Prolog – A (not so) quick introduction

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Prolog Interpreters

- **SICStus Prolog**
  - Probably, the fastest Prolog interpreter and compiler
  - Commercial tool (€€€)
  - https://sicstus.sics.se/

- **SWI-Prolog (my suggested interpreter)**
  - Well supported Prolog, quite used in AI research
  - Free
  - Available as a stand-alone, and also as a web app (nice for quick tests)
  - https://www.swi-prolog.org/
  - **https://swish.swi-prolog.org/**

- **ECLiPSe**
  - Mind the name!!! (indeed, they arrived first…)
  - Strong support to Constraint Logic Programming
  - https://eclipseclp.org/

- **TuProlog**
  - Completely java-based
  - Comes as a jar library, or with a GUI
  - Developed by our colleagues in Cesena
  - https://gitlab.com/pika-lab/tuprolog/2p

Prolog Interpreters – seriously…

- **For small exercise:**
  - TuProlog https://gitlab.com/pika-lab/tuprolog/2p
  - SWI-Prolog web app https://swish.swi-prolog.org/

- **For serious use:**
  - SWI-Prolog
  - Editor: Visual Studio Code + Prolog plugin for syntax highlighting

Few instructions…

- When the prolog interpreter is executed, it assumes a "working directory": it will look for files only in the current working directory.
    - To discover the current working dir: "`pwd.`"
    - To change working dir: "`working_directory(Old, New).`"
- Programs must be loaded, through the pre-processing of source files
    - Given a file "`test.pl`", it can be loaded through "`consult(test).`"
- When a program source is modified, it needs to be reloaded into the database clause
    - Command "`make.`" reloads all the changed files
    - It is always a good practice to close the interpreter, and run it again, from time to time…
- Debugging?
    - Command "`trace.`" … but before, read the official documentation…

Terminology

A Prolog program is a set of definite clauses of the form:

- Fact `A.`
- Rule `A :- B1, B2,…, Bn.`
- Goal `:- B1, B2,…, Bn.`

Where `A` and `Bi` atomic formulas:

- `A` head of the clause
- `B1,B2,…,Bn` body of the clause
- Symbol "`,`" is for conjunction
- Symbol "`:-`" stands for the logical implication, where `A` is the consequent, and `B1,B2,…,Bn` is the antecedent

Terminology

- An **atomic formula** has the form:

  $$p(t1,t2,…,tn)$$

  - p is a predicate symbol (alphanumeric string starting with low-capital letter)
  - t1,t2,…,tn are terms

- A **term** is recursively defined as:
  - **constants** (integer/floating numbers, alphanumeric strings starting with a low-capital letter) are terms
  - **variables** (alphanumeric strings starting with a capital letter or starting with the symbol "_") are terms.
  - $f(t1,t2,…,tk)$ is a term if "f" is a **function symbol** with k arguments and t1,t2,…,tk are terms.
    f(t1,t2,…,tk) is also known as structure. Constants can be viewed as functions with zero arguments (arity zero).

Terminology – few examples

- Constants: `a, goofey, aB, 9, 135, a92`
- Variables: `X, X1, Goofey, _goofey, _x, _`
  - variable _ is usually named as the anonymous variable
- Compound terms: `f(a), f(g(1)), f(g(1),b(a),27)`
- Atomic formulas: `p, p(a,f(X)), p(Y), q(1)`
- Definite clauses:
  - `q.`
  - `p :- q,r.`
  - `r(Z).`
  - `p(X) :- q(X, g(a)).`
- Goal:
  - `:- q, r.`
- **NOTICE: NO DISTINCTION between constants, function symbols and predicate symbols!!!**

Declarative interpretation

Variables within a clause are universally quantified

- For each fact: `p(t1,t2,…,tm).`
  If `X1,X2,…,Xn` are the variables appearing in `t1,t2,…,tm` the intended meaning is:
  `∀X1, ∀X2,…, ∀Xn (p(t1,t2,…,tm))`

- For each rule: `A :- B1,B2,…,Bk.`
  - If `Y1,Y2,…,Yo` are the variables appearing in the body only
  - If `X1,X2,…,Xn` are the variables appearing in both the body and the head

    `∀X1, ∀X2,…, ∀Xn, ∀Y1, ∀Y2,…, ∀Yo ((B1,B2,…,Bk) → A)`

    `∀X1, ∀X2,…, ∀Xn ( (∃Y1, ∃Y2,…, ∃Yn(B1,B2,…,Bk)) → A)`

Declarative interpretation – Examples

`father(X,Y)`   "**X** is the father of **Y**"

`mother(X,Y)`   "**X** is the mother of **Y**"

`grandfather(X,Y):- father(X,Z), father(Z,Y).`
  "for each **X,Y**, **X** is the granfather of **Y** *if it exists* **Z** s.t. **X** is father of **Z** and **Z** is father of **Y**"

`grandfather(X,Y):- father(X,Z), mother(Z,Y).`
  "for each **X,Y**, **X** is the granfather of **Y** *if it exists* **Z** s.t. **X** is father of **Z** and **Z** is mother of **Y**"

Execution of a (Prolog) program

- A computation is the attempt to prove, through resolution, that a formula logically follows from the program (it is a theorem).

- Moreover, it aims to determine a **substitution** for the variables in the goal, for which the goal logically follows from the program.

