'InterLISP sviluppato presso la Xerox e lo StanfordLISP sviluppato presso l'Università di Stanford.

Nel 1970 vede la luce una versione del LISP che unifica un po' tutto: si chiama Scheme.

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

Altro tentativo unificante avviene nel 1984 con la nascita del CommonLISP, tuttora usato.

Nel 1985 Richard Stallman e Guy L. Steel jr. propongono l'EmacsLISP come progetto open source con licenza GNU.

Nel 1991 Lutz Műller sviluppa NewLISP, sintesi di tutto il LISP fino allora noto con selezione di ciò che veramente serve, potente ma con modesto utilizzo di risorse. In breve volgere di tempo compare una versione per Windows 3.0 e, nel 1995, la versione a 32 bit per Windows 95.

Nel 1999 NewLISP diventa open source distribuito con licenza GNU-GPL e trasferisce il proprio ambiente di sviluppo su Linux, diventando software libero a tutti gli effetti.

Anche se, soprattutto nella versione NewLISP, non è un linguaggio funzionale puro il linguaggio LISP è storicamente da considerarsi il primo linguaggio che usa un paradigma di programmazione funzionale.

Indice

1 Installazione **3**

2 Come funziona: il primo programma **3**

3 Basi del linguaggio **4**

4 Tipi fondamentali **5**

5 Variabili e costanti **6**

6 Funzioni preconfezionate **7**
6.1 Input/Output . . . 7
6.2 Aritmetica . . . 8
6.3 Matematica . . . 10
6.3.1 Matematica di base . . . 10
6.3.2 Trigonometria . . . 10
6.3.3 Statistica . . . 11
6.3.4 Matematica finanziaria . . . 12
6.3.5 Array e matrici . . . 13
6.4 Manipolazione di liste . . . 14
6.5 Manipolazione di stringhe . . . 14
6.6 Manipolazione di simboli . . . 14
6.7 Lavorare con i file . . . 15

7 Costruire una funzione **17**

8 Interattività con l'utente **18**

9 Strutture di controllo **19**
9.1 Esecuzione condizionale . . . 19
9.2 Ripetizione . . . 20

10 Estensioni **21**
10.1 Macro . . . 21
10.2 Context . . . 21
10.3 FOOP . . . 22
10.4 Modules . . . 22
10.5 Interfaccia grafica . . . 22
10.5.1 tk . . . 22
10.5.2 Guiserver . . . 22

1 Installazione

NewLISP si trova all'indirizzo *http://www.newlisp.org/*.

Se apriamo la scheda DOWNLOADS abbiamo modo di scaricare l'eseguibile per Windows (a 32 o 64 bit e nelle versioni normale o UTF8, consigliata), l'eseguibile per Mac OS X (solo a 64 bit) e il source. Nel caso di Windows e Mac OS X sono disponibili anche le librerie dinamiche, file `.dll` o `.dylib`, necessarie in questi sistemi operativi, per importare funzionalità LISP in un altro linguaggio di programmazione ma di nessuna utilità per il più normale utilizzo di NewLISP che tratterò in questo manuale.

Windows

Per l'installazione su Windows dobbiamo semplicemente copiare il file `.exe` che abbiamo scaricato in una directory qualsiasi nominata `newlisp` (per coerenza con il resto del sistema è consigliabile creare la directory `newlisp` in `Program Files`); il percorso verso questa directory va ovviamente inserito nella variabile d'ambiente PATH.

Questo tipo di installazione è minimale; per l'installazione completa occorre partire dal source e avere installato MinGW e si tratta di un lavoro non da dilettanti.

Mac OS X

Possiamo scegliere se copiare l'eseguibile scaricato in `/usr/local/bin` o se compilare il source come se fossimo su Linux.

Linux

Sul nostro sistema Linux potrebbe essere già installato NewLISP: conviene pertanto verificarlo, ad evitare lavori inutili, scrivendo a terminale il comando `newlisp`.

Sempre su Linux l'installazione potrebbe avvenire attraverso il gestore pacchetti Synaptic.

Il sistema più laborioso è quello della compilazione del source, che è contenuto nel file `newlisp-xx.x.x.tgz` (per l'ultima versione, del maggio 2019, `newlisp-10.7.5.tgz`).
Una volta scaricato questo file, lo copiamo, con poteri di root, in `/usr/local/src`.
Lo scompattiamo con `tar xzvf newlisp-xx.x.x.tgz`.
Ci posizioniamo nella directory che si è così creata a fianco, `newlisp_xx.x.x`, e diamo il comando `./configure`.
Indi diamo, uno dopo l'altro, i comandi `make` e `sudo make install`.
Se tutto ha funzionato possiamo eliminare il contenuto della directory `/usr/local/src`.

Su Ubuntu a 64 bit potrebbe essere necessario, prima di dare i comandi `make` e `sudo make install`, installare alcune librerie necessarie per la compilazione con i comandi

```
sudo apt install libffi-dev
sudo apt install libreadline6 libreadline6-dev
```

Dopo tutto questo lavoro avremo il nostro eseguibile `newlisp` in `/usr/local/bin`.

* * *

Se apriamo la scheda DOCS, sempre all'indirizzo *http://www.newlisp.org/*, possiamo consultare il manuale di NewLISP. Conviene, anzi, scaricare questo manuale, composto da una sola lunga pagina in formato html, in modo da averlo a disposizione off-line.

2 Come funziona: il primo programma

Una volta installato NewLISP possiamo usare il comando `newlisp` in tre modi:
. scriverlo a terminale da solo,
. scriverlo a terminale seguito dal nome di un file con estensione `.lsp`,

. scriverlo a terminale con sintassi atta a «compilare» un file con estensione `.lsp`.

Nel primo caso si apre nel terminale una shell interattiva in cui possiamo utilizzare il linguaggio NewLISP per esperimento o per ottenere risultati di meno difficoltoso raggiungimento.
La seguente figura 1 mostra la shell nel mio terminale Linux.

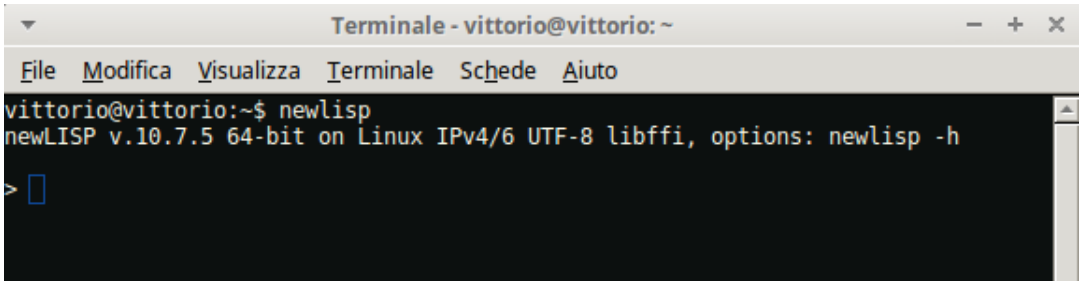

Figura 1: Shell interattiva di NewLISP

Se in corrispondenza del prompt della shell (simbolo >) scriviamo la seguente istruzione NewLISP
`(println "Ciao mondo!")`
nella riga sottostante compare immediatamente il saluto.
Normalmente la shell accetta una sola riga di comando.
Per scrivere un comando su più righe occorre racchiudere queste righe tra un tag di apertura `[cmd]` e un tag di chiusura `[/cmd]`.

Nel secondo caso, previa scrittura in un editor di testo di queste istruzioni
```
(println "Ciao mondo!")
(exit)
```
e memorizzazione in un file con estensione `.lsp`, per esempio `ciao_mondo.lsp`, scrivendo a terminale
`newlisp ciao_mondo.lsp`
sempre a terminale vedremo comparire il saluto.

Nel terzo caso, con l'istruzione
`newlisp -x ciao_mondo.lsp ciao_mondo`
oppure, se siamo in Windows,
`newlisp -x ciao_mondo.lsp ciao_mondo.exe`
otteniamo la «compilazione» del nostro programmino.
Scrivo «compilazione» tra virgolette in quanto non si tratta di una vera e propria compilazione ma della produzione di un file che contiene l'interprete: il risultato è comunque quello di poter distribuire il programma e poterlo eseguire anche su un computer su cui non è installato NewLISP.
Se siamo su Linux dobbiamo preventivamente rendere eseguibile il file con il comando
`chmod 555 ciao_mondo`
o comando similare.

