Generazione di codice

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

La generazione di codice trasla la rappresentazione intermedia in "codice". Assumeremo che il codice finale sia codice assembler che poi verra' assemblato, linkato, caricato ed eseguito.

C'e' una fase di preparazione nella generazione di codice che decide come allocare registri e memoria. Poi la fase di generazione di codice vera e propria trasla la rappresentazione intermedia in codice usando registri e memoria decisi nella fase di preparazione.

Preparazione per la generazione di codice

Il compilatore decide dove i valori di variabili ed espressioni risiederanno durante l'esecuzione.

Le locazioni preferite sono i registri poiche' consentono un'esecuzione piu' veloce. Ma se non sono sufficienti si deve ricorrere alla memoria centrale. I risultati finali vengono invece memorizzati in memoria centrale.

Un problema e' quindi un buon uso dei registri.

Una tecnica e', ad esempio, inserire tutte le variabili che sono all'interno di cicli in registri, poiche' saranno (probabilmente) riferite molte volte.

Comunque si dovrebbero inserire nei registri le variabili maggiormente usate.

Per le macchine a stack, gli accessi allo stack sono piu' rapidi che quelli alla memoria normale. Quindi per le variabili si preferisce un accesso allo stack.

Esempio:

Nel caso di:

X1:=a+bb*12;

X2:=a/2 + bb*12

potremmo generare (per una macchina di nostra invenzione) il seguente codice:

```
PushAddr    X2  Mette l'indirizzo di X2 nello stack
PushAddr    X1   Mette l'indirizzo di X1 nello stack
Push        bb   Mette bb nello stack
Push        a    Mette a nello stack
Load        1(S),R1 Mette bb in R1
Mpy         #12,R1   Mette bb*12 in R1
Load        S,R2     Mette a in R2
Store       R2,R3    Copia a inR3
Add         R1,R3    Mette a+bb*12 in R3
Store       R3,@2(S)     Mette a+bb*12 in X1
Div         #2,R2    Mette a/2 in R2
Add         R1,R2    Mette a/2+bb*12 in R2
Store       R2, @3(S)     Mette a/2+bb*12 in X2
```

Nota:

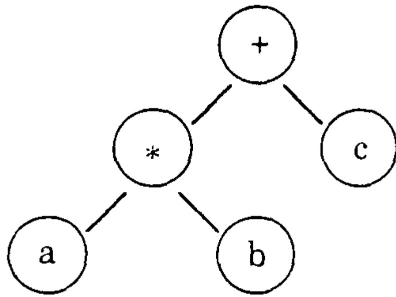
(S), 1(S), 2(S) ecc, significa accedere al contenuto del top dello stack, ad una posizione successiva, due posizioni successive ecc.
@A indica che si vuole accedere alla locazione il cui valore è puntato da A. (indirizzamento indiretto).

La tabella dei simboii serve in questa fase perche' in dipendenza dal fatto che si tratti di una variabile di un certo tipo, l'area di memoria avra' una certa dimensione (si generano particolari direttive in dipendenza dalla macchina utilizzata).

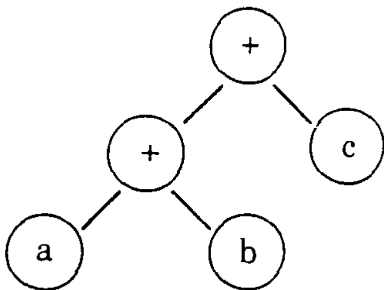
Generazione di codice dagli alberi sintattici astratti

Un generatore di codice molto semplice puo' generare codice dall'albero sintattico astratto seplicemente visitandolo opportunamente.

Dall'albero sintattico astratto di una espressione il compilatore costruisce il codice di valutazione.



