INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
RENNES

Computer Science
Department

Ch 5 – More Prolog features

Towards « real » programming

Mireille Ducassé

Last revision April 2024

Remember: arithmetics

In order to ask Prolog to treat numbers,
use **is/2** (that is **extra-logical**)

```
?- X = 2+3.
X=2+3


?- X is 2+3
X=5


?- X is Y+3
Error
```

Modes

- Not all predicates are fully declarative
- It is important to know the **mode** of the arguments when the goal is called
    - ++: should be ground
    - + : should not be a variable (but can contain variable)
    - - : should be a variable
    - ? : can be not instantiated at all

Crucial information in the library documentation

- Example
    - factorial(++, ?)
        - cannot be used with variables in the first argument

Exercise 5.1: Zebra puzzle 1/2

The "Zebra puzzle":

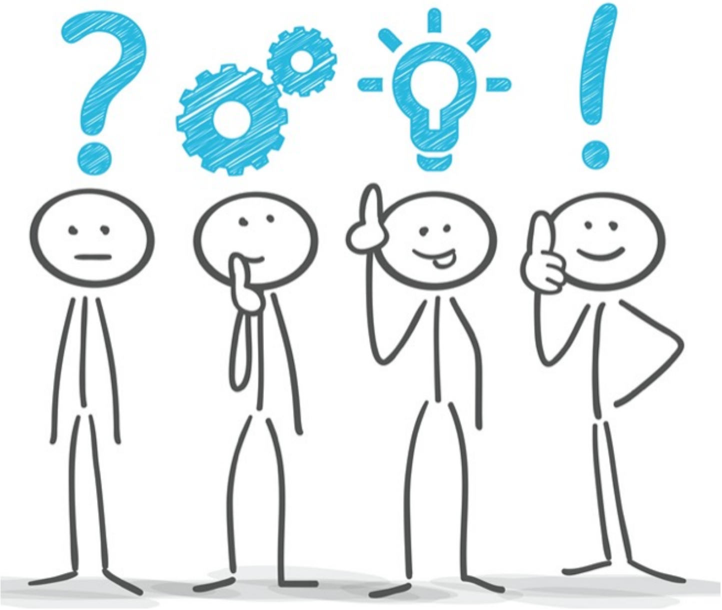
1 There are 5 colored houses in a row, each having an owner, which has an animal, a favorite cigarette, a favorite drink.
2 The English lives in the red house.
3 The Spanish has a dog.
4 They drink coffee in the green house
5 The Ukrainian drinks tea.
6 The green house is next to the white house.
7 The Winston smoker has a serpent.
8 In the yellow house they smoke Kool.
9 In the middle house they drink milk.
10 The Norwegian lives in the first house from the left.
11 The Chesterfield smoker lives near the man with the fox.
12 In the house near the house with the horse they smoke Kool.
13 The Lucky Strike smoker drinks juice.
14 The Japanese smokes Kent.
15 The Norwegian lives near the blue house.

Who has a zebra and who drinks water?

Exercise 5.1: Zebra puzzle 2/2

- Write a Prolog program to solve The Zebra problem
  - The main predicate has 17 subgoals.
- How to proceed
  - Represent the houses as a list with 5 lists **from left to right** in the street:

    Sol = [[Man1, Animal1, Cigarette1, Drink1, Color1],
    [..],[..],[..],
    [Man5, Animal5, Cigarette5, Drink5, Color5]]
  - Define predicate **right(X, Y, L)** that is true if X is just after Y in list L.
  - Define predicate **near(X, Y, L)** that is true if either X is just after Y or Y is just after X is L.
  - Use predicates member/2
  - Test case : ?- zebra(Sol).



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

Then take your time to understand the following solution

Exercise 5.1: Zebra puzzle 2/2 (bis)

```
zebra(Sol):-
    length(Sol, 5),                                      % 1
    member([english,_,_,_,red], Sol),                    % 2
    member([spanish,dog,_,_,_], Sol),                    % 3
    member([_,_,_,coffee,green], Sol),                   % 4
    member([ukrainian,_,_,tea,_], Sol),                  % 5
    right([_,_,_,_,green],[_,_,_,_,white], Sol),         % 6
    member([_,snake,winston,_,_], Sol),                  % 7
    member([_,_,kool,_,yellow], Sol),                    % 8
    Sol= [_,_,[_,_,_,milk,_],_,_],                       % 9
    Sol= [[norwegian,_,_,_,_],_,_,_,_],                  % 10
    near([_,_,chesterfield,_,_],[_,fox,_,_,_], Sol),     % 11
    near([_,_,kool,_,_],[_,horse,_,_,_], Sol),           % 12
    member([_,_,lucky,juice,_], Sol),                    % 13
    member([japonese,_,kent,_,_], Sol),                  % 14
    near([norwegian,_,_,_,_],[_,_,_,_,blue], Sol),       % 15
    member([_,_,_,water,_], Sol),                        % someone drinks water
    member([_,zebra,_,_,_], Sol).                        % someone has a zebra
```

```
near(X,Y,L):-
        right(X,Y,L).
near(X,Y,L):-
        right(Y,X,L)).

right(X, Y, [Y, X | _]).
right(X, Y, [_ | Zs]) :-
        right(X, Y, Zs).

left(X, Y, L) :-
        right(Y, X, L).
```

MORE IMPORTANT FEATURES

More important features

- Input/output
  - read/2, read/3
  - printf/2, printf/3
- Controlling backtracking
  - !/0 (cut)
- Negation as failure
  - not/1
- Operators

Note that there are many more built-in predicates. See documentation: http://eclipseclp.org/doc/bips/

Input: **read(-Term)**

Succeeds if the next term from the input stream is successfully read and unified with Term.

```
?- read(X).
12654.
X = 12654
```

‘.’ is mandatory

```
?- read(X).
hello.
X = hello

?- read(X).
father_of(lali, ana).
X = father_of(lali, ana).

?- read(X).
father_of(lali.
Error
```

read/1 is a **full parser** !!