Abbiamo così costruito il nostro primo programma in NewLISP.

3 Basi del linguaggio

La base del linguaggio LISP (LISt Processing) è la lista.

Una lista è una sequenza di elementi separati da uno spazio e racchiusa tra parentesi tonde.

Normalmente il primo elemento della lista viene visto da NewLISP come una funzione e gli elementi successivi vengono visti come parametri necessari alla funzione per raggiungere il risultato richiesto.

Il nostro primo programmino `(println "Ciao mondo!")` è una lista di due elementi: il primo è la funzione predefinita nel linguaggio per scrivere una cosa e andare a capo (`println`), il secondo è una stringa che contiene ciò che va scritto.

In NewLISP, come in tutti gli altri LISP che esistono, tutto viene eseguito attraverso funzioni che devono occupare il primo posto di una lista.
La stessa semplicissima somma di due numeri interi, che da qualsiasi altra parte viene effettuata attraverso l'operatore + inserito tra i due numeri da sommare, in NewLISP (in LISP non esistono gli operatori) viene effettuata richiamando la funzione +, che è la funzione preposta per sommare numeri con risultato intero, come primo elemento di una lista seguito dai numeri da sommare.
Sicché la somma di 4 e 5 si ottiene con l'istruzione `(+ 4 5)`.

La trasformazione della lista in risultato avviene immediatamente in quanto la lista viene immediatamente valutata.

Per evitare questo possiamo anteporre alla lista un semplice apice ('), altrimenti detto apostrofo (codice ASCII 39).
Se nella shell interattiva scriviamo `(+ 4 5)` otteniamo il risultato `9`.
Se scriviamo `'(+ 4 5)` otteniamo `(+ 4 5)`.
In questo modo abbiamo l'opportunità di utilizzare la lista non come istruzione di programma ma come contenitore di dati.

4 Tipi fondamentali

I tipi di dati manipolabili con NewLISP sono i seguenti.

Numero

I valori numerici vengono trattati come numeri interi (`int`) e numeri in virgola mobile (`float`).

L'intero si scrive con uno o più caratteri numerici e viene letto e ritornato come tale.
Nella shell NewLISP a terminale, `5` ritorna `5`.

Il numero in virgola mobile si scrive con caratteri numerici utilizzando la virgola come separatore decimale e viene letto e ritornato come tale.
`5,6` ritorna `5,6`.
Contrariamente a quanto avviene in altri linguaggi di programmazione, dove impera il punto come separatore decimale, NewLISP, se siamo in Italia ed abbiamo un sistema operativo in lingua italiana, capisce le nostre abitudini e si aspetta che usiamo la virgola.
Possiamo verificarlo dando nella shell il comando (div 3 2) e vedere se il risultato viene scritto 1,5 o 1.5.

Si può trasformare un `float` in `int` con l'istruzione `(int <numero_float>)`. Ciò comporta la sparizione della parte decimale senza arrotondamento:
`(int 5,6)` ritorna `5`.

Si può trasformare in numero una stringa contenente caratteri numerici con le istruzioni `(int <stringa>)` o `(float <stringa>)`:
`(int "5,6")` ritorna `5`,
`(float "5,6")` ritorna `5,6`.

Contrariamente si può anche trasformare un numero in una stringa con l'istruzione `(string <numero>)`:
`(string 5,6)` ritorna `"5,6"`.

Se la stringa contiene caratteri che identificano un numero in base diversa da 10, con la funzione `int` possiamo tradurlo nel corrispondente numero intero in base 10, anteponendo nella stringa il prefisso 0x se si tratta di un numero esadecimale, il prefisso 0 se si tratta di un numero ottale e il prefisso 0b se si tratta di un numero binario.
Il numero esadecimale 5E7A si converte in decimale con `(int "0x5E7A")` che ritorna `24186`.
Il numero ottale 72 si converte in decimale con `(int "072")` che ritorna `58`.

Il numero binario 11101 si converte in decimale con `(int "0b11101")` che ritorna `29`.

Simbolo

Qualsiasi carattere o combinazione di caratteri che non siano numerici è un simbolo.

Il simbolo preceduto da un semplice apice (`'`) viene letto e ritornato come tale.
`'a` ritorna `a`,
`'abc` ritorna `abc`.

Stringa

La stringa è una successione di caratteri inclusa tra doppi apici e viene letta e ritornata come tale.
`"Vittorio"` ritorna `"Vittorio"`.

Lista

La lista, quella preceduta da apice che abbiamo visto prima, detta quoted list, è una successione di elementi di qualsiasi tipo.
`'(abc g x z)` genera una lista di simboli,
`'(2 5 7 9)` genera una lista di numeri,
`'("Pippo" "uccello" "Sabato")` genera una lista di stringhe,
`'(sin sqrt add)` genera una lista di funzioni,
`'(35,8 sqrt "Giuseppe" abc)` genera una lista mista.

Array

L'array, pur avendo maggior utilità pratica se composto da numeri, è un modo di organizzare dati di qualsiasi tipo.

L'array monodimensionale, detto altrimenti vettore, è molto simile alla lista.

L'array multidimensionale è una matrice.

L'array si genera con la funzione `array`, che ha la seguente sintassi.
`(array 3 '(1 3 6))` genera il vettore `(1 3 6)`,
`(array 2 2 '(3 4 1 2))` genera la matrice `((3 4) (1 2))`, altro modo di scrivere $\begin{vmatrix} 3 & 4 \\ 1 & 2 \end{vmatrix}$.

5 Variabili e costanti

Una variabile è un indirizzo di memoria ove si trova un dato e si identifica con un simbolo.

La funzione per creare una variabile è `set` con la seguente sintassi:
`(set <simbolo> <dato>)`
ove `<dato>` può essere una lista che genera un dato.
Con `(set 'x 12,5)` creiamo la variabile `x` che contiene il valore numerico `12,5`.
Con `(set 'a (+ 4 3))` creiamo la variabile a che contiene la somma di 4 e 3, cioè il valore numerico `7`.
Con `(set 'nome "Vittorio")` creiamo la variabile nome che contiene la stringa `"Vittorio"`.
Con `(set 'M (array 2 2 '(3 4 1 2)))` creiamo la variabile `M` che contiene la matrice $\begin{vmatrix} 3 & 4 \\ 1 & 2 \end{vmatrix}$.
Con `(set 'l '(4 5 ab))` creiamo la variabile `l` che contiene la lista `(4 5 ab)`.

La variabile creata con `set` è una variabile globale, cioè una variabile utilizzabile nel corso di un intero programma in più liste.

Se desideriamo creare, all'interno di una lista, una variabile locale che svolga la sua funzione solo all'interno della lista e che scompaia una volta che la lista ha operato possiamo utilizzare la funzione `let`, con la seguente sintassi:

`(let (<simbolo> <dato>) <altre_istruzioni>)`
ove `<dato>` può essere una lista che genera un dato e <altre_istruzioni> è il restante contenuto della lista.
Con `(let (nome "Vittorio") (println "Ciao " nome))` scriviamo un saluto utilizzando la variabile `nome` creata come variabile locale all'interno della lista. Una volta che la lista ha esaurito il proprio compito scrivendo il saluto la variabile `nome` viene eliminata.

Il valore di una variabile si può modificare nel corso del programma.

Se vogliamo avere a disposizione dati utili per sviluppare il nostro programma ma che non siano modificabili, conviene che creiamo delle costanti che li contengano. La sintassi è
`(constant <simbolo> <dato>)`
ove <dato> può essere una lista che genera un dato.
Con `(constant 'c 8)` creiamo la costante `c` che contiene il numero `8`.
Con `(constant 'e (exp 1))` creiamo la costante `e` che contiene il numero *e*, base dei logaritmi naturali (`2,718281828459045`).

6 Funzioni preconfezionate

Tutto ciò che si fa con NewLISP lo si fa utilizzando funzioni: abbiamo visto che la lista normale (quella senza l'apostrofo) deve contenere come primo elemento una funzione e gli altri elementi sono i dati di cui questa funzione ha bisogno per produrre un risultato.

