a cura di Francesco Romani

LISP & *Mathematica*

Continuiamo a ripassare i fondamenti dell'Informatica con l'ausilio di *Mathematica*. Dopo le Macchine di Turing è la volta del LISP: il più elegante dei linguaggi di programmazione e l'unico di quelli "storici" che è ancora usato per i suoi meriti intrinseci e non solo per i miliardi di dollari investiti (vedi COBOL, FORTRAN) o per una sua "supposta" semplicità (vedi BASIC).

Introduzione

Il LISP (**LIS**t **P**rocessing) è un linguaggio di programmazione nato negli anni '60 per il trattamento di liste e, in generale, dell'informazione non numerica. Per la sua eleganza e pulizia è tuttora usato per applicazioni di intelligenza artificiale. Costruire un interprete per il LISP senza "fronzoli" e senza controlli di errore è un semplice e istruttivo esercizio di programmazione. Ovviamente, il codice risultante è privo di ogni utilità pratica dato che mancano tutte quelle aggiunte che rendono la programmazione più agile ed efficiente e che ogni errore nel programma LISP da interpretare si riflette in un errore, spesso letale, nell'interprete.

L'impostazione della trattazione e gli esempi sono tratti dai testi di Fiorentino *et al.* (1996,1997) a loro volta ispirato da Aiello e Montanari (1972).

I dati su cui opera un programma LISP sono le cosiddette S-espressioni che hanno la seguente grammatica:

<atomo> ::= *identificatore*

<S-expr> ::= <atomo> | **(**<S-expr>•<S-expr>**)** | <lista>

<lista> ::= **()** | **(**<S-expr>**)** | **(**<S-expr> ... <S-expr>**)**

Esistono alcuni identificatori speciali (**NIL**, **T**, **QUOTE**, ecc.) a cui viene attribuito un significato speciale (che verrà esposto nel seguito).

Le liste sono considerate equivalenti a S-espressioni del primo tipo nel modo seguente:

() ≡ **NIL** (la lista vuota);

(a) ≡ **(a• NIL)**;

(a b) ≡ **(a• (b • NIL))**;

(a b ... z) ≡**(a• (b• ... (z• NIL)...))**;

dove **a**, **b**, ... **z** sono generiche S-espressioni. Con questa posizione tutte le S-espressioni possono essere rappresentate internamente come alberi binari. L'equivalenza tra liste ed S-espressioni corrisponde alla equivalenza tra alberi liberi e alberi binari mediante la tecnica primogenito-primo fratello.

Ad esempio la lista
(a (b c) d (e f g))

corrisponde alla S-espressione
(a•((b•(c•NIL))•(d•((e• (f•(g• NIL)))• NIL))))

Il LISP mette a disposizione alcune funzioni elementari (molto simili a quelle usualmente definite per gli alberi binari) che operano sulle S-espressioni. Le prime due sono predicati (**T** è l'equivalente del **True** e **NIL** del **False**):

atom[x] vale **T** se **x** è un simbolo atomico, **NIL** altrimenti;

eq[x, y] vale **T** se l'atomo **x** è uguale all'atomo **y**, **NIL** altrimenti.

Per le operazioni seguenti **x** e **y** possono essere S-espressioni qualsiasi:

car[(x•y)] vale **x**;

cdr[(x•y)] vale **y**;

cons[x, y] vale **(x•y)**.

Si noti il comportamento delle operazioni **car**, **cdr** e **cons** sulle liste.

car restituisce il primo elemento di una lista:
car[(a (b c) d)] = a

cdr restituisce la lista privata del primo elemento:
cdr[(a (b c) d)] = = ((b c) d)

cons inserisce il suo primo argomento come primo ele-

mento nella lista passata come secondo argomento:
cons[(a b),(c)] = ((a b) c).

Un programma LISP è scritto a partire da queste definizioni base con la composizione di funzioni e un costrutto condizionale del tipo

$[p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_k \rightarrow e_k]$

che assume il valore della prima espressione e_i per cui il predicato p_i vale **T**. Questo costrutto è del tutto equivalente al **Which** di *Mathematica* (basta cambiare "→" in ",", aggiungere **Which** davanti).

L'uso sistematico della ricorsione, a partire da questi elementi, permette di scrivere un qualunque programma; in altre parole la potenza del linguaggio LISP è la stessa di linguaggi di programmazione tipo Pascal o C.