It reads what is typed in the input stream

It **builds a Prolog term of any complexity**

except if there are syntax errors

Input: read(-Term, ++Stream)

- read/2 behaves like read/1 but it reads from a given stream
- very useful if you want to read from a file
- In that case the programming pattern is

```
top(FileName) :-
     open(FileName, read, s),
     my_program(…, s),
     close(s).
my_program(…., s) :-
     …
     read(s, A),
     do_something(A, …),
     … .
```

See http://eclipseclp.org/doc/bips/kernel/ioterm/read-2.html

Output: **printf(+Format, ?ArgList)**

- The arguments in the argument list ArgList are interpreted according to the Format string and the result is printed to the output stream
- A useful example

```
?- printf("\tHello %w !\n\tYes, %w !!", [you, 'I mean you']).
    Hello you !
    Yes, I mean you !!
```

output: **printf(+Stream, +Format, ?ArgList)**

- Same as printf/2 but can write on any file
- In that case the programming pattern is

```
top(FileName) :-
    open(FileName, write, s),
    my_program(…, s),
    close(s).
my_program(…., s) :-
    …
    printf( "…", […]),
    … .
```

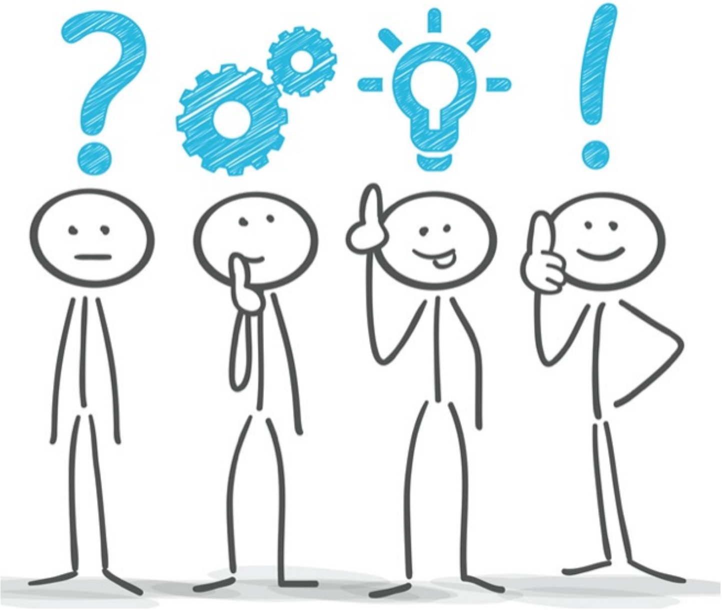
- See the Eclipse documentation for the details of the format

http://eclipseclp.org/doc/bips/kernel/ioterm/printf-3.html

Back to ancestor

- Add a predicate ancestor/0, that
  - asks the user for whom s.he wants to find ancestors
  - prints, for each solution, the ancestor

Do not forget that when you enter the person name you must end up with a '.'



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

Then take your time to understand the following solution

Back to ancestor (bis)

- Add a predicate ancestor/0, that
  - asks the user for whom s.he wants to find ancestors and
  - prints, for each solution, the ancestor

Note that this is what you are used to with procedural programming languages but it is **much less general** than ancestor/2…

Why ?

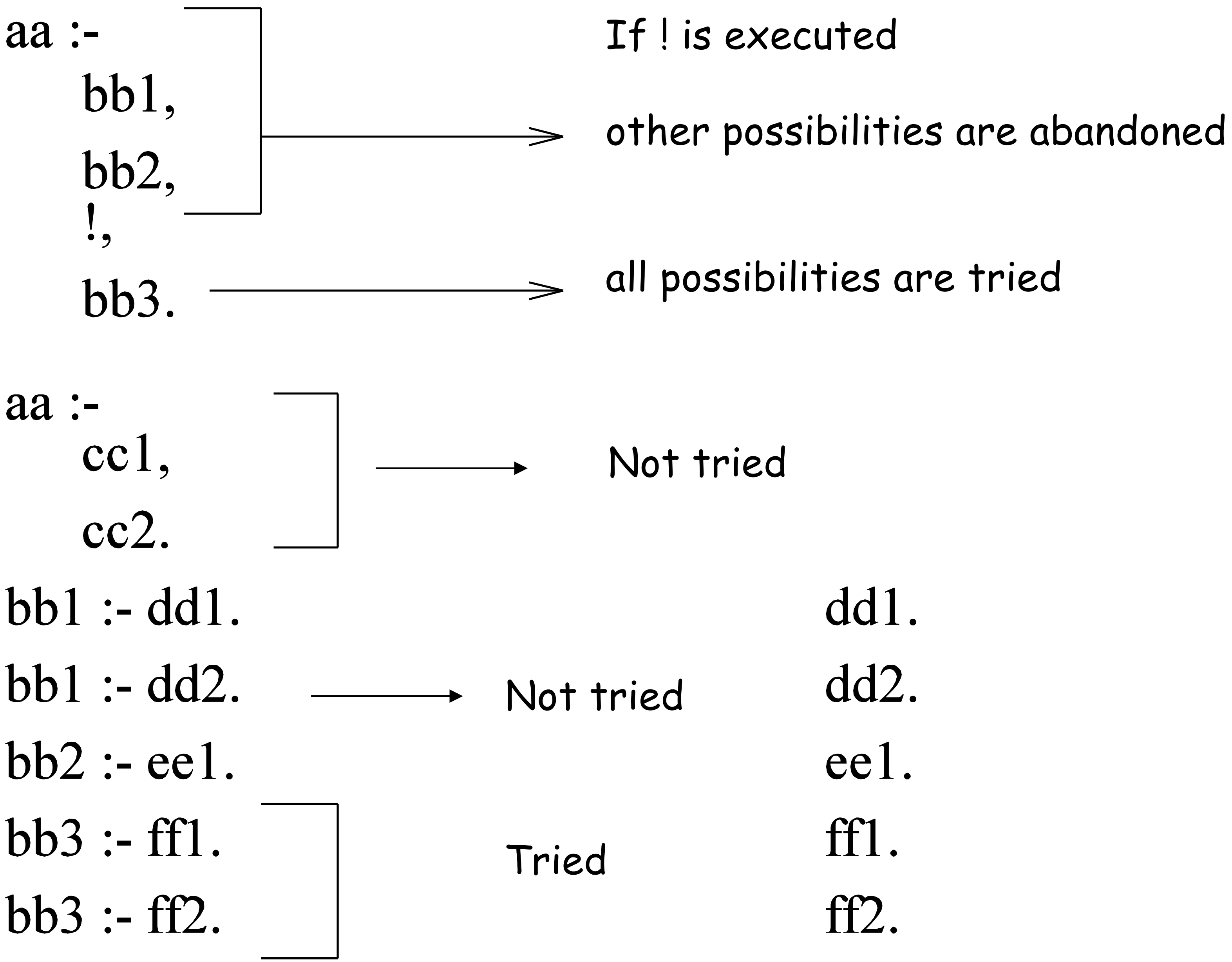
- Code

```
ancestor :-
    printf("Initial child?", []),
    read(C),
    ancestor(A, C),
    printf("%w is an ancestor of: %w\n”, [A, C]).
```