- Given a program `P` and the goal/query:
$$\texttt{:- p(t1,t2,...,tm).}$$

- if `X1,X2,...,Xn` are the variables appearing in `t1,t2,...,tm` then the meaning is:
$$\texttt{∃X1, ∃X2,..., ∃Xn p(t1,t2,...,tm)}$$

- The objective is to determine a substitution
$$\texttt{σ = {X1/s1,X2/s2,...,Xn/sn}}$$

- where `si` are terms such that `P ⊨ [p(t1,t2,...,tm)]σ`

SLD Resolution

- The resolution adopted by LP is the SLD that has two non-determinisms:
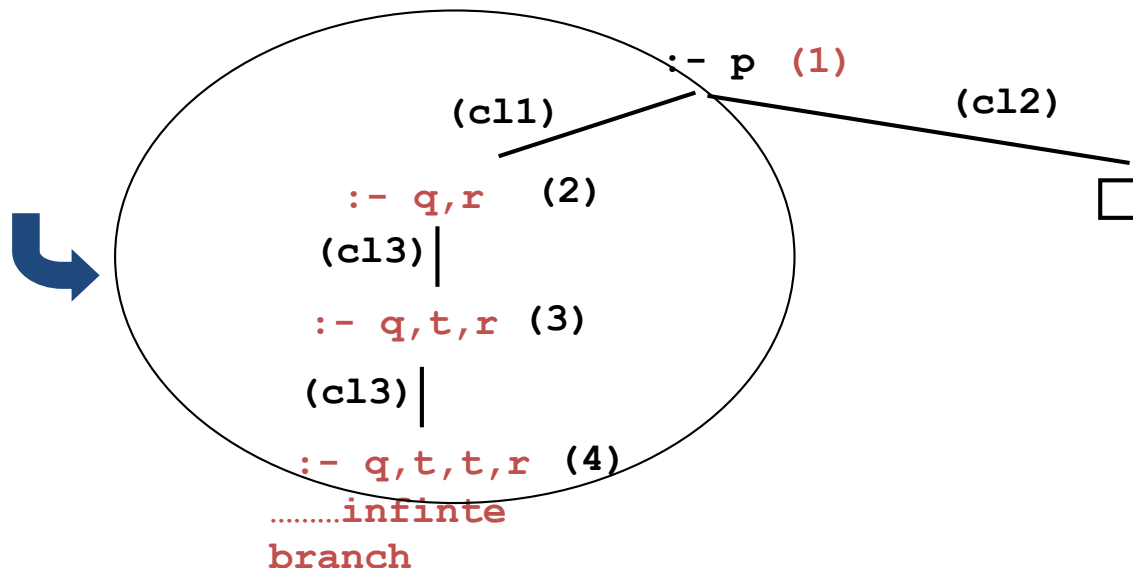   - The rule of computation
    - The search strategy

Prolog adopts SLD with the following choices:

- Rule of computation:
    - **Rule "left-most"**; given a "query":

      ?- G1, G2, ..., Gn.

      It is selected the left-most literal (G1).

- Search strategy
    - **Depth-first** with chronological backtracking

Depth-first and incompleteness in Prolog

- The order of the clauses in the program may greatly affect the termination, and consequently the completeness

P_2

```
(cl1) p :- q,r.
(cl2) p.
(cl3) q :- q,t.
      :- p.
```

```
                    :- p  (1)
        (cl1)                        (cl2)

      :- q,r    (2)

   (cl3)|                              □

    :- q,t,r  (3)

  (cl3)|

  :- q,t,t,r  (4)
.........infinte
branch
```

Order of the clauses in Prolog

- Clauses order in Prolog is highly relevant

    P2

    (cl1) `p :- q,r.`

    (cl2) `p.`

    (cl3) `q :- q,t.`

    P3

    (cl1') `p.`

    (cl2') `p :- q,r.`

    (cl3') `q :- q,t.`

- Programs P2 and P3 are NOT EQUIVALENT
- Given the query "query": `:-p.`   you get:
    - The proof with P2 does not terminate;
    - The proof with P3 successfully terminates (immediately).

- However, a depth first strategy can be efficiently implemented using techniques similar to those adopted in procedural programming (**TAIL RECURSION**).

Multiple solutions, and the disjunction

- There may exist multiple answers for a query.

- How to get them? After we get an answer, we could force a failure: backtracking should be started then, looking for the "next" solution.

- Practically, it means to ask the procedure to explore the remaining part of the SLD tree.

- In Prolog standard we can ask for more solutions through the operator "; ":

```
    :-   sister(maria,W).
yes W=giovanni;
W=anna;
no
```

- Operator ";" can be interpreted as:
  - A disjunction operator, that looks for alternative solutions
  - Within a Prolog program, to express the disjunction.

Procedural Interpretation of Prolog

- A procedure is a set of clauses of a program P whose:
  - Heads have the same predicate symbol
  - Predicate symbols have the same arity (same number of arguments)

- Arguments appearing in the head of the procedure are the formal parameters

$$:- \ \texttt{p(t1,t2,...,tn)}.$$

  This can be viewed as the call to procedure p. Arguments of p (terms t1,t2,....,tn) are the actual parameters.

- Unification then can be viewed as the mechanism for passing the parameters.

- There is no distinction between input parameters and output parameters (reversibility)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Procedural Interpretation of Prolog

- The body of the clause then can be viewed as a sequence of calls to procedures.

- Two clauses with the same head are two alternative definitions of the body of a procedure.

- All the variables are "**single assignment**": they assume a single value along the proof, except when looking for alternative paths in the SLD tree ("backtracking").

Example

```
play_sport(mario,football).
play_sport(giovanni,football).
play_sport(alberto,football).
play_sport(marco,basket).
live(mario,torino).
live(giovanni,genova).
live(alberto,genova).
live(marco,torino).


:- play_sport(X,football). %"does exists X such that X plays
football?"
yes        X=mario;
X=giovanni;
X=alberto;
No


:- play_sport(giovanni,Y).
yes        Y=football;
no
```