Per esempio, la funzione che seleziona il primo simbolo atomico in una S-espressione è la seguente:
ff[x] = [atom[x] → x, T → ff[car[x]]]

Alcune funzioni molto importanti che saranno usate nel seguito sono:

null[x] vale **T** se **x** è **NIL**;

list[x, y, ... z] = (x y … z);

append[x, y] opera su due liste e ne restituisce la concatenazione;

pairlis[x, y, a] concatena ad **a** la lista delle coppie degli elementi corrispondenti in **x** e **y**;

assoc[x; y] ricerca nella lista **y** l'elemento corrispondente a **x**.

Sono anche comuni le seguenti abbreviazioni:

caar[x] è l'abbreviazione di **car[car[x]]**;

cadar[x] è l'abbreviazione di **car[cdr[car[x]]]**, ecc.

Una delle caratteristiche del LISP consiste nel fatto che un programma LISP può essere scritto sotto forma di S-espressione, fatto che permette di modificare un programma durante la sua esecuzione e di scrivere facilmente una funzione universale (detta **evalquote**) che accetta in *input* una lista formata da una funzione e dai suoi argomenti e ne restituisce il valore.

Per scrivere i programmi LISP sotto forma di S-espressioni si deve introdurre il modo di specificare gli argomenti e assegnare il nome di una funzione per poterla richiamare ricorsivamente.

Per il primo scopo si usa il seguente costrutto (detto λ formalismo);

λ[<lista variabili>, <definizione>]

Il nome viene attribuito con il costrutto

label[<nome>, <definizione>]

per esempio:
label[pippo. λ[[x], cons[x, A]]]

definisce la funzione **pippo[x]** che restituisce **(x•A)**.

La traduzione dei programmi LISP in S-espressioni avviene secondo le seguenti regole (la traduzione dell'espressione **e** si indica con **e***).

Variabili e nomi di funzioni si scrivono in caratteri maiuscoli; per esempio:
car* = CAR

Un'applicazione di funzione si scrive come la lista della funzione e dei suoi argomenti:

$f[e_1, e_2, \ldots e_k]^* = (f^* e_1^* e_2^* \ldots e_k^*)$

Il condizionale si scrive come la lista formata dall'atomo **COND** e dalle coppie predicato-espressione:

$[p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_k \rightarrow e_k]^* =$

$(\text{COND } (p_1\, e_1)\ (p_2\, e_2) \ldots (p_k\, e_k))$

In modo analogo si trattano λ e **label**:

$\lambda[[l_1, l_2, \ldots], d]^* = (\text{LAMBDA } (l_1^*\ l_2^*)\ d^*)$

label[n, d] = (LABEL n* d*)

Le S-espressioni (ovvero le costanti del programma) vengono inserite in una lista preceduta dall'atomo **QUOTE**:
s* = (QUOTE s)

Per esempio
label[pippo, λ[[x],cons[x, A]]]* =
(LABEL PIPPO
(LAMBDA (X) (CONS X (QUOTE A))))

In pratica i programmatori LISP esperti scrivono direttamente le S espressioni.

La funzione universale **evalquote** ha la proprietà che data una definizione di funzione **fn** e *k* argomenti $a_1, a_2, \ldots, a_k$ vale:

$\text{evalquote}[fn^*, (a_1^*\ a_2^* \ldots a_k^*)] =$

$fn[a_1, a_2, \ldots, a_k]$

evalquote può essere definita con un programma LISP di poche righe. Per costruire un interprete LISP funzionante bisogna definire:

1) una struttura interna per le S-espressioni;

2) i programmi che convertono le stringhe di ingresso nella struttura interna;

3) i programmi che trasformano la struttura interna in S-espressioni;

4) implementare le funzioni elementari;

5) implementare **evalquote**.

Per un'implementazione Pascal si veda Fiorentino *et al.* (1996), per un'implementazione C si veda Fiorentino *et al.* (1997), per un'implementazione in Mathematica si veda il paragrafo successivo.

L'interprete LISP in *Mathematica*

Vediamo come si realizzano i vari passi dell'implementazione.

1) struttura interna

Il punto 1 non ha una soluzione unica, in genere si può privilegiare l'efficienza o la chiarezza o la semplicità di implementazione. La nostra scelta che cerca di privilegiare quest'ultimo aspetto è la seguente:

gli atomi sono rappresentati con simboli;

(a•b) è rappresentato come **cons[a,b]**;

T è rappresentato con **True**.

Lasciamo come esercizio al lettore la sperimentazione di altre alternative.