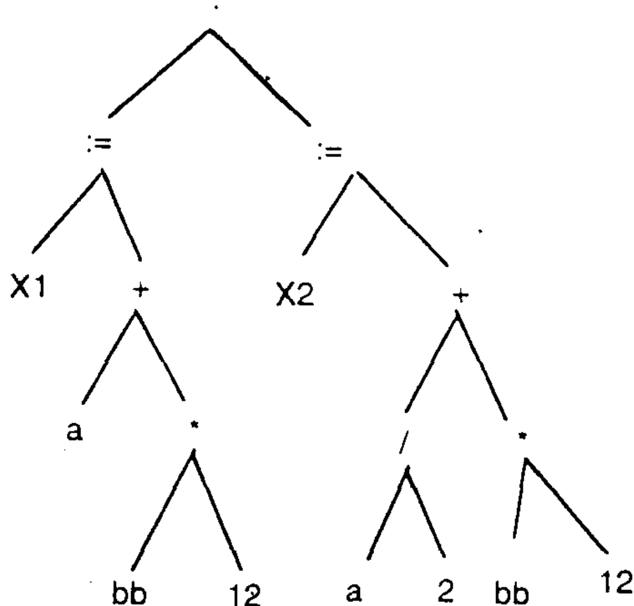
LOADA *a*
LOADB *b*
MUL
LOADB *c*
ADD



LOADA *a*
LOADB *b*
ADD
LOADB *c*
ADD

Dall'albero sintattico relativo a:
X1:=a+bb*12;
X2:=a/2 + bb*12;

si genera il seguente albero sintattico astratto:



Generando una visita dell'albero in post-ordine, cioe' che visita prima i sottoalberi e poi la radice si puo' generare il codice seguente (immaginando per semplicita' di avere un solo registro).

```
Load   BB,R1
Mult   #12,R1
Store  R1,Temp1
Load   A,R1
Add    Temp1,R1
Store  R1,Temp2
Load   Temp2,R1
Store  R1,X1
Load   A,R1
Div    R1,#2
Store  R1,Temp3
Load   BB,R1
Mult   #12,R1
Store  R1,T4
Load   T3,R1
Add    T4,R1
Store  R1,T5
Load   T5,R1
Store  R1,X2
```

Il codice continuamente mette il contenuto del registro in memoria per poi riutilizzarlo (si confronti questo codice con quello prodotto per una macchina a stack).

Algoritmo

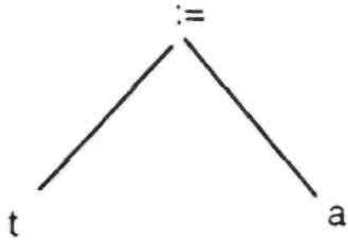
```
Procedure CodeGen(Node)
BEGIN
   CASE Node Type is
   Expression operator, Op: (* result left in Reg1*)
   IF Neither child is a leaf THEN
     BEGIN
        CodeGen (Left Child);
        Emit "Store, Reg1, T1:=GetTemp";
        CodeGen (Right Child);
        Emit "Store, Reg1, T2:=GetTemp";
        Emit "Load Tempi,Reg1";
        Emit"Op Temp2,Reg1"
      END
   ELSE IF only one child is a leaf THEN
     BEGIN
         CodeGen(Other Child);
         Emit "Op Leaf Child, Reg1";
     END
  ELSE (* both children are leaves *)
     BEGIN
        Emit "Load Left Child, Reg1";
        Emit "Op Right Child, Reg1"
     END
":=": CodeGen(Right Child);
   Emit "Store Reg1, @Left Child"
"IF": CodeGen(First Child);
     Emit "Branch on equal Reg1, Label1 = GetLabel";
     CodeGen(Second Child);
     Emit "Branch label2 = GetLabel";
     Emit"Label1:";
     CodeGen(Third Child);
     Emit "Label2:"
"WHILE": Emit "Label1= GetLabel";
     CodeGen(Left Child);
     Emit "Branch on equal Reg1, Label2 =GetLabel"
     CodeGen(Right Child);
     Emit "Branch Label 1"
     Emit "label2:"
```

```
END;
```

In pratica, un generatore di codice puo' essere scritto riconoscendo particolari schemi:

(1) Istruzione di assegnamento

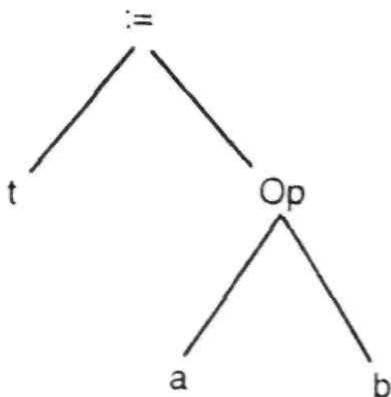
Un albero della forma:



genera una istruzione di Move (Copy):
MOVE aPlace,T
Dove *aPlace* rappresenta il registro, la locazione di memoria o la posizione nello stack assegnata ad a.