Example

```
:- play_sport(X,Y).
```
% "Do exist X and Y s.t. X plays the sport Y?"

```
yes     X=mario          Y=football;
        X=giovanni       Y=football;
        X=alberto        Y= football;
        X=marco          Y=basket;
        no
```


```
:- play_sport(X,football), live(X,genova).
```
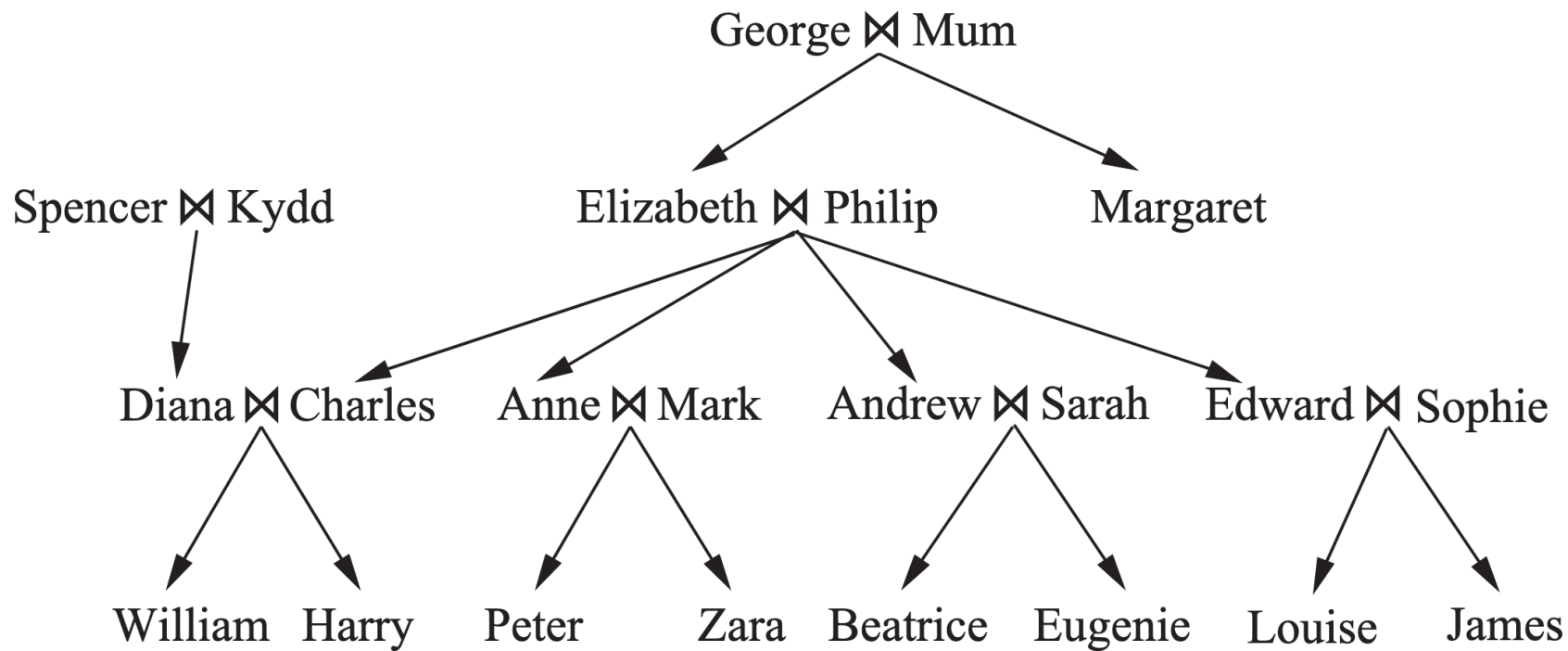% "Does exist X s.t. X plays football and X lives in Genova?"

```
        yes     X=giovanni;
                X=alberto;
        no
```

Exercise – the kinship domain

(Sect, 8.3.2 and Ex. 8.kins from AIMA)

Represent the family relations as function terms and predicates.

A typical family tree. The symbol ⋈ connects spouses and arrows point to children.

Exercise – the kinship domain

"Concepts" we would like to represent:

- Male
- Female

- Parent
- Father
- Mother

- Sibling
- Brother
- Sister

- Child
- Daughter
- Son

- Spouse
- Wife
- Husband

Exercise – the kinship domain

"Concepts" we would like to represent:

- Granparent
- Grandchild

- Cousin
- Aunt
- Uncle

- Greatgrandparent
- Ancestor
- FirstCousin
- BrotherInLaw
- SisterInLaw

Arithmetic and math in Prolog

Arithmetic

- In Logic there is not any way to evaluate functions. That might be a problem for us...

- Integer numbers can be represented through the Peano notation: $s(s(s(...s(0)...)))$

```
prod(X, 0, 0).
prod(X, s(Y), Z):- prod(X, Y, W), sum(X, W, Z).
```

- Non practicable...

Arithmetic

- In Prolog, both integers and floating point numbers are atoms.

- Math operators are function symbols predefined in the language.

- Every expression is then a Prolog term.

- Binary operators are supported with a pre-fix notation, as well as the in-fix notation.

```
Unary operators         -, exp, log, ln, sin, cos, tg
Binary operators        +, -, *, \, div, mod
```

- **+(2,3)** and **2+3** are equivalent.

Evaluation of expressions

However, when we write an expression, we would like to have it evaluated...

- Special pre-defined predicat `is`.

$$T \text{ is } Expr \qquad (\text{is}(T,Expr))$$

  - `T` can be a numerical atom or a variable
  - `Expr` must be an expression

- Expression `Expr` is evaluated and the result is unified with `T`

Variables in `Expr` **MUST BE COMPLETELY INSTANTIATED** at the moment of evaluation.

Evaluation of expressions – examples

```
:-    X is 2+3.
yes X=5


:-    X1 is 2+3, X2 is exp(X1), X is X1*X2.
yes X1=5    X2=148.413    X=742.065


:-    0 is 3-3.
yes


: -    X is Y-1.
No
(or Instantion Fault, depending on the prolog system)


:-    X is 2+3, X is 4+5.
no
```

Evaluation of expressions – examples

```
:- X is 2+3, X is 4+1.
```

yes X=5

In this example the second goal is:

```
:- 5 is 4+1.
```

X has been instantiated by the evaluation of the first goal.

```
:-    X is 2+3, X is X+1.
```

No

No way: there is not the assignment like in procedural languages.
Variables are write-once….

Evaluation of expressions – examples

With the operator **is**, the order of the goals is very important:

```
(a)      :-   X is 2+3, Y is X+1.
(b)      :-   Y is X+1, X is 2+3.
```

– Goal **(a)** succeeds and returns

    X=5, Y=6

– Goal **(b)** fails.

- The predefined predicate "is" is not reversible. Procedures that use it are not (generally speaking) reversible.