2) input

Il punto 2 viene di conseguenza. Bisogna realizzare un analizzatore sintattico che trasforma una stringa nella rappresentazione interna della S-espressione corrispondente. Rinunciando alla gestione degli errori, in *Mathematica* questo si fa in poche righe.

In[1]:=
```
parse[Sesp_]:=
 ToExpression[
  StringReplace[Sesp, {"("->"{",
     ")"->"}", "."->",dot,",
     " "->","}]]/.List->list
```

In[2]:=
```
list[a_,dot,b_]:=cons[a,b]
list[x_]        :=cons[x,NIL]
list[x_,y__]    :=cons[x,list[y]]
```

In[3]:=
```
T=True;
```

Ecco un esempio di funzionamento:

In[4]:=
```
s=parse["((a.b) (c.d) (QUOTE T))"]
```

Out[4]=
```
cons[cons[a,b],cons[cons[c,d],
   cons[cons[QUOTE, cons[True, NIL]], NIL]]]
```

3) output

La funzione **format** traduce una S-espressione in stringa stampabile. Viene realizzato il formato semplificato per le liste, ovvero espressioni come **(a•(b•NIL))** vengono stampate con la rappresentazione alternativa **(a b)**.

In[1]:=
```
c1[a_,NIL]:=l1[a];
c1[a_,l1[b__]]:=l1[a,b];
```

In[2]:=
```
tost[True] :="T"
tost[False]:="NIL"
tost[a_?AtomQ]:=ToString[a]
tost[a_String]:=a
tost[c1[a_,b_]]:=
  "("<>tost[a]<>"."<>tost[b]<>")";
tost[l1[a__]]:=
  "("<>seq[tost/@{a}]<>")";
```

In[3]:=
```
seq[a_List]:=
 First[a]<>StringJoin[{" "<>#}&/@Rest[a]]
```

In[4]:=
```
format[sesp_]:=tost[sesp/.cons->c1]
```

Ecco un esempio di funzionamento (si stampa la S-espressione precedente):

In[5]:=
```
format[s]
```

Out[5]=
```
((a.b) (c.d) (QUOTE T))
```

4) funzioni elementari

L'implementazione delle funzioni elementari è grandemente facilitata dal fatto che quasi tutte hanno un equivalente in *Mathematica*.

In[1]:=
```
atom=AtomQ;
```

```
eq=SameQ;
car=First;
cdr[cons[a_,b_]]:=b;
```

La funzione **cons** non viene implementata in quanto realizza la rappresentazione interna.

Le abbreviazioni ci permetteranno di semplificare l'interprete.

In[2]:=
```
caar[x_]:=car[car[x]];
cadr[x_]:=car[cdr[x]];
...
```

Vediamo ora l'implementazione delle funzioni ausiliarie. Si noti l'uso di **===** al posto di **==** per forzare il risultato a **True** o **False**.

In[3]:=
```
null[x_]:= x===NIL
```

La funzione **list** è già stata definita per realizzare le funzioni di *input*.

In[4]:=
```
list[a,b,c]
%//format
```

Out[4]=
```
cons[a, cons[b, cons[c, NIL]]]
(a b c)
```

In[5]:=
```
append[x_, y_] := If[null[x], y,
     cons[car[x], append[cdr[x], y]]]
```

In[6]:=
```
append[list[a,b,c],list[e,f]]
%//format
```

Out[6]=
```
cons[a, cons[b, cons[c, cons[e, cons[f,
NIL]]]]]
(a b c e f)
```

In[7]:=
```
pairlis[x_, y_, a_] := If[null[x] , a,
       cons[cons[car[x],car[y]],
            pairlis[cdr[x], cdr[y],a]]]
```

In[8]:=
```
la=pairlis[list[1,2,3],list[a,b,c],NIL]
%//format
```

Out[8]=
```
cons[cons[1,a],
    cons[cons[2,b],cons[cons[3,c], NIL]]]
((1.a) (2.b) (3.c))
```

In[9]:=
```
assoc[x_, y_] := If[eq[caar[y],x], car[y],
      assoc[x, cdr[y]]]
```

In[10]:=
```
assoc[2,la]
%//format
```

Out[10]=
```
cons[2, b]
(2.b)
```

5) evalquote

Siamo ora pronti a scrivere l'interprete vero e proprio. In pratica basta prendere l'interprete LISP scritto in LISP (si vedano gli articoli citati) e con pochi semplici cambiamenti sintattici si ottiene l'interprete scritto in *Mathematica*

Le modifiche da apportare sono le seguenti:

I caratteri "**;**" e "→" divengono virgole. Si premette **Which** ai condizionali, si applica parse agli argomenti di **evalquote**. Una modifica più sottile consiste nelll'applicare **TrueQ**, quando necessario, per forzare il risultato della valutazione dei condizionali a **True** o **False**.

