Logic Paradigm (LP)

Striking features about the "*pure*" LP: (e.g., *Mercury*, *Starlog*)

1) All about "what" is the problem environment; and nothing about "how" to do the problem solution.
2) Total separation between the "logic" and the "control" of the solution; making it very amenable for asynchronous execution of program code (exploitation of concurrency and parallel processing).**Hence, the order of program "rules", or order of "sub-goals" within a rule, is not important.**
3) The program is expressed in the form of propositions (predicates or relations) that "assert" the existence of a desired "goal". It consists of the following types of clauses, in any order:
   i) Hypotheses (Axioms): number of "facts" on "atomic" (concrete) individuals; a fact is always "asserted" true.
      Example: father (jim, dan).
      Mother (cathy, mike).

   ii) Rules of Inference (conditions): Number of general principles which define the problem domain.
      Example: a) grandparent (X,Z) :- parent (X,Y) , parent(Y,Z).
      b) parent (X,Y) :- father (X,Y).
      parent (X,Y) :- mother (X,Y).

   As we see, a rule has a "head" and a "body" that consists of one or more subgoals. In order to prove the assertion of the "head" we need to prove the assertion of all of its body's subgoals.

For example, in rule a, X is the grandparent of Z if it is the parent of some Y **and** Y is the parent of Z. In rule b, X is the parent of Y if it is its father **or** mother.

iii) Theorems (Goals/Queries):

They are the input to the prolog system seeking proof of assertion. If the goal has no variables then the answer is true of fail (asserted or not asserted). If it has some variables then the system will answer fail, or tallies a sequence of one or more value(s) of the input variable(s), as the programmer enters ";", until there is no more answers to be provided by the system.

Example: ?grandparent(alex, john).
?grandparent(X, john).

PROLOG

An "impure" logic programming, where the logic is not totally separated from the control and the order of the program clauses is important factor in the execution efficiency. The programmer can alter the execution control via predefined and universally asserted clauses, e.g., **cut** "**!**" operator, **fail**.
Moreover, the **order** of program **rules** or that of **sub-goals** within a rule is **important.** All of the above violate the steel separation ***logic*** and ***control*** threads of the ***pure*** LP's programs.

More violations are: The relations (predicates) -- ***asserta***, ***assertz***, and ***retract***. (read text pages 475). *Satisfied* subgoals such as ***asserta(C)***, ***assertz(C)***, and ***retract(C)*** will ***insert*** at beginning, ***insert*** at the end, ***delete*** the *clause* ***C*** ***in***/***from*** the database, respectively.
Such relations, when included in the program rules, will imply that the logical derivation will change the control (e.g., adding or removing facts) which is not at all part of the ***pure*** LP basic philosophy.
For more ***impurity***, Prolog includes data ***types*** and ***structures*** versus objects (atomic items) with meaning that is not part of the ***pure*** LP languages.

The Goal *Deduction* Process in Prolog:

It is carried out by the system inference engine which takes a *"goal"* to be proven and the program clauses (facts/rules "*database*") as inputs and performs the following on them:

A) Resolution: It is the process of finding a chain of inference rules and/or facts that connects the input goal (or every of its subgoals, in case of a compound goal) to one or more facts in the database. This is done by ***matching*** the **input goal's** (or subgoal) **name with every clause's name in the program**, hence matching will be with:

i) matching **rules' heads**-- top-down, one at a time, **resolving** (**expanding**) the matched rule's body subgoal(s); and a resolution process will start for the new subgoal(s), or

ii) matching **facts** – top-down, one at a time, until a match between the goal and fact names is found, then if there is no need to ***unify*** any names (i.e., all inputs in the *goal* are literal-constants-- father (*john*, *alex*)), an "assertion" (true) will be reported to the upper level, for the given goal (or subgoal), followed by an exit from the current level of the search tree. Otherwise, see below for the unification process (guess&verify).

B) Unification: The presence of, non-terminating symbols (***variables*** such as X, Y, Z, etc) in the input goal (or subgoals) requires the resolution process to **find values of those goals' variables that makes the matching process to succeed**; which is done via the ***unification*** process that determines useful values for variables. The temporary assignment (***guessing***) of values to variables (binding) is called ***instantiation***.

