Logic Programming in Prolog

There are important advantages of using programming systems based on logic:

- Data are neatly separated from the inference engine, which is incorporated within the programming system.
- There is a uniform way of representing axioms (i.e. program data) and sentences that play the role of inference rules.
- Data and programs are represented in symbolic form.
- Sentences are largely independent one from another (the program gains in modularity).
- Programs can be dynamically modified and generated.

As a representative of logic programming languages, Prolog offers all of the mentioned advantages. Prolog works with Horn clauses and its inference engine is using the refutation-base resolution. However, apart form FOL, Prolog is a practical programming language. From this perspective, certain characteristics separate neatly Prolog from FOL in the sense that not all the power of FOL can be used with Prolog.

An important characteristic is that Prolog is using depth-first backward chaining to control the inference process. As discussed, this is to say that Prolog cannot be used as a complete theorem prover. For example, the crime story from the previous lecture cannot run as it is in Prolog. It will cycle forever. On the other hand the depth-first controlled of inference save space and can be efficiently implemented as a backtracking procedure. Apart from being incomplete as a proof system, Prolog may allow unsound inferences to take place. This is due to omitting the occurrence test from the unification process. However, an informed programmer can avoid the mentioned traps and can program complex applications with only very few lines of code at efficiency level that nears that of the C language.

Basic Prolog

A Prolog program consists of a sequence of INF sentences - called clauses - that are implicitly conjoined. An implication $q_1 \wedge q_2 \wedge ... \wedge q_n \Rightarrow q$ is called a *rule* and is written

$$q :- q_1, q_2, ..., q_n.$$

The terms q_i, which are the antecedents of the rule, form the *body* of the rule; q is the consequent (or the *head*) of the rule. Neither q_i nor q can be negated. Negation has a particular meaning in Prolog as discussed later. An implication $true \Rightarrow q$ is called a *fact* (or a unit clause) and is represented as: `q.` The syntax of a clause is:

```
term ::= variable | atom | number | structure
structure ::= functor(term,...)
variable ::= an identifier starting with an upper case letter or with _
number ::= an integer constant
atom ::= an identifier starting with a lower case letter

head ::= structure
body ::= structure | structure,body
clause ::= head :- body. | head.
```

Atoms stand for constants. Functors stand both for predicate and function identifiers. Prolog makes no difference between predicate identifiers and function identifiers. In the Prolog term `sell(seller(m1,foo),m1,X)` `m1,foo` are atoms while `sell` and `seller` are functors; `X` is a variable.

As a simple example of a Prolog program consider deciding on the grandparent relationship of two persons.