Controlling backtracking: !/0

- Backtracking is very powerful but sometimes we need to control it
  - built-in predicate '**!**' (called ‘cut’) is used to tell the interpreter not to backtrack
  - it is always true and works by **side effects** on the interpreter internal (hidden) structure
  - it cuts branches in the search tree
    - **within a certain scope**
  - this is very useful but **extra-logical**

cut : example 1



```
aa :-
    bb1,
    bb2,
    !,
    bb3.
```

If ! is executed

other possibilities are abandoned

all possibilities are tried

```
aa :-
    cc1,
    cc2.
```

Not tried

```
bb1 :- dd1.
bb1 :- dd2.
bb2 :- ee1.
bb3 :- ff1.
bb3 :- ff2.
```

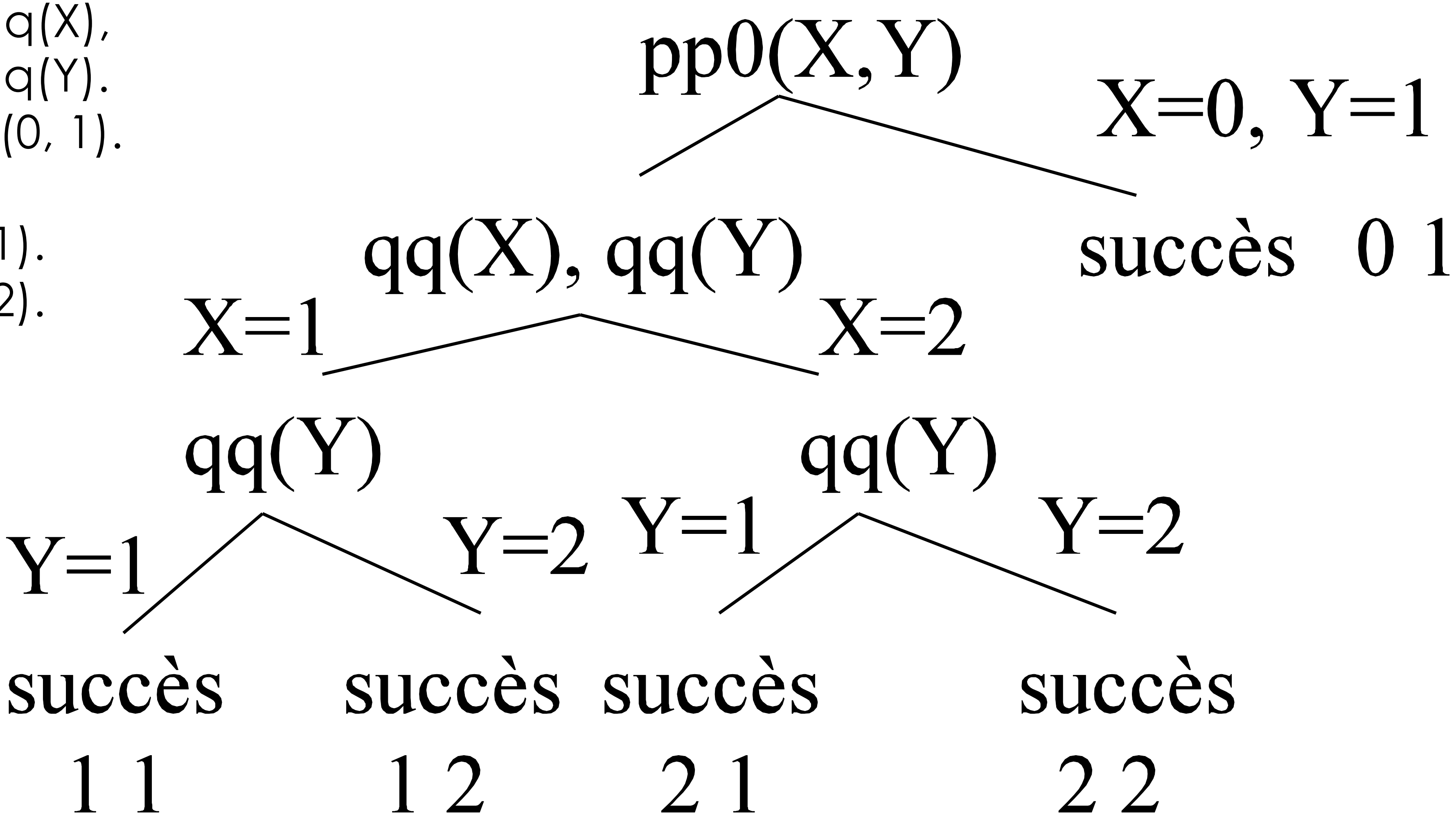
Not tried

Tried

```
dd1.
dd2.
ee1.
ff1.
ff2.
```

Cut: example 2 1/3

```
pp0(X, Y) :-
    qq(X),
    qq(Y).
pp0(0, 1).

qq(1).
qq(2).
```



pp0(X,Y)