Expressions and Terms

A term representing an expression is evaluated only if it is the second argument of a predicate "is"

```
p(a,2+3*5).
q(X,Y)  :-  p(a,Y), X is Y.


:-  q(X,Y).
yes  X=17 Y=2+3*5    (Y=+(2,*(3,5)))
```

Notice: Y is not evaluated, but unified with a structure that has "+" as operator, and arguments "2" and another structure with * as operator and arguments 3 and 5.

Relational operators

- It is possible to compare expression values
- Such operators can be used as goals within clauses, and have in-fix notation

```
Relational operators:
>, <, >=, =<, ==, =/=
```

Comparing expressions

`Expr1 REL Expr2`

- where **`REL`** is a relational operator, and **`Expr1`** and **`Expr2`** are expressions

- **`Expr1`** and **`Expr2`** are evaluated: Mind it! They both have to be completely instantiated.
- The results are then compared on the basis of **`REL`**

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Math functions...

- Given the math operators and "is", it is possible to define functions in Prolog
- Given f with arity n:
  - it can be implemented through a (n+1)-arity predicate
    - $f:x_1, x_2, \ldots, x_n \rightarrow y$ is written as

    ```
    f(X1,X2, ...,Xn,Y)    :- <compute Y>
    ```

  - Example: compute the factorial

    ```
    fatt:  n → n !                 (n positive integer)
    fatt(0) = 1
    fatt(n) = n * fatt(n-1)    (for n>0)

    fatt(0,1).
    fatt(N,Y):- N>0, N1 is N-1, fatt(N1,Y1), Y is N*Y1.
    ```

Iteration and recursion

Iteration and recursion in Prolog

- In Prolog there is no iteration (no while, for, repeat…)
- **But… you can get iterative behaviour through recursion!!!**

- A function f is tail-recursive if f is the "most external call" in the recursive definition
- In other terms, if the result of the recursive call is not subject of any other call.
- Tail-recursion is equivalent to iteration
  - It can be evaluated in constant space, like in iteration.
  - However, in Prolog things get complicated: non determinism, choice points…

Non-tail recursion, and conversion to tail recursion

- There are many cases where a non-tail recursion can be re-written as a tail recursion

```
fatt1(N, Y):-  fatt1(N,1,1,Y).
fatt1(N, M, ACC, ACC) :-   M > N.
fatt1(N, M, ACCin, ACCout) :- ACCtemp is ACCin*M,
                                M1 is M+1,
                                fatt1(N,M1,ACCtemp,Accout).
```

Input accumulator

Output accumulator

Non-tail recursion, and conversion to tail recursion

- The factiorial is computed using an accumulator, initialized to 1, and incremented at every step, and unified only at the termination of the recursion.

```
ACC₀=1
ACC₁= 1 * ACC₀ = 1 * 1
ACC₂= 2 * ACC₁ = 2 * (1*1)
...
ACC_{N-1}= (N-1) *ACC_{N-2} = N-1*(N-2*(...*(2*(1* 1)) ...))
ACC_N = N * ACC_{N-1} = N*(N-1*(N-2*(...*(2*(1* 1)) ...)))
```

Non-tail recursion, and conversion to tail recursion

Another solution to the tail-recursive version of the factorial:

```
fatt2(N,Y)
        "Y is the factorial of N"


fatt2(N,Y)   :-  fatt2(N,1,Y).

fatt2(0,ACC,ACC).

fatt2(M,ACC,Y)   :- ACC1 is M*ACC,
                      M1 is M-1,
                      fatt2(M1,ACC1,Y).
```

Few exercises

Excercise

Define a Prolog program that receives in input a number N, and print all the numbers between 1 and N.

Solution:

```prolog
specialPrint(1) :-
   write(1), nl.
specialPrint(N) :-
    N>1,
    write(N), nl,
    TheNext is N-1,
    specialPrint(TheNext).
```

Excercise

Define a Prolog program that receives in input a number N, and print all the numbers between 1 and N, from the smallest to the greatest.

Solution:

```prolog
specialPrint(1) :-
   write(1), nl.
specialPrint(N) :-
    N>1,
    TheNext is N-1,
    specialPrint(TheNext),
    write(N), nl.
```

Excercise

Define a Prolog program that computes the Fibonacci number.

Definition of Fibonacci:

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

Solution:

```prolog
% fib(Num, Result)
% Num is the input, Result is the fibonacci number of Num
fib(0,0).
fib(1,1).
fib(N, Result) :-
  N>1, % test to avoid infinite recursion/loops
  N1 is N-1, fib(N1,Part1),
  N2 is N-2, fib(N2,Part2),
  Result is Part1 + Part2.
```

Excercise

Write a predicate about a number N, that is true if N is prime

Solution:

```
isPrime(2).
isPrime(N) :-
    N>2,
    TheDiv is N-1,
    cannotBeDividedBy(N, TheDiv).


cannotBeDividedBy(N, 2) :-
    1 is N mod 2.
cannotBeDividedBy(N, TheDiv) :-
    TheDiv>2,
    Rest is N mod TheDiv,
    Rest > 0,
    TheNextDiv is TheDiv-1,
    cannotBeDividedBy(N, TheNextDiv).
```

Excercise

Write a predicate that, given a number N, prints out all the prime numbers between 2 and N.

Solution:

```prolog
printOnlyPrimes(2) :-
    printOnlyIfItIsPrime(2).
printOnlyPrimes(N) :-
    N>2,
    printOnlyIfItIsPrime(N),
    TheNext is N-1,
    printOnlyPrimes(TheNext).


printOnlyIfItIsPrime(N) :-
    isPrime(N),
    write('The number '), write(N), write(' is prime!'), nl.
printOnlyIfItIsPrime(N) :-
    \+isPrime(N),
    write('The number '), write(N), write(' is NOT prime!'), nl.
```

Lists

Lists

- One of the most used among the primitive data structures; languages exists that are mainly based on lists (e.g., LISP)

- In Prolog, lists are terms build upon the special atom "empty list, `[ ]`" and the constructor operator "."

Recursive definition:

- the atom `[ ]` represents the empty list

- the term `.(T, List)` is a list if `T` is any term, and `List` is a list. `T` is named also "head of the list", while `List` is named "tail of the list"

Lists – Examples

The following are lists:

- `[ ]`

- `.(a, [])`

- `.(a, .(b, [ ]))`

- `.(f(g(X)), .(h(z), .(c, [ ])))`

- `.([ ], [ ])`

- `.( .(a,[]),    .(b, []))`

- The notation with the binary functor "." might be difficult...