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
parent(bibo,foo).
parent(bibo,fred).
parent(fred,dino).
```

The program corresponds to the following FOL description, written in INF:

$$\texttt{parent(X,Z)} \wedge \texttt{parent(Z,Y)} \Rightarrow \texttt{grandparent(X,Y)}$$
$$\texttt{true} \Rightarrow \texttt{parent(bibo,foo)}$$
$$\texttt{true} \Rightarrow \texttt{parent(bibo,fred)}$$
$$\texttt{true} \Rightarrow \texttt{parent(fred,dino)}$$

The way Prolog is working to prove the goal **grandparent(bibo,dino)** is by depth-first controlled resolution starting with the clause :- **grandparent(bibo,dino)**. Prolog is maintaining a goal list and, at each inference step, the goal at the top of the goal list is solved. The clauses are traversed sequentially in the order they appear in the program. If the head of a clause matches (unifies) the current goal, then the body of the clause is pushed at the top of the goal list, replacing the matched goal. The inference backtracks when no sentence can match the current goal. The process terminates successfully when the list of goals is empty; it fails when the bottom goal in the goal list cannot be satisfied.

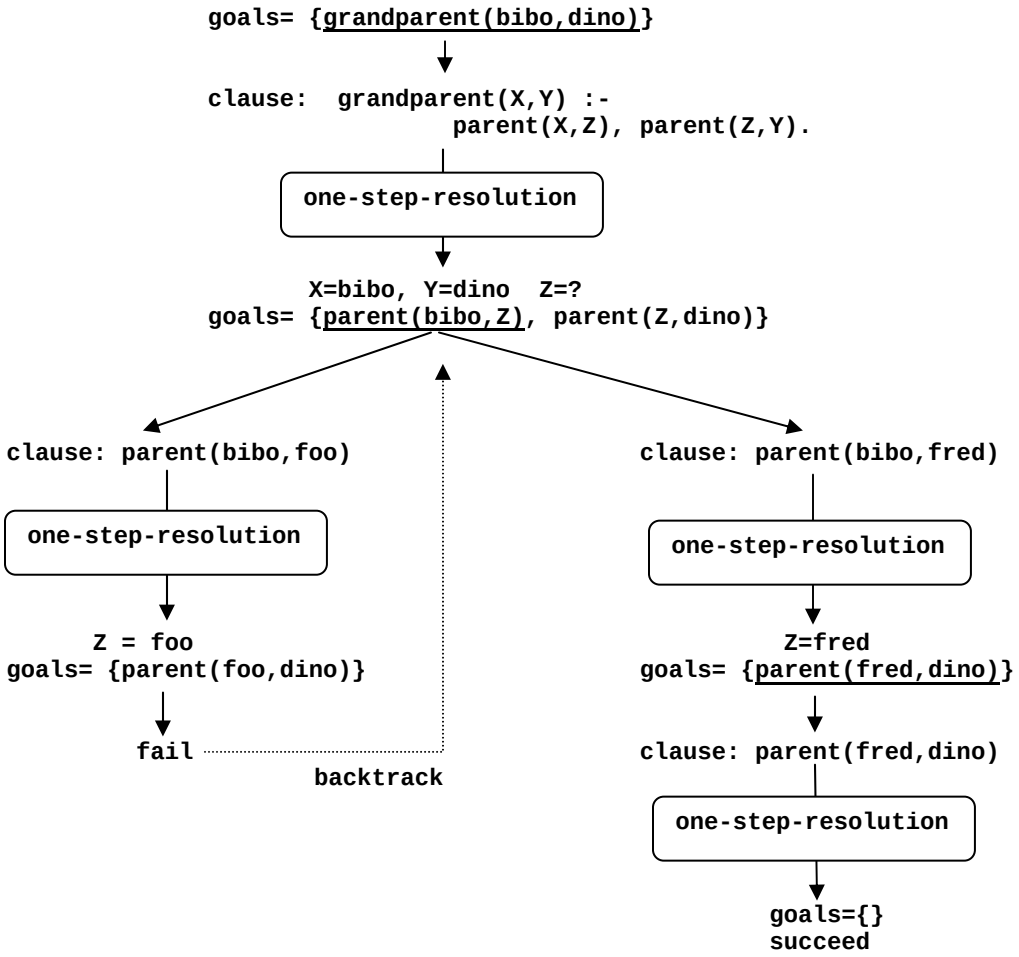

Figure 1. Depth-first proof of a goal

The general description of the inference process in figure 1 is encoded as the Caml function **Backward_chaining** from the previous lecture, slightly modified for depth-first traversal of the proof tree.

```
type 'subst Result = fail | succeed of 'subst list;;

let Backward_chaining all_rules goal =
    let rec solve =
        fun []    rules substitution -> succeed(substitution)
         |  goals []    substitution -> fail
         |  (goal::rest_goals as goals) (rule::rest_rules) substitution ->
                match unify(goal,head(rule),substitution)
                with
                    fail -> solve goals rest_rules substitution
                 |  succeed(new_substitution) ->
                        match solve ((body rule)@rest_goals) all_rules
                                    new_substitution
                        with
                            fail -> solve goals rest_rules substitution
                          | succeed(new_subst) -> succeed(new_subst)
    in
       solve [goal] all_rules [];;
```

It is obvious that the order of clauses in a Prolog program matters. Equally important is the order of the terms in the body of a rule. If the clause **parent(bibo,fred)** would be in front of the clause **parent(bibo,foo)** then the proof of the goal **grandparent (bibo,dino)** does not backtrack. In other cases, a wrong order of rules can make the program loop forever.

Apart from the structures defined by the user, Prolog is able to work with lists. The empty list is designated by **[]**. A non-empty list is specified by enumerating its head elements and, eventually, its tail.

