Logic Programming and Prolog

martedì 24 febbraio 2015
11:12

Logic programming and PROLOG

Prolog 1

Summary

- Logic programming
- Knowledge base
- Queries
- Recursive rules
- Program execution
- Operational model

Logic programming

Declarative programming.

- Program = problem description
- Execution = check the truth of an assertion (goal)

R. Kowalski : Algorithm = Logic + Control.

(2D or 3D maps)

Can be used with SEMANTIC MAPPING

Logic Based programming language (subset of FOL)

Applications of logic programming

PROLOG is the major logic-based programming language (subset of *First Order Logic*).

Resources (implementation):

`http://www.swi-prolog.org/`

Resources (textbook):
L. Sterling, E. Shapiro, The Art of Prolog, 2nd Ed., MIT Press, 1994.

Resources (other book):

`http://www.learnprolognow.org/`

Applications of logic programming

- deductive databases
- expert systems
- knowledge representation for robots!!

Check for example:

`http://stackoverflow.com/questions/130097/real-world-prolog-usage`

Basic intuition

Logic program: We need SYMBOLS to interact with the robot

1. definition of the problem through the assertion of l'asserzione **facts** and **rules**;
2. Querying the system which **infers** the answer to the query given known facts and rules (theorem provers).

Inference is done giving an answer Yes/No

- Variables CAPITALS names
- Constants/Predicate Names/Functions LOWERCASE names

head :- list of positive atoms ? query

Aristotelic Syllogism

- *All men are mortal*
- *Socrates is a man*

we can infer: *Socrates is mortal.*

We can get more than one result using ;

Comments %

Every line ends with .

Aristotelic Syllogism in PROLOG

```
mortal(X) :- man(X).
man(socrates).
```

The inference is started by:

```
?mortal(socrates)
```

Knowledge base

A PROLOG program is composed by a set of *clauses*, i.e. *conditional* and *unconditional assertions*.

unconditional assertion (*fact*):

```
father(daniele,jacopo).
```

```
loves(enzo,X).
```

In PROLOG the names of predicates and constants start with a capital letter.

Rules

conditional assertion (*rule*):

```
A :- B,C,...,D.
```

A is true if B, C,..., D are true,

- `A` is the **conclusion**,
- `B, C,... , D` are the **premises**
- `A,B,C, D` are **atoms**

If $t_1, \ldots, t_n$ are terms and P is an n-ary predicate $P(t_1, \ldots, t_n)$ is an atom.

We start simple (without function symbols), so terms are either constants or variables.

Examples of rules

```
grandFather(X,Z) :- father(X,Y), father(Y,Z).
grandFather(X,Z) :- father(X,Y), mother(Y,Z).

son(X,Y) :- father(Y,X).
son(X,Y) :- mother(Y,X).
```

`grandFather`, `son` can be seen as **procedures**

Querying the system

goal (query)

```
? A,B,C,...,D.
```

```
? father(daniele,jacopo).
```

YES.

Knowledge base

```
father(daniele,michela).
father(daniele,jacopo).
father(eriberto,daniele).
father(antonio,eriberto).
mother(alma,eriberto).
mother(annamaria,daniele).
mother(annamaria,marcello).
mother(annamaria,sandro).

nice(michela).
nice(anna).

fem(michela).
```

Queries

```
? nice(X).
```

YES michela.

to get other answers: ;

YES anna

goal conjunction:

```
? grandFather(eriberto,X), nice(X).
```

```
? grandFather(X,Z), nice(Z).
```

Recursive rules

```
descendant(X,Y):-son(X,Y). % 1
descendant(X,Y):-son(Z,Y),descendant(X,Z). % 2
son(X,Y):-father(Y,X). % 3
son(X,Y):-mother(Y,X). % 4

? descendant(michela,eriberto).
```

Prolog is based on a resolution resolver that recurs to find the answer T or F on a query

T ⇝ { } CONTRADICTION

F ⇝ { x, y ... }

Directed Graph

```
/* Directed Graph */
arc(a,b).
arc(a,c).
arc(b,d).
arc(c,d).
arc(d,e).
arc(f,g).

/* Transitive closure of the arc relation
connected(Node1,Node2) :- Node1 connected to Node2
*/
connected(Node,Node).
connected(Node1,Node2) :- arc(Node1,NodeInt),
                          connected(NodeInt,Node2).
```

Multiple roles of the arguments

```
? descendant(X,daniele).
? descendant(daniele,X).
? descendant(X,Y).

? connected(a,X).
? connected(X,a).
? connected(X,Y).
```

PROLOG operational model

- abstract interpreter
- search of the solution
- unification

At each step the resolvent shrinks or grows [if : have functions]
[if : have facts that prove]

This process can be represented with a TREE

- Goal is the Root
- Branches will be selected in order (can be alternatives)
  DEPTH-FIRST from left to right

Abstract interpreter

Input: a goal G and a program P
Output: an instance of G logical consequence of P if it exists, otherwise NO

begin
  $R := G$; R resolvent
  finished := false;
  prove the goal in the resolvent;
  if $R = \{\}$
    then return G
    else return NO
end

Prove the goal

while not $R = \{\}$ and not finished **do**
begin
  choose a goal A in the resolvent
  choose a clause `A' :- B1,...,Bn` (renaming variables)
    such that $\theta = unify(A, A')$
  if no more choices
    then finished:=true;
    else begin
        substitute A with $B1, ..., Bn$ in R
        apply θ to R and G;
      end
end

The search tree

- the root is the initial goal;
- every node has one successor for each clause whose head unifies with a goal in the node. Every successor has a resolvent obtained by the parent node by replacing the chosen goal with the body of the clause, after applying the unifier.

Every node contains a resolvent. If it is empty the node is a *success node*. A node without successors, not a success node, is a *failure node*.

Every success node represents a solution. If the tree cannot be further expanded ant it does not have any success node then the goal fails.

The design choices of PROLOG

- the goal to be resolved determines the structure of the search tree;
- the clause determines the order of the successors of a node.

The PROLOG interpreter chooses the goals from left to right and the clauses are chosen wrt the order specified in the program. The resolvent is a stack. The search tree is built depth-first.

Is this computational complete? NO! ["greedy approach"]

For completeness: ITERATIVE DEEPENING

It's impossible to do complex things with Breadth First!