(2) Operazioni Aritmetiche

Supponiamo che Op rappresenti un'operazione aritmetica e si consideri un albero sintattico astratto per l'istruzione: t := a Op b



Una possibile sequenza di istruzioni e':

```
MOVE        aPlace,     Reg
OP          bPlace, Reg
MOVE        Reg, T
```

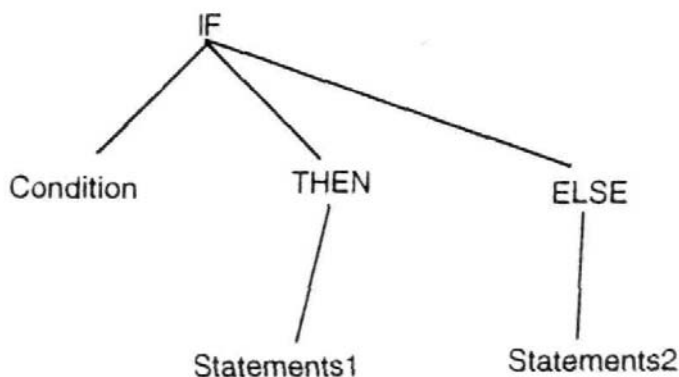
Ad esempio per:
T:= A-B

avremo

```
MOVE aPlace, Reg
SUB bPlace, Reg
MOVE   Reg,T
```

(3) Istruzione di IF

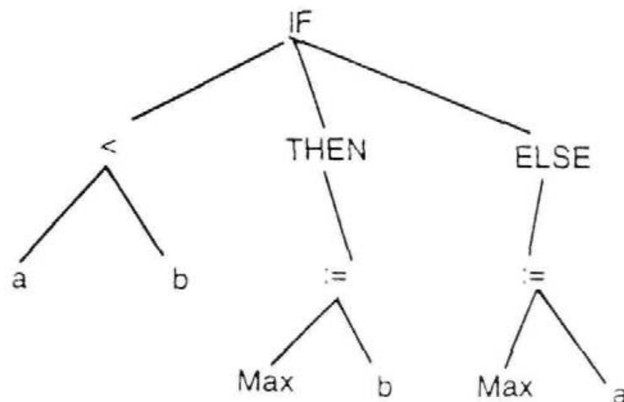
Puo' essere rappresentata da un albero sintattico astratto del tipo:



Una tipica sequenza di codice e':

```
        (code for condition)
        BRANCHIFFALSE     Label1
        (code for Statements1)
        BRANCH            Label2
Label1: (Code for Statements2)
Label2:
```

Esempio per:
IF a < b THEN Max := b ELSE Max := a



Il codice allora potrebbe essere:

```
          (Code for comparison)
          COMPARE      aPlace, bPlace
          BGEQ         Label1    (NOT <)
          (Code for Statements1)
          MOVE         bPlace,Max
          BRANCH       Label2
          (Code for Statements2)
Label1:   MOVE         aPlace,Max
Label2:
```

aPlace e bPlace possono comparire direttamente nell'istruzione: COMPARE aPlace, bPlace solo se la macchina prevede due operandi direttamente in memoria. Per le macchine che consentono solo un operando in memoria (vedi la famiglia 86 di cui fa parte il PC-IBM) l'istruzione COMPARE deve essere cosi' modificata:

```
MOVE   aPlace,Reg
COMPARE   Reg, bPlace
```

Gli operandi sono scritti nell'ordine come Sorgente, Destinazione (mentre nella famiglia 86 e' il contrario).

L'istruzione:

```
    MOVE bPlace, Max;
```

viene quindi sostituita da:

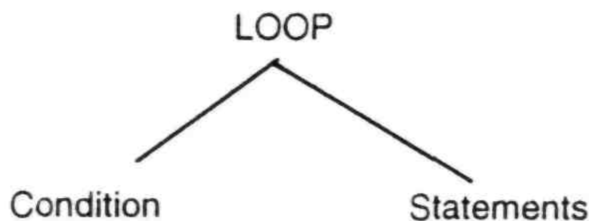
```
    MOVE bPlace, Reg;
    MOVE Reg, Max;
```

(4) Istruzioni iterative

Ad esempio:

```
LOOP    While condition DO
            Statements
ENDLOOP
```

Albero sintattico astratto:

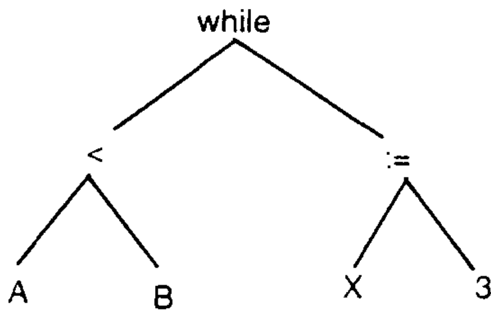


Una sequenza di codice ragionevole potrebbe essere:

```
Label 1: (Code for NOT condition)
        BRANCHIFTRUE     Label2
        (Code for statements)
        BRANCH           Label1
Label2:
```

Esempio:

WHILE A < B DO X:=3



Un possibile codice e':

```
Label1:  COMPARE aPlace, bPlace
         BRANCHGE      Label2
         MOVE          #3, X
         GOTO          Label1
Label2:
```

BRANCHGE sta per Branch on Greater Than or Equal to.

Concludendo

La generazione di codice puo' essere svolta in differenti modi.

Qui abbiamo presentato solo alcuni cenni relativi a come generare codice partendo dagli alberi sintattici astratti.

Si percorrono tali alberi e si utilizzano alcuni "schemi" di traduzione.

La generazione di codice diventa un CASE. Ogni nodo non foglia viene analizzato ed in base al suo valore viene generato il codice opportuno

Si potrebbe ovviamente pensare di utilizzare uno stack o piu' registri.

La generazione di codice puo' quindi essere molto semplice, ma la generazione di "buon" codice (cioe' ottimizzato) e' invece molto complicata!

Noi non tratteremo questo problema.

GENERAZIONE DI CODICE IN PROLOG

Descriviamo un semplice programma che genera codice per una macchina con un singolo registro ed ha le usuali perazioni aritmetiche piu' le operazioni di:
LOAD Var
STORE Var
dove Var e' la locazione di una variabile.

L'algoritmo (simile a quello per la generazione della forma polacca postfissa) e' il seguente:

(1)Quando riconosce una variabile la inserisce nello stack;

(2)Quando riconosce una operazione, gli operandi sono al top dello stack. Se sono variabili genera le seguenti istruzioni:

```
LOAD         1st Operand
Operation    2nd Operand
```

Il passo due continua rimpiazzando gli elementi al top dello stack con il marker acc per ricordarsi che a tempo di esecuzione il risultato sara' nell'accumulatore.

Per tenere conto di questo marker speciale si introducono i punti revisionati di (1) e (2).

(1a) Prima di inserire una variabile nello stack e' necessario controllare se il mark acc occupa la posizione giusto sotto il top dello stack. Questa situazione indica la necessita' di una locazione di memoria temporanea poiche' l'accumulatore contiene un valore che non puo' essere distrutto.
Allora il marker acc e' rimpiazzato da Ti (locazione di memoria temporanea) e si genera l'istruzione: STORE Ti. Poi si puo' inserire la varibile nello stack.

(1b) Se il penultimo elemento dello stack non e' acc, semplicemente si inserisce la variabile nello stack.

Si devono poi considerare operazioni commutative e non commutative. Sia S1 l'elemento al top dello stack ed S2 l'elemento immediatamente dopo.

(2a) Se ne' S1 ne' S2 e' un acc, il codice e' generato come gia' spiegato al passo 2;

(2b,c) Per le operazioni commutative (addizione e moltiplicazione) basta generare:
    Operation S1 (se S2 e' in acc) o
    Operation S2 (se S1 e' in acc).

(2d) Per le operazioni non commutative (sottrazione e divisione) si controlla se S1 e' in acc ed in questo caso si genera l'istruzione STORE Ti, sostituendo acc con Ti; poi si procede come in (2). Nel caso invece in cui S2 e' in acc si procede come per le operazioni commutative (2b).

Traslazione in Prolog

Assumiamo per semplicità che le espressioni siano in notazione Polacca postfissa. Le variabili sono rappresentate da termini v(Name) e gli operatori da op(Op). t(X) rappresenta una locazione temporanea.