NewLISP ci mette a disposizione tantissime funzioni preconfezionate, utilizzabili per creare i nostri programmi: esse sono elencate nel manuale di NewLISP, che possiamo consultare o scaricare da *http://www.newlisp.org/*. Le troviamo nella sezione Function Reference, organizzate per gruppi o in ordine alfabetico. Per ciascuna funzione preconfezionata possiamo così conoscere a cosa serve, come si richiama, con quali parametri e di che tipo.

Qui richiamo dettagliatamente solo quelle che è necessario conoscere per le funzionalità di base o poco più.

6.1 Input/Output

`read-line`
legge l'input dal device corrente (per default la tastiera) come stringa delimitata dal carattere di fine riga.
Richiamata questa funzione il computer si mette in attesa che l'utente digiti sulla tastiera ciò che vuole inserire e prema INVIO.
Dal momento che quanto digitato viene acquisito come stringa, se ciò che interessa acquisire è un numero occorre fare la trasformazione.
Se rispondiamo a `(read-line)` digitando la parola ciao il computer legge la stringa "ciao".
Se rispondiamo digitando 16,25 il computer legge la stringa "16,25".
Se usiamo il costrutto `(float(read-line))` e rispondiamo digitando 16,25 il computer legge il numero 16,25.
In genere questa funzione si utilizza per memorizzare il dato acquisito in una variabile, nel qual caso si utilizza la sintassi
`(set <nome_variabile> (read-line))` se si vuole inserire nella variabile una stringa,
`(set <nome_variabile> (int(read-line)))` se si vuole inserire nella variabile un numero intero,
`(set <nome_variabile> (float(read-line)))` se si vuole inserire nella variabile un numero in virgola mobile,
dove `<nome_variabile>` è il simbolo (una o più lettere precedute a apostrofo `'`) con cui intendiamo nominare la variabile.

`println`
scrive sul device corrente (per default lo schermo) andando a capo dopo la scrittura,

`print`
scrive senza andare a capo dopo la scrittura.
Il parametro di questa funzione (ciò che deve essere scritto) si può indicare creandolo al momento con un'istruzione o richiamando una variabile che contiene quanto si vuole scrivere.
Esempi:
`(println "ciao")` scrive la parola `ciao`,
`(println (* 3 2))` scrive il numero `6` (risultato della moltiplicazione tra 3 e 2),
`(println 5,28)` scrive il numero `5,28`,
`(println x)` scrive il contenuto della variabile `x`.
Quando si scrivono numeri a volte si sente l'esigenza di formattarli, o per allinearli o per fissare il numero di decimali. Possiamo farlo applicando la funzione `format` al numero da scrivere, indicato come tale, con un'espressione che lo genera o richiamando la variabile che lo contiene.
La sintassi della funzione format è
`(format {<direttiva_di_formattazione>} <numero>)`
dove `<direttiva_di_formattazione>` è
`%<spazi_allineamento>.<decimali>f`
con `<spazi_allineamento>` ad indicare il numero di spazi entro cui fissare l'allineamento a destra: se il numero indicato è inferiore al numero dei caratteri da scrivere l'allineamento avviene a sinistra, per ottenere il quale basta indicare 1,
e <decimali> ad indicare il numero di cifre decimali da scrivere.
Esempi:
`(println (format {%1.3f} 7,895746))` scrive `7,896` allineato a sinistra,
`(println (format {%10.2f} (div 2 3)))` scrive `0,67` allineato a destra in uno spazio di 10 caratteri.

6.2 Aritmetica

Abbiamo innanzi tutto funzioni che operano su numeri interi. Se passiamo a queste funzioni numeri in virgola mobile esse operano solo sulla parte intera dei numeri e ritornano come risultato un numero intero, senza approssimazioni di sorta: le parti decimali vengono semplicemente ignorate, come se non ci fossero. Si tratta delle seguenti funzioni:
`+` somma tutti i numeri inseriti nella lista,
`-` sottrae al primo numero inserito il secondo e via via tutti gli altri inseriti,
`*` moltiplica tra loro tutti i numeri inseriti nella lista,
`/` divide il primo numero inserito e il risultato intero lo divide per il numero successivo e così via,
`%` ritorna il resto della divisione tra il primo numero e il secondo e lo divide per l'eventuale numero successivo e così via,
`++` oppure `inc`, incrementa un numero di una unità,
`--` oppure `dec`, decrementa un numero di una unità.
Nella lista si possono inserire numeri, espressioni che li generano o variabili che li contengono.
Esempi:
`(+ 1 3 5)` ritorna `9`,
`(+ 1,9 2,9)` ritorna `3`,
`(- 10 3 4 1)` ritorna `2`,
`(* 2 3 6)` ritorna `36`,
`(/ 25 2 3)` ritorna `4`,
`(/ 12 (+ 3 2))` ritorna `2`,
`(% 16 7)` ritorna `2`,
`(% 16 7 2)` ritorna `0`,
`(+ (/ 6 3) (* 2 4))` ritorna `10`,
`(++ 3)` ritorna `4`,
`(dec (* 3 2))` ritorna `5`,
se esiste una variabile `x` contenente il valore `15`, `(/ x 3)` ritorna `5`.

Normalmente queste operazioni producono numeri interi con un massimo di 19 cifre (per l'esattezza il numero massimo generabile con queste operazioni è 9 223 372 036 854 775 807. Tuttavia, se almeno uno dei numeri indicati nella lista per l'operazione è superiore a questo limite, interviene automaticamente la gestione big integer e il risultato viene indicato come big integer, che è un intero con il suffisso L e può arrivare ad essere un intero di ben 1634 cifre.
Per poter esprimere un risultato che possa raggiungere queste dimensioni operando soltanto con interi di dimensione inferiore occorre che il primo di questi interi sia espresso come big integer, cioè con il suffisso L.
Esempi:
`(* 70000000000000 160000)` genera stack overflow e produce un risultato inaccettabile consistente in un numero negativo,
`(* 70000000000000L 160000)` ritorna `11200000000000000000L` che è un big integer di 20 cifre: si è potuto superare il limite in quanto il primo intero della lista è stato qualificato big integer,
`(+ 9223372036854775810 100)` ritorna `9223372036854775910L` che è un big integer superiore all'intero massimo generabile: ciò avviene in quanto il primo intero della lista è, sia pure di sole 3 unità, superiore all'intero massimo generabile e si apre così automaticamente la gestione big integer.

Esiste la funzione `bigint` con la quale possiamo trasformare un intero normale in big integer:
`(bigint 24)` ritorna `24L`, che è un big integer,
`(bigint a)` acquisisce come big integer il contenuto della variabile `a`,
`(bigint (int (read-line)))` acquisisce come big integer il numero che scriviamo con la tastiera[1].

Per fare le stesse operazioni che abbiamo visto con numeri in virgola mobile, quelli con i decimali, dobbiamo usare, nell'ordine, le seguenti funzioni, che possiamo comunque usare anche per i numeri interi (senza tuttavia la possibilità di gestire i big integer):
`add` per la somma,
`sub` per la sottrazione,
`mul` per la moltiplicazione,
`div` per la divisione,
`mod` per il resto della divisione,
`inc` incrementa un numero di una unità,
`dec` decrementa un numero di una unità.
Esempi:
nel secondo esempio di prima, se scriviamo `(add 1,9 2,9)`, otteniamo il risultato `4,8`,
nel quinto esempio di prima, se scriviamo `(div 25 2 3)`, otteniamo il risultato `4,166666666666667`.

Queste operazioni producono numeri in virgola mobile di qualsiasi dimensione ma precisi nell'ambito di 17 cifre complessive (tra parte intera e parte decimale), virgola compresa.
Esempi:
`(div 12,5 3)` ritorna, come viso in altro modo nell'esempio di prima, `4,166666666666667`,
`(div 8967542 3)` ritorna `2989180,666666667`;

prima una cifra per la parte intera, 15 cifre per la parte decimale, per un totale di 16 più la virgola,

poi 7 cifre per la parte intera, 9 cifre per la parte decimale, per un totale di 16 più la virgola.
`(mul 9223372036854775810 1114)` ritorna `1,027483644905622e+22`, un bel numero di 23 cifre, che è tuttavia preciso per le solite 16 cifre e, dopo l'arrotondamento alla sedicesima cifra si completa con degli zeri: `10274836449056220000000`; il vero risultato, `10274836449056220252340L`, lo otterremmo sostituendo alla funzione `mul` la funzione `*`.