Change the rule order

```
descendant(X,Y):- son(X,Y).
descendant(X,Y) :- son(Z,Y),descendant(X,Z).
son(X,Y) :- mother(Y,X).
son(X,Y):- father(Y,X).

? descendant(daniele,X).
```

Change the order of the conjuncts in the rule body

```
descendant(X,Y):- son(X,Y).
descendant(X,Y):- descendant(X,Z),son(Z,Y). %1'
son(X,Y):- father(Y,X).
son(X,Y):- mother(Y,X).
```

Change the rule order II

```
descendant(X,Y):- descendant(X,Z),son(Z,Y). %1'
descendant(X,Y):- son(X,Y).
son(X,Y):- father(Y,X).
son(X,Y):- mother(Y,X).
```

Exercises

1. Define the relation `brother` and then the relation `cousin`
2. Build the search tree for the goal:
   `?- descendant(michela,eriberto).`
   and check the differences with
   `?- descendant2(michela,eriberto).`

   ```
   descendant2(X,Y):-son(Z,Y),descendant2(X,Z). % 2
   descendant2(X,Y):-son(X,Y). % 1
   ```

3. Build the search tree for the goal: `?- sg(jacopo,Y)` and explain the behavior of the program `sg`:

   ```
   sg (X,X).
   sg (X,Y):-
      parent(Z,X),sg(Z,W),parent(W,Y).
   ```

- Always get the first in the res stack

  Unification is key concept

- When we resolve we substitute the unification in the remaining clauses

[sg finds all pairs that have the same generation]

Building the tree can be tricky but it's important to understand PROLOG

- Conflict names must be handled carefully if they still happpear in the stack

Prolog: Data Structures

lunedì 2 marzo 2015
10:25

Data structures in PROLOG

Lecture 2

Summary

- Esercise solutions
- "is" Predicate
- Term Definition
- Unification
- Natural numbers
- Lists
- Esercises

Prolog: is predicate

`A is B` is a system predicate, true when the *evaluation* of the expression B returns a value, that is **assigned** to the variable A.

The evaluation of B is done using system operators.

Predicates defined by `is` are NOT invertible:
`?5 is X+Y.`
does not return the values of X and Y making the atom true.
`?X is 3+4.`
succeeds and returns `X=7`.

Prolog: programs with `is`

```
factorial(0,1).
factorial(Y,X):-
        Y1 is Y-1,
        factorial (Y1,X1),
        X is Y*X1.
```

result of the computation

Recap of Unification

Terms

The set TERM of *terms* is inductively defined as:

1. Every constant symbol is a term;
2. Every variable symbol is a term;
3. If $t_1 \ldots t_n$ are terms and f is an n-ary, $f(t_1, \ldots, t_n)$ is a term (called *functional term*).

Examples: x, c, $f(x, y + c)$,...

Atoms and clauses are defined as before.

Unification recap: Substitutions

A *substitution* is a function from the set of variables VAR to the set of terms TERM, (initially restricted to variables and constants):
$\sigma : Var \mapsto Term.$

Given t, $t\sigma$ is defined (without function symbols) as follows:

- if c is a constant symbol, $c\sigma = c$;
- if x is a variable symbol, $x\sigma = \sigma(x)$;

The substitution σ of a variable x by a term t is denoted by $x = t$ (or x/t).

OCCOUR CHECK
not in prolog because
it would make things too slow

Unification

An expression s is *more general* than an expression t, if t is an instance of s, but not viceversa.

Example: $p(a, X)$ is more general than $p(a, b)$.

more general if : still have open variables

A *unifier* of two expressions is the substitution, that makes them identical (when applied to them).

Example: $\{X = b\}$ is a unifier of $p(a, X)$ and $p(a, b)$.

Most general unifier

Intuitively, the *most general unifier* of two expressions, is the unifier that gives the most general instance of the two expressions.

Example: $\{X = b, Y = b, Z = a\}$ e $\{X = Y, Z = a\}$ are both unifiers of $p(a, X)$ and $p(Z, Y)$,

but $\{X = Y, Z = a\}$ is more general than $\{X = b, Y = b, Z = a\}$.

This unifier is unique up to variable renaming and is called mgu (most general unifier).

Unification (review)

1. $t_i = s_i$ identical variables or constants: skip to the next pair.
2. t_i variable: if t_i occurs in s_i then failure, otherwise $t_i = s_i$ is added to the unifier and all the occurrences of t_i are replaced by s_i.
3. s_i variable: as the previous one.
4. let t_i $f(tt_1, \ldots, tt_n)$ and s_i $g(ss_1, \ldots, ss_m)$ if $\neg(f = g) \vee \neg(n = m)$ then failure, otherwise unify $< tt_1, ss_1 >, \ldots < tt_n, ss_n >$.

Unification algorithm (full)

Input: C a set of pairs $< t_1, s_2 >$ where t_i, s_i are terms
Output: most general unifier θ, if exists, otherwise false
begin
  $\theta := \{\}$; success := true;
  while not empty(C) and success **do**
  begin
    choose $< t_i, s_i >$ in C;
    if $t_i = s_i$ **then** C:=C/$\{< t_i, s_i >\}$
      else if var(t_i)
        then if occurs(t_i, s_i) ← OCCOUR CHECK
            then success:=false;
            else begin
              $\theta :=$ subst(θ, t_i, s_i) $\cup \{t_i = s_i\}$;

```
                C:=subst(rest(C), t_i, s_i)
                end
        else if var(s_i)
          then if occurs(s_i, t_i)
            then success:=false;
            else begin
                 θ := subst(θ, s_i, t_i) ∪ {s_i = t_i};
                 C:=subst(rest(C), s_i, t_i)
                 end
          else if t_i = f(tt_1, ..., tt_n) and
                 s_i = g(ss_1, ..., ss_m) and
                 f = g ∧ n = m
                 then C:= rest(C) ∪{< tt_1, ss_1 >, ... < tt_n, ss_n >
                 else success := false
end;
```

if not success **then** output false **else** output true, θ
end

Unification: examples

$p(f(X,Y),a,g(b,W))$ unifies with $p(Z,X,g(b,Y))$.

Unifier $Z=f(x,y)$ $x=a$ $b=b$ $W=Y$

$p(f(X,Y),a,g(b,W))$ does not unify with $p(Z,f(a),g(b,Y))$.

$p(f(X,Y),a,g(b,W))$ does not unify with $p(X,a,g(b,Y))$.

Unification

Check whether the following pairs of expressions are unifiable, and write the mgu if they unify.
Otherwise, change the second term so that they unify. X, Y, Z are variables and a, b are constants.

(1u) $f(cons(car(X), cdr(Y)), Z, X)$ and $f(Z, Z, cons(car(X), cdr(a)))$
(2u) $f(g(x, a), g(b, a))$ and $f(y, y)$
(3u) $P(g(x, a), f(b, a))$ and $P(g(f(b, y), y), f(z, y))$
(4u) `P([X|[a,b]])` and `P([a|[a|[Xs]]])`

A program for the class timetable

```
course(ai,timetab(tu,10,12),
       teacher(nardi,d),room(diag,b2)).
course(ai,timetab(we,10,12),
       teacher(nardi,d),room(diag,b2)).
course(ai,timetab(fr,10,12),
       teacher(nardi,d),room(diag,b2)).
...
```

A program for the class timetable

```
teaches(Tea,Course) :- course(Course,Timetab,Tea,Room).
length(Course,Len) :-
    course(Course,timetab(Day,Start,End),Tea,Room),
    plus(Start,Len,End).
hasClass(Tea,Day) :-
    course(Course,timetab(Day,Start,End),Tea,Room).
busy(Room,Day,Time) :-
    course(Course,timetab(Day,Start,End),Tea,Room),
    Start =< Time, Time =< End.
```

Natural numbers

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).

plus1(0,X,X) :- natural_number(X).
plus1(s(X),Y,s(Z)):- plus1(X,Y,Z).

