INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Computer Science Department

Ch 7 – Synthesis

Mireille Ducassé

Last revision April 2024

Why Prolog ?

- A new programming philosophy
- Language relevant for
  - Knowledge management
  - Artificial intelligence (reasoning, planning, expert systems, games, etc.)
  - Automatic language processing
  - E-learning
  - Bioinformatics
  - Optimization, decision support
- Used in industry, in particular for its constraint programming aspect

Specificity of Prolog

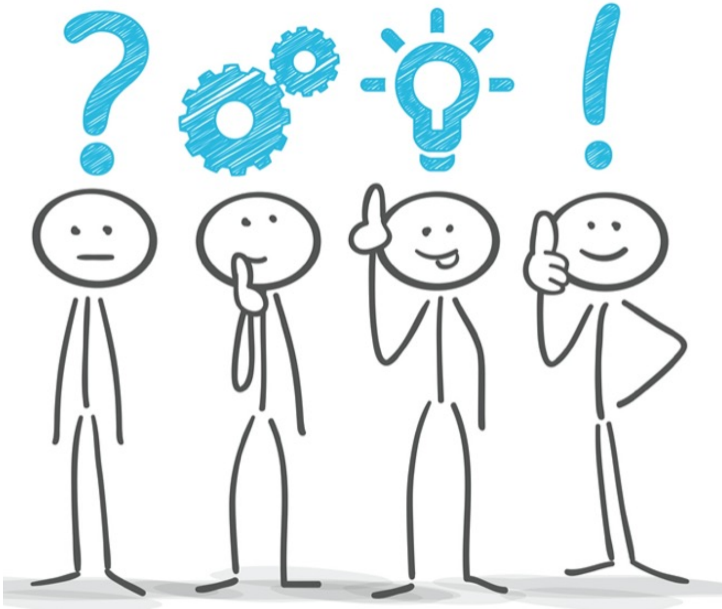
- Logic =>
  - You specify **what** is true
  - You let the interpreter prove queries and build solutions for you
    - it handles **how** to do it

➢ Much less low-level aspects to care about

Exercise 7.1: code reading

- It is crucial to “read” code as logical assertions
- Paraphrase in English the following code (make sure to “translate” everything)
- How could you test it ?
- When and how was it used in exercises ?

```
member(X, [X | _]).
member(X, [_ | T]):-
    member(X, T).
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 7.1: code reading (bis)

- Paraphrase in English the following code (make sure to "translate" everything)

```
member(X, [X | _]).
member(X, [_ | T]):-
    member(X, T).
```

It is true that an element X is a member of a list L
if X is the first element of L
or
if X is a member of the tail of L.

- How could you test it ?

