L'interprete LISP

Il LISP (LISt Processing) è un linguaggio di programmazione nato negli anni '50-'60 per il trattamento di liste e, in generale, di informazione non numerica.

http://www-formal.stanford.edu/jmc/history/lisp/lisp.html

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 tipicamente ogni errore nel programma LISP da interpretare si riflette in una eccezione nel codice Java che lo interpreta.

L'impostazione della trattazione e gli esempi sono tratti dal testo di Aiello, Albano, Attardi e Montanari, *Elementi di teoria della computabilità, logica, teoria dei linguaggi formali* (ETS, 1972).

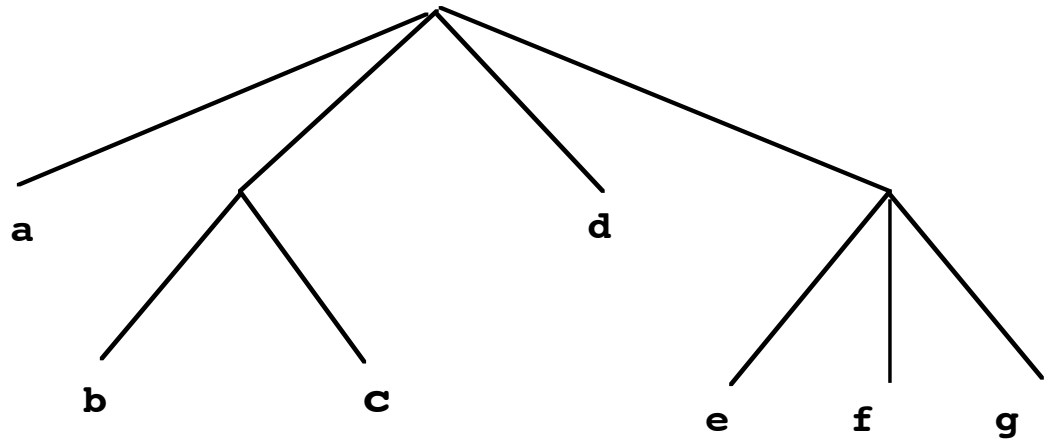
I dati su cui opera un programma LISP sono le cosiddette S-espressioni che hanno la seguente grammatica:

```
<S-expr> ::= <simbolo atomico> | (<S-expr>•<S-expr>) | <lista>
<simbolo atomico> ::= identificatore
<lista> ::= () | (<S-expr>) | (<S-expr> … <S-expr>)
```

Esistono alcuni identificatori speciali (**NIL**, **T**, ecc.) a cui viene attribuito un significato (che verrà esposto nel seguito).
Le liste sono considerate equivalenti a S-espressioni del primo tipo nel modo seguente:

- **()** equivale al simbolo atomico **NIL** (la lista vuota);
- **(a)** equivale a **(a• NIL)**;
- **(a b)** equivale a **(a• (b • NIL))**;
- **(a b** … **z)** equivale **a (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.
Si noti che le liste del LISP sono in realtà alberi con l'informazione (i simboli atomici) memorizzata nelle foglie.
Per esempio la lista **(a (b c) d (e f g))** corrisponde all'albero seguente.