- Unification is a pattern matching operation between two terms, both of which can contain variables.
- A **substitution** is an assignment of values to variables.
- Two terms unify if there is a substitution that makes the terms identical.

  Unifying *f(X,2)* and *f(3,Y)* produces the following assignments *X=3*, *Y=2*

It is common in the resolution process above, in matching goals (or subgoals) with rules heads and facts, if a goal (or subgoal) has a variable then the system will instantiate (allocate memory and assign value) such variable to its corresponding value (that completes the match (***guessing*** it), and then ***verifies*** it through the rest of the ***resolution*** process.

We can say that the goal **deduction** process, in general, is a continuous process of "***guess***" and "***verify***" operations.

There are three majors goal matching approaches, in the resolution process:

A) *Bottom-up* Resolution (forward chaining): (text pages 463-5)

The system starts from the facts (the leaves of the searching tree space) combining them to prove the above subgoals true, then use the proven subgoals to prove their above level subgoals true, and so on, until it reaches the input goal, hence asserting it true (fact). It is an efficient approach when there are many answers to the variables in the input goal; also in some application we might avoid going through calculating redundant subgoals' solutions (as done in the next mechanism of the top-down approach).

B) *Top-down* Resolution (backward chaining):(text pages 461-3)

The system starts from the input goal resolving until it reaches a set of original facts. Good when we have small set of possible answers, better yet, only one. Two techniques are used to traverse the resolution search tree:

i) **Breadth-first:** The system works all expanded subgoals (in the resolution process) in parallel! Hence, it gets fast to **all** answers that prove the goal true. Moreover, if one resolution path has an infinite length (black-hole that holds the system), other solution would be still obtained (not like the next Depth-first).

It is good for “pure” LP languages, but **NOT** Prolog, since in Prolog the order of traversing (resolving) subgoals is important. Even though, it gets to many answers fast, it requires a huge memory space to store all intermediate nodes in the tree (**exponential *space*** complexity).

ii) **Depth-first**: The prolog approach where resolution is done in the order of left to right in the subgoals (Left-Node-Right policy of visiting the tree nodes). After resolving a subgoal, if *true* and not a black-hole, the system proceeds to the next one (its right). It is clear that the system does not need to store all intermediate subgoals (as in the Breadth-first), since it only goes one sub-tree (subgoal) path at a time. Hence, it has a reasonable memory space complexity. Yet, if one search results in traversing an infinite path, the entire process is held hostage to such ***black-hole***, even with the existence of other paths that might prove the goal true!

An enhancement to the Top-down is when the system remembers the already satisfied subgoals not to resolve them again!

Backtracking:

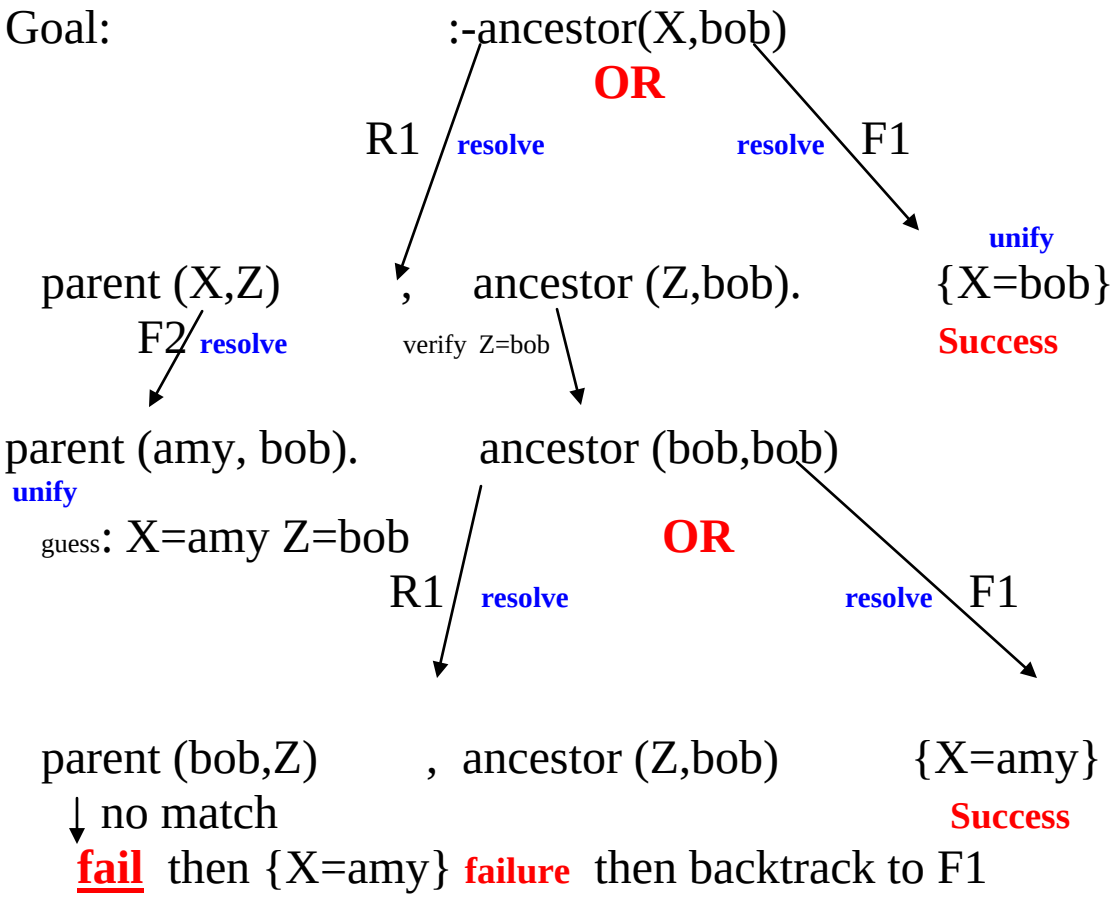
When the system fails to prove a subgoal, instead of abandoning it, it goes back and attempts to find one of its **alternative** paths (*OR*'ed rules, if exists) (i.e., tree path) . Hence, if there exists an alternative (sibling path) rule that matches the previous subgoal, then it guesses different variables' values (instantiations) for the failing subgoal, and resolves it again with the newly obtained values. If all sibling alternative rules fail for such subgoal, the system backtracks further up the solution tree, and tries an alternative rule for the parent subgoal, and so on, until it gets to the ***root*** goal then it declares goal failure.

C) *Hybrid* Top-down and Bottom-up: the two techniques are applied and they meet in the middle for performance enhancement.

Example of Prolog Dependency on Rules' and Subgoals Order:

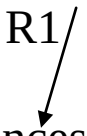
(R1): ancestor (X,Y) :- parent (X,Z), ancestor (Z,Y).
(F1): ancestor (X,X).
(F2): parent (amy, bob).

Depth-first search:



There are two values found for X while that prove the assertion of the target goal: amy and bob.
Yet if we just switch the two subgoals of R1, then we get into a "***black hole***" (left-recursion)!

(R1): ancestor (X,Y) :- ancestor (Z,Y), parent (X,Z).



ancestor (Z,bob) → «***black hole***»-- recursion !

The system will never leave this black-hole because of the depth first approach of Prolog. Yet if we swap "R1" and "F1" (write F1 before R1), then we will get at least one solution (X=bob) before going into the blackhole!

In other logic programming (not Prolog) with ***breadth first***, this problem will never happen, since the system will try ***all*** possible solutions (i.e, all alternative solutions, rules), hence "F1" will be also tried and solution is found, X= bob.