X=0, Y=1

qq(X), qq(Y)

succès 0 1

X=1

X=2

qq(Y)

qq(Y)

Y=1

Y=2

Y=1

Y=2

succès 1 1

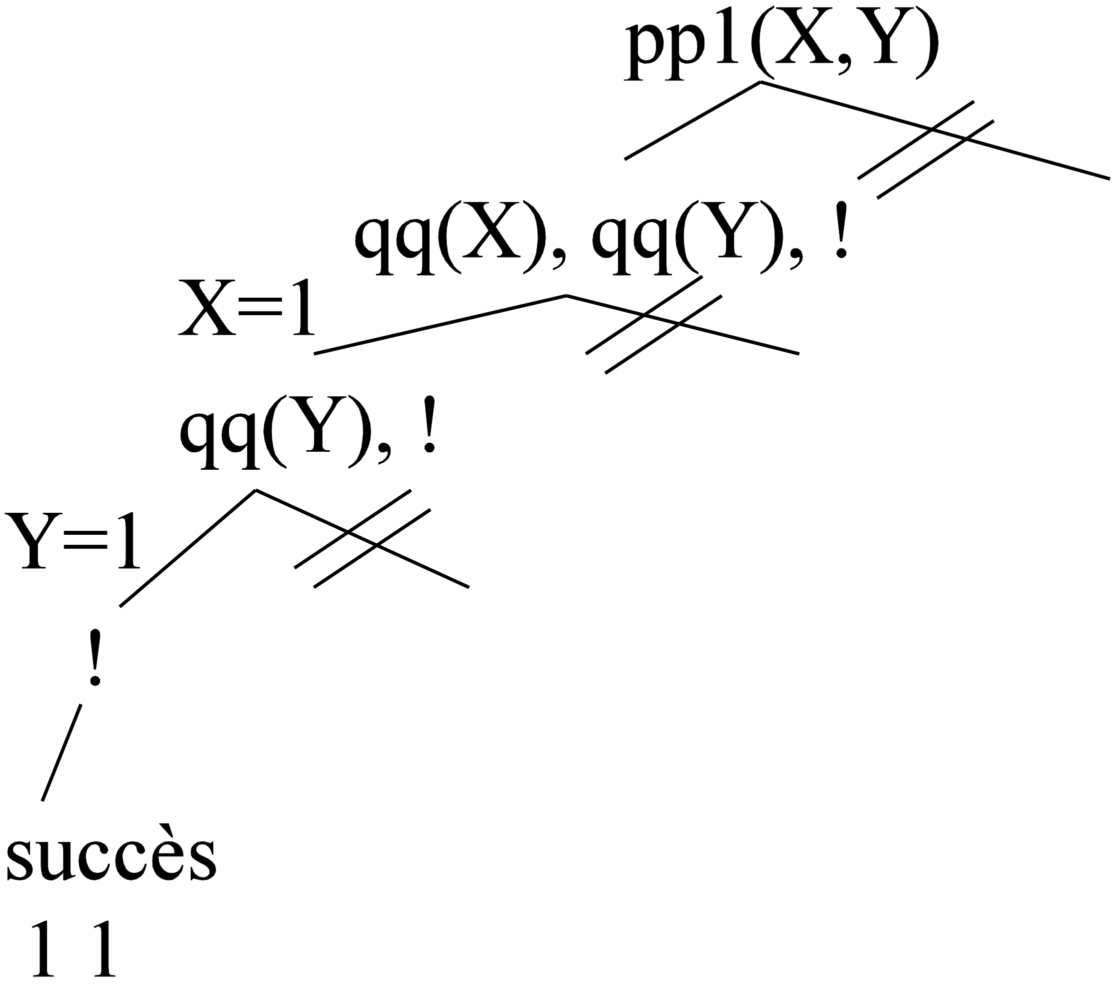
succès 1 2

succès 2 1

succès 2 2

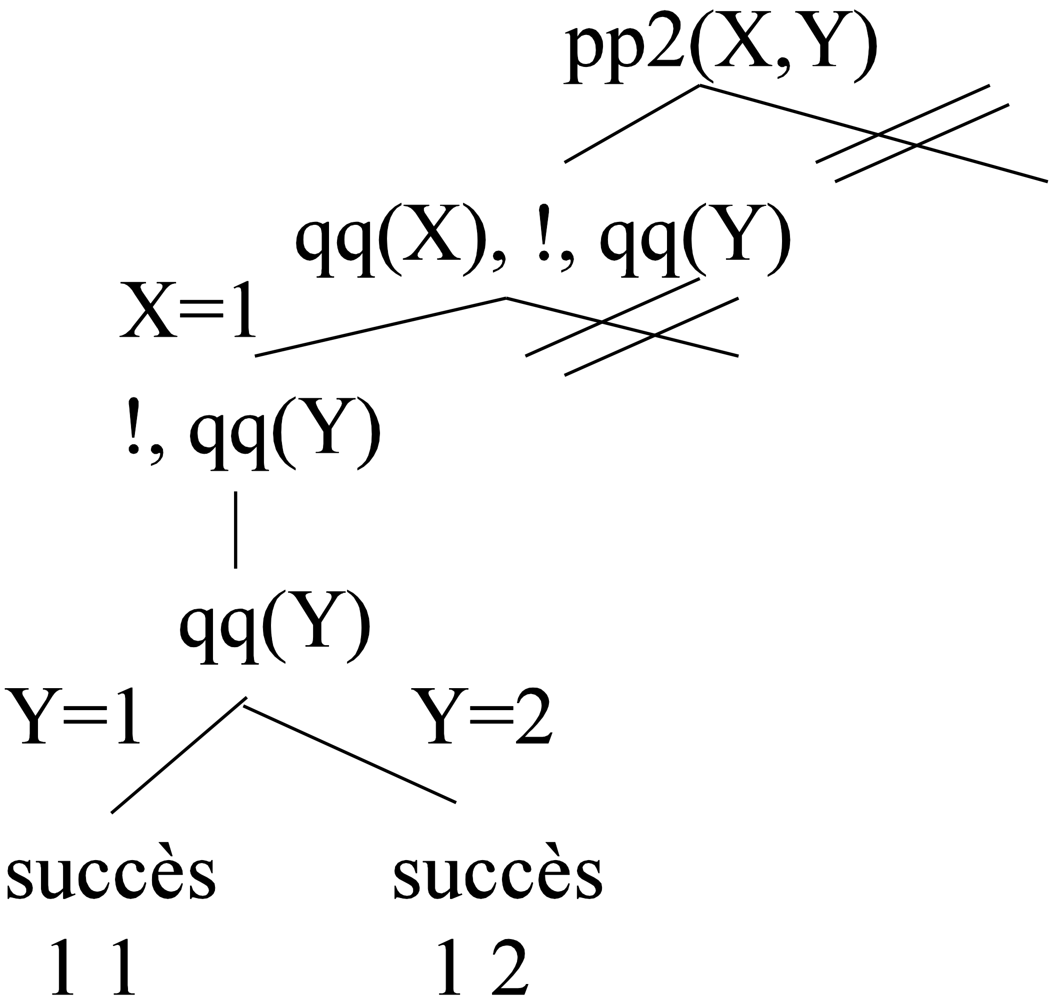
Cut: example2 2/3

```
pp1(X, Y) :-
    qq(X),
    qq(Y),
    !.
pp1(0, 1).

qq(1).
qq(2).
```