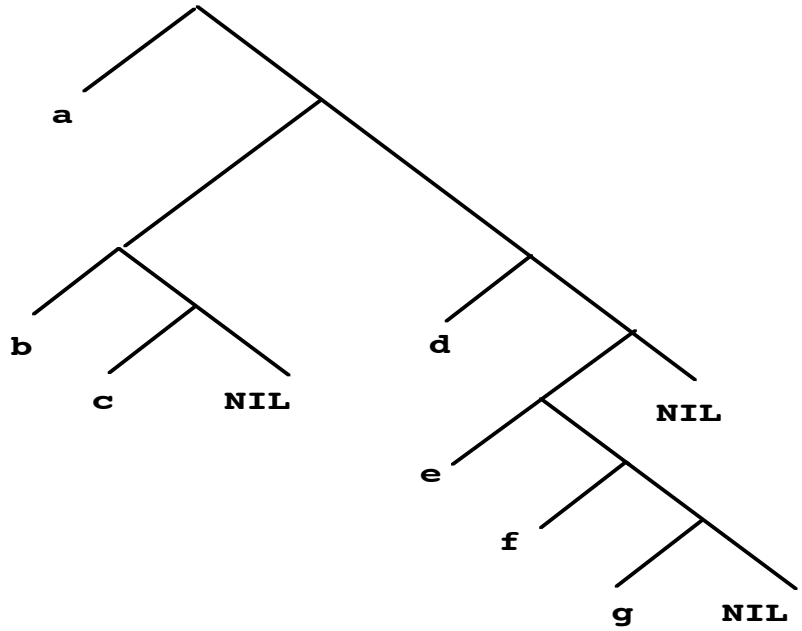
L'equivalenza tra liste ed S-espressioni corrisponde alla rappresentazione di alberi qualsiasi con alberi binari mediante la tecnica primogenito-primo fratello. In questo modo, un nodo con *n* figli e *m* fratelli minori è rappresentato con un nodo che ha come figlio sinistro il primogenito e come figlio destro il primo dei fratelli minori. Applicando la tecnica ricorsivamente a partire dalla radice si ottiene l'albero binario corrispondente.
Continuando l'esempio precedente abbiamo che la lista

```
(a (b c) d (e f g))
```

corrisponde alla S-espressione

```
(a•((b•(c• NIL))•(d•((e•(f•(g• NIL)))•NIL))))
```

ovvero all'albero binario:



Il LISP mette a disposizione alcune funzioni elementari che operano sulle S-espressioni. Le prime due sono predicati (**T** è l'equivalente del `true` e **NIL** del `false`; esiste anche **F** come sinonimo di **NIL**)

- **atom[x]** vale **T** se **x** è un simbolo atomico, **NIL** altrimenti.
- **eq[x; y]** vale **T** se **x** è un atomo ed è uguale a **y**, **NIL** altrimenti.
- **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)] =
  = car[(a•((b•(c•NIL))•(d•NIL)))] =
  = a
```

cdr restituisce la lista, privata del primo elemento:

```
cdr[(a (b c) d)] =
  = cdr[(a•((b•(c•NIL))•(d•NIL)))] =
  =((b•(c•NIL))•(d•NIL)) = ((b c) d)
```

cons inserisce il suo primo argomento come primo elemento nella lista passata come secondo argomento.

```
cons[(a b);(c)] =
  = cons[(a•(b•NIL));(c•NIL)] =
  = ((a•(b•NIL))•(c•NIL)) =
  = ((a b) c)
```

Esiste anche un costrutto condizionale del tipo

[p1 → e1; p2 → e2; ...; pk → ek]

che assume il valore della prima espressione **ei** per cui il predicato **pi** vale **T**.

Un programma LISP è scritto a partire da queste definizioni base con la composizione di funzioni e l'uso sistematico della ricorsione. La potenza del linguaggio LISP è la stessa di linguaggi di programmazione tipo Java o C.

Per esempio, la funzione che seleziona il primo simbolo atomico in una S-espressione si scrive nel modo seguente

ff[x] = [atom[x] → x, T → ff[car[x]]]

Alcune funzioni molto importanti che saranno usate nel seguito sono:

```
equal[x;y] =[
  atom[x] → [atom[y]→ eq[x;y]; T → NIL];
  equal[car[x]; car[y]] →equal[cdr[x]; cdr[y]]];
  T → NIL]
```

vale **T** se **x** e **y** sono la stessa S-espressione.

```
null[x] = [atom[x] → eq[x,NIL]; T → NIL]
```

vale **T** se **x** è la lista vuota **() = NIL**.

```
list[x; y; ... z] = (x y ... z)
```

```
append[x; y] = [
  null[x] → y;
  T → cons[car[x]; append[cdr[x]; y]]]
```

opera su due liste e ne restituisce la concatenazione

```
pairlis[x; y; a] = [
  null[x] → a;
  T→ cons[cons[car[x];car[y]]; pairlis[cdr[x]; cdr[y];a]]]
```

concatena ad **a** la lista delle coppie degli elementi corrispondenti in **x** e **y**.