lesseq1(0,X) :- natural_number(X).
lesseq1(s(X),s(Y)) :- lesseq1(X,Y).
```

Lists

- `[a,b,c,d]` is a 4 element list;
- `[a | X]` is a list whose first element is a and the rest of the list is denoted by the variable `X`;
- `[Y | X]` is a list whose first element is denoted by the variable `Y` and il resto della lista è denotato dalla variable `X`.

`[a | X]` is the same as `cons(a,X)`

cons(a, X) a: head, X: rest of the list

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

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

[a, b, c]

[a | X]

[a | [b, c]]

needed for recursion

Programs using lists

```
append1([],Ys,Ys).
append1([X|Xs],Ys,[X|Zs]) :- append1(Xs,Ys,Zs).

prefix([],_Ys).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

reverse1([],[]).
reverse1([X|Xs],Zs) :- reverse1(Xs,Ys),
                       append1(Ys,[X],Zs).
```

Programs using lists and numbers

```
len([],0).
len([_X|Xs],s(N)) :- len(Xs,N).
```

```
len([],0).
len([_X|Xs],N) :- len(Xs,N1), N is N1 + 1.
```

Sorting lists

simpliest way (not efficient)

```
sort1(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys),
                          permutation(Ys,Zs).
permutation([],[]).

ordered([]).
ordered([_X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

We can put `X \= Y` so that it doesn't unify them

Home exercises

1. build the search tree for:
   `?- member(c,[a,c,b]).`
   `?- plus1(Y,X,s(s(s(s(0))))).` and
   `?- reverse([a,b,c],X).`
2. Write the PROLOG programs `times`, `power`, `factorial`, `minimum` using the definitions given for natural numbers.
3. Write the PROLOG programs `suffix`, `subset`, `intersection` using lists to represent sets.
4. Write a PROLOG program for a depth-first visit of possibly cyclic graphs, represented through the relation `arc(X,Y)`
5. Write a PROLOG program implementing insertion sort on lists.

Esercitazione 1

lunedì 2 marzo 2015
12:00

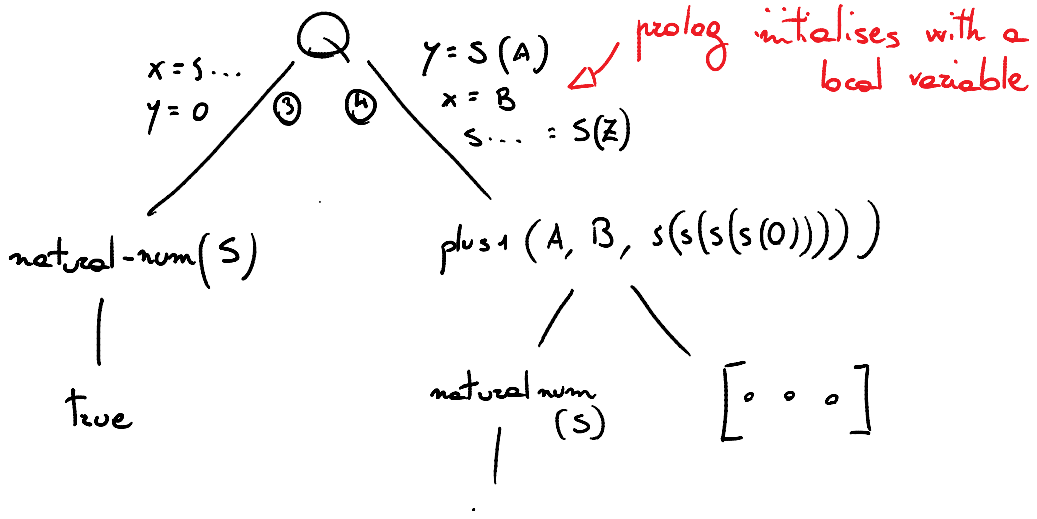
① ? member(c, [a, c, b])

Prog member(x, [x | xs])
member(x, [y | ys]) :- member(x, ys)

Q
|
member(c, [c, b]) true

② ? plus1(y, x, s(s(s(s(s(0))))))

Prog: ① natural_num(0)
② natural_num(s(x)) :- natural_num(x)
③ plus1(0, x, x) :- natural_num(x)
④ plus1(s(x), y, s(z)) :- plus1(x, y, z)



true

? reverse([d, b, c], X)

Program:

reverse([], [])

reverse([X | Xs], Zs) :- reverse(Xs, Ys)
append(Ys, X, Zs)

append([], Ys, Ys)

append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs)

Q
X = Zs = G4
Xs = [b, c]
reverse([b, c], G3)
append(G3, d, G4)
G3 = G9
reverse([c], G10)
append(G10, b, G9)
append(G9, d, G4)
G10 = G20
reverse([], G25)
append(G25, c, G20)
append(G20, b, G9)

append (G9, d, G4)

| G25 = []

append ([], c, G20) . . .

| G20 = c

append ([c], b, G9)

append (G9, d, G4)

④ | G9 = [c | G30]

append ([], b, G30)

append ([c|G30], d, G4)

| G30 = b

append ([c|b], d, G4)

④ | G4 = [c | G70]

append (b, d, G70)

| G70 = [b, G80]

append ([], d, G80)

| G80 = d

G70 = [b, d]

G4 = [c | b, d] done

(5) times

```
times(0, -x, 0).
times(s(0), X, X).
times(s(x), Y, Z) :- times(x, Y, U)
                     plus1(U, Y, Z)
```

power

```
power(s(x), 0, s(0))
power(0, s(x), 0)
power(s(x), Y, Z) :- power(x, Y, U)
                     times(U, Y, Z)
```

factorial

```
fac(0, s(0))
fac(s(x), Y) :- fac(x, Z)
                times(s(x), Z, Y)
```

suffix

```
suff(Xs, Xs)
suff(Xs, [Y|Ys]) :- suff(Xs, Ys)
```

subset

```
sub([], Xs)
sub([Y|Ys], Xs) :- member(Y, Xs)
                   subset(Ys, Xs)
```

intersection

```
int([], X, [])
int([Y|Ys], X, [X|Zs]) :- member(X, Y)
                          intersection(Ys, X, Zs)
int([Y|Ys], X, Zs) - intersection(Ys, X, Zs)
```

```
arc(X,Y)

connected(X, X, [])
connected(X, Y, [X|Q]) :- arc(X,Z), nonmember(Z,Q)
                          connected(Z, Y, Q)
```

where Q is the visited list

to avoid loops: visited list

```
nonmember(X, [Y|Ys]) :- X \== Y,
                        nonmember(X, Ys, Z)
nonmember(-X, [])
```

Prolog: Non-Linear Data Structures

giovedì 5 marzo 2015
10:41

PROLOG: NON LINEAR DATA STRUCTURES

LECTURE 3

Summary

- Lists of lists
- Binary trees

Starts from the outermost wk to create

the tree

Evaluating Expressions

Write a PROLOG program, whose input is an arithmetic expression represented by a term and computes its value.

```
evalExpression(plus(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1+N2.
evalExpression(minus(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1-N2.
evalExpression(mult(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1*N2.
evalExpression(frac(A,B), N):-  evalExpression(A, N1),
        evalExpression(B, N2), N is N1 // N2.
evalExpression(X, X).
```

Lists of lists

`memberlist(X,L)` X is an element of the list of lists L.