La funzione **evalquote** riceve una funzione e una lista di argomenti e li passa ad **apply** insieme ad una lista associativa vuota. Durante il funzionamento dell'interprete la lista associativa si comporterà come lo *stack* del Pascal e del C e permetterà la gestione delle chiamate ricorsive.

In[1]:=
```
evalquote[fun_,args_]:=
   apply[parse[fun],parse[args],NIL]
```

La funzione **apply[fn, x, a]** valuta la funzione **fn** sugli argomenti **x** usando la lista associativa **a**; **apply** chiama ricorsivamente **eval**.

In[2]:=
```
apply[fn_, x_, a_] := Which[
 atom[fn] ,Which[
  eq[fn,CAR],    caar[x],
  eq[fn, CDR],   cdar[x],
  eq[fn, CONS],  cons[car[x], cadr[x]],
  eq[fn, ATOM],  atom[car[x]],
  eq[fn, EQ] ,   eq[car[x], cadr[x]],
  True,          apply[eval[fn, a],x,a]],
 eq[car[fn], LAMBDA] , eval[
      caddr[fn], pairlis[cadr[fn],x,a]],
 eq[car[fn], LABEL] ,  apply[
      caddr[fn], x,
      cons[cons[cadr[fn],caddr[fn]],a]]]
```

La funzione **eval[e, a]** valuta l'espressione **e** usando la

lista associativa **a**; **eval** chiama ricorsivamente **apply**.

In[3]:=
```
eval[e_, a_] := Which[
 atom[e],       cdr[assoc[e,a]],
 atom[car[e]], Which[
       eq[car[e], QUOTE], cadr[e],
       eq[car[e], COND],  evcon[cdr[e],a],
       True,              apply[
        car[e], evlis[cdr[e],a],a]],
 True, apply[
        car[e], evlis[cdr[e],a],a]])
```

La funzione **evcon[c, a]** valuta il condizionale **v** usando la lista associativa **a**.

In[4]:=
```
evcon[c_, a_] := Which[
 TrueQ[eval[caar[c],a]], eval[cadar[c],a],
 True,                   evcon[cdr[c],a]]
```

La funzione **evlis[m, a]** valuta una lista **m** usando la lista associativa **a**.

In[5]:=
```
evlis[m_, a_] := Which[
 null[m], NIL,
 True,    cons[eval[car[m], a],
               evlis[cdr[m], a]]]
```

Tutto qui! Abbiamo scritto l'interprete di un linguaggio di programmazione potente quanto il C. Vale pena di fare le seguenti considerazioni:

- Una volta tanto il merito dell'eleganza non va a Mathematica ma al LISP stesso.
- Poiché *Mathematica* è un sovrainsieme del LISP la traduzione dell'interprete LISP da LISP a *Mathematica* è stata particolarmente semplice.
- Il risultato ottenuto è privo di efficienza (per il doppio livello di interpretazione) e di praticità per la totate mancanza di controlli di errore. Provate a dimenticare una parentesi o mettere un *blank* di troppo e vedete cosa succede!
- Nonostante tutto, questo non è solo un gioco. Un interprete LISP scritto in *Mathematica* permette di fare interessanti esperimenti di informatica teorica; Gregory Chaitin ha usato questo approccio per ottenere importanti risultati costruttivi sulla complessità Program-size e i limiti della matematica.

Si veda il libro elettronico gratuito *The Limits of Mathematics* all'indirizzo:

www.cs.auckland.ac.nz/CDMTCS/chaitin/lm.ps

Programma di prova

Il programma di prova si limita a leggere due S-espressioni e ad applicare **evalquote**. L'esecuzione avviene con la seguente funzione che trova l'elemento più a destra in un albero (l'erede al trono)

In[1]:=
```
fn="(LABEL FF (LAMBDA (X) (COND ((ATOM X) X)
((QUOTE T) (FF (CAR X))))))";
```

e applicata alla S-espressione ((A.B))

In[2]:=
```
evalquote[fn, "(((A.B)))"
```

Out[2]=
```
A
```

La traccia di tutta l'elaborazione si può trovare nei due testi citati.