```
assoc[x; y] = [
  eq[car[car[y]]; x] → car[y];
  T → assoc[x; cdr[y]]]
```

ricerca nella lista di coppie **y** l'elemento corrispondente a **x**.

Una delle caratteristiche del LISP consiste nel fatto che un programma LISP può essere scritto sotto forma di S-espressione; ciò permette di modificare un programma durante la sua esecuzione e di scrivere facilmente una funzione universale (detta **evalquote**) che accetta in *input* una funzione e la lista dei 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 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[e1; e2; ... ek;]* = (f* e1* e2* ... e*)
```

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

```
[p1 → e1; p2 → e2; ...; pk → ek]* =
          (COND (p1* e1*) (p2* e2*) ... (pk* ek*))
```

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

```
λ[l; d]* = (LAMBDA (l* d*))

label[n; d] = (LABEL (n* d*))
```

- Le costanti del programma, che sono S-espressioni 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 **a1**, **a2**, …, **ak** vale:

```
evalquote[fn*; (a1* a2* ... ak*)] = fn[a1; a2; ...; ak]
```

Per definire la funzione **evalquote** servono alcune funzioni ausiliarie:

caar[x] è l'abbreviazione di **car[car[x]]**;
cadr[x] è l'abbreviazione di **car[cdr[x]]**;
cadar[x] è l'abbreviazione di **car[cdr[car[x]]]**, ecc:

```
evalquote[fn; x] = apply[fn; x; NIL]


apply[fn; x; a] = [
 atom[fn] → [eq[fn; CAR] → caar[x];
             eq[fn; CDR] → cdar[x];
             eq[fn; CONS] → cons[car[x]; cadr[x]];
             eq[fn; ATOM] → atom[car[x]];
             T → 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]]]


eval[e; a] = [
 atom[e] → cdr[assoc[e;a]];
 atom[car[e]] → [eq[car[e]; QUOTE] → cadr[e]];
                 eq[car[e]; COND] → evcon[cdr[e];a];
                 T → apply[car[e]; evlis[cdr[e];a];a]];
 T → apply[car[e]; evlis[cdr[e];a];a]]
```

```
evcon[c; a] = [
  eval[caar[c];a] → eval[cadar[c];a];
  T → evcon[cdr[c; a]]


evlis[m; a] = [
  null[m] → NIL;
  T → cons[eval[car[m]; a]; evlis[cdr[m]; a]]]
```

Un interprete LISP in *Mathematica*

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`**

Vediamo come si realizzano i vari passi dell'implementazione di un interprete LISP in *Mathematica*, si noti che il linguaggio di programmazione di *Matematica* è un sovrainsieme del LISP per cui la cosa viene abbastanza naturale.

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 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.

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

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

T=True;
```

Ecco un esempio di funzionamento:

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

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)`**.

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

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}]<>")";


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

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

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

```
format[s]

((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*.

```
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.

```
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`**.

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

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

```
list[a,b,c]
%//format

cons[a, cons[b, cons[c, NIL]]]
(a b c)

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

```
append[list[a,b,c],list[e,f]]
%//format

cons[a, cons[b, cons[c, cons[e, cons[f, NIL]]]]]
(a b c e f)


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

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

cons[cons[1,a], cons[cons[2,b],cons[cons[3,c], NIL]]]
((1.a) (2.b) (3.c))


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

assoc[2,la]
%//format

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'intetrprete 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 nell'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.

```
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**.

```
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`**.

```
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`**.

```
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`**.

```
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 totale 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 testo *The Limits of Mathematics*

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)

```
fn = "(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))";
```

e applicata alla S-espressione

```
((A.B))
```

```
evalquote[fn, "(((A.B)))"
A
```

Costruzione di un semplice interprete LISP in Java

L'interprete LISP in Java è diviso in tre classi, appartenenti al **package** `lisp`.

La classe **Tree** definisce la struttura interna, esportando alcuni metodi protetti usati dall'interprete.