[1] Tutto questo casino non lo troviamo nel normale LISP, dove, come in Python, il limite dimensionale degli interi è dato solo dall'hardware, per cui è possibile manipolare enormi interi di dimensioni illeggibili. Lutz Műller penso abbia introdotto queste novità in NewLISP per rendere più veloce il trattamento delle dimensioni sotto il limite, limite che, peraltro, corrisponde a quelli imposti da tutti gli altri linguaggi di programmazione e fogli di calcolo.

Altre semplici funzioni per la manipolazione di numeri sono:
`abs` che ritorna il valore assoluto di un numero,
`ceil` che arrotonda un numero in virgola mobile per eccesso,
`floor` che arrotonda un numero in virgola mobile per difetto,
`round` che applica a un numero in virgola mobile il così detto arrotondamento commerciale.

6.3 Matematica

Oltre alle funzioni per l'aritmetica di base che abbiamo appena visto, NewLISP ci mette a disposizione moltissime funzioni per eseguire calcoli più complicati.

Per queste altre funzioni non vale più la distinzione tra numeri interi e numeri in virgola mobile e il risultato fornito è sempre un numero in virgola mobile (float).

Qui elencherò le funzioni di più diffusa utilità classificate in gruppi. Riferimenti completi si possono trovare nella documentazione che ho indicato all'inizio del Capitolo.

6.3.1 Matematica di base

`sqrt`
ritorna la radice quadrata di un numero,

`pow`
ritorna la potenza all'esponente indicato come secondo parametro (se non indicato calcola il quadrato)
`(pow 5 3)` ritorna `125`
`(pow 5)` ritorna `25`
`pow` serve anche per calcolare radici ennesime se il secondo parametro è indicato come reciproco dell'indice di radice:
`(pow 125 (div 1 3)` ritorna `5`, radice cubica di 125 (attenzione ad usare, per indicare il reciproco, la funzione `div` e non `/`).

6.3.2 Trigonometria

abbiamo tutte le funzioni trigonometriche dirette (`sin cos tan`) e inverse (`acos asin atan`).
Il parametro per le dirette va espresso in radianti.
Le indirette ritornano radianti.

NewLISP non ha costanti predefinite, per cui non ha nemmeno la costante π. Possiamo tuttavia definirla noi ricorrendo a una funzione trigonometrica inversa, per esempio acos, in questo modo:
`(constant 'pi (acos -1))`, che immette il valore `3,141592653589793` nella costante `pi`.
In questo modo possiamo richiamare in tutto il programma il valore della costante `pi` per indicare i radianti.
`(sin (div pi 2))` ritorna `1`,
`(cos (mul pi 2))` ritorna `1`,
`(sin pi)` ritorna `0`.

Definita la costante `pi` possiamo anche utilizzarla per trasformare i radianti in gradi con
`(div(mul <radianti> 180) pi)`
e i gradi in radianti con
`(div(mul <gradi> pi) 180)`.

6.3.3 Statistica

`stats`
ritorna una lista contenente indicatori statistici relativi ad una serie di numeri; la serie di numeri, argomento della funzione, è contenuta in una lista o in un vettore;
gli indicatori statistici sono, nell'ordine:
. numero dei valori,
. media aritmetica,
. media aritmetica degli scarti dalla media,
. deviazione standard campionaria,
. varianza campionaria,
. indice di asimmetria di Fisher,
. indice di curtosi di Pearson.
`(stats '(1 2 3 4 5 4 3 2 1))` ritorna la lista
`(9 2,777777777777778 1,135802469135802 1,394433377556793 1,944444444444445 0,1011833743351869 -1,543431594860166)`
rammento che l'indice di asimmetria di Fisher:
. compreso tra -0,5 e 0,5 indica simmetria perfetta,
. compreso tra -1 e -0,5 indica moderata asimmetria (coda lunga) a sinistra,
. compreso tra 0,5 e 1 indica moderata asimmetria (coda lunga) a destra,
. oltre -1 indica forte asimmetria a sinistra,
. oltre 1 indica forte asimmetria a destra;
e l'indice di curtosi di Pearson:
. uguale a 0 indica perfetta sovrapponibilità alla campana di Gauss,
. maggiore di 0 indica una curva più appuntita (leptocurtica),
. minore di 0 indica una curva più piatta (platicurtica).
La distribuzione dell'esempio è perfettamente simmetrica (lo si vede anche a occhio)
ed è platicurtica, cioè più piatta della campana di Gauss.

`corr`
ritorna una lista contenente una serie di indicatori relativi alla correlazione esistente tra dati contenuti in due liste o vettori aventi lo stesso numero di elementi;
gli indicatori sono, nell'ordine:
. coefficiente di correlazione `r`,
. intercetta della retta di regressione,
. coefficiente angolare della retta di regressione,
. test di significatività `t`,
. gradi di libertà del test di significatività `t`,
. probabilità a due code di `t` relativa all'ipotesi nulla.
Se abbiamo due serie di dati in altrettante variabili:
```
(set 'x '(7 12 22 35 46 58))
(set 'y '(8 14 20 38 50 56))
```
`(corr a b)` ritorna la lista
`(0,9919130494367538 1,302724520686176 0,9899091826437941 15,63061323392482 4 9,78337158110687e-05)`
dove il coefficiente di correlazione quasi pari a 1 indica perfetta correlazione
e i due dati successivi sottendono l'equazione della retta di regressione
`y = 1,302724520686176 + 0,9899091826437941x`
infine il valore prossimo a 0 della probabilità dell'ipotesi nulla conferma l'esistenza di un legame tra i dati della variabile `x` e quelli della variabile `y`.

`t-test`
ritorna una lista di valori utili per valutare la differenza tra medie campionarie; dal manuale possiamo vedere quattro possibili sintassi, qui evidenzio la più classica.
Indichiamo come argomenti della funzione i risultati di due osservazioni campionarie;
la lista ritornata dalla funzione contiene, nell'ordine:
. media delle osservazioni del primo campione,
. media delle osservazioni del secondo campione,

. deviazione standard delle osservazioni del primo campione,
. deviazione standard delle osservazioni del secondo campione,
. valore del test `t` di Student,
. gradi di libertà,
. probabilità a due code di `t` relativa all'ipotesi nulla.
Se abbiamo due serie di dati
`(set 'a '(97 86 70 68 74 71 48 65))` pulsazioni cardiache di un campione di individui presi a caso,
`(set 'b '(62 72 65 60 48 52 60))` pulsazioni di individui che hanno assunto un betabloccante,
`(t-test a b)` ritorna la lista
`(72,375 59,85714285714285 14,49076060312718 7,967194642795556 2,027114439618418 13 0,0636665985911502)`
dal bassissimo valore della probabilità del test `t` corrispondente all'ipotesi nulla dobbiamo dedurre che il betabloccante fa effetto.

A proposito di test statistici abbiamo poi funzioni che, per i quattro più famosi, ritornano la probabilità (a una coda) che il valore osservato possa essere uguale o più grande nell'ipotesi nulla:
`prob-t` è relativa al test `t` di Student;
alla funzione si passano come argomenti il valore del test e i gradi di libertà;
`prob-f` è relativa al test `f` di Snedecor;
gli argomenti da passare sono il valore del test e i due gradi di libertà;
`prob-z` è relativa al test `z` di Fisher;
l'unico argomento da passare è il valore del test;
`prob-chi2` è relativa al test χ^2 di Pearson;
alla funzione si passano come argomenti il valore del test e i gradi di libertà.
A queste funzioni corrispondono le funzioni inverse `crit-t`, `crit-f`, `crit-z` e `crit-chi2`, che ritornano il valore critico minimo del test corrispondente ad una certa probabilità in ipotesi nulla.

Mi sono limitato a citare le funzioni statistiche di più ricorrente utilizzo. Altre ce ne offre NewLISP e le troviamo nel manuale che ho già citato.