```
list ::= [ list_head | list_tail ]
list_head ::= term, term,...,term
list_tail ::= variable | []
```

For instance, the notation **[a,b,c | []]** is equivalent to **[a,b,c]**. The head and/or the tail of a list can be specified using variables. The list **[H | T]** is interpreted as a list with an element identified by **H** as the head, and some other elements identified collectively by **T** as the tail. As an example of list processing, consider appending two lists.

```
concat([A|R],L,[A|Result]):- concat(R,L,Result).
concat([],L,L).

test :- concat(X,Y,[1,2]),
        write(X), write(‘ ‘), write(Y), nl, fail.
```

When executed with the goal **test**, the program produces the following output, which enumerates all the possible alternatives to producing the list **[1,2]** by appending two lists.

```
[1, 2] []
[1] [2]
[] [1, 2]
```

The example illustrates a valuable characteristic of Prolog programs. With the same program we are able to compute the program output as a function of given input and, moreover, we are able to derive the right input that corresponds to an expected output. Not all Prolog programs behave automatically in this way. They have to be written on purpose for being bi-directional.

Controlling backtracking

The main implicit control process performed in Prolog is backtracking. When a goal fails to be satisfied for the current proof state, Prolog backtracks to find another possible alternative for satisfying the current goal (another sub-tree in the AND-OR proof tree). This is a powerful and useful device. However, in some cases the backtracking action should be explicitly forced or inhibited for selected goals. The control of the backtracking engine, and implicitly, the control of the program behavior, is performed using predefined control predicates and operators.

An important control predicate has been used in the program above: **`fail`**. When "executed" from within a clause it unconditionally fails the proof of the goal that matched the head of the clause. Opposite to **`fail`**, another built-in predicate, called **`true`**, always succeeds.

Another remarkable control mechanism is provided by the cut operator **`!`**. Consider that the *parent goal* is the goal that matches the head of the rule that contains the cut operator. All the goals from the top of the rule to the cut symbol are subject to backtracking as long as the cut is not processed. When the cut is processed it succeeds instantly and the goals following the cut symbol in the rule are subject to backtracking. However, if the backtracking resumes the processing of cut then the parent goal is failed instantly and all the remaining alternatives of satisfying the parent goal are discarded.

As an example of using **`fail`** and cut, consider the implementation of the Prolog **`not`** operator.

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

The meaning of **`not`** is different from the meaning of the logical negation. In Prolog, **`not(P)`** simply means that the goal **`P`** cannot be satisfied in the current program. This has nothing to do with the truth value that can be associated to **`P`**. Negation is treated as failure considering the *closed world* assumption. The program works as follows:

- If the first rule is used to prove **`not(P)`** and the goal **`P`** can be satisfied then **`not(P)`** will fail and, as the effect of cut, the second rule will not be considered as an alternative for satisfying the goal. Therefore **`not(P)`** will definitely fail.

- If the first rule is used but the goal **`P`** cannot be satisfied (the cut is not active), then the second rule is considered as an alternative of satisfying **`not(P)`** and the goal will succeed.

The cut mechanism plays an important role for making programs efficient and deterministic. Consider the Prolog procedure below that tests for the membership of an element within a list.

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

The program is non-deterministic. If **`X`** occurs several times in **`L`** any occurrence can be found when the goal that uses **`member`** fails. Assume we are interested only in a single occurrence, say the first one. To make the program behave in this deterministic way, a cut is used. It inhibits backtracking as soon as the first occurrence of **`X`** is found in **`L`**.

```
memberchk(X,[X|_] :- !.
memberchk(X,[Y|L]) :- memberchk[X,L].
```

Some Prolog implementations provide both variants of membership testing function. As illustrated in the sequel the non-deterministic variant can prove useful in some cases.

Declarative versus procedural semantics

Each rule **h :- p_1,p_2,...,p_n.** of a Prolog program has two possible interpretations: a declarative interpretation and a procedural interpretation.

- The declarative reading of the rule is: the goal **h** is satisfied if all goals **p_i** are satisfied.
- Procedurally, the rule reads: in order to satisfy the goal **h** satisfy the sub-goals **p_1,p_2,...,p_n** from left to right. When trying to satisfy goal **p_i** use the clauses of the program in the order they appear in the program.

The declarative interpretation forces a particular (declarative) programming style. The procedural interpretation is useful when efficiency problems are to be dealt with or when a special behavior of the program is thought for, as in the case of membership testing above.

As a first example of declarative style programming in Prolog consider quick-sorting a list of numbers.