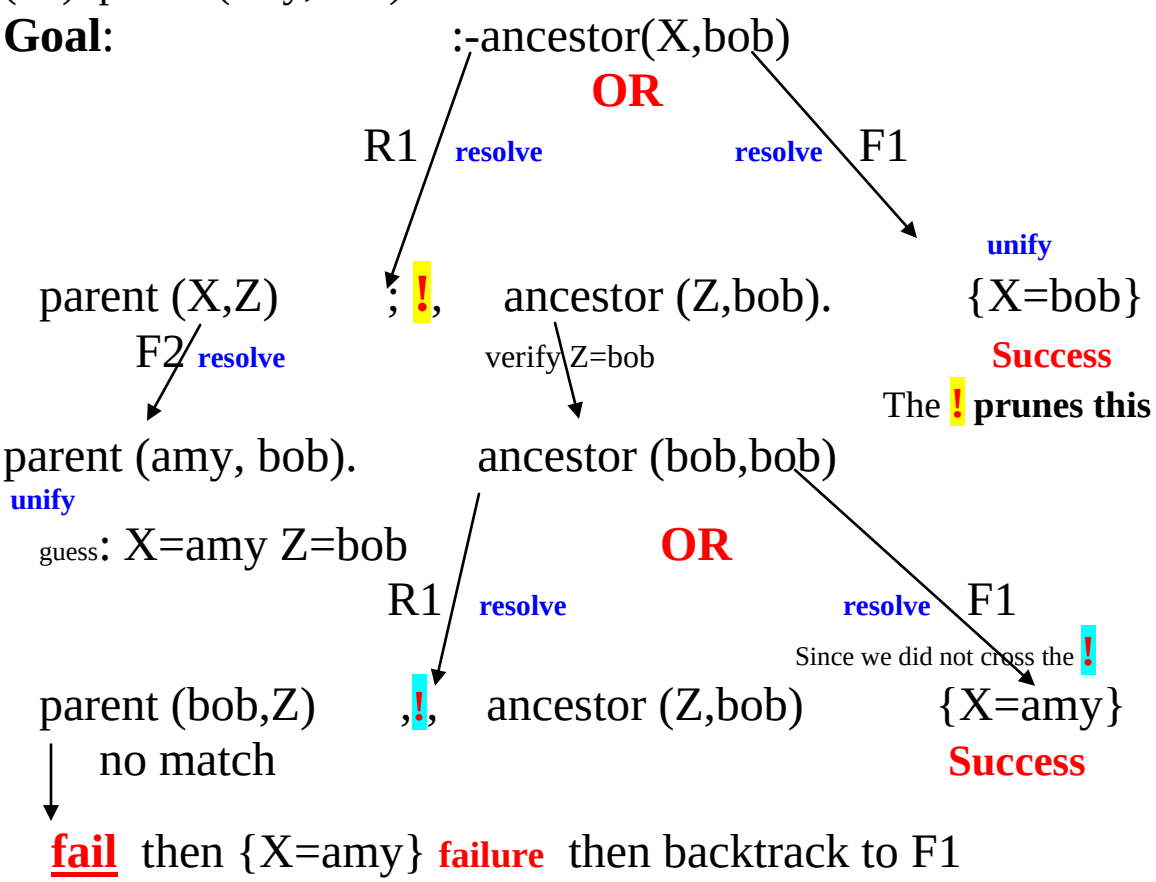
The "cut" operator "!" in Prolog: (**one way non-revolving door**)
It is a system built in subgoal that is always "asserted" (universally true) to stop, on purpose, the backtracking process. When the system encounters (gets to) a cut operator, in the resolution process, it passes it (since it is automatically asserted), but never backtrack to try any alternatives of variables' instantiations, or an alternative rules (of the ORed possibilities), in case of a failing subgoal after the passed cut.

Rule-1: **b** :- c , d , **!** , e , f . % Rule for b
Rule-2: **b** :- g , h. % an alternate Rule for b

A cut "!" is a one way door, you get in, locked, no return.
If subgoals **c** and **d** are asserted (successful) then we pass through the ***cut*** (one way non-revolving door), proceeding to resolve subgoals **e** and **f**. If subgoal **e** fails, the system is prevented from trying other possible resolutions for **c** and **d**, or even try the alternative rule for **b** (Rule 2). If subgoal **e** fails, then we can try alternative rule for **e**, but if **e** is asserted, then subgoal **f** failed, then **we can try all alternatives of resolving e & f, BUT**, never back through the cut to retry **c** or **d**, or **b**.