Cut: example2 3/3

```
pp2(X, Y) :-
    qq(X),
    !,
    qq(Y).
pp2(0, 1).

qq(1).
qq(2).
```



pp2(X,Y)

qq(X), !, qq(Y)

X=1

!, qq(Y)

qq(Y)

Y=1

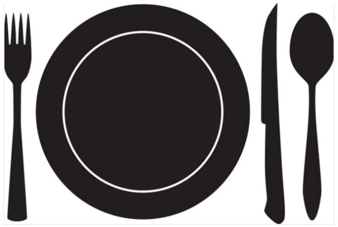
Y=2

succès

1 1

succès

1 2



Back to french_menu/2

```
french_menu([A, M], Cal) :-
        appetizer(A , ApCal),
        main_course(M , MaCal),
        check_cal([ApCal, MaCal], Cal).
french_menu([M, D] , Cal) :-
        main_course(M , MaCal),
        dessert_or_cheese(D , DeCal) ,
        check_cal([MaCal, DeCal], Cal).
…
```

What are the answers to the following queries ?

```
?- french_menu(M, C), !.
```

```
?- french_menu([X, Y], C), main_course(X, _), !.
```

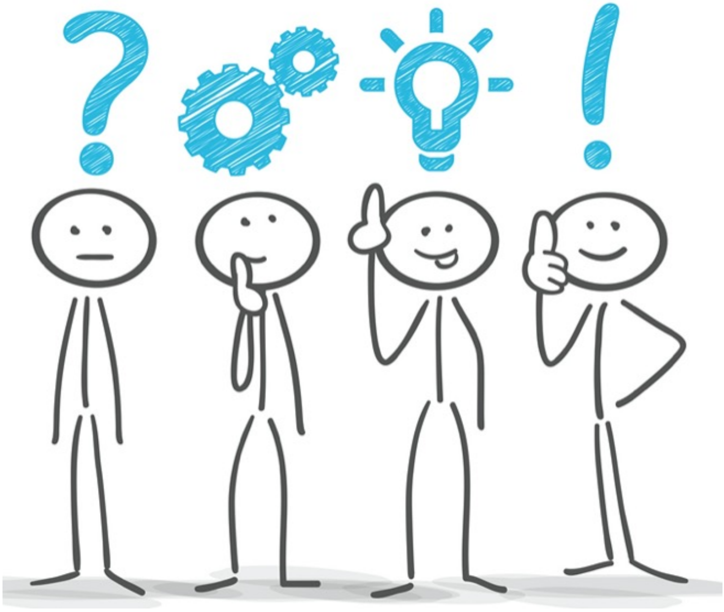
```
?- french_menu([X, Y], C) , !, main_course(X, _).
```

Back to ex 3.5 deleteXs(X, L1, L2)

We designed

```
deleteXs(_X, [], []).
deleteXs(X, [X | L1], L2) :-
    deleteXs(X, L1, L2).
deleteXs(X, [Z | L1], [Z | L2]) :-
    X \= Z,
    deleteXs(X, L1, L2).
```

How could we prune the search tree without loosing any solution ?



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

Then take your time to understand the following solution

Back to ex 3.5 deleteXs/3: update 1 (bis)

We designed

```
deleteXs(_X, [], []).
deleteXs(X, [X | L1], L2) :-
        deleteXs(X, L1, L2).
deleteXs(X, [Z | L1], [Z | L2]) :-
        X \= Z,
        deleteXs(X, L1, L2).
```

How could we prune the search tree without loosing any solution ?

```
deleteXs(_X, [], []).
deleteXs(X, [X | L1], L2) :-
        deleteXs(X, L1, L2),
        !.
deleteXs(X, [Z | L1], [Z | L2]) :-
%       X \= Z,
        deleteXs(X, L1, L2).
```

%.... <end of line>: comment

Note that the base clause does not need a cut because Prolog compiler is clever enough to deduce that it is exclusive