`memberlist(X,Y)` is true if one of the following conditions holds:
- X is the first element of L
- X is not the first element of L, the first element of L is a list and X is a member of the first element of L.
- X is a member of the rest of L.

```
memberlist(X,[X|_Xs]).
memberlist(X,[Y|_Ys]) :- memberlist(X,Y).
memberlist(X,[_Y|Ys]) :- memberlist(X,Ys).
```

NB. always put at start the easiest teem (conclusion condition) → otherwise might loop forever

Prolog: binary trees

binary_tree(X) is true if X is a binary tree, that is:
- X is the empty tree.
- X is a structure with a node label binary trees.

```
binary_tree(void).
binary_tree(tree(_Element,Left,Right)) :-
       binary_tree(Left), binary_tree(Right).
```


Isomorphism - same tree structure
proved if left and right branches are both isomorphic (recursion)

Binary trees: isomorphism

isotree(X,Y) is true if X and Y are isomorphic, that is:
- X and Y are both the empty tree
- X and Y have the same root element and the left and right subtrees are isomorphic

```
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
         isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
         isotree(Left1,Right2), isotree(Right1,Left2).
```

Binary trees: pre-order

`preorder (X,Y)`: Y is the list containing the elements of the tree X, as encountered in the pre-order visit.

```
preorder (void, [void]).
preorder (tree(X, Left, Right), CompleteList):-
        preorder(Left, LeftList),
        preorder(Right, RightList),
        append([X|LeftList], RightList, CompleteList).
```

Binary trees: breadth-first visit

`breadth_first(X,Y)`: Y is the list containing the elements of the tree X, as encountered in the breadth-first visit.

```
bf([], []).
bf([void | Rest], Ls):- bf(Rest, Ls).
bf([tree(I, Dx, Sx)| Rest], [I|Ls]):-
    append(Rest, [Sx, Dx], Nodes), bf(Nodes, Ls).
```

Breadth First

prolog goes depth-first, so we need a queue

AI is search, so this is extremely important

STACK - depth first

QUEUE - breadth first

Exercises

1. Write a PROLOG program counting the elements of a list of lists.
2. Write a PROLOG program which implements member for a binary tree.
3. Write a PROLOG program which returns a list containing all the nodes at a given depth D of a binary tree.

Exercises on binary trees

a) Consider the PROLOG terms representing the binary trees whose nodes are labelled by a constant symbol and, in addition, store the depth of the node. Write a PROLOG program that returns true if its argument is a binary tree as above specified.

b) Write a PROLOG program that, given in input a binary tree and a constant, returns the depth of a node containing the given constant.

c) Write a PROLOG program that, given in input a binary tree without the depth information on the nodes and a constant, returns the depth of a node containing the given constant.

d) Write a PROLOG program that, given in input a binary tree without the depth information on the nodes, returns an iso-

morphic binary tree with the depth information stored in the nodes.

Esercitazione 2

lunedì 16 marzo 2015
11:58

$[X|X_s]$ is a list of lists

- countListElements ([], 0)

countListElements ([X|Xs], N) :- countListElements (X, N1),
countListElements (Xs, N2),
N is N1 + N2.

countListElements (-X, 1).

separate search for left and right tree (avoid semicolumns!)

- hasElem (X, tree(X, -L, -R)) .

hasElem (X, tree (-E, L, -R)) :- hasElem (X, L) .

hasElem (X, tree(-E, -L, R)) :- hasElem (X, R).

- pzintElem (void, [], -h, -D).

pzintElem (tree(X, -L, -R), [X], -D, -D).

pzintElem (tree(-X, L, R), Xs, In, D) :-
In < D, In1 is In+1,
pzintElem (L, Ls, In1, D),
pzintElem (R, Rs, In1, D),
append (Ls, Rs, Xs).

add a variable to see if depth is stored right

- `bt(void).`
  `bt(tree(node(_E,_D),L,R)) :- bt(L), bt(R).`

- `assignD(void,_D,void).`
  `assignD(tree(X,L,R),D,tree(node(X,D),Ls,Rs)) :-`
  `Z is D+1, assignD(L,Z,Ls), assignD(R,Z,Rs).`

Prolog: Inference Control

mercoledì 11 marzo 2015
09:47

Inference control and meta-programming

Lecture 4

AI2, Daniele Nardi, 2014-2015 Lecture 4 1

Inference control and meta-programming

(in PROLOG)

- Negation as Failure
- Inference control
- Cut to control backtracking
- Meta-predicates (Any Prolog textbook e.g. SS)
- Terms and their representation
- Meta-interpreters

negation as finite failure

SLDNF = SLD-resolution + NF

NF (Negation as Finite Failure):
an atom $\neg A$ succeeds if the derivation tree corresponding to the goal A is finite and its leaves are all failure leaves.

Consistent (with the various characterizations of negation), but incomplete.

Conditions for completeness are very stringent:

- instantiated variables in negated atoms
- constraints on the program structure

Answer Set Programming extends SLDNF, computing the answer on the basis of the stable model semantics

if A is not provable ¬A is T

Negation as Failure in PROLOG

In prolog negation is realized as **failure** in the search.

If `X` is a list of atoms, `not(X)` is true if the interpreter cannot find a proof for `X`.

```
student(bill).
student(joe).
married(joe).

unmarried_student(X) :- not(married(X)), student(X).
? unmarried_student(bill/joe).
```

Termination of negation

The negation mechanism of Prolog is neither correct nor complete: it depends on the ordering of evaluation of clauses

The termination of `not(X)` depends on the termination of `X`:
- If `X` terminates, `not(X)` terminates.
- If `X` has infinite solutions, `not(X)` terminates if a success node is found before an infinite branch is encountered.

```
married(gino, remberta).
married(X,Y):-married(Y,X).

? not(married(remberta, gino)).
```

Example

```
p(s(X)):-p(X).
q(a).
```

The query `not(p(X),q(X))` does not terminate. - there's a loop, variable never instantiated

Instead, `not(q(X),p(X))` terminates. - variable instantiated

Using negation with completely instantiated atoms does not generate incorrect behaviors, in case the atoms terminate. If negation is used with variables we must take into account the execution mechanism of prolog

- Termination of negation is problematic!
- Atoms must be instantiated before functions!

Problems with negation

```
unmarried_student(X):-not(married(X)),student(X).
student(bill).
married(joe).
?- unmarried_student(X).
```

```
unmarried_student1(X) :- student(X), not(married(X)).
student(bill).
student(joe).
married(joe).
?- unmarried_student1(X).
```

Sufficient condition for correctness:
variables in atoms must be instantiated before they are executed

Instantiate variables before calling a not on them!

Inference control in PROLOG programs

- Program specification
- Clause ordering
- Conjunct ordering
- Cut to control backtracking

depends on the branching factor

bigger search space is worse!

PROLOG program specification

```
descendant(X,Y):-son(X,Y). % 1
descendant(X,Y):-son(Z,Y),descendant(X,Z).

descendant(X,Y):-son(X,Y). % 2
descendant(X,Y):-son(X,Z),descendant(Z,Y).

descendant(X,Y):-son(X,Y). % 3
descendant(X,Y):-descendant(X,Z),descendant(Z,Y).
```

From ancestor to descendant

From descendant to ancestor

Backtrack causes problems because of the variables!

Explicit Inference Control

"What is the income of the US president' wife?"

```
income(S,I),married(S,P),job(P,uspresident).
```

"Do I have an american cousin?"