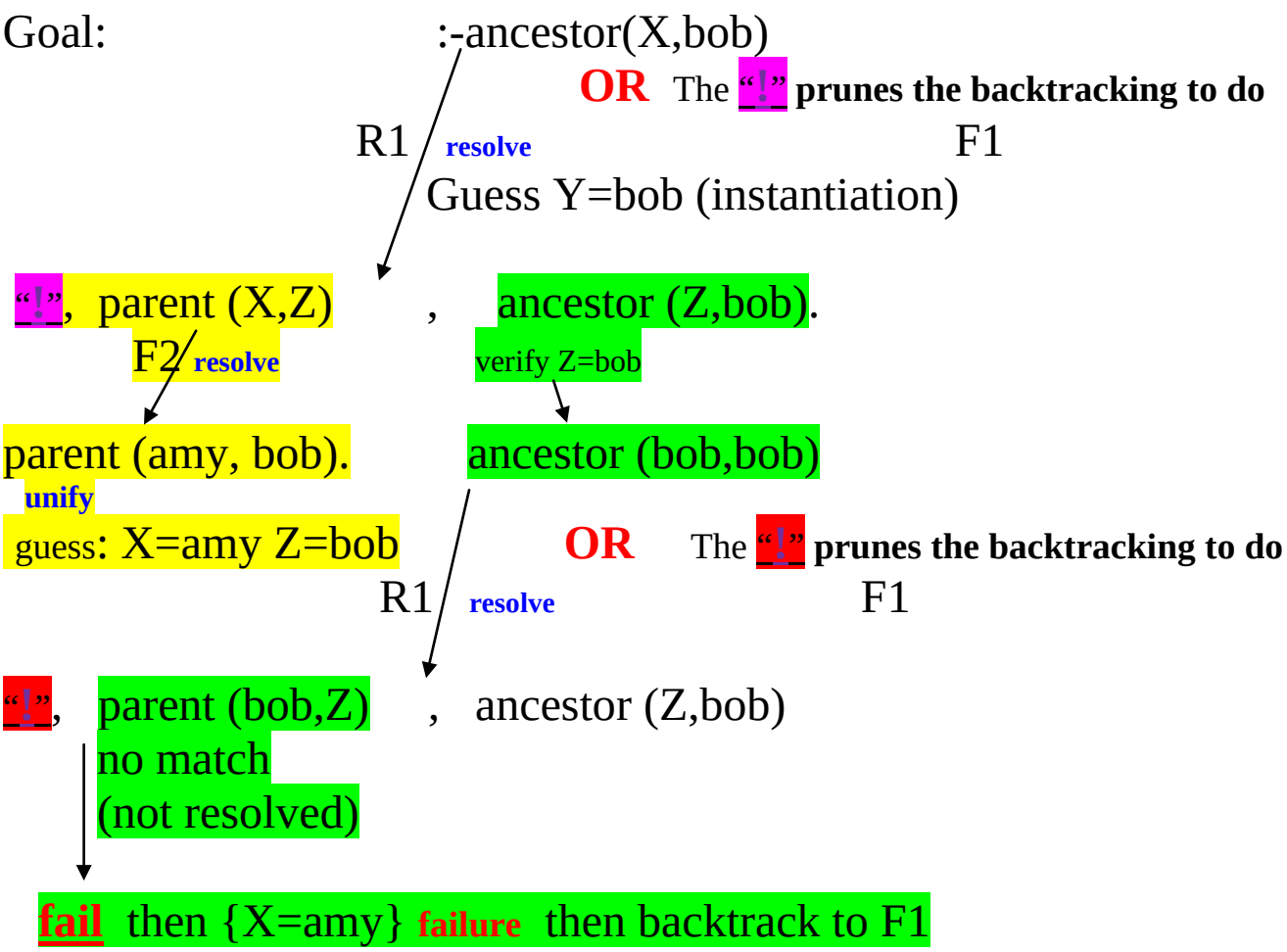
Example using the ***cut*** operator:

(R1): ancestor (X,Y) :- parent (X,Z), **!**,ancestor (Z,Y).
(F1): ancestor (X,X).
(F2): parent (amy, bob).



Hence, in such position of the cut, **one** solution only would be found {X=amy}.

(R1): ancestor (X,Y) :- **!**, parent (X,Z) , ancestor (Z,Y).
(F1): ancestor (X,X).
(F2): parent (amy, bob).