```
protected boolean isAtom() //vale true se l'oggetto è un atomo
protected boolean isNull() //vale true se l'oggetto è un atomo e vale NIL
protected Tree car()       //rende il figlio sinistro
protected Tree cdr()       //rende il figlio destro

protected static boolean isEq(Tree a, Tree b)
 //vale true se l'atomo a è uguale all'atomo b
protected static Tree cons(Tree a, Tree b)
 //rende l'albero che ha a come figlio sinistro e b come figlio destro
protected static Tree quote(String s)
 //rende l'atomo che ha s come informazione
public String toString()
 //converte l'oggetto in una S-espressione
protected static Tree readSE(LispTokenizer lt)
 //legge una S-espressione e rende l'oggetto corrispondente
```

I costruttori di **Tree** sono inaccessibili in quanto le strutture vengono create leggendo S-espressioni con `readSE`, manipolate eseguendo i programmi e infine convertite in S-espressioni con `toString`.

La classe **Lisp** implementa l'interprete vero e proprio. Gli unici metodi pubblici sono

```
public static Tree evalquote(Tree fn, Tree x)
// calcola fn[x]

public static Tree read(Reader r)
//legge una S-espressione e rende l'oggetto corrispondente
```

La classe **LispTokenizer** effettua il parsing di un stringa fornita da un **Reader** individuando i caratteri speciali **'(', ')', '.', ' '** e gli atomi, è costruita da **Lisp** e usata da **Tree**.

I metodi **protected** di **LispTokenizer** sono

```
protected int next()
//rende il tipo del token successivo

protected String getAtom()
//rende l'atomo corrispondente se il token successivo è un atomo
```

Anche il costruttore è protetto

```
protected LispTokenizer(Reader r)
```

La classe `LispTokenizer`

```
package lisp;
import java.io.*;
import java.util.*;

public class LispTokenizer {

   protected final static int
      OPEN = 0, CLOSED = 1, DOT = 2, BLANK = 3, CHAR = 4,EOF = 5, ATOM = 6;

   protected LispTokenizer(Reader r) {
      this.r = r;
      data = null;
      notToRead = false;
   }

   protected int next() {
      if (notToRead) {notToRead = false; data = null; return res;}
      data = "";
      res = read();
      if (res != CHAR) {data = null; return res;}
      while (res == CHAR) {res = read();}
      notToRead = true;
      return ATOM;
   }
```

```
protected String getAtom() {
   if (data == null) throw new NoSuchElementException();
   return data;
}

private Reader r;
private boolean notToRead;
private int res;
private String data;

private int read() {
   char c;
   try {
      int i = r.read();
      if (i == -1) return EOF;
      c = (char) i;
   } catch (IOException e) {new RuntimeException(("IOException"));}
   switch (c) {
      case '(': return OPEN;
      case ')': return CLOSED;
      case '.': return DOT;
      case ' ': return BLANK;
   }
   data += c;
   return CHAR;
}
```

La classe Tree

```
package lisp;
public class Tree {

// costanti
   private static final String NIL = "NIL";


// variabili di istanza
   private Tree car, cdr;

   private String data = null;


// metodi pubblici
   public String toString() {
      if (isAtom()) return data;
      if (isList(this)) return "(" + restList(this);
      return "(" + car + "." + cdr + ")";
   }


// metodi protected
   protected boolean isAtom() {return data != null;}

   protected boolean isNull() {return data == NIL; }
```

```
protected static boolean isEq(Tree a, Tree b) {
   return a.isAtom() && b.isAtom() && a.data.equals(b.data);
}

protected Tree car() {return this.car;}

protected Tree cdr() {return this.cdr;}

protected static Tree cons(Tree a, Tree b) {return new Tree(a, b);}

protected static Tree quote(String s) {return new Tree(s);}

protected static Tree readSE(LispTokenizer lt) {
   int c = lt.next();
   if (c == LispTokenizer.ATOM) return new Tree(lt.getAtom());
   if (c != LispTokenizer.OPEN) throw new RuntimeException(" SyntaxError 1");
   Tree first = readSE(lt);
   c = lt.next();
   if (c == LispTokenizer.DOT) return new Tree(first, readSE(lt));
   if (c == LispTokenizer.CLOSED) return new Tree(first, new Tree(NIL));
   if (c == LispTokenizer.BLANK) return new Tree(first, readLIST(lt));
   throw new RuntimeException(" SyntaxError 2");
}
```