- ... Prolog provides a simpler notation: `.(T, List)` can be represented also as `[T | List]`

Lists – Examples

Lists again, but with the simplified notation:
- `[  ]`
- `[a  |  []  ]`
- `[a  |  [b  |  []]]`
- `[f(g(X))  |  [h(z)  |  [c  |  []  ]]]`
- `[[  ]  |  [  ]]`
- `[  [a  |[]]  |    [b  |  []]]`

- Yet, the recursive notation is still "difficult"
- Lists like, e.g. `[a  |  [b  |  [c|[]]]]` can be written as `[a, b, c]`

Lists – Examples

Lists again, but with the simplified notation:

- `[ ]`
- `[a]`
- `[a, b]`
- `[f(g(X)), h(z), c]`
- `[ [] ]`
- `[ [a], b]`

Unification and Lists

The unification algorithm is extended to the lists, and provide a powerful method for accessing their content:

```
p([1, 2, 3, 4, 5, 6, 7, 8, 9]).


:- p(X).
yes X=[1,2,3,4,5,6,7,8,9]


:-  p([X|Y]).
yes    X=1    Y=[2,3,4,5,6,7,8,9]


:-  p([X,Y|Z]).
yes    X=1 Y=2 Z=[3,4,5,6,7,8,9]


:-  p([_|X]).
yes    X=[2,3,4,5,6,7,8,9]
```

Operations – isList/1

Define a predicate **isList/1** that is true only if the argument is a list.

Starting from the recursive definition, T is a list if:

- T is an empty list, or
- T is a non empty list, and its tail is a list

```
isList([]).
isList([ X | Tail]) :- isList(Tail).


isList([]).
isList([ _ | Tail]) :- isList(Tail).


:- isList([1,2,3]).
yes


:- isList([a|b]).
no
```

Operations – member/2

Define a predicate **member/2** that is true if the first argument is an element of the list passed as a second argument.

```
member( El, [El|_]).
member(El, [_ | Tail]) :- member(El, Tail).



:-  member(2, [1,2,3]).
yes


:-  member(1, [2,3]).
no


:-  member(X, [1,2,3]).
yes      X=1;
         X=2;
         X=3;
no
```

Operations – member/2

The proposed definition of member can be used also to generate lists:

```
member( El, [El|_]).
member(El, [_ | Tail]) :- member(El, Tail).
```

```
:-  member(42, X).
yes,     X = [42|_1648];
         X = [_1364, 42|_1372];
         X = [_1364, _1370, 42|_1378];
         X = [_1364, _1370, _1376, 42|_1384];
         X = [_1364, _1370, _1376, _1382, 42|_1390];
         X = [_1364, _1370, _1376, _1382, _1388, 42|_1396];
          ...
```

Operations – length/2

Define a predicate length/2 that takes as first argument a list, and the second argument is the number of elements contained in the list.

```
length([], 0).
length([_ | Tail], N) :-
        length(Tail, NT),
        N is NT+1.


:- length( [9,8,7], 3).
yes


:- length([9,8,7], Result).
yes      Result = 3


:- length(AList, 3).
yes      AList = [_, _, _]
```

Operations – append/3

Define a predicate append/3 that takes as first and second arguments two lists, and the third argument is the list obtained by concatenating the two lists.

```
append([], L1, L1).
append([H |Rest1], L2, [H | NewTail]) :-
        append(Rest1, L2, NewTail).


:- append([1,2],[3,4,5],L).
yes  L = [1,2,3,4,5]


:- append([1,2],L2,[1,2,4,5]).
yes  L2 = [4,5]


:- append([1,3],[2,4],[1,2,3,4]).
no
```

Operations – deleteFirstOccurence/3

Define a predicate deleteFirstOccurence/3 that takes as first and second argument an element and alist respectively, and the third argument is the list without the first occurrence of the element (without the term that unifies with the element).

```
deleteFirstOccurence(El, [], []).
deleteFirstOccurence(El, [El|T], T).
deleteFirstOccurence(El, [H|T], [H|T1]) :-  deleteFirstOccurence(El, T, T1).

% This solution is not correct when backtracking… why?
```

Operations – deleteAllOccurences/3

Define a predicate deleteAllOccurences/3 that takes as first and second argument an element and a list respectively, and the third argument is the list without all the terms that unify with the element.

```
deleteAllOccurences(El, [], []).
deleteAllOccurences(El, [El|T], Result):- deleteAllOccurrences(El, T, Result).
deleteAllOccurences(El, [H|T], [H|T1]) :- deleteAllOccurences(El, T, T1).

% This solution is not correct when backtracking… why?
```

Operations – reverse/2

Define a predicate reverse/2 that takes as first and second argument two lists, where one is the reversed of the second.

```
reverse([],[]).
reverse([H|T], Result) :-
        reverse(T, Partial),
        append(Partial, [H], Result).


:- reverse([],[]).
yes


:- reverse([1,2], Lr).
yes Lr = [2,1]


:- reverse(L, [2,1]).
yes L = [1,2]
```

Lists in Prolog – some exercises

1. Write a predicate that given a list, it returns the last element.

2. Write a predicate that given two lists L1 and L2, returns true if and only if L1 is a sub-list of L2.

3. Write a predicate that returns true if and only if a list is a palindrome.

4. Write a predicate that, given a list (possibly with repeated elements), returns a new list with repeated elements.

5. Write a predicate that given a term T and a list L, counts the number of occurrences of T in L.

6. Write a predicate that, given a list, returns a new list obtained by flattening the first list. Example: given the list [1,[2,3,[4]],5,[6]] the predicate should return the list [1,2,3,4,5,6].