```
quick-sort([],[]).

quick-sort([X | Unsorted], Sorted) :-
     partition(Unsorted,X,Left,Right),
     quick-sort(Left,Sorted_left),
     quick-sort(Right,Sorted_right),
     append(Sorted_left,[X | Sorted_right], Sorted).

partition([],_,[],[]).

partition([X|L],Y,[X|L1],L2) :- X < Y,
                                partition(L,Y,L1,L2).

partition([X|L],Y,L1,[X|L2]) :- X >= Y,
                                partition(L,Y,L1,L2).
```

The program encodes almost directly the following declarative description of the quick-sort algorithm for sorting a list of numbers.

- The empty list is sorted.
- If a non empty list **L** can be partitioned according to an element **X** into two partitions, **Left** and **Right** such that the elements from **Left** are smaller than **X** and the elements from **Right** are greater than or equal to **X** and if **Sorted_left** and **Sorted_right** are the sorted lists corresponding to **Left** and **Right** then the sorted list corresponding to **L** is obtained by appending the lists **Sorted_left**, **[X]**, and **Sorted_right** in the mentioned order.
- The **Left** and **Right** partitions of an empty list are empty.
- Consider a list **List** with the head **X** and the tail **L** such that the lists **L1** and **L2** are the result of partitioning the tail **L** according to an element **Y**. Then:

  a) If **X < Y**, the list obtained by inserting **X** at the top of **L1** is the **Left** partition of **List**, and **L2** is the **Right** partition of **List** according to **Y**.

b) If **X** ≥ **Y**, the list obtained by inserting **X** at the top of **L2** is the **Right** partition of **List**, and **L1** is the **Left** partition of **List** according to **Y**.

Although the clauses of the program can also be read in a procedural fashion, the declarative reading fits better the program.

As a second example of Prolog declarative programming style consider the **n×n-Queens** problem: place **n** non-attacking queens on an **n×n** chessboard. First call **k×n-Queens** the problem of placing **k** non-attacking queens on an **n×n** chessboard. The reasoning that encourages the declarative programming style is the following.

- An empty chessboard is a solution of the **0×n-Queens** problem.
- A queen **q** placed on the chessboard is part of the solution if it does not attack the non-attacking queens placed already on the chessboard.
- Two queens are attacking one another if they are placed in the same line or in the same column or in the same diagonal of the chessboard. The queens are placed in the same diagonal if the absolute value of their line difference is equal to the absolute value of their column difference.

Observe that the way of computing a partial solution is not specified. Only the properties of a solution are outlined. However, the above pure declarative specification cannot be coded straight away in Prolog. The second rule needs more insight: what is it meant by the non-attacking queens already placed on the board? The following rewriting of the rule explicates the meaning.

- A queen **q** placed on chessboard is part of the solution for the **n×n-Queens** problem if it does not attack the queens that form a solution of the **(n-1)×n-Queens** problem.

The meaning is now clear but, at the same time, we introduced some procedural bias in our story. The rule can be read: to solve the **n×n-Queens** problem, solve the **(n-1)×n-Queens** problem and then try to place the n^{th} queen. We will add to the procedural bias by labeling the queens and assuming that the queen **i** is placed on the i^{th} chessboard line. The assumption helps represent the **k×n-Queens** problem as **solution(Queens,Chessboard)**.

The variable **Queens** is bound to the list $[l_1/\_, l_2/\_, ..., l_k/\_]$, where the term $l_i/\_$ corresponds to the queen from line l_i. The anonymous variable _ has to be bound to the column occupied by the queen. Which are the **k** queens that are placed is not at all important. What is important is that $l_i \neq l_j$ for $i \neq j$.

The **Chessboard** is a template encoded as the list **[1/_, 2/_, ..., n/_]**, where the term **j/_** corresponds to the j^{th} column, and the anonymous variable _ is a placeholder for the queen that will occupy this column. The template has only one role: to specify the available columns. The anonymous variables have no importance and they will stay unbound.

The program that follows is a Prolog transcription of the reasoning above. Apart from the conjunction of terms in a clause, it uses the disjunction operator, represented as a semicolon. The term **(C1 =:= C; abs(C1-C) =:= abs(L1-L))** reads: the numerical value of **C1** is equal to the value of **C** *or* the numerical values of the expressions **abs(C1-C)** and **abs(L1-L)** are equal. The disjunction helps condense several rules into a single rule. The program is mostly declarative except for the head and the first term of the second rule.