```
?- member(a, [c, b, a]).    -> Yes
?- member(X, [c, b, a]).    -> Yes X=c ; X=b ; X=a
?- member(d, [c, b, a]).    -> No
```

- When and how was it used in exercises ?
  - extensively in the Zebra code

Summary of this course

(MAIN) KEY FEATURES OF PROLOG

(Main) Key features of Prolog

- Unification
- Recursion
- Lists
- Search tree
- Extra-logical predicates
- Compiler and interpreter

Exercise 7.2: Unification

- Unification is the key stone of Prolog interpreters
- Answer the following queries

?- hello = 3.

?- A=3.

?- A=Y.

?- p(a,b) = p(A,B,C).

?-p(p(a), p(p(a))) = p(X, Y).

?- p(p(a), Y) = p(X, p(p(a))).

?- p(A) = A.

?- [3, Y] = [A, foo].

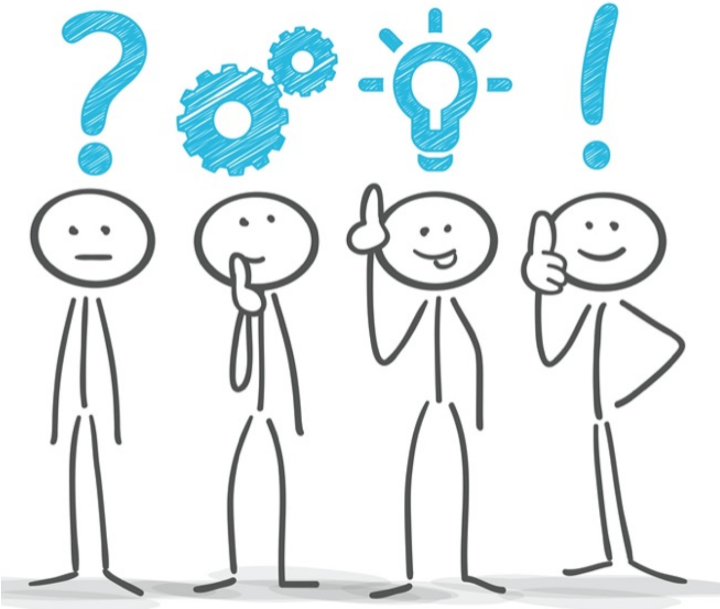
?- [3 | Y] = [A, foo].

?- [3, a, hello | Y] = [A | Foo].

?- X = 3*7.

?- X is 3*7.

?- 21 is 3*X.



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 7.2: Unification (bis)

?- hello = 3.
 No
?- A=3.
 A=3, Yes
?- A=Y.
 A=Y, Yes
?- p(a,b) = p(A,B,C).
 No
?-p(p(a), p(p(a))) = p(X, Y).
 X=p(a), Y=p(p(a)), Yes
?- p(p(a), Y) = p(X, p(p(a))).
 X=p(a), Y=p(p(a)), Yes
?- p(A) = A.
 Error

?- [3, Y] = [A, foo].
 Y=foo, A=3, Yes
?- [3 | Y] = [A, foo].
 Y=[foo], A=3, Yes
?- [3, a, hello | Y] = [A | Foo].
 Foo=[a, hello | Y], A=3, Yes
?- [3 | Y] = [A | foo].
 No
?- X = 3*7.
 X = 3*7
?- X is 3*7.
 X = 21
?- 21 is 3*X.
 Error

Recursion and Lists

- Recursion replaces iteration of imperative programming
- Much safer to program with
  - … once well understood ☺
- Lists are the main data structures of Prolog
  - Remember [Head | Tail]
  - In case of doubts check chapter 3

Design pattern: list processing

Pattern 1: Computing a result list

```
do_list([], <base result>).
do_list([Head | Tail], [HRes |TRes]) :-
    do_one(Head , HRes),
    do_list(Tail, TRes).
```

Equivalent to

```
do_list([], <base result>).
do_list(Arg1, Arg2) :-
    Arg1= [Head | Tail],
    Arg2= [HRes |TRes]
    do_one(Head , HRes),
    do_list(Tail, TRes).
```

End result is concatenated **at the end of the recursions**

Design pattern: list processing and counting

```
do_list([], <base result>, 0).
do_list([Head | Tail], [Head_Res |Tail_Res], N) :-
        do_one(Head , Head_Res, N1),
        do_list(Tail, Tail_Res, Nt),
        N is N1+Nt.
```

Remember that is/2 must be called only when the right-hand side variables have become ground

Design pattern: directed graph traversal with intermediate results collected