7. Write a predicate that given a list, returns a new list that is the first one, but ordered.

The CUT

Controlling a (Prolog) program

- There are a number of pre-defined predicates that allows to interfere and control the execution process of a goal.
- The cut "!" is among them.
- No logic meaning, no declarative semantics…
- … but it heavily affects the execution process

Controlling a (Prolog) program – some insight on the execution process

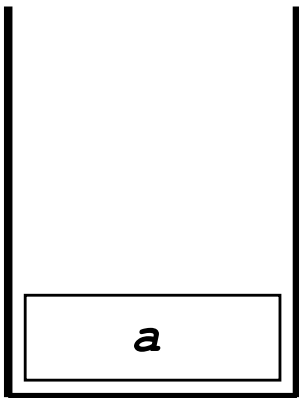The execution process is build upon (at least) two stacks:

- **Execution stack**: it contains the activation records of the procedures/predicates

- **Backtracking stack**: it contains the set of open choice points.

Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
```
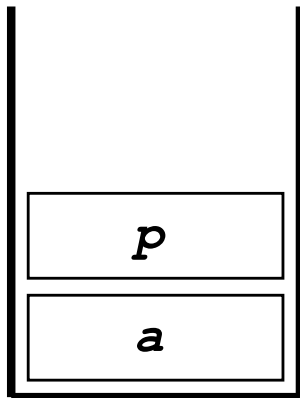
– Query `:-a.`

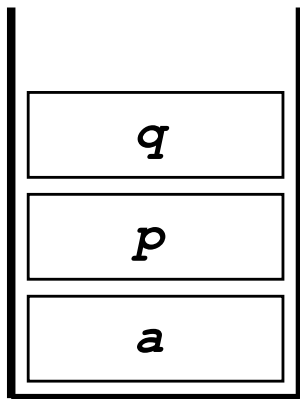*Execution stack*   *Backtracking stack*
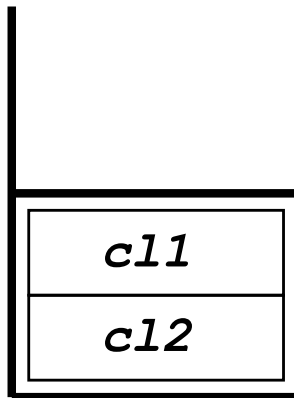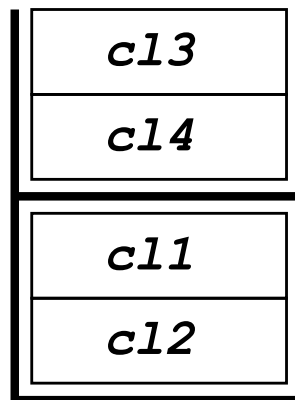
Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
```


Execution stack

Backtracking stack

Execution stack contains: p, a

Backtracking stack: c13, c14 (Current choice), c11, c12 (Current choice)

Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
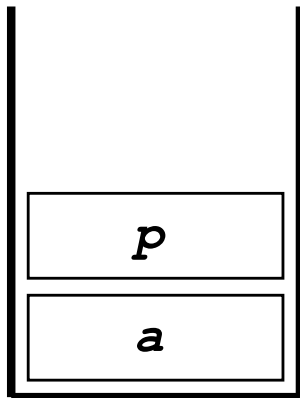```
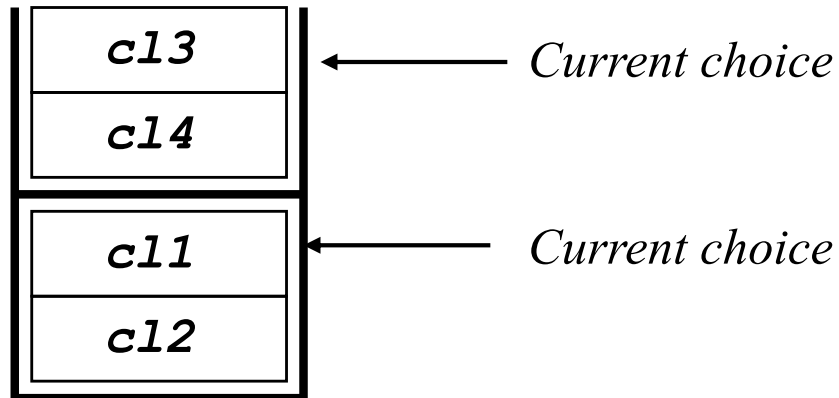


Execution Stack

Backtracking stack

Current choice

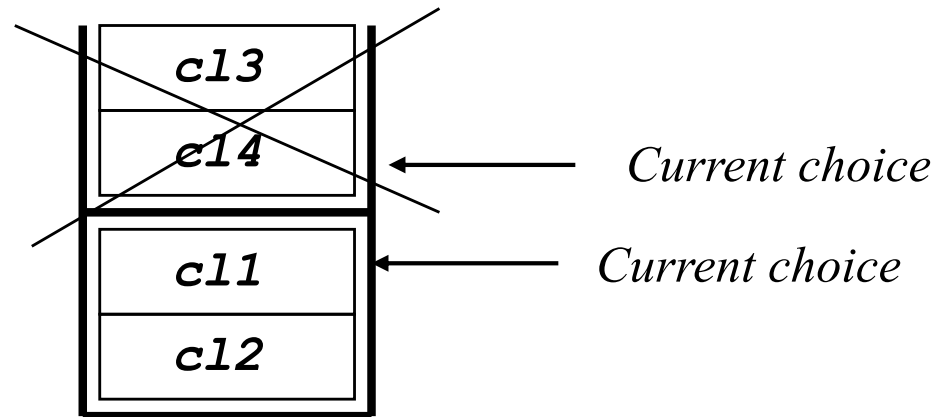Current choice

Failure

# Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
```


Execution stack   *Backtracking stack*
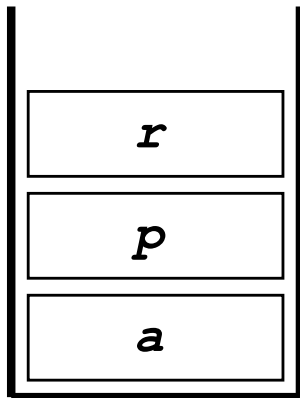
Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
```

```
        ┌───────┐              ┌───────┐
        │   r   │              │       │
        ├───────┤              ├───────┤
        │   p   │              │  cl1  │ ← Current choice
        ├───────┤              ├───────┤
        │   a   │              │  cl2  │
        └───────┘              └───────┘
```

Execution stack   *Backtracking stack*
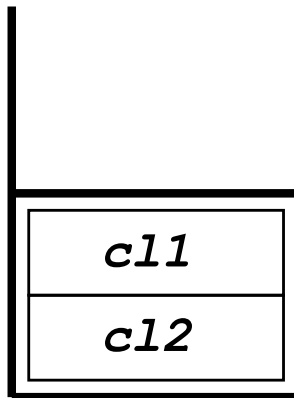
r succeed!

Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
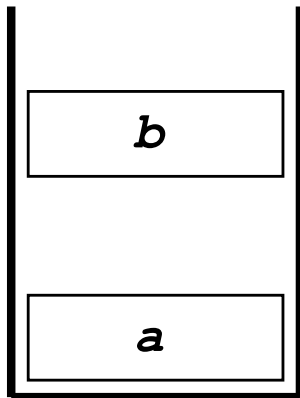```



*Execution stack*          *Backtracking stack*

*Current choice*

**b** fails

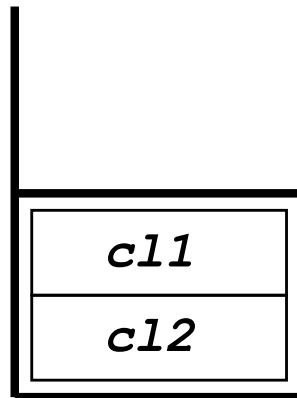# Controlling a (Prolog) program – example

```
(cl1)        a :- p,b.
(cl2)        a :- r.
(cl3)        p :- q.
(cl4)        p :- r.
(cl5)        r.
```

Execution stack: `a`

Backtracking stack: `cl1`, `cl2`

Current choice

Execution stack

Backtracking stack

Controlling a (Prolog) program – example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
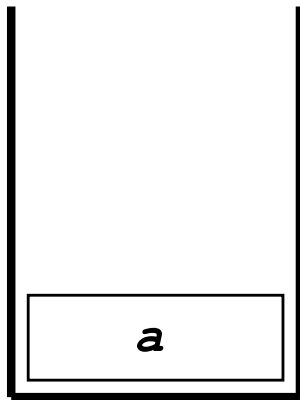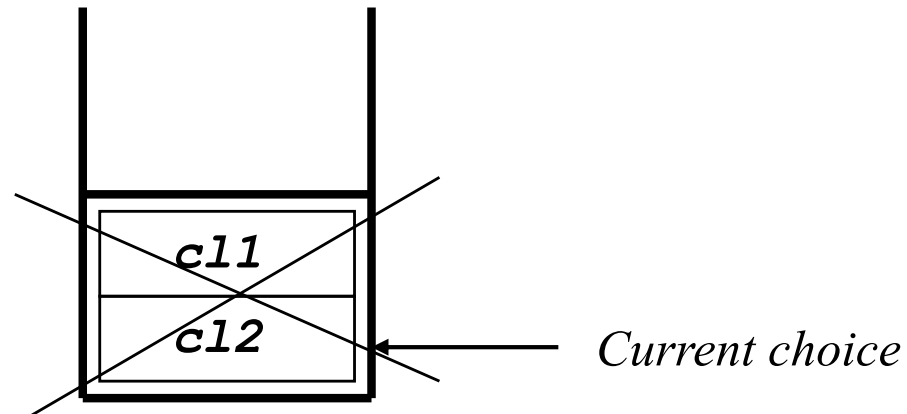```
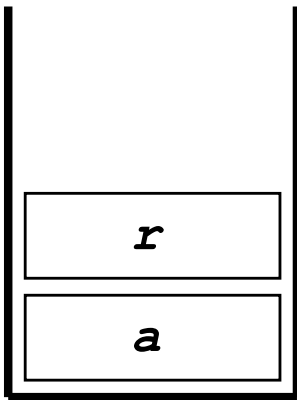


*Execution stack*

*Backtracking stack*

**Success**

## How CUT works

- The evaluation of the CUT is to make some choices as definitive and non-backtrackable

- In other words, some block are removed from the backtracking stack.

- The CUT alters the control of the program....

- When used, declarativeness is partially lost

## How CUT works

Consider the clause:

$$p \; :- \; q_1, \; q_2, \ldots, \; q_i, \; !, \; q_{i+1}, \; q_{i+2}, \ldots, \; q_n.$$

- The evaluation of ! succeeds, and it is ignored in backtracking;
- All the choices made in the evaluation of goals $q_1$, $q_2$, . . . , $q_i$ and goal $p$ are made definitive; in other words, the choice points are removed form the backtracking stack.
- Alternative choices related to goals placed after the cut are not touched/modified.

## How CUT works

Consider the clause:

$$p \text{ :- } q_1, q_2, ..., q_i, \text{ ! }, q_{i+1}, q_{i+2}, ..., q_n.$$

- If the evaluation of $q_{i+1}, q_{i+2}, ..., q_n$ fails, then the "whole" $p$ fails. Even if there were other alternatives for $p$, these would have been removed by the cut.

- The cut removes branches of the SLD tree, hence it cannot be defined in a declarative way.

# CUT – example

```
a(X,Y):- b(X), !, c(Y).
a(0,0).
b(1).
b(2).
c(1).
c(2).


:-  a(X,Y).
yes  X=1    Y=1;
     X=1    Y=2;
no
```

CUT – example

```
p(X):- q(X),r(X).
q(1).
q(2).
r(2).


:- p(X).
yes   X=2
```

```
p(X):- q(X),!,r(X).
q(1).

q(2).

r(2).


:- p(X).
no
```

The CUT to achieve mutual exclusion between two clauses

Suppose you have a condition a(X), used to choose between two different program paths:

```
if a(.) then b else c
```

Using the "cut":

```
p(X) :- a(X), !, b.
p(X) :- c.
```

- *If `a(X)` is true, then the cut is evaluated and the choice point for `p(X)` is removed.*
- *If `a(X)` fails, backtracking is started before the cut.*

The CUT to achieve mutual exclusion between two clauses – example

Write a predicate that receives a list of integers, and returns a new list containing only positive numbers.

```
filter([], []).
filter([H|T], [H|Rest]) :- H>0,  filter(T, Rest).
filter([H|T], Rest)      :- H=<0, filter(T, Rest).
```

```
filter([], []).
filter([H|T], [H|Rest]) :- H>0, !, filter(T, Rest).
filter([_|T], Rest)      :- filter(T, Rest).
```

The Negation

The problem of the negation

- Prolog allows only definite clauses, no negative literals
- Moreover SLD does not allow to derive negative information

```
person(mary).
person(john).
person(anna).
dog(fuffy).
```

- Intuitively, fuffy is not a person, since it cannot be proved by using the facts in our KB.