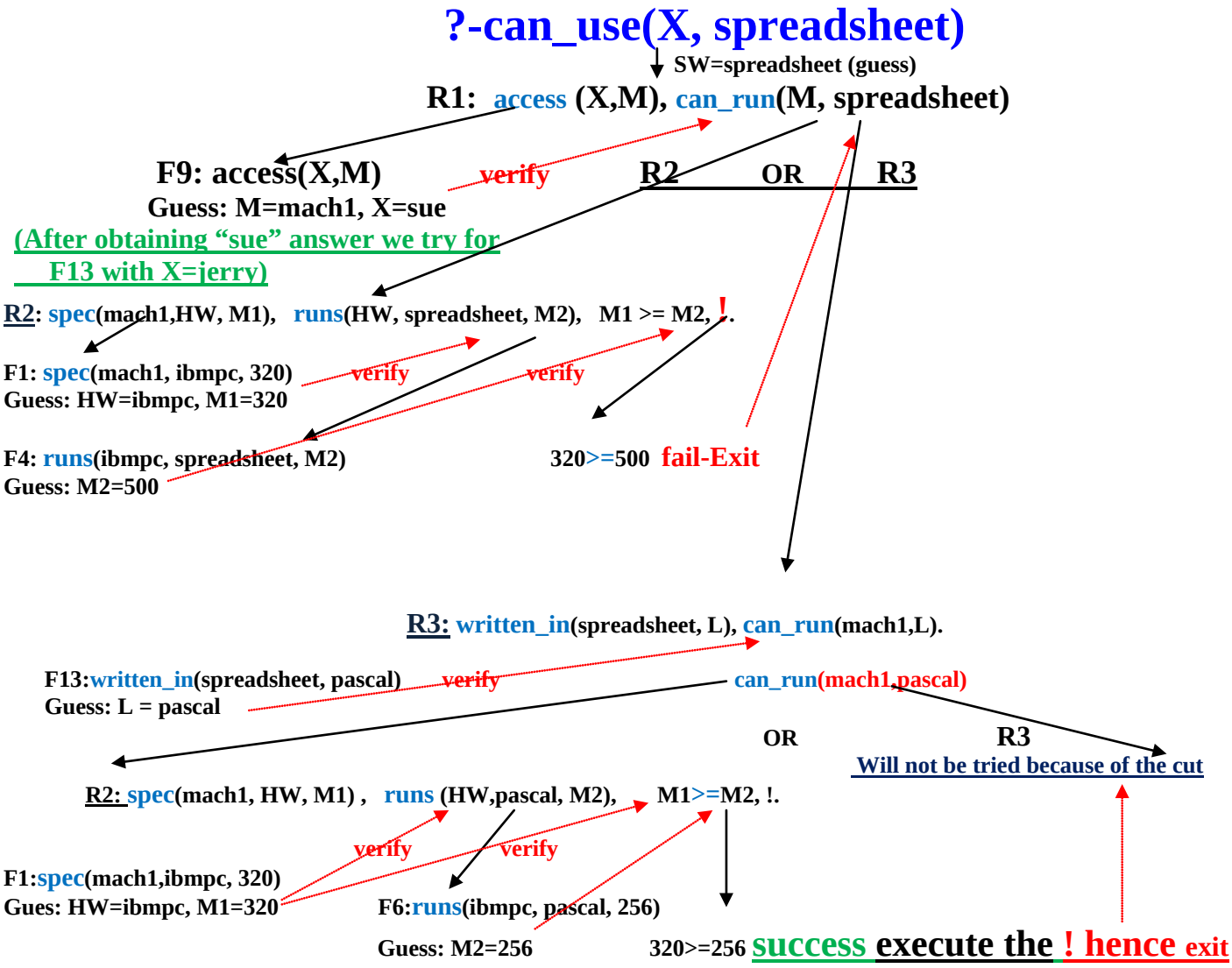
Hence, in such position of the cut, **no** solution would be found.

Prolog program for logic model program

```
F1:    spec(mach1,ibmpc,320).
F2:    spec(mach2,mac,1000).
F3:    spec(mach3,ibmpc,640).
F4:    runs(ibmpc,spreadsheet,500).
F5:    runs(ibmpc,basic,128).
F6:    runs(ibmpc,pascal,256).
F7:    runs(mac,basic,200).
F8:    runs(mac,smalltalk,1000).
F9:    access(sue,mach1).
F10:   access(jerry,mach3).
F11:   access(sam,mach1).
F12:   access(sam,mach2).
F13:   written_in(spreadsheet,pascal).
R1:    can_use(P,SW) :- access(P,M),
                        can_run(M,SW).
R2:    can_run(M,SW) :- spec(M,HW,Mem1),
                           runs(HW,SW,Mem2),
                           Mem1>=Mem2 ,!.
R3:    can_run(M,SW) :- written_in(SW,L),can_run(M,L).
```

```
?-can_use(X,spreadsheet).
```

Goal.