6.3.4 Matematica finanziaria

Delle sei funzioni che troviamo nel manuale segnalo le tre che possono essere utili a chi abbia intenzione di contrarre un prestito.

`pmt`
ritorna l'importo della rata necessaria per estinguere un prestito;
gli argomenti necessari, nell'ordine, sono:
. tasso d'interesse unitario (il 5% si indica con 0,05),
. numero delle rate,
. importo del prestito.
`(pmt 0,05 12 1000)` ritorna `-112,8254100208153`
che è la rata (indicata in negativo, essendo un pagamento) necessaria per estinguere in 12 anni un prestito di 1000 al tasso del 5%.

`pv`
ritorna il valore attuale di pagamenti periodici costanti (importo del prestito);
gli argomenti necessari, nell'ordine, sono:
. tasso d'interesse unitario (il 5% si indica con 0,05),
. numero delle rate,
. importo della rata.
`(pv 0,05 12 112,82541)` ritorna `-999,9999998155085`, cioè 1000
che è l'importo di un prestito che si può rimborsare in 12 anni, al tasso del 5%, con una rata di 112,82541.

`nper`
ritorna il numero di rate necessarie per estinguere un prestito;
gli argomenti necessari sono:
. tasso d'interesse unitario (il 5% si indica con 0,05),
. importo della rata,
. importo del prestito (indicato con segno negativo).
`(nper 0,05 112,82541 -1000)` ritorna `12,00000000300939`, cioè 12
che è il numero di rate da 112,82541 necessarie per estinguere un prestito di 1000 regolato al tasso del 5%.

Gli esempi prevedono un tasso del 5% che, salvo strozzinaggio, è un tasso su base annua, pertanto la rata si intende annuale e il relativo numero è rappresentato in anni.
Per avere il tutto rapportato a mesi, semestri, ecc., occorre inserire il tasso mensile, semestrale, ecc.

6.3.5 Array e matrici

Oltre alla funzione `array` per costruire array e matrici, già vista nel Capitolo 4, abbiamo una serie di funzioni per lavorare con questo tipo di dati.

`append`
crea un nuovo array mettendo insieme più array, senza modificare gli array di partenza;
se abbiamo l'array a (1 2 3 4) e l'array b (5 6)
`(set 'c (append a b))` crea l'array c (1 2 3 4 5 6)
gli elementi da mettere insieme devono essere dello stesso tipo array;
per esempio, per aggiungere l'elemento 7 all'array c
non possiamo fare `(append c 7)` e nemmeno `(append c '(7))`
ma dobbiamo fare `(append c (array 1 '(7)))`

`nth`
ritorna un elemento di un array;
gli argomenti da indicare sono l'indice dell'elemento (partendo da 0) e l'array;
con riferimento all'array c dell'esempio di prima, `(nth 1 c)` ritorna 2.

`mat`
introduce un'operazione scalare tra due matrici o tra una matrice e un numero;
gli argomenti da indicare sono:
. la funzione aritmetica da utilizzare (+ - * /, validi, in questo caso, anche per numeri decimali),
. una matrice,
. l'altra matrice o un numero;
`(mat * M1 M2)` ritorna il prodotto scalare tra le matrici M1 e M2
`(mat * M1 5)` ritorna il prodotto scalare tra la matrice M1 e il numero 5.

`multiply`
esegue la moltiplicazione tra due matrici;
gli argomenti sono le due matrici da moltiplicare.

`transpose`
traspone la matrice indicata come argomento.

`invert`
calcola la matrice inversa di quella indicata come argomento.

`det`
calcola il determinante della matrice indicata come argomento.

6.4 Manipolazione di liste

Alle liste si applicano innanzi tutto le due funzioni `append` e `nth` che abbiamo visto per gli array: ovviamente negli argomenti, in questo caso, abbiamo liste anziché array.

Sono poi previste una serie di funzioni, applicabili sia alle liste sia alle stringhe per selezionare elementi (`select`, `find`, `first`, `last`, `rest`), sostituirli (`replace`), spostarli (`reverse`, `rotate`). Per la sintassi, peraltro molto semplice e intuitiva di queste funzioni rimando al manuale.

Se la quoted list ha come primo elemento una funzione, con eval possiamo valutarla.

`'(* 3 4)` ritorna `'(* 3 4)`,

`(eval '(* 3 4))` ritorna `12`.

6.5 Manipolazione di stringhe

Alle stringhe si applicano le funzioni `append` e `nth` che abbiamo visto per gli array e le liste: negli argomenti, in questo caso, abbiamo stringhe anziché array o liste.

Abbiamo poi tutte le funzioni per selezionare elementi (`select`, `find`, `first`, `last`, `rest`), sostituirli (`replace`), spostarli (`reverse`, `rotate`) che abbiamo visto per le liste.

Infine abbiamo funzioni riservate alle stringhe.

Oltre a quelle viste nel Capitolo 4 per le trasformazioni di tipo (`int`, `float`, `string`), ne abbiamo altre (`upper-case`, `lower-case`, rispettivamente per trasformare il contenuto della stringa in caratteri minuscoli o maiuscoli, `explode`, per trasformare la stringa in una lista di singoli caratteri) ed altre ancora, di sempre meno ricorrente utilità, che troviamo nel manuale.

Si può anche trasformare la stringa in una lista di token con la funzione `parse`.

Se a questa funzione passiamo, come solo argomento, una stringa, essa ritorna una lista di token isolandoli in presenza di uno spazio bianco, di una parentesi, di una virgola o di due punti: lo spazio bianco e i due punti si perdono nella tokenizzazione, mentre la virgola e la parentesi rimangono come token.

`(parse "Ieri sono andato a casa di Giuseppe")` ritorna `("Ieri" "sono" "andato" "a" "casa" "di" "Giuseppe")`
`(parse "Ieri(mercoledì)sono andato a casa")` ritorna `("Ieri" "(" "mercoledì" ")" "sono" "andato" "a" "casa")`
`(parse "lunedì,martedì,mercoledì")` ritorna `("lunedì" "," "martedì" "," "mercoledì")`
`(parse "lunedì:martedì:mercoledì")` ritorna `("lunedì" "martedì" "mercoledì")`

Se passiamo come secondo parametro opzionale una stringa la divisione dei token avviene in presenza di questa stringa, che viene eliminata dalla stringa di partenza.

`(parse "lunedìmartedìmercoledì" "edì")` ritorna `("lun" "mart" "mercol" "")`.

Altra funzione che può essere utile la funzione `char` che tramuta un carattere contenuto in una stringa, se necessario identificato dal suo indice, nel corrispondente codice ASCII e viceversa.

`(char "a")` ritorna `97`,

`(char 97)` ritorna `"a"`,

`(char "Vittorio" 1)` ritorna `105`, corrispondente a i, secondo carattere della stringa "Vittorio".

6.6 Manipolazione di simboli

La funzione `sym` trasforma una stringa o un numero in simbolo.

Argomento della funzione è l'elemento da trasformare, che può essere indicato anche richiamando la variabile che lo contiene.

`(sym "pippo")` ritorna `pippo`, che non ha più le virgolette in quanto è diventato un simbolo,

`(sym 12)` ritorna `12`, che non è più un numero ma un simbolo e non è utilizzabile per le funzioni aritmetiche o matematiche,

`(+ 2 (sym 3))` genera un errore per invalidità del secondo parametro.

6.7 Lavorare con i file

Nel paragrafo 6.1, dedicato all'Input/Output, ho mostrato le funzioni che ci consentono di lavorare con i device tastiera, per l'input, e schermo, per l'output: tastiera e schermo sono classificati da NewLISP con il numero `0` e sono i device di default.

Spesso è comodo o necessario avere altre forme di input (per esempio inserire i dati da elaborare prendendoli da un supporto che li contiene, evitando così di doverli digitare sulla tastiera) e/o altre forme di output (per fissare i risultati delle nostre elaborazioni su un supporto rileggibile e non volatile come lo schermo). In entrambi i casi dobbiamo poter lavorare con file da cui prendere o in cui inserire dati e informazioni.

Abbiamo innanzi tutto le seguenti tre funzioni che ci consentono di lavorare direttamente su file.