Nonostante la diversità di implementazione sia il programma C che quello Pascal che quello *Mathematica* effettuano l'interpretazione nello stesso modo con gli stessi passaggi intermedi. MC

Bibliografia

Aiello e Montanari, **Elementi di teoria della computabilità, logica, teoria dei linguaggi formali** (ETS, Pisa, 1972).

Fiorentino, Lagana, Romani, Turini, **Pascal, Laboratorio di Programmazione** (MacGraw Hill, 1996)

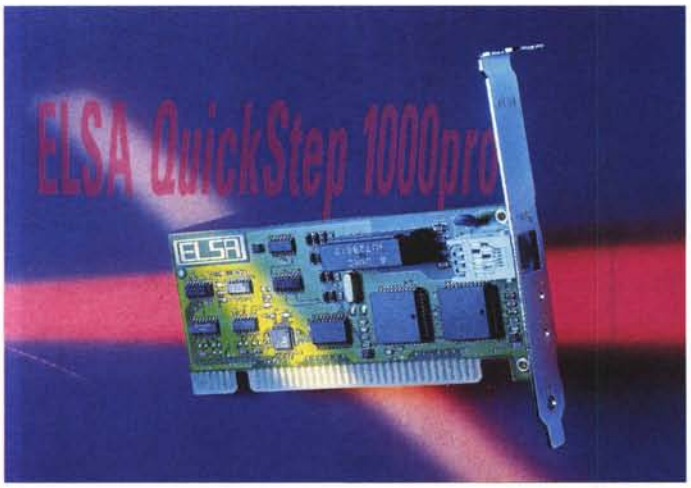
Fiorentino, Lagana, Romani, Turini, **C e Java, Laboratorio di Programmazione** (MacGraw Hill, 1997)

Stephen Wolfram, **The Mathematica Book**, 3rd ed. (Cambridge University Press, 1996)

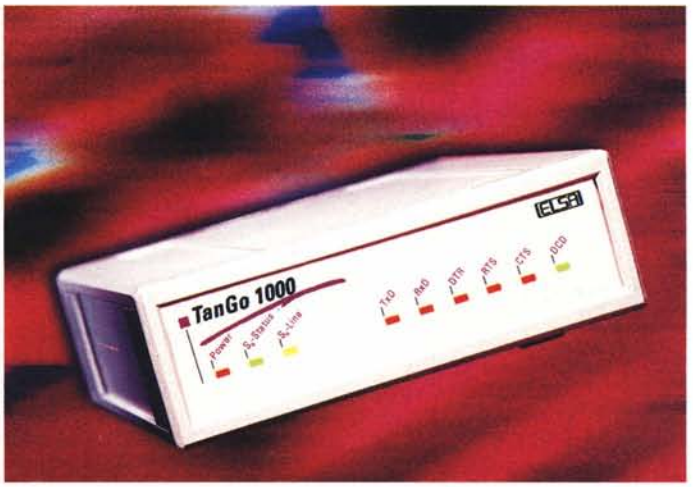
RICHIEDI IL CATALOGO
COMPLETO !!!

ELSA
Data Communications
Computer Graphics

Internet,Intranet e BBS non sono stati mai così rapidi!



ELSA Quickstep1000pro



ELSA Tango 1000

Adesso non ci sono più scuse per non passare alla linea ISDN.

Una volta il problema era il costo dei prodotti e dei scatti, la complessità dell'installazione, la mancanza di Internet provider che offrivano abbonamenti ISDN. Ora la situazione è ben differente, non solo gli scatti di utenza ISDN si sono equiparati a quelli analogici, ma ELSA ha risolto anche il problema della complessità d'uso. Vi serve una scheda ISDN interna Plug&Play, trasferimento dati a 64 e 128Kbps, standard CAPI2.0, emulatore Fax Gruppo III via software, funzioni di telefonia e segreteria telefonica? Il tutto gestito a 32bit per Windows 95/NT4.0? La risposta è semplice ELSA Quickstep 1000. Ma forse a voi serve qualche cosa di esterno che si colleghi ad una semplice porta seriale, consentendo la gestione tramite semplici comandi AT Hayes, che supporti sia gli standard necessari (HDLC,V.120, x.75), che il protocollo PPP e che faccia anche da fax analogico. E magari il tutto essere contenuto nelle dimensioni e nel prezzo. Questo é ELSA Tango 1000. Come dite? Internet? No problem!

www.mavian.com

AGENZIA ESCLUSIVA-*MAVIAN LABS s.r.l-Via Modena 50-Roma Tel.06/48904891 Fax 06/4814017*