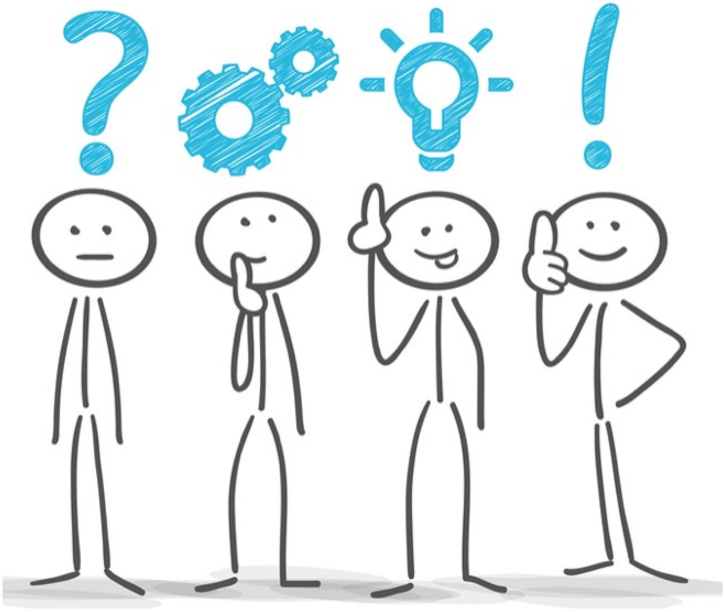
The test is not necessary, but keeping it would be correct

Cut and declarative programming

- Cut is extra-logical
  - testing all cases is especially crucial
  - **remember that you should always test**
    - verification of at least a correct solution (that should get ‘yes’)
    - verification of at least an incorrect solution (that should get ‘no’)
    - generations of solutions (at least one test case per argument with that argument variable)

Exercise 5.2: minimum of 2 integers

- min(M, X, Y) is true if M is the minimum of X and Y
- Write 2 versions
  - one without cut and one with cut that prunes the search tree **without changing the results**
  - specify the mode of the arguments when the goal is called
    - ++: should be ground
    - + : should not be a variable (but can contain variable)
    - - : should be a variable
    - ? : can be not instantiated at all



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

Then take your time to understand the following solution

Ex. 5.2: minimum of 2 integers (bis)

- min(M, X, Y) is true if M is the minimum of X and Y
- Write 2 versions
  - one without cut and one with cut that prunes the search tree without changing the results

```
:- mode mini(?, ++, ++).
mini(X, Y, X) :-
    X < Y.
mini(X, Y, Y) :-
    Y =<X.
```

Optimized version

```
mini(X, Y, X) :-
    X < Y,
    !.
mini(X, Y, Y) :-
    Y =<X.
```

Ex. 5.2: minimum of 2 integers (ter)

This is incorrect

```
mini(X, Y, X) :-
    X < Y,
    !.
mini(X, Y, Y).
```

```
?- min(2, 5, 5).
yes
```

(It should be “No”, 2 is not the minimum of 5 and 5!)

Assumption of a closed world

- **Negation as failure**
  - if it cannot be proved = it is considered negated
- The standard procedure is to check if a goal **succeeds**
- You can explicitly check if a goal **fails**
  - using predicate not/1
  - But you have to be careful

Predicate not/1

- Existing built-in **meta-**predicate
  - namely a predicate that takes a **predicate as argument**

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

fail/0 is another built-in predicate
It forces the execution to fail.

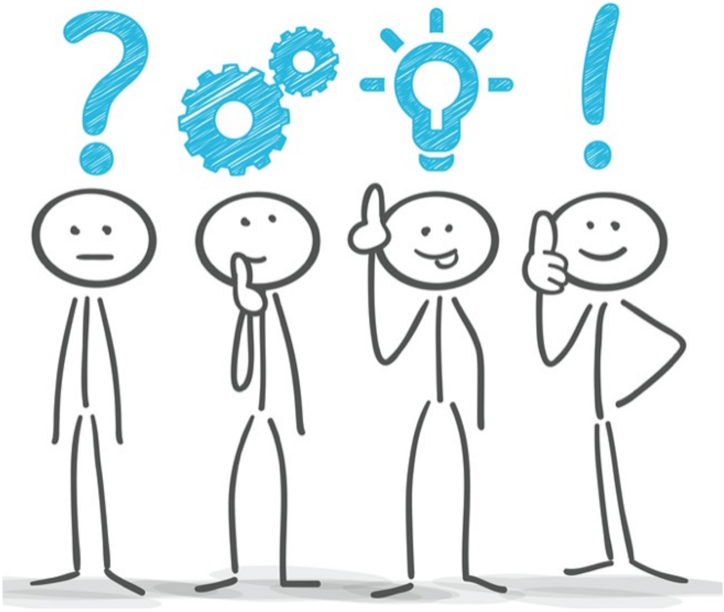
The second clause is only tried if P did not previous succeed, hence telling that P fails 🤪

Exercise 5.3: small/1

```
short(X) :-
    not(tall(X)).
tall(peter).
tall(paul).
```

- **What are the answers to**

```
?-short(mary).
?-short(peter).
?-short(X), X=mary.
```



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

Then take your time to understand the following solution

Exercise 5.3: small/1 (bis)

```
short(X) :-
    not(tall(X)).
tall(peter).
tall(paul).
```

When using not/1 you should be careful about non ground arguments !

⚠ not/1 is not fully logical

- **What are the answers to**

?-short(mary).	yes (valid)
?-short(peter).	no (valid)
?-short(X), X=mary.	**no (invalid)**

Operators

Operators help to write more readable code

- **useful when your code is to be read by non experts**

you have to declare

- priority
- whether is an infix, prefix or suffix operator
- (sometimes tricky)

Examples

- X parent_of Y is the same as parent_of(X, Y)

  `:- op(500, xfx , parent_of).`

  infix operator with priority 500

- X is 3+2 is the same as is(X, 3+2)