X = sue is a solution

There are more solutions (trying another resolution for "*access*" ,e.g., F10 guessing "jerry" and verifying it, and so on) but we will stop at this point, continuation is in the example handout.

Negation in Prolog: It is a challenging problem, in ***pure*** LP domain, since the inability to assert a goal does not mean the negation of the goal.

Hence, in Prolog, we are forced to use the "impure" **cut (!)** operator and the system subgoal **fail** in order to implement negation, where both the **cut** and **fail** are always universally asserted "true".

To implement *negation* of X-- not(X):

R1: not(X):- call(X),**!**, **fail.** **%** rule
F1: not(X). **%** fact

The order of writing the above clauses is important; also, the order of the sub-goals in R1 is important; hence **impure LP (mixing logic and control!)**.

If **X is true**, then "call(X)" succeeds, and we proceed to pass the cut, then "**fail**" which is pre-asserted. Hence, its negation, **not(X) fails** (as expected) since X is true. Notice that because we went through the **!** we will not try the other alternative for not(X), i.e., F1, and failure is the result.

But, if **X is *false***, then "call(X)" fails and we do not go through the cut; hence we can try the second clause for "not(X)", which is a fact (**asserted**) and, henec **not(X) is asserted, then *true* is returned** , as expected, since X is ***false***.

Example of the goal deduction process:

Given the following database:
% X is a parent of Y if it is the father or mother of Y.
R1: parent (X,Y) :- father (X,Y).
R2: parent (X,Y) :- mother (X,Y).

R3: grandparent (X,Z) :- parent (X,Y) , parent (Y,Z).
% X is the grand parent of Z if it is the parent of Y and
Y is the parent of Z.

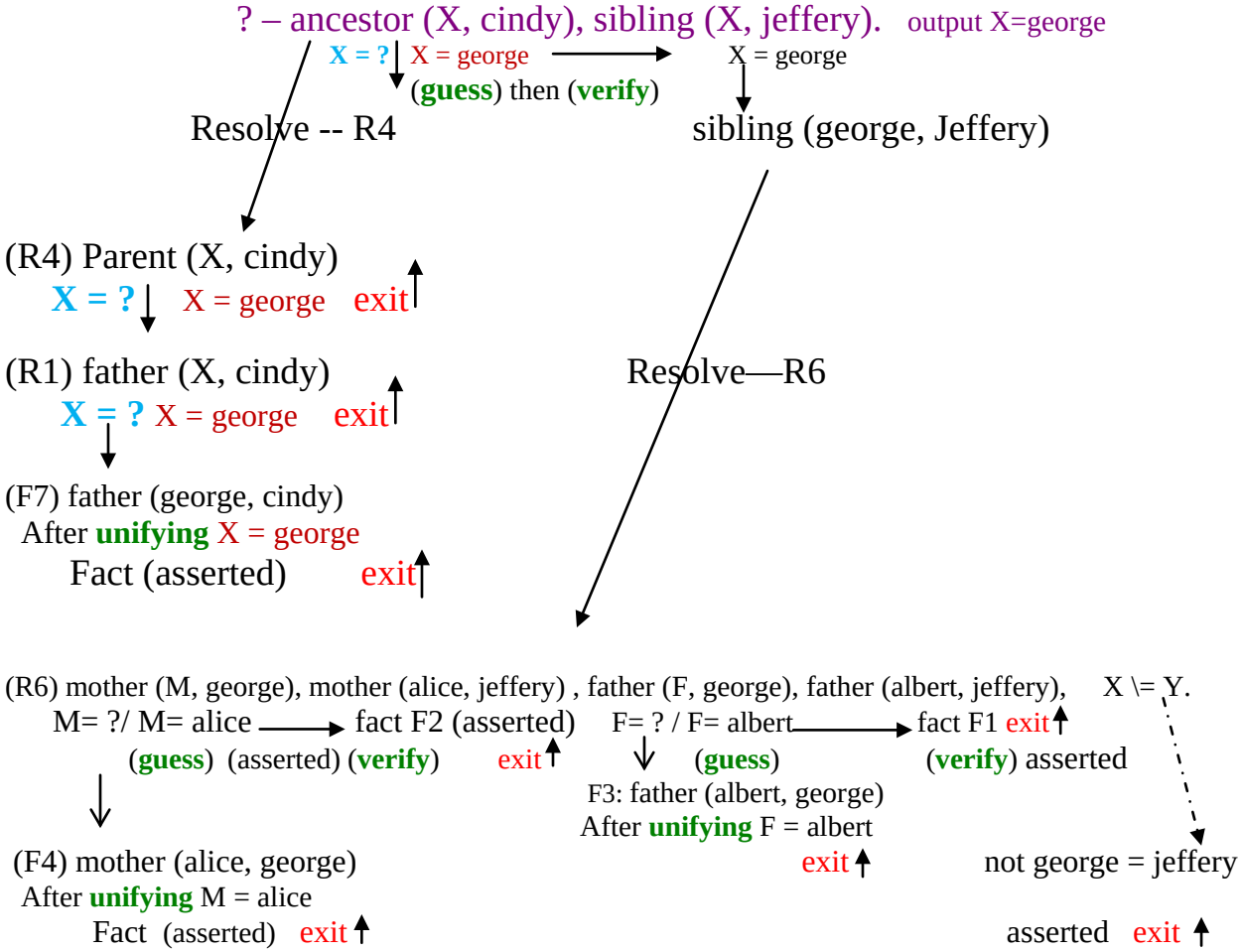
R4: ancestor (X,Z) :- parent (X,Z).
R5: ancestor (X,Z) :- parent (X,Y) , ancestor (Y,Z).