```
solution([L/C | Queens], Chessboard):-
     solution(Queens,Chessboard), place the queen from line L in column C
```

They show that for solving the **k×n-Queens** problem we have to solve the **(k-1)×n-Queens** problem first and, afterwards, to place the additional queen.

```
solution([],_).

solution([L/C | Queens], Chessboard):-
     solution(Queens,Chessboard),
     member(C/_,Chessboard),
     not(attacks(L/C,Queens)).

attacks(L/C,Queens) :-
      member(L1/C1,Queens),
      (C1 =:= C; abs(C1-C) =:= abs(L1-L)).

solve(S):- solution(S,S),write(S),nl,fail.
```

The high declarative level of the program is partly due to the non-deterministic behavior of the **member** predicate. Since the variables of the term **C/_** are not bound when the term is searched in the **Chessboard** list, **member** enumerates all the terms from the list, binding **C** to the integers **1,2,...** until a solution is found. The program prints all possible solutions.

```
? - solve([1/_,2/_,3/_,4/_]).
[1/3, 2/1, 3/4, 4/2]
[1/2, 2/4, 3/1, 4/3]
```

The program above forces the filling of some free slots in a given template. The user must be aware of the template structure. We consider writing a user-friendly variant of the program. The program must hide the chessboard template and use instead a simpler initial goal of the form **solve(n,S)**, where **n** is the dimension of the chessboard and **S** is the variable that represents the solution. The program is more procedural than the first variant in the way the **non_attacks** predicate is programmed.

```
solution([],_,[]).

solution([L | Lines],Columns,[L/C | Queens]):-
     solution(Lines,Columns,Queens),
     member(C,Columns),
     non_attacks(L/C,Queens).

non_attacks(_,[]).

non_attacks(L/C,[L1/C1 | Queens]) :-
      C1 =\= C,
      L1-L =\= abs(C1-C),
      non_attacks(L/C,Queens).

columns(N,Limit,[]):- N > Limit, !.

columns(N,Limit, [N | Columns]) :-
    N1 is N+1,
    columns(N1,Limit,Columns).

solve(N,S):-
    columns(1,N,Positions),
    solution(Positions,Positions,S),
    write(S),nl,fail.

?- solve(4,S).
[1/3, 2/1, 3/4, 4/2]
[1/2, 2/4, 3/1, 4/3]
```

Meta-Programming

A meta-program takes programs as input data. A meta-interpreter is a meta-program written in a language **L** that executes programs written in the same language **L**. A meta-interpreter apparently modifies in a specific way the behavior of the core machine able to run **L** programs. Prolog is able of such a performance with an outstanding conciseness and clarity. As an example, consider the variant below of the meta-interpreter of Prolog from the Ivan Bratko's book "Prolog programming for Artificial Intelligence" (third edition), Addison Wesley 2001. The meta-interpreter "executes" the program rules and moreover, traces the action it is performing. It is modified to work with the built-in functors from SWI-Prolog. The following messages are displayed:

- **Call *goal***: means that the ***goal*** is going to be solved.
- **Exit goal**: the solving process of the ***goal*** has succeeded.
- **Fail *goal***: failing to solve the ***goal***.
- **Redo goal**: forcing backtracking to find other possible solutions for the ***goal***.

The interpreter uses three Prolog built-in (powerful) functors:

1. **clause(Goal,Body)** - finds the next rule the head of which unifies with the **Goal** and binds the variable **Body** to the body of that rule. For a unit clause (a fact) the **Body** is bound to **true**. For a rule the **Body** is bound to a conjunction of terms. The terms can be accessed using the functor explained next.

2. **,(Goal1,RestGoals)** - splits a conjunction of goals into a first goal, bound to the variable **Goal1**, and the rest of the conjunction, which is bound to the variable **RestGoals**. The functor has an infix form. We can write simply **(Goal1,RestGoals)**.

3. **call(Goal)** – invokes the Prolog inference engine for solving the goal that is the value of the variable **Goal**.

The meta-interpreter uses an additional predicate, specific to SWI-Prolog. The predicate **predicate_property(Goal,built_in)** succeeds if the principal functor of the **Goal** is a built-in functor. For example, **+,-, >, <** etc. are built-in functors. A structure with a principal built-in functor cannot be processed using the mechanism provided by **clause**. Instead, the term **call(Goal)** directly invokes Prolog for processing the **Goal**.

For a nice indentation of messages, as a function of the depth of the traced goals, the interpreter is maintaining throughout the processing the **Depth** information of the processed goals.