```
path(A, B, []) :-
    edge(A, B).
path(A, B, [C |Path]) :-
    edge(A, C),
    path(C, B, Path).
```

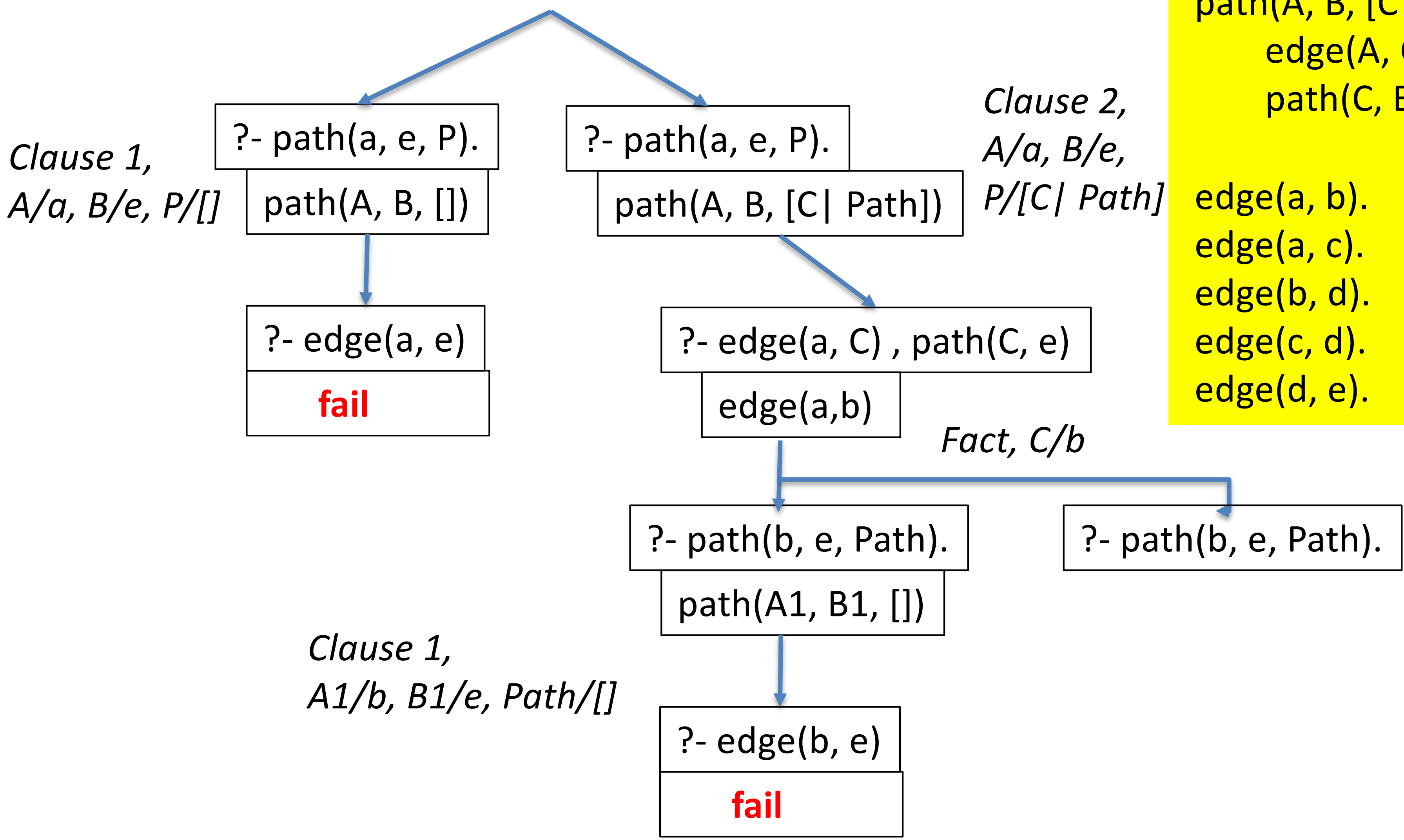
Equivalent to

```
path(A, B, Path) :-
    Path = [],
    edge(A, B).
path(A, B, Path0) :-
    edge(A, C),
    path(C, B, Path),
    Path0 = [C |Path].
```

Search tree: ?- path(a, e, P).

```
path(A, B, []) :-
      edge(A, B).
path(A, B, [C |Path]) :-
      edge(A, C),
      path(C, B, Path).

edge(a, b).
edge(a, c).
edge(b, d).
edge(c, d).
edge(d, e).
```



Write the next steps of execution until the first solution, then compute "Path" using the chain of substitutions

Success branch: ?- path(a, e, P).

```
path(A, B, []) :-
        edge(A, B).
path(A, B, [C |Path]) :-
        edge(A, C).
        path(C, B, Path).