R6: sibling (X,Y) :- mother (M,X) , mother (M,Y) ,
father (F,X) , father (F,Y) , X \= Y.

R7: cousin (X,Y) :- parent (U,X) , parent (V,Y) , sibling (U,V).

F1: father (albert, jeffery).
F2: mother (alice, jeffery).
F3: father (albert, george).
F4: mother (alice, george).
F5: father (john, mary).
F6: mother (sue, mary).
F7: father (george, cindy).
F8: mother (mary, cindy).
F9: father (george, victor).
F10: mother (mary, victor).

Deduce the following goal:



Factorial (N):-

F1: fac(0,1).
R1: fac (N,F) :- N1 := N -1, fac(N1, X1), F := X1 * N.

?- fac(5,N).
 N = 120 ;

What about the deduction of this goal: ?- fac(5,N), N=10.(pp476/7)

system output: **ERROR**: Out of local stack **(problem)**
Since N=10 is always unsatisfied, the system will back track to the last subgoal fac(0,_) and try R1. R1 will lead to calculate "fac(-1,_)" and fail again and try R1 again getting fac(-2,_),... forever!

Solutions: 1) F1:fac(0,1):- ! **2)** R1:fact(N,F):- N>0, …

Binary Search Tree:

Define the term node (K, LS, RS) for tree representation, where K is its root and LS is left-subtree and RS right-subtree.

Operations on BST:

member (K, node (K, _, _)).
                %% "_" is a don't-care, wildcard.
member (K, node (N, LS, _)) :- K<N, member (K,LS).
member (K, node (N, _,RS)) :- K>N, member (K,RS).

List Membership:

member (E, [E, _]). %% the element E is found at the head of the list (exit)

member (E, [_,T]) :- member (E, T).
        %% if E is not at the list head, search the tail T recursively

Defining the predicate "not-in (Element, List)" using the "!" operator:

not_in (E, []) :- **!**.

%% it is a rule (syntactically) but also a fact (semantically).
%% if the list is empty, it means that we looked all of its elements and we did not find the E, then the recursive call below matches this rule (fact) and we execute the cut. Hence, "not_in" is asserted, i.e., it is true that the element is not in the list.

not_in (E, [Head | Tail]) :- E <>Head, not_in (E,Tail).

%% if the list is not empty, this rule matches, and it is asserted if the element E is not the head (i.e., <>), and not in tail (recursive call). But, if E is found to be the head element, then the first subgoal "fails", hence the "not_in" fails, asserting that the searched element E is in the list.