```
% A simple meta-interpreter of Prolog

trace_goal(Goal) :-                     % Initialize processing
    trace_goal(Goal,0).                 % Depth is 0

trace_goal(true,_) :- !.                % An unit clause succeeds once only

trace_goal((Goal1,Goals),Depth) :- !,   % Process a conjunction of goals.
    trace_goal(Goal1,Depth),
    trace_goal(Goals,Depth).

trace_goal(Goal,Depth) :-               % Process a single goal
    display('Call: ',Goal,Depth),
    process(Goal,Depth),
    display('Exit: ',Goal,Depth),
    redo_goal(Goal,Depth).
```

```
trace_goal(Goal,Depth) :-                % End of rules that can be used
    display('Fail: ',Goal,Depth),        % to satisfy the Goal
    fail.

process(Goal,_) :-                       % Goal with a main built-in functor
    predicate_property(Goal,built_in),   % Call Prolog to solve it
    !,call(Goal).

process(Goal,Depth) :-                   % Goal with a non built-in functor
    clause(Goal,Body),                   % Find next rule that unify Goal
    Depth1 is Depth+1,
    trace_goal(Body,Depth1).             % Solve the body of the rule

display(Message,Goal,Depth) :-
    tab(2*Depth),
    write(Message),
    write(Goal),nl.

redo_goal(Goal,Depth) :-                 % Succeed on the first call from
    true ;                               % the rule trace_goal. On the second
    display('Redo: ',Goal,Depth),        % call fail, thus forcing Prolog to
    fail.                                % backtrack in order to find other
                                         % possible solutions for Goal
```

The interpreter works well with most programs apart from those that are using the cut and, implicitly, the negation predicate. When fails, the interpreted cut has no control effect on the backtracking process of the interpreted program. As an example of using the interpreter to trace a program, consider the **concat** predicate for appending two lists.

```
concat([],L,L).
concat([A|R],L,[A|Result]):- concat(R,L,Result).
```

```
?- trace_goal(concat(X,Y,[1,2])).

Call: concat(_G318, _G319, [1, 2])
Exit: concat([], [1, 2], [1, 2])
X = []
Y = [1, 2] ;

Redo: concat([], [1, 2], [1, 2])
  Call: concat(_G406, _G319, [2])
  Exit: concat([], [2], [2])
Exit: concat([1], [2], [1, 2])
X = [1]
Y = [2] ;
```

```
Redo: concat([1], [2], [1, 2])
  Redo: concat([], [2], [2])
    Call: concat(_G433, _G319, [])
    Exit: concat([], [], [])
  Exit: concat([2], [], [2])
Exit: concat([1, 2], [], [1, 2])
X = [1, 2]
Y = [] ;