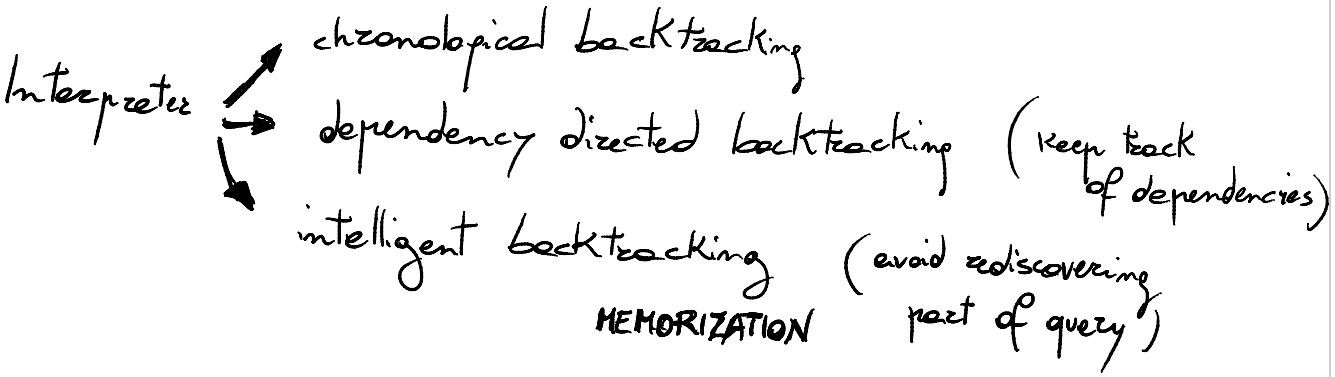
```
american(Y),cousin(daniele,Y).
```

♢ **meta-reasoning** to decide which order of the atoms is more effective → less computation needed

Given a predicate returning the number of instances of a relation, one can sort the conjuncts to optimize the computation of the answer.

Efficiency of PROLOG implementations

- **chronological backtracking**
- **dependency directed backtracking**
- **intelligent backtracking** memorizing the previous derivation to avoid re-discovering them



cuts portions of the search tree

(decision made by the programmer)

Cut

Generic clause with cut:

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n.$$

If the current goal G unifies with A and $B_1, \ldots, B_k$ are successfully proven,

- any other proof achievable by unifying G with another clause is eliminated from the search tree;
- any alternative proof of $B_1, \ldots, B_k$, is also eliminated from the search tree.

Discard alternative proofs

cousin(x, y), !, living(x, y) →

I don't backtrack on cousin

if this fails the first time i return to "before proving cousin(x, y)" (again)

Cut: example

```
merge1([X|Xs],[Y|Ys],[X|Zs]):-
       X<Y,!,
       merge1(Xs,[Y|Ys],Zs).
merge1([X|Xs],[Y|Ys],[X,Y|Zs]):-
   X=:=Y,!,
   merge1(Xs,Ys,Zs).
merge1([X|Xs],[Y|Ys],[Y|Zs]):-
   X>Y,!,
   merge1([X|Xs],Ys,Zs).
merge1([],X,X):- !.
merge1(Y,[],Y):- !.
```

Green and red cuts: discarding valid solutions

A cut is **green**, when it does not affect the program solutions, **red** otherwise.

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- ancestor(X,Z),!,parent(Z,Y).
```

Stops the search at the first ancestor of X: Z. Since Y may not be parent of Z, the search fails.

- If things are mutually exclusive I can put a cut
- Be careful not to cut **valid solutions**! (red/green cut)
- Can be used to control backtracking

Controlling backtracking

```
cousin(massimo,daniele),!,american(daniele)

member(a,C), !, q(a).

member1(X,[X|Xs]) :- !.
member1(X,[Y|Ys]) :- member1(X,Ys).

member1(a,C).
```

If it's already true there's no need to backtrack

All green cuts!

Meta-logical predicates

♢ Needed to deal with a program element as data.

♢ Access the program's internal representation.

♢ Enable meta-programming.

♢ Meta-programming makes it possible to **explicitly** control execution

Explicitly control execution

findall( ) - collect all solutions to understand execution process

Meta predicates are used to control execution

Collecting solutions

It is often useful to compute all the solutions of a program.

`findall(Term,Goal,List).`

`List` is the list, obtained computing `Goal` and taking the bindings of `Term`.

`findall(X,grandfather(giovanni,X),Grandchildren).`

Variables as programs

- `call(X)` executes the goal `X` which must be instantiated, (simply written `X`).

Es: Disjunction

```
X ; Y :- X.
X ; Y :- Y.
```

Negation as failure

```
not(G):- G, !, fail.
not(G).
```

`fail` always fails.

Type meta-logical predicates

- `var(Term)` succeeds if `Term` is a variable T if variable (not constant)
- `nonvar(Term)` succeeds if `Term` is not a variable T if not variable (constant)

```
grandfather(X,Z):- nonvar(X),
                   parent(X,Y),
                   parent(Y,Z).
grandfather(X,Z):- nonvar(Z),
                   parent(Y,Z),
                   parent(X,Y).
```

all these are meta-predicates

Ground terms

```
ground(Term) :- nonvar(Term), constant(Term).
ground(Term) :-  nonvar(Term), compound(Term),
                functor(Term,F,N), ground(N,Term).

ground(N,Term) :- N > 0, arg(N,Term,Arg),
                  ground(Arg), N1 is N-1,
                 ground(N1,Term).
ground(0,Term).
```

a(b,c) funzione composta

Ground terms are without variables
just constants

Equality and Unification

- `X == Y` succeeds if `X` and `Y` are the same constant, the same variable, or structures with the same functor and recursively == (negated \ ==)
- `X = Y` succeeds if `X` and `Y` unify

`is` numerical assignment

`=` assignment

`==` checks if they are the same or not (\==)

`=:=`

Unification

```
unify1(X,Y) :- var(X), var(Y), X=Y.
unify1(X,Y) :- var(X), nonvar(Y),
               not_occurs_in(X,Y), X=Y.
unify1(X,Y) :- var(Y), nonvar(X),
               not_occurs_in(Y,X), Y=X.
unify1(X,Y) :- nonvar(X), nonvar(Y),
               constant(X), constant(Y), X=Y.
unify1(X,Y) :- nonvar(X), nonvar(Y), compound(X),
               compound(Y), term_unify1(X,Y).

term_unify1(X,Y) :- functor(X,F,N), functor(Y,F,N),
                    unify1_args(N,X,Y).
% Program 10.6    Unification with the occurs check
```

Argument unification

```
unify1_args(N,X,Y) :- N > 0, unify1_arg(N,X,Y),
                      N1 is N-1, unify1_args(N1,X,Y).
unify1_args(0,X,Y).

unify1_arg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY),
                     unify1(ArgX,ArgY).
```

Occurs check

```
not_occurs_in(X,Y) :- var(Y), X \== Y.
not_occurs_in(X,Y) :- nonvar(Y), constant(Y).
not_occurs_in(X,Y) :- nonvar(Y), compound(Y),
                      functor(Y,F,N),
                      not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :- N > 0, arg(N,Y,Arg),
                        not_occurs_in(X,Arg),
                        N1 is N-1,
                        not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).
```

VANILLA META INTERPRETER

Basic meta interpreter

A PROLOG **meta-interpreter** is a PROLOG interpreter written in PROLOG.