`write-file`
la sintassi è
`(write-file <file> <contenuto>)`
dove
`<file>` è una stringa contenente l'indirizzo e il nome del file;
`<contenuto>` è una stringa contenente ciò che vogliamo scrivere nel file.
Se il file non c'è viene creato. Se c'è viene sovrascritto, cioè il precedente contenuto va perso.
Il contenuto viene inserito senza a capo: per andare a capo dobbiamo terminare la stringa del contenuto con `\n`.

`append-file`
la sintassi è
`(append-file <file> <contenuto>)`
dove
`<file>` è una stringa contenente l'indirizzo e il nome del file;
`<contenuto>` è una stringa contenente ciò che vogliamo scrivere nel file.
Se il file non c'è viene creato. Se c'è, il contenuto viene aggiunto al contenuto precedente.
Il contenuto viene inserito senza a capo: per andare a capo dobbiamo terminare la stringa del contenuto con `\n`.

`read-file`
la sintassi è
`(read-file <file>)`
dove
`<file>` è una stringa contenente l'indirizzo e il nome del file.
Questa funzione ritorna il contenuto del file come stringa in cui compaiono anche gli eventuali simboli \n (eliminabili ma non sostituibili con la funzione `replace`). Essa non si presta per l'acquisizione di dati da elaborare.

Esistono modi più sofisticati per lavorare con i file e vi dobbiamo ricorrere soprattutto per l'acquisizione di dati da elaborare, in superamento dei limiti della funzione `read-file` che abbiamo appena visto. Alla base di tutto sta la funzione

`open`
questa funzione ritorna un numero intero che può essere la chiave attraverso cui interagire con un file; la sua sintassi è:
`(open <file> <modalità>)`
dove
`<file>` è una stringa contenente l'indirizzo e il nome del file;
`<modalità>` è una stringa per indicare cosa vogliamo fare con il file:
`"write"` o semplicemente `"w"` per scriverci sopra,
`"read"` o semplicemente `"r"` per leggerlo,
`"update"` o semplicemente `"u"` per scriverci sopra e leggerlo,
`"append"` o semplicemente `"a"` per aggiungervi dati e leggerlo.

Occorre fare molta attenzione alle modalità `write` e `update`. La modalità `write`, se il file indicato non c'è ancora lo crea e lo rende disponibile per accogliere informazioni, se il file indicato c'è già tutto quanto vi inseriamo va a sostituire il precedente contenuto. Quest'ultima cosa avviene anche inserendo informazioni in un file aperto in modalità `update`.
Se vogliamo scrivere ulteriori informazioni in un file salvando quelle che vi sono già contenute dobbiamo usare la modalità `append`.
Il numero intero generato dalla funzione `open` può essere convenientemente utilizzato per attivare un device alternativo a quello di default, come abbiamo già detto identificato con il numero 0, identificandolo più comodamente con un simbolo.
A questo scopo usiamo la funzione `device` che ha la seguente sintassi:
`(device <simbolo>)`
dove `<simbolo>` sottende il numero generato dalla funzione open oppure, per tornare al default (tastiera e schermo), 0.
Fintanto che è attivo il device collegato al file possiamo lavorare sul file stesso con le solite funzioni `println`, `print` e `read-line` che abbiamo visto per il normale Input/Output nel paragrafo 6.1.
Il device si disattiva con
`(close <simbolo>)`
e si torna così al default tastiera/schermo.

Un paio di esempi utili per capirci:

```
(set 'f (open "/home/vittorio/nomi" "read"))
(device f)
(set 'lista_nomi '())
(while(read-line f)(set 'lista_nomi(append lista_nomi(list(sym(current-line))))))
(close f)
```

Con questo codice ci predisponiamo a lavorare sul file `nomi`, che contiene l'elenco dei nomi dei miei amici, proponendoci di inserire questi nomi in una lista di simboli.
Assegniamo al simbolo f il numero corrispondente all'apertura del file per lettura e, con questo simbolo creiamo il device di lavoro.
Creiamo una variabile globale `lista_nomi` contenente, per intanto, una lista vuota.
Avviamo un ciclo di lettura delle righe dell'elenco contenuto nel file, trasformiamo in simbolo la stringa letta (`current_line`), ne facciamo una lista e andiamo ad aggiungere questa lista alla lista dei nomi nella variabile globale `lista_nomi`.
Chiudiamo il device.
Ora abbiamo a disposizione una variabile contenente la lista dei simboli dei nomi dei miei amici e possiamo lavorare con questi dati: rammento che in LISP tutto avviene attraverso liste e quanto abbiamo fatto si è reso necessario per trasferire in una lista il contenuto di un qualsiasi elenco che avevamo a disposizione.

```
(set 'f (open "/home/vittorio/dati.csv" "read"))
(device f)
(set 'lista_dati '())
(while(read-line)(set 'lista_dati(append lista_dati(list(float(current-line))))))
(close f)
(stats lista_dati)
```

Con quest'altro codice facciamo un po' la stessa cosa di prima, con la differenza che l'obiettivo è quello di creare una lista contenente dati numerici da elaborare, traendoli da un file .csv.
In questo caso il codice contiene anche un esempio di come sulla nostra lista possiamo intervenire: nel caso calcolando le statistiche che caratterizzano la serie numerica ivi contenuta.
Unica precauzione: nel file .csv il separatore non può essere la virgola (rammento che il NewLISP italiano usa la virgola per separare le cifre decimali) e va eventualmente sostituito con il punto e virgola o con uno spazio, inoltre i dati devono essere elencati in colonna. Le predisposizioni possono eventualmente essere messe a punto ripassando preventivamente il file .csv con un foglio di calcolo.

7 Costruire una funzione

Può accadere che tra le funzioni preconfezionate, anche se sono tantissime, non troviamo quella che ci serve o che vogliamo fare una certa cosa in maniera diversa da come la si potrebbe fare con una funzione preconfezionata.

Per questi casi NewLISP ci mette a disposizione la funzione `define` attraverso la quale possiamo costruire noi la funzione che ci serve con la sintassi
`(define(<nome_funzione> <parametro> <parametro>..)(<istruzione>)(<istruzione>)..)`
dove
`<nome_funzione>` è il nome che diamo alla funzione per poterla richiamare,
`<parametro>` indica uno o più parametri richiesti perché la funzione produca risultato,
`<istruzione>` indica una o più istruzioni sul compito o sui compiti che la funzione debba svolgere.

Se non si indicano parametri, la funzione potrà essere richiamata senza indicare parametri, a patto che le istruzioni non abbiano bisogno di dati o li trovino disponibili altrimenti.

Esempi:

`(define(raddoppia x)(* x 2))`
è una funzione che serve per raddoppiare il numero indicato;
richiamandola con l'istruzione `(raddoppia 5)` essa ritorna il numero 10.

`(define(triplica)(* y 3))`
è una funzione che serve per triplicare un numero, ma non lo richiede;
essa, se richiamata con l'istruzione `(triplica)`, che non richiede parametri, produce un risultato solo se trova una variabile y contenente un numero.

`(define(salutami)(println "Ciao, Vittorio!"))`
è una funzione attraverso la quale mi faccio salutare dal computer;
essa, se richiamata con l'istruzione `(salutami)`, che non richiede parametri, produce il risultato, scrivendo il saluto, in quanto ha a disposizione tutto ciò che serve (la stringa da scrivere).

* * *

Esiste un elegante modo alternativo per costruire una funzione in NewLISP, che consiste nel creare una variabile che contiene una funzione scritta con un'espressione lambda. La sintassi è la seguente:
`(set <simbolo> (lambda(<parametro> <parametro>..)(<istruzione>)(<istruzione>)..))`

Le tre funzioni create prima, con questa sintassi si creerebbero così:

```
(set 'raddoppia (lambda(x) (* x 2)))
(set 'triplica (lambda() (* y 3)))
(set 'salutami (lambda() (println "Ciao, Vittorio!")))
```

* * *

Faccio un esempio meno banale.