Redo: concat([1, 2], [], [1, 2])
  Redo: concat([2], [], [2])
    Redo: concat([], [], [])
    Fail: concat(_G433, _G319, [])
  Fail: concat(_G406, _G319, [2])
Fail: concat(_G318, _G319, [1, 2])
No
```

Prolog is not the only language that can be used as a platform for easy prototyping. Scheme, Lisp and functional languages in the ML category are also good in this respect although, comparatively, the conciseness of Prolog is remarkable. Nevertheless, it is difficult to say that Prolog is more powerful than other languages that offer symbolic processing mechanisms.

On one hand Prolog programs are difficult to control. A slight modification can have dramatic effects on the logic of the program as shown in the sequel. On other hand, Prolog programming requires training and a way of thinking that is quite different from what the common programmer is expecting. There are lots of non-obvious tricks. The bonus is the substantial time saving for programming complex applications that include natural language processing, verifying program correctness, language implementation etc.

Programs with problems

It seems that Prolog can easily accept transcriptions of stories from FOL, provided the stories contain Horn clauses only. This is not quite so. An example of program with problems is obtained by transcribing the crime story from the previous lecture in Prolog. Here, the clause

```
person(seller(W,P)) :- person(P), owns(P,W).
```

is a trap for an endless cycle. The goal **person(P)** from the body of this rule is resolved by using the same rule, which generates another goal **person(P)** and so on.

Direct transcription from FOL The program loops for ever	A modified program that eventually terminates

`criminal(P) :-`
`  person(P),weapon(W),person(Q),`
`  hostile(Q),sells(P,W,Q).`
 	
`weapon(M) :- missile(M).`
 	
`person(seller(W,P)) :-`
`  person(P),owns(P,W).`
 	
`sells(seller(W,P),W,P) :-`
`  person(P),owns(P,W).`
 	
`hostile(P) :-`
`  person(P),person(Q),hates(P,Q).`
 	
`owns(foo,m1).`
`missile(m1).`
`person(foo).`
`hates(foo,_).`
	`criminal(P) :-`
	`  person(P),weapon(W),person(Q),`
	`  hostile(Q),sells(P,W,Q).`
	
	`weapon(M) :- missile(M).`
	
	`person(foo).`
	`person(seller(W,P)) :-`
	`  owns(P,W),person(P),!.`
	
	`sells(seller(W,P),W,P) :-`
	`  person(P),owns(P,W).`
	
	`hostile(P) :-`
	`  person(P),person(Q),hates(P,Q).`
	
	`owns(foo,m1).`
	`missile(m1).`
	`hates(foo,_).`

Possible solutions are to change the order of terms in the rule, to change the order of the rules, and, moreover, to inhibit the backtracking as soon as the goal **person(P)** is satisfied. However, these solutions will not work unless there is a fact that satisfies the goal **person(P)** prior to applying any other rule and if **P** is leading to the solution. In addition, the modifications will commit Prolog to finding a single solution, although several solutions may exist.

Another possible modification is to remove completely the term **person(P)** from the rule. But in this case, the story may lose meaning. We want to be sure that a weapon seller can deal only with persons. He does not sell weapons to pussycats. Fortunately, since we are working with the rule

```
criminal(P):- person(P),weapon(W),person(Q),hostile(Q),sells(P,W,Q).
```

we know sure that **P** and **Q** are persons and indeed we can remove the term **person(P)** from the rules with the headers **person(seller(W,P))**, **sells(seller(W,P),W,P)**, and **hostile(P)**. Now the program works almost all right.

```
criminal(P) :-
     person(P),weapon(W),person(Q),hostile(Q),sells(P,W,Q).
weapon(M) :- missile(M).
person(seller(W,P)) :- owns(P,W).
sells(seller(W,P),W,P) :- owns(P,W).
hostile(P) :- person(Q),hates(P,Q).

owns(foo,m1).
missile(m1).
person(foo).
hates(foo,_).
```

Why the program is almost right? What is still wrong? If the program above is executed and Prolog is forced to *redo* the top goal, the same solution is obtained twice.