edge(a, b).
edge(a, c).
edge(b,d).
edge(c,d).
edge(d, e)
```

?- path(a, e, P).

path(A, B, [C| Path])

Clause 2, A/a, B/e, P/[C| Path]

↓

?- edge(a, C)

edge(a,b)

Fact, C/b

↓

?- path(b, e, Path).

path(A1, B1, [C1|Path1])

Clause 2, A1/b, B1/e, Path/[C1|Path1]

↓

?- edge(b, C1)

edge(b, d)

Fact, C1/d

↓

?- path(d, e, Path1).

path(A2, B2, [])

Clause 1, A2/d, B2/e, Path1/[]

↓

?- edge(d, e)

Success

The result comes from the series of substitutions:
P/[C| Path], C/b, Path/[C1|Path1], C1/d, Path1/[]

P= [b, d]

Extra-logical predicates

- Extra-logical predicates
  - is/2
    - right-hand side argument must be ground at calling time
  - comparison operators (</2, >/2, =</2, >=/2)
    - all arguments must be ground at calling time
  - not P
    - P arguments must be ground at calling time
  - !/1 (cut)
    - prunes branches in the search tree
    - beware not to lose solutions

- ⚠️ To be tested even more thoroughly than the other predicates

Compiler and Interpreter

When programming

- edit one or several files to **define** the predicates related to a given subject, domain or problem
- **compile** the files
- make sure there are **no more compilation errors** or warnings
  - Remember that an error can occur **earlier** than the place where the compiler detects it
- run queries under the **interpreter**
  - **Any predicate** defined in your compiled files (or in the built-in predefined libraries) can be called directly
- test each predicate as soon as you define it
  - Do not wait that the job is finished
  - The answer would most probably be “No”

Flexibility

- Cf french_menu exercises
- we started with very simple solutions and easily improved them step by step

➢ Prototyping language

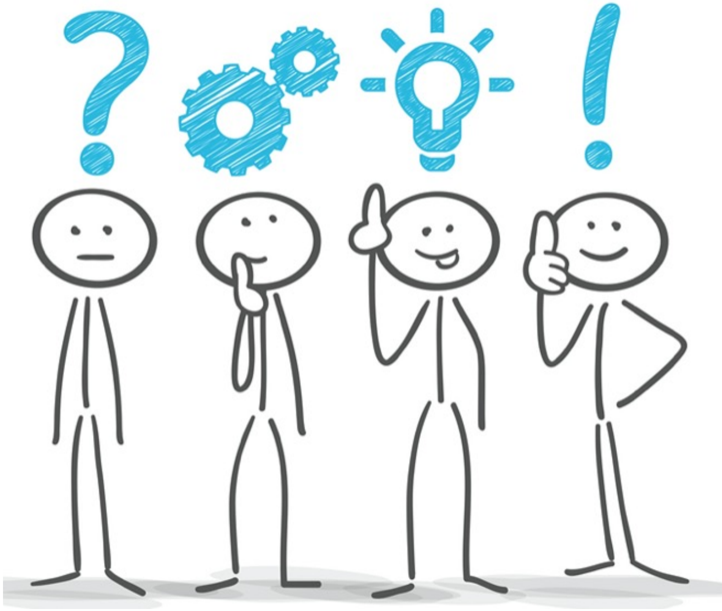
  - easy to test new ideas
  - often efficient even if you have to program in another language afterwards

Exercise 7.3: ground_list/1

- Write predicate ground_list(+List) that succeeds if every element of List is ground (namely it does not contain any variable).
- Hint: use predefined predicate ground/1.

```
?- ground_list([a, 1, [x, y]]).
yes
?- ground_list([a, 1, [X, y]]).
no
```

- **Once your code it tested, paraphrase it.**



Take your time to search, code and test your own program

Then take your time to understand the following solution

ex. 7.3: ground_list/1 (bis)

Write predicate ground_list(+Pred, +List) that succeeds if every element of List is ground.

```
?- ground_list([a, 1, [x, y]]).
yes
?- ground_list([a, 1, [X, y]]).
no
```

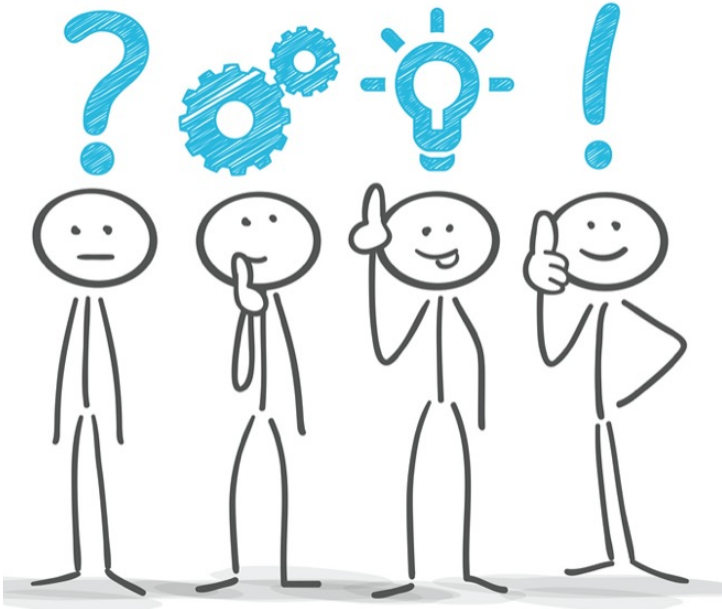
```
ground_list([]).
ground_list([H | T]) :-
    ground(H),
    ground_list(T).
```

A list is said to be ground if
   it is empty
or
   its head is ground
   (it contains no variable)
   and
   its tail is recursively a ground list

Exercise 7.4: separate_numbers/3

- Write predicate separate_numbers(+L, ?LN, ?LO) that succeeds if the arguments of list L that are numbers are extracted into list LN, the other arguments are in list LO.
- Note that we do not ask for numbers inside structures.
- Hint: use predefined predicate number/1.