```
solve1(true).
solve1((A,B)) :- solve1(A), solve1(B).
solve1(A) :- clause(A,B), solve1(B).
```

`clause/2` returns the head and the body of the clause (at least one arg must be instantiated).

Meta interpreter with unification

```
solve2(true).
solve2((A,B)) :- solve2(A), solve2(B).
solve2(A) :- clause(X,B), unify1(X,A), solve2(B).
```

Meta interpreter tracing the proof

```
solve_trace(Goal) :- solve_trace(Goal,0).

solve_trace(true,Depth) :- !.
solve_trace((A,B),Depth) :-  !, solve_trace(A,Depth),
                             solve_trace(B,Depth).
solve_trace(A,Depth) :- builtin(A), !, A,
                        display(A,Depth), nl.
solve_trace(A,Depth) :- clause(A,B), display(A,Depth),
                        nl, Depth1 is Depth + 1,
                        solve_trace(B,Depth1).

display(A,Depth) :-  Spacing is 3*Depth,
                     put_spaces(Spacing), write(A).
```

```
put_spaces(N) :- between(1,N,I), put_char(' '), fail.
put_spaces(N).

between(I,J,I) :- I =< J.
between(I,J,K) :- I < J, I1 is I + 1, between(I1,J,K).

% Program 17.7 A tracer for Prolog
```

Meta interpreter building the proof tree

```
solve(true,true) :- !.
solve((A,B),(ProofA,ProofB)) :- !, solve(A,ProofA),
                                 solve(B,ProofB).
solve(A,(A:-builtin)) :- builtin(A), !, A.
solve(A,(A:-Proof)) :- clause(A,B), solve(B,Proof).

% Program 17.8
% A meta-interpreter for building a proof tree
```

Meta interpreter with certainty factors

```
solve(true,1) :- !.
solve((A,B),C) :- !, solve(A,C1),
                  solve(B,C2),
                  minimum(C1,C2,C).
solve(A,1) :- builtin(A), !, A.
solve(A,C) :-clause_cf(A,B,C1),
              solve(B,C2), C is C1 * C2.

minimum(X,Y,X) :- X =< Y, !.
minimum(X,Y,Y) :- X > Y, !.

%  Program 17.9
% A meta-interpreter for reasoning with uncertainty
```

Esercises (i)

- Define in PROLOG the relation `onlychild(X)`, exploiting the family defined before.
- Define the predicate `notMember(X,L)`, true if `X` does not occur in the list `L`. Provide a definition using NAF and one without it, and compare them.
- Define union using the negation of `member`. Build the serach tree for the goal

  ```
  ?  union([a,b],[b,c],Z).
  ```

Esercises (ii)

- Apply the cut operator to the program `insert` of assignment 2, and draw the search tree for the goal `insert(4, [3,5,7], X)`.
- Define a predicate `sortm(X,Y,Z)`, to order lists `X` and `Y` and return the result in `Z`, through merge-sort.
- Add cuts to the program `sortm` if needed to obtain a single solution.

Esercises (iii)

- Check the execution of the vanilla meta-interpreter by computing queries of predicates `grandfather`, `descendant` of the family program.
- Consider the vanilla meta-interpreter returning the proof tree. Add a few built-in predicates and verify the behaviour of the meta interpreter on built-in and user-defined `member`. Compare the proof-tree obtained with the meta-interpreter tracing the execution of the same query with the standard interpreter.

Esercitazione 3

lunedì 16 marzo 2015
12:58

- siblings(a, b).

  onlyChild(X) :- not(hasSiblings(X)).
  hasSiblings(X) :- siblings(X, Y), X \== Y

- notMember(X, []).
  notMember(X, [H|B]) :- X \== H, notMember(X, B)

or with negation as failure

notMember(X, L) :- not(member(X, L))

- union1([], L, L)
  union1(L, [], L)
  union1([X|Xs], Y, [X|Zs]) :- notMember(X, Y), union1(Xs, Y, Zs).

  union1([X|Xs], Y, Zs) :- member(X, Y), union(Xs, Y, Zs)

↝ or with cut to avoid backtracking on the 4° from the 3°

union1([], L, L)

```
union1 ( L, [], L)

union1 ([X|Xs], Y, [X|Zs]) :- notMember(X,Y), !,
                               union1(Xs, Y, Zs).

union1 ([X|Xs], Y, Zs) :- union(Xs, Y, Zs)
```

- ```
  insert ( X, [], [X])

  insert ( X, [Y|Ys], [Ys|Zs]) :- X>Y, !, insert (X, Ys, Zs)

  insert ( X, [Y|Ys], [X, Y|Zs]) :- X<=Y !
  ```

- ```
  sortm ( [], []).

  sortm ([X], [X]).

  sortm ([X1, X2], [X1, X2]) :- X1 <= X2 !

  sortm ([X1, X2], [X2, X1]) :- X1 > X2 !

  sortm (X, Z) :- append([X1, X2], Y, X), !,
                  sortm ([X1 X2], Z1), sortm (Y, W),
                  merge (Z1, W, Z)
  ```

Prolog: AI classics

sabato 12 ottobre 2013
11:43

PROLOG: AI Classics

Lecture 5

Summary

- Automated search
- Non deterministic programming
- Other AI implementations

Remember: negation as failure

Search in PROLOG

- Modularizing search;
- Predicates **applicable(Action,State)** and **apply(Action,State,NewState)**;
- Avoid repeated states; identical states; predicata **sameState(S1,S2)**;
- Predicates **initState(S)** and **finalState(S)**;
- Depth first search using PROLOG backtracking.

↳ Space for search proportional to depth

PROLOG program structure

```
solution(ActionList):-init(I),
   solution(I,[],ActionList).
solution(State,VisitedStates,[]) :- final(State).
solution(State,VisitedStates, [Action|Rest]) :-
   applicable(Action,State),
   apply(Action,State,NewState),
   not visited(NewState,VisitedStates),
   solution(NewState,[State|VisitedStates],Rest).
```

1° example

```
visited(State,[VisitedState|OtherVisitedStates]) :-
   sameState(State,VisitedState).
visited(State,[VisitedState|OtherVisitedStates]) :-
   visited(State,OtherVisitedStates).