```
// metodi privati
   private Tree(String id) {
      if (id == null)
         throw new IllegalArgumentException("Tree: Atoms cannot be null");
      this.data = id;
   }
   private Tree(Tree car, Tree cdr) {
      this.car = car;
      this.cdr = cdr;
   }
   private static boolean isList(Tree a) {
      if (a.isAtom())return a.data.equals(NIL);
      return isList(a.cdr);
   }
   private static String restList(Tree a) {
      if (a.isAtom() && a.data.equals(NIL)) return ")";
      if (a.cdr.isAtom() && a.cdr.data.equals(NIL)) return a.car +")";
      return a.car + " " + restList(a.cdr);
   }
   private static Tree readLIST(LispTokenizer lt) {
      Tree first = readSE(lt);
      int c=lt.next();
      if (c == LispTokenizer.CLOSED) return new Tree(first, new Tree(NIL));
      if (c == LispTokenizer.BLANK) return new Tree(first, readLIST(lt));
      throw new RuntimeException(" SyntaxError 3");
   }
```

}

La classe `Lisp`

```
package lisp;

import java.io.Reader;

public class Lisp {

// metodi pubblici statici

   public static Tree evalquote(Tree fn, Tree x) {
      System.out.println("evalquote[" + fn + "; " + x + "]");
      System.out.println();
      return apply(fn, x, QNIL);
   }

   public static Tree read(Reader r) {
      lt = new LispTokenizer(r);
      return Tree.readSE(lt);
   }

// metodi privati

   private Lisp() {}

   private static LispTokenizer lt;
```

```
private final static
      Tree QT = quote("T"), QNIL = quote("NIL"),
      QCAR = quote("CAR"), QCDR = quote("CDR"), QCONS = quote("CONS"),
      QATOM = quote("ATOM"), QEQ = quote("EQ"), QQUOTE = quote("QUOTE"),
      QCOND = quote("COND"), QLABEL = quote("LABEL"),
      QLAMBDA = quote("LAMBDA");

private static boolean isNull(Tree a) {return a.isNull();}
private static boolean isAtom(Tree a) {return a.isAtom();}
private static boolean isEq(Tree a, Tree b) {return Tree.isEq(a, b);}
private static Tree atom(Tree a) {return a.isAtom() ? QT : QNIL;}
private static Tree eq(Tree a, Tree b) {return Tree.isEq(a, b) ? QT : QNIL;}
private static Tree car(Tree a) {return a.car();}
private static Tree cdr(Tree a) {return a.cdr();}
private static Tree quote(String s) {return Tree.quote(s);}
private static Tree cons(Tree a, Tree b) {return Tree.cons(a, b);}
private static Tree caar(Tree x) {return car(car(x));}
private static Tree cadr(Tree x) {return car(cdr(x));}
private static Tree cdar(Tree x) {return cdr(car(x));}
private static Tree caddr(Tree x) {return car(cdr(cdr(x)));}
private static Tree cadar(Tree x) {return car(cdr(car(x)));}


private static Tree pairLis(Tree x, Tree y, Tree a) {
   If (x.isNull() return a;
   return cons(cons(car(x), car(y)), pairLis(cdr(x), cdr(y), a));
}
```