```
?- separate_numbers([a, 1, 2, X, [1, 2], 3], LN, LO).
X = X
LN = [1, 2, 3]
LO = [a, X, [1, 2]]
?- separate_numbers([a, 1, 2, X, [4, 5], 3],[1, 2, 4, 5, 3], LO).
No
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Ex. 7.4: separate_numbers/3 (bis)

```
?- separate_numbers([a, 1, 2, X, [1, 2], 3], LN, LO).
X = X
LN = [1, 2, 3]
LO = [a, X, [1, 2]]
?- separate_numbers([a, 1, 2, X, [4, 5], 3],[1, 2, 4, 5, 3], LO).
No

/* predicate separate_numbers(+L, ?LN, ?LO) */
separate_numbers([], [], []).
separate_numbers([H | T], [H | LN], LO) :-
	number(H),
	separate_numbers(T, LN, LO).
separate_numbers([H | T], LN, [H | LO]) :-
	not number(H),
	separate_numbers(T, LN, LO).
```

exercise 7.5: using arguments to collect/verify properties

Given facts

m(a, 2, v).

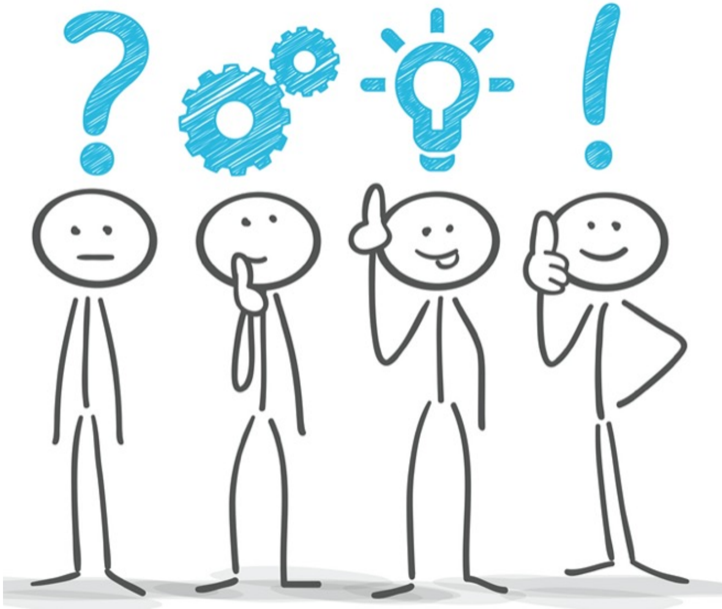
m(b, 5, nv).

d(c, 7, v).

d(e, 10, nv).

Write a predicate p/3 that is true for p([M, D], N, V) where

- M satisfies m(M, N1, V1)
- D satisfies d(D, N2, V2)
- N is the sum of N1 and N2
- V unifies to v if V1 and V2 are equal to v, to nv otherwise



Take your time to search, code and test your own program

Then take your time to understand the following solution

exercise 7.5: using arguments to collect/verify properties (bis)

```
m(a, 2, v).
m(b, 5, nv).
d(c, 7, v).
d(e, 10, nv).
```

```
p([M, D], N, V) :-
    m(M, Nm, V1),
    m(D, Nd, V2),
    N is Nm + Nd,
    check_v(V1, V2, V).

check_v(v, v, v).
check_v(v, nv, nv).
check_v(nv, v, nv).
check_v(nv, nv, nv).
```

More logic programming languages

Prolog is a starting point to

- Constraint Logic programming
- Answer set programming
- Concurrent (constraint) logic programming
- …

check sites of

- **Association for Logic programming** https://logicprogramming.org
- Association for constraint programming: https://www.a4cp.org

You can go on learning by yourself

- **Learn Prolog now !**
  - slightly larger than this lecture
  - 12 chapters
  - by Patrick Blackburn, Johan Bos, and Kristina Striegnitz
  - https://lpn.swi-prolog.org/lpnpage.php?pageid=online
- **ECLiPSE ELearning Website of Helmut Simonis**
  - video lectures, slides, handouts and other material
  - mainly ***Constraint Logic programming***
  - 20 (!) chapters
  - **An impressive lists of applications**
  - by Helmut Simonis
  - http://www.eclipseclp.org/ELearning/