Tra le funzioni di matematica finanziaria precostituite ne manca una di fondamentale importanza: quella per determinare il tasso equivalente, cioè il tasso per una frazione di anno equivalente ad un tasso annuo.
Quando usiamo la funzione `pmt` per determinare la rata di rimborso di un prestito, infatti, dobbiamo inserire il parametro tasso di interesse riferito al periodo di rata: nell'esempio che avevamo visto l'ipotesi era di un prestito di 12 anni con rata di rimborso annuale e avevamo indicato il tasso annuo 0,05 (5%).
Ma supponiamo ora che il prestito duri un solo anno e sia da rimborsare in dodici rate mensili. In questo caso dobbiamo inserire il tasso mensile, anche se, quando concordiamo le condizioni del prestito, parliamo di tasso annuo. In genere, concordato, per esempio, il tasso annuo del 5%, per semplicità si determina il tasso mensile dividendo per 12 il tasso annuo. Si tratta di

una pratica semplificatoria tollerabile se siamo in presenza di tassi bassi e di prestiti di importi modesti, ma si compie un grossolano errore, tra l'altro andando contro la legge che regola il TAEG: in regime di capitalizzazione composta mensile, infatti, non è vero che che il tasso mensile corrisponde a 1/12 del tasso annuo e, in presenza di tassi e importi di prestito non modesti, la differenza si nota.
Per fare le cose per bene, pertanto, conviene che disponiamo di una funzione che calcola i tassi equivalenti utilizzando la nota formula $i_k = (1+i)^{\frac{1}{k}} - 1$ che fornisce il tasso per $\frac{1}{k}$ di anno equivalente al tasso annuo i.
Ce la possiamo costruire con il seguente codice

```
(define (tasso_equivalente i k) (sub (pow (add 1 i) (div 1 k)) 1))
```

* * *

Le funzioni che ci servono possiamo costruirle nel corso del programma e richiamarle quando le utilizziamo.

Ma, se la funzione è laboriosa da costruire, come comincia ad esserlo quella appena vista del tasso equivalente, ci conviene salvarla, con un nome sufficientemente descrittivo, in un file con estensione .lsp e in una directory dedicata.
Nel mio sistema Linux, per esempio, creo nella mia home una directory chiamata `funzioni_newlisp`.
Scrivo il codice per creare la funzione tasso_equivalente con un editor di testo e salvo il file `tasso_equivalente.lsp` nella directory `funzioni_newlisp`.
Ho così a disposizione la funzione per richiamarla quando e dove serve semplicemente caricandola nella shell interattiva o nel programma, senza più riscriverla, con l'istruzione
`(load "/home/vittorio/funzioni_newlisp/tasso_equivalente.lsp")`.

Data questa istruzione possiamo calcolare la rata mensile del nostro prestito di 1000 con l'istruzione
`(pmt (tasso_equivalente 0,05 12) 12 1000)` che ritorna `-85,55659945661426`
solo lievemente inferiore al valore che otterremmo con la formula semplificata per ottenere il tasso mensile dividendo per 12 il tasso annuo
`(pmt (div 0,05 12) 12 1000)` ritorna `-85,60748178846745`.
Se però abbiamo a che fare con un mutuo per la casa di 250000, al tasso del 6%, rimborsabile in 15 anni con rate semestrali
`(pmt (tasso_equivalente 0,06 2) 30 250000)` ritorna `-12682,8735102168`
`(pmt (div 0,06 2) 30 250000)` ritorna `-12754,81483006314`
con una differenza di 72 per ogni rata, che, proiettata su 30 rate genera un maggiore rimborso di 2160 che non sarebbe dovuto.

8 Interattività con l'utente

Spesso, soprattutto quando si tratta di programmi destinati ad eseguire calcoli, diventa utile creare un programma che astragga dai parametri necessari alle varie funzioni per ottenere risultati e che chieda questi parametri all'utente in sede di esecuzione.

Il seguente programma, che scriviamo con un editor di testo e salviamo in un file cerchio.lsp, calcola circonferenza e area del cerchio in presenza del raggio 2,5.

```
(constant 'pi (acos -1))
(mul 2,5 pi 2)
(mul (pow 2,5 2) pi)
```

ma ha due difetti:
. innanzi tutto tiene per sé i risultati,
. inoltre è fissamente dedicato ad un cerchio di raggio 2,5 e dovremmo riscriverlo ogni volta che vogliamo fare i calcoli con un raggio diverso.

Al primo difetto possiamo ovviare utilizzando la funzione per l'output `println`. Se riscriviamo il programma in questo modo

```
(constant 'pi (acos -1))
(println (mul 2,5 pi 2))
(println (mul (pow 2,5 2) pi))
(exit)
```
otteniamo la stampa del valore della circonferenza e del valore dell'area del cerchio e l'uscita dall'interprete grazie alla funzione `exit`.
Se ci posizioniamo nella directory dove è memorizzato il file cerchio.lsp e scriviamo il comando
`newlisp cerchio.lsp`
otteniamo in risposta
```
15,70796326794897
19,63495408493621
```
nell'ordine circonferenza ed area del cerchio di raggio 2,5.

Al secondo difetto ovviamo ricorrendo alla funzione `println` per l'output di un messaggio di richiesta all'utente e successivamente alla funzione `read-line` per l'input di quanto l'utente scriverà sulla tastiera. Il nostro programma diventa
```
(constant 'pi (acos -1))
(println "Inserisci il raggio del cerchio")
(set 'raggio (float(read-line)))
(println (mul raggio pi 2))
(println (mul (pow raggio 2) pi))
(exit)
```
Se lanciamo questo programma ci viene chiesto di indicare il raggio del cerchio e, una volta che l'avremo digitato sulla tastiera e premuto INVIO, otterremo circonferenza e area del cerchio avente il raggio da noi indicato.

Se vogliamo perfezionare il tutto, accontentandoci dei risultati espressi con sole due cifre decimali, possiamo modificare così il nostro programmino cerchio.lsp:
```
(constant 'pi (acos -1))
(println "Inserisci il raggio del cerchio")
(set 'raggio (float(read-line)))
(set 'circonferenza (mul raggio pi 2))
(set 'area (mul (pow raggio 2) pi))
(println "Per un cerchio di raggio " raggio)
(println "la circonferenza è:" (format {%15.2f} circonferenza))
(println "e l'area è:        " (format {%15.2f} area))
(exit)
```

9 Strutture di controllo

NewLISP ci offre il modo di controllare l'esecuzione del programma attraverso funzioni per condizionare l'esecuzione di una o più istruzioni al verificarsi di certe condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

Per esprimere condizioni abbiamo a disposizione le funzioni di confronto
< minore
> maggiore
= uguale
<= minore o uguale
>= maggiore o uguale
!= diverso

9.1 Esecuzione condizionale

`when`
la sua sintassi è

`(when <condizione> <istruzione>)`
se la condizione è vera viene eseguita l'istruzione, altrimenti non succede nulla e il programma continua con le altre istruzioni, se ce ne sono, non dipendenti dalla condizione.
```
(when (> x y) (println "SI"))
(println "Ciao")
```
se il valore della variabile `x` è superiore a quello della variabile `y` in output abbiamo la scritta `SI` e la scritta `Ciao`, in caso contrario in output abbiamo la sola scritta `Ciao`.

`if`
la cui sintassi prevede tre possibilità
```
(if <condizione> <istruzione>)
(if <condizione> <istruzione> <istruzione>)
(if <condizione> <istruzione> <condizione> <istruzione> ...)
```
Nel primo caso otteniamo, tali e quali, gli effetti della precedente funzione `when`.
Nel secondo caso, se la condizione è vera viene eseguita la prima istruzione, altrimenti viene eseguita la seconda (è ciò che in altri linguaggi si ottiene con l'istruzione if ... else).
```
(if (> x y) (println "SI") (println "NO"))
(println "Ciao")
```
se il valore della variabile `x` è superiore a quello della variabile `y` in output abbiamo la scritta `SI` e la scritta `Ciao`, in caso contrario in output abbiamo la scritta NO e la scritta `Ciao`.
Nel terzo caso possiamo abbinare quante istruzioni vogliamo ad altrettante condizioni.

```
(if
(> x y) (println "maggiore")
(< x y) (println "minore")
(= x y) (println "uguale"))
(println "Ciao")
```
se il valore della variabile `x` è superiore a quello della variabile `y` in output abbiamo la scritta `maggiore` e la scritta `Ciao`, se il valore della variabile `x` è inferiore a quello della variabile `y` in output abbiamo la scritta `minore` e la scritta `Ciao`, se il valore della variabile `x` è uguale a quello della variabile `y` in output abbiamo la scritta `uguale` e la scritta `Ciao`.