```
?- criminal(P).
P= seller(m1,foo);
P= seller(m1,foo);
No
```

The reason lies deep in the program. When forced to redo, the backtracking Prolog engine restarts to find other possible alternatives for satisfying the different sub-goals, in the reverse order they were previously solved. Consider that, when the first solution is found, the variables of the main clause

```
criminal(P) :- person(P),weapon(W),person(Q),hostile(Q),sells(P,W,Q).
```

are bound as follows: **`P= seller(m1,foo)`**, **`Q= foo`**, **`W= m1`**. When forced to *redo*, Prolog backtracks on the **`sells(P,W,Q)`** sub-goal to find another possible alternative of satisfying the goal. There is no additional alternative, and the backtracking goes up to the sub-goal **`hostile(foo)`** that is solved by the clause:

```
hostile(P) :- person(Q),hates(P,Q).
```

```
hostile(foo) :- person(seller(m1,foo)), hates(foo,seller(m1,foo)).
hostile(foo) :- person(foo), hates(foo,foo).
```

Both **`seller(m1,foo)`** and **`foo`** are persons since: a) **`person(seller(m1,foo))`** is true if **`owns(foo,m1)`** as indeed is the case, and b) **`person(foo)`** is an axiom. Moreover, we have the axiom **`hates(foo,_)`**, i.e. **`foo`** is so wicked that he hates everything, including his own weapon seller and himself. Indeed, there are two different ways of satisfying the goal **`hostile(foo)`** and it is for this reason there are two identical solutions.

If we are not interested of the multiple reasons why **`foo`** can be hostile, and it is sufficient to know just that he is hostile, then we want that the goal **`hostile(P)`** succeeds once only. The cut symbol does the job.

```
hostile(P) :- person(Q),hates(P,Q),!.
```

With the rule **`hostile(P)`** modified as shown, the program produces a single solution:

```
?- criminal(P).
P= seller(m1,foo);
No
```

Another problem occurs with the cut mechanism. A strong point of Prolog is that a pure declarative program has a declarative meaning that is independent of the order of the rules. The order of the rules can influence only the termination or the efficiency of the program, and not its declarative meaning. This is not the case when the cut mechanism is used. Consider the variant 1 of the program in the table below.

variant 1	*variant 2*	*variant 3*
`p:- a, b.` `p :- c.`	`p:- a, !, b.` `p:- c.`	`p:- c.` `p:- a, !, b.`

The declarative meaning of the program is **`(a ∧ b) ∨ c ⇒ p`**, i.e. if both **`a`** and **`b`** are true or **`c`** is true then **`p`** is true. This meaning is independent of the order of the clauses in the program. Consider now the variant 2. The meaning is changed to **`(a ∧ b) ∨ (~a ∧ c) ⇒ p`**, where **`~a`** means: the goal **`a`** cannot be satisfied in the program. The variant 3 has the initial meaning: **`c ∨ (a ∧ b) ⇒ p`**.

The variants 2 and 3 are sensitive to the order of the rules. The declarative meaning of the program is blurred by its procedural semantics.

The third problem may occur when negation is used. An interesting phenomenon happens in Prolog when negation is used with a goal that contains non-instantiated variables. Consider the program:

```
parrot(pipo).
parrot(nino).
swears(pipo).

decent(X) :- not(swears(X)).

?  parrot(X),decent(X).
X = nino
Yes
```

Here, the variable **X** is already instantiated when the goal **not(swears(X))** is processed. Assume **X=nino**. Since the goal **swears(nino)** cannot be satisfied, the negated goal **not(swears(nino))** succeeds and the conjunction of goals **parrot(X),decent(X)** succeeds with the binding **X=nino**. Now solve the apparently equivalent goal:

```
? decent(X),parrot(X).
No
```

Here when the goal **not(swears(X))** is processed the variable **X** is not yet instantiated. We expect that **not swears(X)** is interpreted as ∃**X.~swears(X)**, where **~g** means "the goal **g** cannot be satisfied in the program". However, the predicate **not** is trying to verify if the goal **swears(X)** cannot be satisfied at all (for any **X**), i.e. the interpretation of the goal **not(swears(X))** is ∀**X.~swears(X)**. Since the goal **swears(X)** can be satisfied by the fact **swears(pipo)** the goal **not(swears(X))** fails. The conjunction of goals **decent(X), parrot(X)** fails as well.

The examples above show that a logic program having Prolog as target should be thought of carefully, according to the limitations and particularities of the language. As already mentioned the real Prolog power must not lure one in taking Prolog programs as FOL descriptions. May be the nicest thing about Prolog is that remarkably compact programs pack large loads of thinking of how to be at peace with logic while computing for practical purpose.