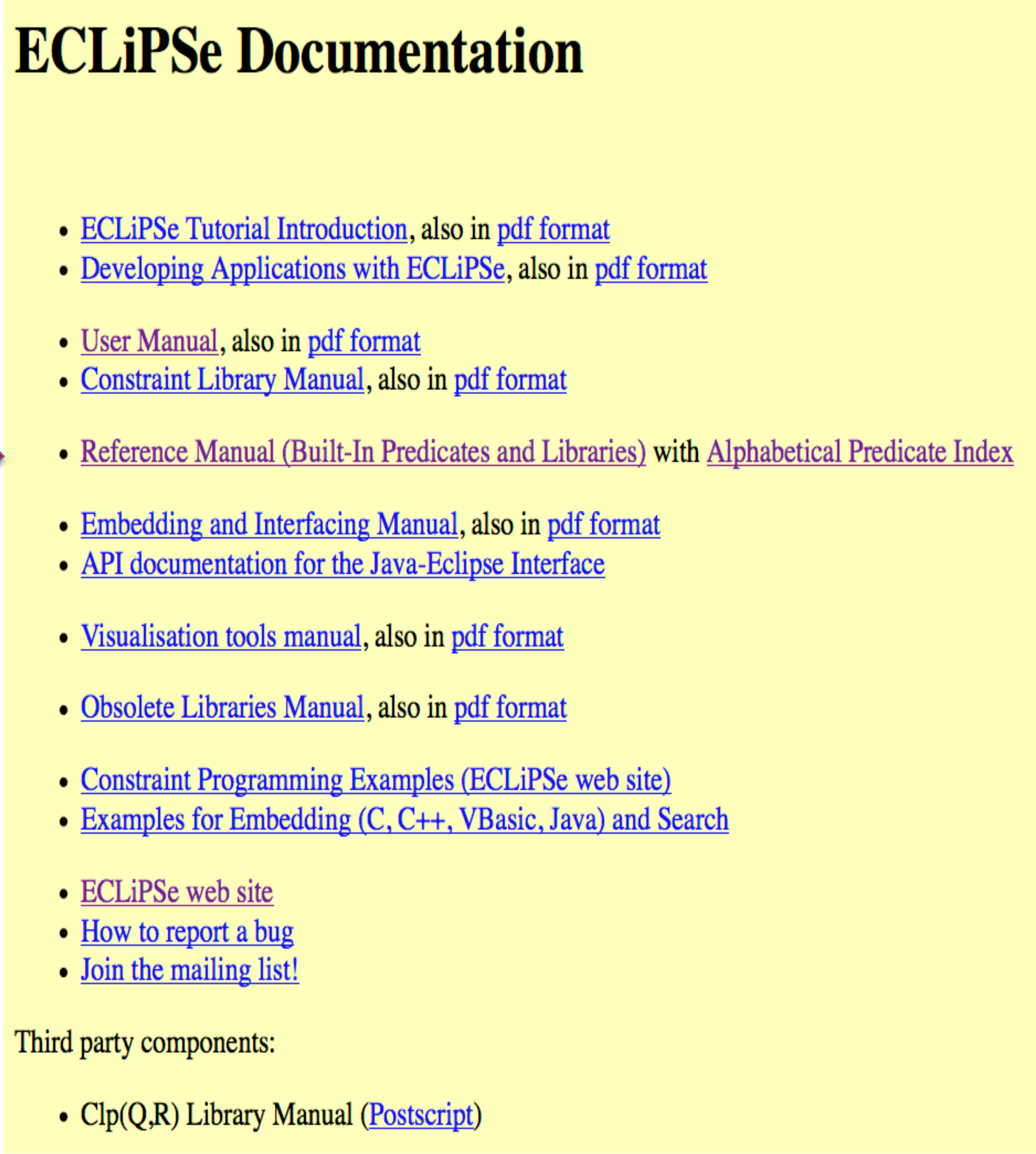
Note that the project does not need operators

ECLiPSe Documentation

- User manual
- Tutorials
- **The Reference Manual(s)**
  - **must always be open while programming**
  - predicate with the largest arity often the most general one, with the most detailed help
    - eg min_max/8
- **On line help**
  - same contents as reference manual

  :- help <keyword>.

http://eclipseclp.org/doc/index.html



ECLiPSe Documentation

- ECLiPSe Tutorial Introduction, also in pdf format
- Developing Applications with ECLiPSe, also in pdf format

- User Manual, also in pdf format
- Constraint Library Manual, also in pdf format

- Reference Manual (Built-In Predicates and Libraries) with Alphabetical Predicate Index

- Embedding and Interfacing Manual, also in pdf format
- API documentation for the Java-Eclipse Interface

- Visualisation tools manual, also in pdf format

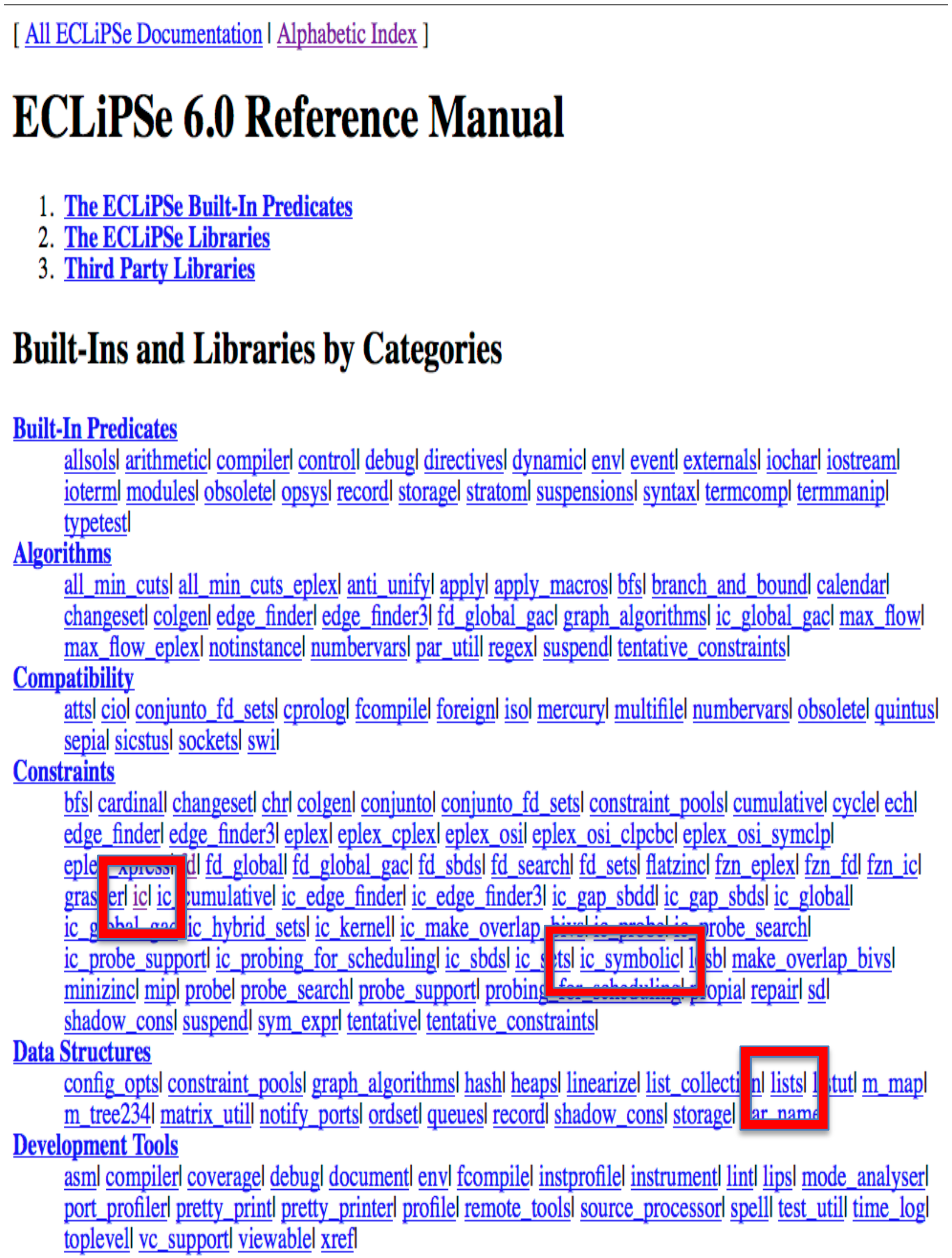
- Obsolete Libraries Manual, also in pdf format