```
private static Tree assoc(Tree x, Tree y) {
   if (Tree.isEq(car(car(y)), x)) return car(y);
   return assoc(x, cdr(y));
}

private static Tree eval(Tree e, Tree a) {
   if (isAtom(e)) return cdr(assoc(e, a));
   if (isAtom(car(e))) {
      if (isEq(car(e), QQUOTE))  cadr(e);
      if (isEq(car(e), QCOND)) return evcon(cdr(e), a);
      return apply(car(e), evLis(cdr(e), a), a);
   }
   return apply(car(e), evLis(cdr(e), a), a);
}

private static Tree evcon(Tree c, Tree a) {
   if (isEq(eval(caar(c), a), QT)) return eval(cadar(c), a);
   return evcon(cdr(c), a);
}

private static Tree evLis(Tree m, Tree a) {
   if (isNull(m)) return QNIL;
   return cons(eval(car(m), a), evLis(cdr(m), a));
}
```

```
   private static Tree apply(Tree fn, Tree x, Tree a) {
      if (isAtom(fn)) {
         if (isEq(fn, QCAR)) return caar(x);
         if (isEq(fn, QCDR)) return cdar(x);
         if (isEq(fn, QCONS)) return cons(car(x), cadr(x));
         if (isEq(fn, QATOM)) return atom(car(x));
         if (isEq(fn, QEQ)) return eq(car(x), cadr(x));
         return apply(eval(fn, a), x, a);
      } else if (isEq(car(fn), QLAMBDA))
         return eval(caddr(fn), pairLis(cadr(fn), x, a));
      else if (isEq(car(fn), QLABEL))
         return apply(caddr(fn), x, cons(cons(cadr(fn), caddr(fn)), a));
      return null;
   }
}
```

Programma di prova

La classe **LispTest** non appartiene al **package** lisp. L'esecuzione di prova avviene con la funzione **ff[x]**, vista prima, tradotta in S-espressione:

```
(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))
```

e applicata alla S-espressione

```
((A.B))
```

```
import java.io.StringReader;
import lisp.*;

public class LispTest {
    final static String
  Sfunc= "(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))";
  Sexpr= "((A.B))";

   public static void main(String[] args) {
      System.out.println("FUNC   "+ Sfunc);
      Tree func = Lisp.read(new StringReader(Sfunc));
      System.out.println("PARSED "+ func);
```

```
        System.out.println("EXPR   "+ Sexpr);
        Tree expr = Lisp.read(new StringReader(Sexpr));
        System.out.println("PARSED "+ expr);
        System.out.println();
        Tree res =Lisp.evalquote(func, expr);
        System.out.println("STOP   "+ res);
    }
}
```

Ed ecco il risultato

```
FUNC   (LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))
PARSED (LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))
EXPR   ((A.B))
PARSED ((A.B))

evalquote[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))));
((A.B))]

apply[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))));
((A.B)); NIL]

apply[(LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))); ((A.B)); ((FF
```

```
LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

eval[(COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))); ((X.(A.B)) (FF LAMBDA (X)
(COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

evcon[(((ATOM X) X) ((QUOTE T) (FF (CAR X)))); ((X.(A.B)) (FF LAMBDA (X) (COND
((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

eval[(ATOM X); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

eval[X; ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

apply[ATOM; ((A.B)); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF
(CAR X))))))]

evcon[(((QUOTE T) (FF (CAR X)))); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X)
((QUOTE T) (FF (CAR X))))))]

eval[(QUOTE T); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

eval[(FF (CAR X)); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF
(CAR X))))))]

eval[(CAR X); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]
```

```
eval[X; ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

apply[CAR; ((A.B)); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF
(CAR X))))))]

apply[FF; (A); ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

eval[FF; ((X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

apply[(LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))); (A); ((X.(A.B))
(FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

eval[(COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))); ((X.A) (X.(A.B)) (FF LAMBDA
(X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

evcon[(((ATOM X) X) ((QUOTE T) (FF (CAR X)))); ((X.A) (X.(A.B)) (FF LAMBDA (X)
(COND ((ATOM X) X) ((QUOTE T) (FF (CAR X))))))]

eval[(ATOM X); ((X.A) (X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF
(CAR X))))))]

eval[X; ((X.A) (X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]
```

```
apply[ATOM; (A); ((X.A) (X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T)
(FF (CAR X))))))]

eval[X; ((X.A) (X.(A.B)) (FF LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR
X))))))]

STOP     A
```