La procedura principale e':
gen_code(Polish, Stack, Temps)

dove Polish (una lista) e' l'ingresso in forma postfissa, Stack lo stack e Temps la lista di indirizzi temporanei. Il risultato e' stampato usando write.

```
gen_code([op(Op)|[Rest], Stack,Temps):-
    operator(Op, Stack, NewStack.Temps, NewTemps),
    gen_code(Rest, NewStack,NewTemps).

gen_code([v(X)|Rest], Stack.Temps):-
    operand(X,Stack, NewStack.Temps, NewTemps),
    gen_code(Rest, NewStack,NewTemps).

gen_code([],AnyStack,AnyTemps).
```

% caso (1a):

```
operand(X,[A,acc|Stack],[v(X),A,t(I)|Stack],
                              Temps, NewTemps):-
    get_temp(t(I),Temps, NewTemps),
    write(store,t(I)).
```

% caso (1b):

```
operand(X,Stack,[v(X)|Stack],Temps,Temps).
```

% caso (2b):

```
operator(Op,[A,acc|Stack],[acc|Stack],
                              Temps, NewTemps):-
    codeop(Op,Instruction,AnyOpType),
    gen_inst(Instruction,A,Temps,NewTemps).
```

% caso (2c):

```
operator(Op,[acc,A|Stack],[acc|Stack],
                              Temps, NewTemps):-
    codeop(Op,Instruction,commute),
    gen_inst(Instruction,A,Temps,NewTemps).
```

% caso (2d):

```
operator(Op,[acc,A|Stack],[acc|Stack],
                              Temps, NewTemps):-
    codeop(Op,Instruction,noncommute),
    get_temp(t(I),Temps, Temps0),
    write(store,t(I)),
    gen_instr(load,A,Temps0,Temps1),
    gen_inst(Instruction,t(I),Temps1,NewTemps).
```

% caso (2a):

```
operator(Op,[A,BIStack],[acclStack],
                                        Temps, NewTemps):-
      A=\=acc, B=\= acc, codeop(Op,Instruction,OpType),
      gen_instr(load,B,Temps,Temps1),
      gen_inst(Instruction,A,Temps1,NewTemps).
```

% procedure ausiliarie

```
codeop(+,add,commute).
codeop(-,sub,noncommute).
codeop(*,mult,commute).
codeop(/,div,noncommute).

get_temp(t(I),[I,J|R],[J|R]).
get_temp(t(I),[I],[J]):-
                  J is I +1.

gen_instr(Instruction,t(I),Temps,[I|Temps]):-
      write(Instruction,t(I)).

gen_instr(Instruction,v(A),Temps,Temps):-
      write(Instruction,A).
```

Il codice generato dall'espressione:
A * (A*B+C-C*D) e'

```
LOAD      A
MULT      B
ADD       C
STORE     T0
LOAD      C
MULT      D
STORE     T1
LOAD      T0
SUB       T1
MULT      A
```

Generazione di codice dagli alberi sintattici

Si tratta, in questo caso, di visitare opportunamente l'albero sintattico astratto.
Gli argomenti sono inseriti in un dizionario, ma sono lasciati non legati fino al caricamento del programma.

Ad esempio, l'assegnamento di un'espressione Expr ad una variabile X è traslata in una lista la cui testa e' il codice generato per l'espressione seguito da STORE X.

La procedura encode_statement ha tre argomenti:
- l'albero sintattico;
- il dizionario Dict;
- il codice risultante (es:
  [...label(L1),[instr(LOAD,X), ...],...]

```
encode_statement(assign(name(X), Expr), Dict,
                        [Exprcode,instr(sto,Addr)]):-
            lookup(X,Addr,Dict),
            encode_expr(Expr,Dict,Exprcode).
```

La procedura lookup memorizza la nuova variabile X nel dizionario, se ancora non è stata inserita e la lega ad una variabile che rappresenta il suo indirizzo.

encode_expr puo' manipolare due tipi di alberi sintattici per espressioni: nel primo l'operando di destra e' una costante c variabile (foglia), nel secondo un generico sottoalbero.