- Constraint Programming Examples (ECLiPSe web site)
- Examples for Embedding (C, C++, VBasic, Java) and Search

- ECLiPSe web site
- How to report a bug
- Join the mailing list!

Third party components:

- Clp(Q,R) Library Manual (Postscript)



[All ECLiPSe Documentation | Alphabetic Index]

ECLiPSe 6.0 Reference Manual

1. The ECLiPSe Built-In Predicates
2. The ECLiPSe Libraries
3. Third Party Libraries

Built-Ins and Libraries by Categories

Built-In Predicates
allsols| arithmetic| compiler| control| debug| directives| dynamic| env| event| externals| iochar| iostream| ioterm| modules| obsolete| opsys| record| storage| stratom| suspensions| syntax| termcomp| termmanip| typetest|

Algorithms
all_min_cuts| all_min_cuts_eplex| anti_unify| apply| apply_macros| bfs| branch_and_bound| calendar| changeset| colgen| edge_finder| edge_finder3| fd_global_gac| graph_algorithms| ic_global_gac| max_flow| max_flow_eplex| notinstance| numbervars| par_util| regex| suspend| tentative_constraints|

Compatibility
atts| cio| conjunto_fd_sets| cprolog| fcompile| foreign| iso| mercury| multifile| numbervars| obsolete| quintus| sepia| sicstus| sockets| swi|

Constraints
bfs| cardinal| changeset| chr| colgen| conjunto| conjunto_fd_sets| constraint_pools| cumulative| cycle| ech| edge_finder| edge_finder3| eplex| eplex_cplex| eplex_osi| eplex_osi_clpcbc| eplex_osi_symclp| eple xpress| d| fd_global| fd_global_gac| fd_sbds| fd_search| fd_sets| flatzinc| fzn_eplex| fzn_fd| fzn_ic| gras er| ic| ic_ umulative| ic_edge_finder| ic_edge_finder3| ic_gap_sbdd| ic_gap_sbds| ic_global| ic_g bal_gac| ic_hybrid_sets| ic_kernel| ic_make_overlap_bivs| ic_probe| ic_probe_search| ic_probe_support| ic_probing_for_scheduling| ic_sbds| ic_ ets| ic_symbolic| l sb| make_overlap_bivs| minizinc| mip| probe| probe_search| probe_support| probing_for_scheduling| opia| repair| sd| shadow_cons| suspend| sym_expr| tentative| tentative_constraints|

Data Structures
config_opts| constraint_pools| graph_algorithms| hash| heaps| linearize| list_collecti n| lists| l tut| m_map| m_tree234| matrix_util| notify_ports| ordset| queues| record| shadow_cons| storage| ar_name

Development Tools
asm| compiler| coverage| debug| document| env| fcompile| instprofile| instrument| lint| lips| mode_analyser| port_profiler| pretty_print| pretty_printer| profile| remote_tools| source_processor| spell| test_util| time_log| toplevel| vc_support| viewable| xref

flatten(+NestedList, ?FlatList)

Succeeds if FlatList is the list of all elements in NestedList, as found in a left-to-right, depth-first traversal of NestedList.

+NestedList
Ground List.

?FlatList
List or variable.

Description

FlatList is the list built from all the non-list elements of NestedList and the flattened sublists. The sequence of elements in FlatList is determined by a left-to-right, depth-first traversal of NestedList.

The definition of this Prolog library predicate is:

```
flatten(List, Flat) :-
        flatten_aux(List, Flat, []).

flatten_aux([], Res, Cont) :- -?-> !, Res = Cont.
flatten_aux([Head|Tail], Res, Cont) :-
        -?->
        !,
        flatten_aux(Head, Res, Cont1),
        flatten_aux(Tail, Cont1, Cont).
flatten_aux(Term, [Term|Cont], Cont).
```

This predicate does not perform any type testing functions.

Modes and Determinism

- flatten(+, -) is det

Fail Conditions

Fails if FlatList does not unify with the flattened version of NestedList.

Resatisfiable

No.

Examples

```
Success:
    [eclipse]: flatten([[1,2,[3,4],5],6,[7]], L).
    L = [1, 2, 3, 4, 5, 6, 7]
    yes.

Fail:
    [eclipse]: flatten([1,[3],2], [1,2,3]).
    no.
```

See Also

flatten / 3, sort / 2, sort / 4, length / 2, member / 2