The Close World Assumption

- This intuition is usually granted in the database field, where only positive information are recorded

- In Logic, this intuition is formalized as Closed World Assumption (CWA) [Reiter '78].

    If a ground atom A is not logical consequence of a program P, then you can infer ~A

CWA(P) = {~A | it does not exists a refutation SLD for P∪{A} }

Close World Assumption – example

```
capital(rome).
city(X) :- capital(X).
city(bologna).
```

- Using CWA, we can infer `~capital(bologna).`

- CWA is a non-monotonic inference rule: adding new axioms to the program might change the set of theorems that previously held.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Close World Assumption – are we happy?

- Due to FOL undecidability, there is no algorithm that establishes in a finite time if A is not logical consequence of P

- Operationally, it happens that if A is not logical consequence of P, SLD resolution is not guaranteed to terminate.
  ```
  city(rome):- city(rome).
  city(bologna).
  ```

- SLD cannot prove, in finite time, that city(rome) is not logical consequence

- Consequence: use of CWA must be restricted to those atoms whose proof terminates in finite time, i.e. those atoms for which SLD does not diverge.

Close World Assumption… and Negation as Failure

- Let's drop the CWA: Prolog adopts a less powerful rule, the Negation as Failure

- Negation as Failure [Clark 78], derives only the negation of atoms whose proof terminates with failure in a finite time.

- Given a program P, we name FF(P) the set of atoms for which the proof fails in a finite time

- NF rule:

    NF(P)={ ~A | A ∈ FF(P)}

Negation as Failure

- If an atom A belongs to FF(P), then A is not logical consequence of P
- Not all the atoms that are not logical consequence of P belong to FF(P)

```
city(rome):- city(rome).
city(bologna).
```

- city(rome) is not logical consequence of P… but it does not belong to FF(P)

SLDNF

- To solve goals containing also negative atoms, SLDNF has been proposed [Clark 78]. It extend SLD resolution with Negation as Failure (NF).

- Let `:-L`$_1$`,...,L`$_m$ be the goal, where `L`$_1$`,...,L`$_m$ are literals (atoms or negation of atoms).
- A SLDNF step is defined as:
  - Do not select any negative literal `L`$_i$, if it is not "ground";
  - If the selected literal `L`$_i$ is positive, then apply a normal SLD step
  - If `L`$_i$ is ~A (with A "ground") and A fails in finite time (it has a finite failure SLD tree), then `L`$_i$ succeeds, and the new resolvent is:

$$:-\ \texttt{L}_1,\dots,\texttt{L}_{i-1},\texttt{L}_{i+1},\dots,\texttt{L}_m$$

SLDNF

- Proving with success a negative literal does not introduce any substitution, since the literal is ground
- No substitution is applied to the new resolvent

$$:- \; \texttt{L}_1, \ldots, \texttt{L}_{i-1}, \texttt{L}_{i+1}, \ldots, \texttt{L}_m$$

- A selection rule is **safe** if it selects a negative literal only when it is ground
- The selection of **ground** negative literals only is needed to ensure correctness and completeness of SLDNF

SLDNF and Prolog

- SLDNF is used in Prolog to implement the Negation as Failure

- To prove ~A, where A is an atom, the Prolog interpreter try to prove A
  - If the proof for A succeed, then the proof of ~A fails
  - If the proof for A fails in a finite time, then ~A is proved successfully

NAF in Prolog – example

```
capital(rome).
chief_town(bologna)
city(X):- capital(X).
city(X):- chief_town(X).

:- city(X), ~capital(X).
```

```
                    :- city(X), ~capital(X).


:- capital(X),                        :- chief_town(X),
   ~capital(X).                          ~capital(X).
        |   X/rome                           |   X/bologna
:- ~capital(rome).                    :- ~capital(bologna).

:- capital(rome).                     :- capital(bologna).
        |                                     |
        □                                    fail

      fail                                 success
```

NAF in Prolog – Are we happy now?

- Prolog does not use a safe selection rule: it selects ALWAYS the left-most literal, without checking if it is ground

- Indeed, it is a non-correct implementation of SLDNF

- Mmmmh… okay but… what happens if a non-ground negative literal is selected?

SLDNF in Prolog – example

```
capital(rome).
chief_town(bologna)
city(X):- capital(X).
city(X):- chief_town(X).

:- ~capital(X), city(X).
```

```
:- ~capital(X), city(X).
            |
   ┌─────────────────────┐
   │   :- capital(X)      │
   │          │ X/roma    │
   │          □           │
   └─────────────────────┘
         fail
```

INCORRECT!!!

SLDNF and Quantification of variables

- The problem lies in the meaning of the quantifiers of variables appearing in negative literals:

```
capital(rome).
chief_town(bologna)
city(X):- capital(X).
city(X):- chief_town(X).
```

```
:- ~capital(X).
```

The intended meaning is: "does exist an X that is not capital?"

$$F = \exists \; X \; \texttt{~capital(X).}$$

Answer: it exists an entity (bologna) that is not a capital.

SLDNF and Quantification of variables

- Instead, in SLDNF, we are looking a proof for

    `:- capital(X).`

    with the explicit quantifier:

    F = ∃`X capital(X).`

- Then, the result is negated:

    F = ~ `(`∃`X capital(X)).`

    syntactic transformation:

    F = ∀ `X (~ capitale(X))`

- Summing up, if there is **x** that is a capital, the proof for F fails.