sameState(S1,S2):-... (depends on the state representation)
```

2° example

Automated Search Problem

Crossing the river: LPC

A man has a wolf, a sheep and a cabbage; he is on a river bench with a boat, whose maximum load for a single trip is the man, plus only one of his 3 goods. The man wants to cross the river with his goods, but he must avoid that (when he is far away) the wolf eats the sheep or the sheep eats the cabbage. what should he do?

We need to cast the problem to be able to use algorithms on it

↓ MODELLING IS ESSENTIAL!

FORMALIZATION OF THE PROBLEM

LPC: state space S

Let $S = D \times D \times D \times D$ where $D = \{a, b\}$, a and b are the 2 river benches
$< U, L, P, C > \in S$ represents the position of the man, the wolf and the cabbage.

Moreover, in D $\overline{a} = b$ and $\overline{b} = a$. ← complement

Initial State: $< a, a, a, a >$
Goal state $< b, b, b, b >$

LPC: operators

Operator	Condition	From state	To state
carryNothing	$L \neq P, P \neq C$	$< U, L, P, C >$	$< \overline{U}, L, P, C >$
carryWolf	$U = L, P \neq C$	$< U, L, P, C >$	$< \overline{U}, \overline{L}, P, C >$

LPC: a solution

$< a, a, a, a > -carrySheep\rightarrow < b, a, b, a >$
$< b, a, b, a > -carryNothing\rightarrow < a, a, b, a >$
$< a, a, b, a > -carryWolf\rightarrow < b, b, b, a >$
$< b, b, b, a > -carrySheep\rightarrow < a, b, a, a >$

Operator	Condition	From state	To state
carryNothing	$L \neq P, P \neq C$	$< U, L, P, C >$	$< \overline{U}, L, P, C >$
carryWolf	$U = L, P \neq C$	$< U, L, P, C >$	$< \overline{U}, \overline{L}, P, C >$
carrySheep	$U = P$	$< U, L, P, C >$	$< \overline{U}, L, \overline{P}, C >$
carryCabbage	$U = C, L \neq P$	$< U, L, P, C >$	$< \overline{U}, L, P, \overline{C} >$

no other constraints (wolf and cabbage can stay together)

$< a, a, a, a > -carrySheep\rightarrow < b, a, b, a >$
$< b, a, b, a > -carryNothing\rightarrow < a, a, b, a >$
$< a, a, b, a > -carryWolf\rightarrow < b, b, b, a >$
$< b, b, b, a > -carrySheep\rightarrow < a, b, a, a >$
$< a, b, a, a > -carryCabbage\rightarrow < b, b, a, b >$
$< b, b, a, b > -carryNothing\rightarrow < a, b, a, b >$
$< a, b, a, b > -carrySheep\rightarrow < b, b, b, b >$

variable = constant UNIFICATION
constant = constant EQUALITY

Crossing the river in PROLOG

Order in which we write things changes execution!

```
init(positions(0,0,0,0)).
final(positions(1,1,1,1)).
applicable(carryNothing,positions(U,L,P,C)):-L =/= P,P =/= C.
applicable(carryWolf,positions(UeL,UeL,P,C)):-P =/= C.
applicable(carrySheep,positions(UeP,L,UeP,C)).
applicable(carryCabbage,positions(UeC,L,P,UeC)):-L =/= P.
apply(carryNothing,positions(U,L,P,C),positions(NewU,L,P,C)):-
   NewU is 1-U.
apply(carryWolf,positions(U,L,P,C),positions(NewU,NewL,P,C)):-
   NewU is 1-U, NewL is 1-L.
apply(carrySheep,positions(U,L,P,C),positions(NewU,L,NewP,C)):-
   NewU is 1-U, NewP is 1-P.
apply(carryCabbage,positions(U,L,P,C),positions(NewU,L,P,NewC)):-
   NewU is 1-U, NewC is 1-C.
```


The 2-jug problem

Two jugs A (4 liters) and B (3 liters) are initially empty and do not have any marks allowing to determine the exact amount of a partial filling. Given a water source and a sink, we need to fill exaclty 2 liters in jug A

2J: State space

$S = \mathcal{N} \times \mathcal{N}$, where $< Q_a, Q_b > \in S$ denotes the amount of water in the 2 jugs;

Initial state $< 0, 0 >$
Goal State $< 2, \_ >$ don't care

The goal state is partially specified. It corresponds to a set of goal states:

$\{< 2, 0 >, < 2, 1 >, < 2, 2 >, < 2, 3 >\}$

We only care about the water in jug A.

2J: operators

Operator	Condition	From state	To state
emptyA	$Q_a > 0$	$< Q_a, Q_b >$	$< 0, Q_b >$
emptyB	$Q_b > 0$	$< Q_a, Q_b >$	$< Q_a, 0 >$
fillA	$Q_a < 4$	$< Q_a, Q_b >$	$< 4, Q_b >$
fillB	$Q_b < 3$	$< Q_a, Q_b >$	$< Q_a, 3 >$
emptyAinB	$Q_a + Q_b \leq 3, Q_a > 0$	$< Q_a, Q_b >$	$< 0, Q_a + Q_b >$
emptyBinA	$Q_a + Q_b \leq 4, Q_b > 0$	$< Q_a, Q_b >$	$< Q_a + Q_b, 0 >$
fillAwithB	$Q_b \geq 4 - Q_a, Q_a < 4$	$< Q_a, Q_b >$	$< 4, Q_b - (4 - Q_a) >$
fillBwithA	$Q_a \geq 3 - Q_b, Q_b < 3$	$< Q_a, Q_b >$	$< Q_a - (3 - Q_b), 3 >$

must have some water!

we don't empty completely a container

2J: solution

$< 0, 0 > -fillB\rightarrow < 0, 3 >$
$< 0, 3 > -emptyBinA\rightarrow < 3, 0 >$
$< 3, 0 > -fillB\rightarrow < 3, 3 >$
$< 3, 3 > -fillAwithB\rightarrow < 4, 2 >$
$< 4, 2 > -emptyA\rightarrow < 0, 2 >$
$< 0, 2 > -emptyBinA\rightarrow < 2, 0 >$

Prolog: The 2-jug problem

```
init(state(0,0)).
final(state(2,X)).
applicable(emptyA,state(Qa,Qb)) :- Qa > 0.
applicable(emptyB,state(Qa,Qb)) :- Qb > 0.
applicable(fillA,state(Qa,Qb)) :- Qa < 4.
applicable(fillB,state(Qa,Qb)) :- Qb < 3.
applicable(emptyAinB,state(Qa,Qb)) :- Qa>0, Qtot is Qa + Qb,
                                      Qtot=<3.
applicable(emptyBinA,state(Qa,Qb)) :- Qb>0, Qtot is Qa + Qb,
                                      Qtot=<4.
applicable(fillAwithB,state(Qa,Qb)) :- Qa<4, Qtrasf is 4 - Qa,
                                       Qtrasf=<Qb.
applicable(fillBwithA,state(Qa,Qb)) :- Qb<3, Qtrasf is 3 - Qb,
                                       Qtrasf=<Qa.
```

2J in PROLOG

```
apply(emptyA,state(Qa,Qb),state(0,Qb)).
apply(emptyB,state(Qa,Qb),state(Qa,0)).
apply(fillA,state(Qa,Qb),state(4,Qb)).
apply(fillB,state(Qa,Qb),state(Qa,3)).
apply(emptyAinB,state(Qa,Qb),state(0,Qtot)) :- Qtot is Qa+Qb.
apply(emptyBinA,state(Qa,Qb),state(Qtot,0)) :- Qtot is Qa+Qb.
apply(fillAwithB,state(Qa,Qb),state(4,NewQb)) :-
                                NewQb is Qb-(4-Qa).
apply(fillAwithB,state(Qa,Qb),state(NewQa,3)) :-
                                NewQa is Qa-(3-Qb).
```


We use parameters differently if not instantiated

Non deterministic programming

```
% true if Xs and Ys have an empty intersection
nonEmptyIntersection(Xs,Ys):-member(X,Xs),member(X,Ys).

member(X,Xs):- append(A,[X|B],Xs).
```

test
X is now instantiated!

generate
X not yet instantiated
function picks an element
(in this case the first)

Better ways to split!
Not optimized

Generate and test

```
find(X) :- generate(X), test(X).