Consideriamo il caso piu' generale:

expr(Op,Expr1 ,Expr2)
in cui Expr2 è della forma:
expr(Op,Any1,Any2).
Sara' traslato in:

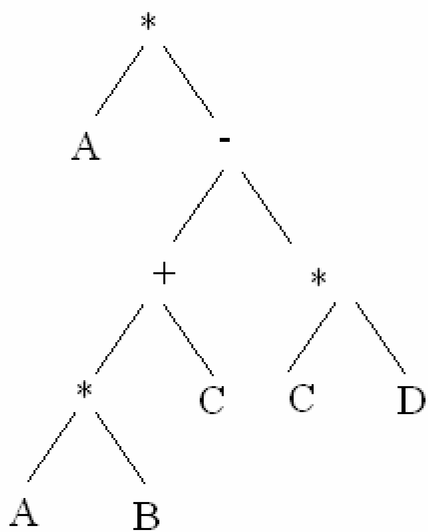
(1) Il codice per Expr2;
(2) STORE temp;
(3) Il codice per Expr1 ;
(4) Il codice per l'istruzione specificata da Op.

E' necessaria una variabile per indicare l'insieme delle variabili temporanee: N = 0 , il suo valore iniziale è zero.

```
encode_subexpr(expr(Op,Expr1,Expr2),N,Dict
      [Expr2code,instr(store,Addr),Expr1code,
      instr(Opcode,Addr)]):-
      complex(Expr2),
      lookup(N,Addr,Dict),
      encode_subexpr(Expr2,N,Dict,Expr2code),
      N1 isN+1,
      encode_subexpr(Expr1,N1,Dict,Expr1code),
      memoryop(Op,Opcode).

complex(expr(Op,Any1 ,Any2)).
memoryop(+,add).
memoryop(*,mult).
memoryop(-,sub).
memoryop(/,div).
```

Il codice generato dall'espressione:
A * (A*B+C-C*D) è

```
LOAD    C
MULT    D
STORE   T0
LOAD    A
MULT    B
ADD     C
SUB     T0
STORE   T0
LOAD    A
MULT    T0
```



Si noti che poiche' l'albero destro e' valutato prima del sinistro, il codice C*D è il primo ad essere generato.

L'utilizzo delle labels e' illustrato dall'esempio seguente per l'istruzione di while.

La traslazione consiste nel trasformare l'albero sintattico while(Test,Do) nel codice:

```
label(L1):    <encode Test>
              <encode Do>
              jumpL1
label(L2):
```

Si noti che e' necessario un nuovo argomento (L2) alla procedura che codifica il test, affinche' generi il salto all'uscita del ciclo nei casi opportuni.

```
encode_statement(while(Test,Do),Dict,
[label(L1),Testcode,Docode,instr(jump,L1),label(L2)]):-
     encode_test(Test,Dict,L2,Testcode),
     encode_statement(Do,Dict,Docode).
```

Tabella dei simboli

Prolog e' molto adatto anche per la gestione della tabella dei simboli. In pratica, si tratta di gestire opportunamente degli alberi.

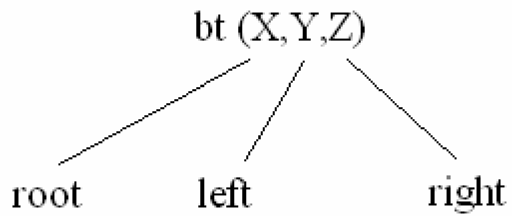
```
lookup( Name, Dic, Value):-
                        root( Dic, bind( Name,Value )),!.
lookup( Name, Dic, Value):-   root( Dic, bind( X , _)),
                        lessthan( Name,X),!,
                        left( Dic, LDic),
                        lookup( Name, LDic, Value ).
lookup( Name, Dic, Value):-   right( Dic, RDic ),
                        lookup( Name, RDic, Value ).

empty( BT):-     var(BT).
/*un albero e' vuoto se la var che lo denota e' unbound*/

root( bt( X,_,_), X).
left( bt( _,X,_), X).
right( bt( _,_,X), X).

show( BT ):- write(BT), nl.

lessthan( X,Y) :-
            name( X,SX ),name( Y,SY), SX < SY.
```

Se invocato con Name bound e Dic unbound o meno lookup funge da costruttore.

Il contenuto informativo del dizionario puo' essere parzialmente specificato, cioe' bind(Name,Value) puo' avere Value unbound.

RAPPRESENTAZIONE DELLA TABELLA

Albero = struttura Prolog



X = bind (Name, Value)

Y = bt (X1,Y1,Z1)

Z= bt (X2,Y2,Z2)