9.2 Ripetizione

NewLISP prevede molte funzioni per il looping. Si va da quelle che ripetono azioni per ciascun elemento di una lista o di una stringa a quelle che ripetono azioni per un numero prefissato di volte, da quelle che ripetono fino a quando succede qualche cosa a quelle che ripetono fino a quando si verificano certe condizioni.

`for`
è la classica per ottenere una ripetizione per un numero prefissato di volte; la sintassi è
`(for(<simbolo> <da> <a>)<istruzione> <istruzione>...)`
dove
`<simbolo>` crea una variabile locale che funge da contatore e, volendo, da variabile
`<da>` è il numero intero da cui si comincia a contare
<a> è il numero intero con cui si finisce di contare
`<istruzione>` è ciò che si deve fare e ce ne possono essere più di una.
`(for (i 1 5) (println "Ciao"))` scrive cinque volte la parola `Ciao`.
`(for (i 1 5) (println (pow i)))` scrive i numeri 1 4 9 16 25.

`dotimes`
altra per ottenere una ripetizione per un numero prefissato di volte; la sintassi è
`(dotimes (<simbolo> <numero>) <istruzione> <istruzione> ... )`
dove
`<simbolo>` crea una variabile locale che funge da contatore (partendo da 0)
`<numero>` è il numero intero di volte per cui si devono eseguire le istruzioni

<istruzione> è ciò che si deve fare e ce ne possono essere più di una.
(dotimes (i 5) (println "Ciao")) scrive cinque volte la parola Ciao.
(dotimes (i 5) (println (pow (+ i 1)))) scrive i numeri 1 4 9 16 25.

dolist
applica una o più istruzioni ad ogni elemento di una lista o array; la sintassi è:
(dolist(<simbolo> <lista>) <istruzione> <istruzione> ...)
dove
<simbolo> crea una variabile locale che assume via via i valori degli elementi della lista
<lista> è la lista dei valori da scorrere, può essere richiamata se contenuta in una variabile
<istruzione> è ciò che si deve fare e ce ne possono essere più di una.
(set 'l '())(dolist (i '(1 2 3)) (set 'l (append l (list (pow i 3))))) inserisce in una lista l i valori 1 8 27.

dostring
applica una o più istruzioni ad ogni elemento di una stringa; la sintassi è:
(dostring (<simbolo> <stringa>) <istruzione> <istruzione> ...)
dove
<simbolo> crea una variabile locale che assume i codici ASCII degli elementi della stringa
<stringa> è la stringa da scorrere, può essere richiamata se contenuta in una variabile
<istruzione> è ciò che si deve fare e ce ne possono essere più di una.
(dostring (i "abc") (println i)) scrive i numeri 97 98 99.

until
esegue una o più istruzioni fino a quando diventa vera una condizione; la sintassi è:
(until <condizione> <istruzione> <istruzione> ...)
(set 'contatore 0)
(until (= contatore 4) (println "Ciao") (inc contatore)) scrive 4 volte Ciao

while
esegue una o più istruzioni fino a quando è vera una condizione; la sintassi è:
(while <condizione> <istruzione> <istruzione> ...)
(set 'contatore 0)
(while (< contatore 4) (println "Ciao") (inc contatore)) scrive 4 volte Ciao

10 Estensioni

Tutto ciò che abbiamo visto finora è la struttura di base del linguaggio NewLISP ed è quanto basta per famigliarizzare con il linguaggio e creare qualche programma di utilità.

Per chi si appassioni e intenda andare oltre segnalo ora le direzioni dei possibili approfondimenti.

Documentazione utile, oltre al più volte citato manuale che possiamo consultare o scaricare da *http://www.newlisp.org/*, è il file PDF Introduction to Newlisp scaricabile dal sito Wikimedia (per trovarlo scrivere nella barra di ricerca Google "introduction to newlisp" e scegliere la voce Introduction to newLISP - Wikimedia Commons).

10.1 Macro

La macro è una funzione creata da noi che ci dà modo di controllare se e quando siano da valutare gli argomenti.

10.2 Context

L'uso dei context ci consente di separare il programma che stiamo creando in compartimenti all'interno dei quali valgono definizioni di simboli, funzioni e quant'altro che non sono visti in altri compartimenti.

Senza che ce ne accorgiamo, se non creiamo noi un context, NewLISP lavora in un context definito MAIN: lo possiamo verificare scrivendo `(context)` nella shell di NewLISP e battendo INVIO.

10.3 FOOP

Il Functional Object-Oriented Programming è lo stile di programmazione per oggetti che possiamo utilizzare in NewLISP.

10.4 Modules

Il module è una raccolta di funzioni inserita in un context.

Se abbiamo installato NewLISP compilando il source, nella directory di installazione abbiamo una sotto-directory `modules` che ne contiene una ventina: per interfacciamenti con vari tipi di database, per calcoli statistici, per crittografia, ecc.

10.5 Interfaccia grafica

E' mia convinzione che NewLISP, per dare il meglio di sé, non abbia bisogno di interfaccia grafica e si comporti benissimo con il vecchio sistema del terminale a riga di comando.

Per chi voglia vestirlo con questa modernità esistono due strade, alle quali si può accedere andando sul sito *http://www.newlisp.org/*, nella scheda LINKS, nella zona GUI DEVELOPMENT ENVIRONMENTS, dove troviamo i link per quanto brevemente descritto qui di seguito.

In ogni caso bene sapere che il programma contenente una interfaccia grafica può essere "compilato" come ogni altro programma scritto in NewLISP come abbiamo visto nel Capitolo 2: tuttavia sul computer dove si vuole far girare l'eseguibile può non esserci installato NewLISP ma deve essere installato tutto ciò che serve per eseguire l'interfaccia grafica. Il file eseguibile dopo la "compilazione", infatti, contiene semplicemente l'interprete NewLISP ma tutto il resto deve trovarsi esattamente dove l'interprete lo cerca.

10.5.1 tk

Il programma viene scritto in linguaggio NewLISP richiamando le funzioni tk necessarie per creare la GUI.

Per l'esecuzione occorre un interprete NewLISP particolare e, per il sistema Windows, lo troviamo nel file `newlisptk.exe` che scarichiamo seguendo il link. Per i sistemi Linux e Mac troviamo le istruzioni, sempre sul sito *http://www.newlisp.org/*, nella scheda TIPS & TRICKS, nella zona CONNECT TO OTHER LANGUAGES AND LIBRARIES, sotto la voce TK AND NEWLISP.

Si tratta di una strada abbastanza complicata che non consiglio ai principianti.

10.5.2 Guiserver

Percorribile con un po' meno fatica la strada del Guiserver.

Guiserver è un server grafico scritto in linguaggio Java e contenuto in un file denominato `guiserver.jar`: esso opera a patto che sul computer sia installata la Java Virtual Machine. Esiste poi un file denominato `guiserver.lsp` che lo interfaccia.

Ci procuriamo quanto serve scaricando il file `guiserver-x.xx.tgz` (nel momento in cui scrivo `guiserver-1.67.tgz`) dal sito *http://www.newlisp.org/*, scheda LINKS, zona GUI DEVELOPMENT ENVIRONMENTS.

Scompattiamo il file compresso `.tgz`
. se siamo su Windows in `C:\Program Files (x86)\newlisp`,
. se siamo su Linux o Mac OS X in `/usr/local/share/newlisp`.
Lo scompattamento crea la directory `guiserver`.

Da questa directory, con TAGLIA/INCOLLA, estraiamo i file `guiserver.lsp` e `guiserver.jar` e li trasferiamo fuori dalla directory `guiserver` in modo che siano direttamente accessibili nella directory `newlisp`.
Ciò che rimane nella directory `guiserver`, che, a questo punto, potremmo trasferire dove ci è più comodo, è una raccolta di programmi studiando i quali vediamo come si lavora con `guiserver`, una guida alla sintassi dei comandi accessibile cliccando su `index.html` e il file `newlisp-edit.lsp` che, lanciato con il comando a terminale `newlisp newlisp-edit.lsp`, ci offre un ottimo editor per programmare con il linguaggio NewLISP.