%ordinamento
sort(X,Y):- permutation(X,Y), ordered(Y).
%parsing
parse(X):- append(Y,Z,X), soggetto(Y), verbo(Z).
```

tries all the possible sublists

Pushing the test inside the generator

- In sort generate a permutation whose first element is the smallest.
- In parsing do only the decompositions such that the prefix matches.

N-queens

```
queens(N,Qs) :- range(1,N,Ns), permutation(Ns,Qs), safe(Qs).
safe([Q|Qs]) :- safe(Qs), not attack(Q,Qs).
safe([]).
attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|Ys]) :- X is Y+N ; X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).
range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).
```


XML is based on parsing (efficient)
* symbols tell right away which rules to apply

Anticipate the test

```
queens1(N,Qs) :- range(1,N,Ns), queens1(Ns,[],Qs).

queens1(UnplacedQs,SafeQs,Qs) :-
        select(Q,UnplacedQs,UnplacedQs1),
        not attack(Q,SafeQs),
        queens1(UnplacedQs1,[Q|SafeQs],Qs).
queens1([],Qs,Qs).
```

First always think about the nondeterministic solutions that are possible!

Map Coloring

Given a map find an assignment of colors such that nations having a common border are colored differently.

The map can be represented as a graph, whose nodes represent the regions and whose arcs between nodes a and b represents a common border between a and b.

Map Coloring(i)

The map is a list of nations.
Nations have a name, a color and a list of Neighbors.

```
[region(a,A,[B,C,D]),     region(b,B,[A,C,E]),
```

Map Coloring(ii)

- (*suitably*) choose the color for a nation.
- check the choice with other nations.

```
color_map([Region|Regions],Colors) :-
        color_region(Region,Colors),
```

Nations have a name, a color and a list of Neighbors.

```
[region(a,A,[B,C,D]),     region(b,B,[A,C,E]),
region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
region(e,E,[B,C,F]),     region(f,F,[C,D,E])]
```


- *(suitably)* choose the color for a nation.
- check the choice with other nations.

```
color_map([Region|Regions],Colors) :-
        color_region(Region,Colors),
        color_map(Regions,Colors).
color_map([],_Colors).
```


Choose the color for a nation

```
color_region(region(_Name,Color,Neighbors),Colors) :-
        /*select a color, in Colors1 there are all remaining colors*/
        select(Color,Colors,Colors1),
        /*check the colors of the neighbor nations*/
        members(Neighbors,Colors1).

members([X|Xs],Ys) :- member(X,Ys), members(Xs,Ys).
members([],_Ys).
```


ANALOGY

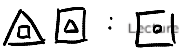
Given 3 figures, a, b, c, and a set of answers R, find a relation between a and b, and apply the relation to c.

Applying the relation to c gives a new candidate figure: d.
Find figure $f \in R$, best approximating d.

Base algorithm:

find the relation between 1.2 to find which is the missing pic based on 3.

$r = findRelation(a, b)$
$f = applyRelation(c)$
if $f \in R$ return f else return fail



ANSWERS:

ANALOGY: program

```
/*Definition of the infix operator is_to, priority 100
A is_to B is equivalent to a is_to(A,B). */

:- op(100,xfx,[is_to]).

analogy(A is_to B,C is_to X,
        match(A,B,Match),
        match(C,X,Match),
        member(X,Answers).
```


ANALOGY: test

```
test_analogy(Name,X) :-
    figures(Name,A,B,C),
    answers(Name,Answers),
    analogy(A is_to B,C is_to X,Answers).

figures(test1,inside(square,triangle),inside(triangle,square),
inside(circle,square)).
answers(test1,[inside(circle,triangle),inside(square,circle),
inside(triangle,square)]).
```


ANALOGY: example

```
analogy(A is_to B,C is_to X,Answers) :-
    match(A,B,Match),
    match(C,X,Match),
    member(X,Answers).

match(inside(Figure1,Figure2),inside(Figure2,Figure1),invert).
match(above(Figure1,Figure2),above(Figure2,Figure1),invert).
```

The ANALOGY program did also the image processing to build the symbolic representation of the problem.

Question answering program

ELIZA

```
?- eliza.
|: [i, am, unhappy].
How long have you been unhappy ?
|: [six, months, can, you, help, me].
What makes you think I help you ?
|: [you, make, me, think, of, my, father, and, my, brother].
Please you tell me more about father
|: [I, like, teasing, father].
Does anyone else in your family like teasing father ?
|: [no, only, me].
Please go on .
|: [bye].
[Goodbye. I hope I have helped you]

Yes
?-
```

ELIZA-control loop

```
eliza :- read(Input), eliza(Input), !.

eliza([bye]) :-
   writeln(['Goodbye. I hope I have helped you']).
eliza(Input) :-
   pattern(Stimulus,Response),
   match(Stimulus,Table,Input),
   match(Response,Table,Output),
   reply(Output),
   read(Input1),
   !, eliza(Input1).

reply([Head|Tail]) :- write(Head), write(' '), reply(Tail).
reply([]) :- nl.
```

- instantiate table
- read table

ELIZA - question-answer schemes

```
pattern([i,am,1],['How',long,have,you,been,1,?]).
pattern([1,you,2,me],['What',makes,you,think,'I',2,you,?]).
pattern([i,like,1],['Does',anyone,else,in,your,family,like,1,?]).
pattern([i,feel,1],['Do',you,often,feel,that,way,?]).
pattern([1,X,2],['Please',you,tell,me,more,about,X]) :-
important(X).
pattern([1],['Please',go,on,'.']).

important(father).
important(mother).
important(sister).
important(brother).
important(son).
important(daughter).
```


ELIZA - answer selection

```
match([N|Pattern],Table,Target) :-
   integer(N),
   lookup1(N,Table,LeftTarget),!,
   append(LeftTarget,RightTarget,Target),
   match(Pattern,Table,RightTarget).
match([Word|Pattern],Table,[Word|Target]) :-
   atom(Word),
   match(Pattern,Table,Target).
match([],Table,[]).
```


ELIZA - dictionary

```
lookup1(X,[(X,V)|XVs],V).
lookup1(X,[(X1,V1)|XVs],V) :- X \== X1, lookup1(X,XVs,V).
```


Exercises

1. Write the PROLOG specification to solve: (i) missionaries and cannibales; husbands problem.
2. Modify the PROLOG program scheme to realize limited depth search with parameter `K`.
3. Extend the program ANALOGY to deal with other types of tests.
4. Create stimulus-response schemas for an italian version of ELIZA.

Exercise

1. Write a PROLOG program to find the safe combination, represented a sequence of digits of unknown length. The program can execute the test to open the safe by the predicate `openSafe(X)`, true if the combination X opens the safe.

   Try a *generate and test* solution, by defining:
   `generateCombination(L,X)`, generating a combination of length L,
   `generateCombinations(X,...)`, generating all combinations in lexicographic order
   `findCombination(X)`, generating all combinations and checks whether the safe is open.