An Introduction to Prolog Programming

Ulle Endriss

King's College London

Contents

What is Prolog?

- Prolog (*pro*gramming in *log*ic) is a logical programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
- Prolog is a *declarative* language: you specify *what* problem you want to solve rather than *how* to solve it.
- Prolog is very useful in *some* problem areas, like artificial intelligence, natural language processing, databases, . . . , but pretty useless in others, like graphics or numerical algorithms.

Literature

- Prolog lecture notes
- I. Bratko. *Prolog Programming for Artificial Intelligence.* 2nd edition, Addison-Wesley Publishing Company, 1990.
- F. W. Clocksin and C. S. Mellish. *Programming in Prolog.* 4th edition, Springer-Verlag, 1994.
- L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
- any other textbook on Prolog

Facts

A little Prolog program consisting of four *facts*:

```
bigger( elephant, horse).
bigger( horse, donkey).
bigger( donkey, dog).
bigger( donkey, monkey).
```


Queries

After compilation we can *query* the Prolog system:

```
?- bigger( donkey, dog).
Yes

?- bigger( monkey, elephant).
No
```

A Problem

The following query does not succeed!

```
?- bigger( elephant, monkey).
No
```

The *predicate* `bigger/2` apparently is not quite what we want.

What we'd really like is the transitive closure of `bigger/2`. In other words: a predicate that succeeds whenever it is possible to go from the first animal to the second by iterating the previously defined facts.

Rules

The following two *rules* define `is_bigger/2` as the transitive closure of `bigger/2` (via recursion):

```
is_bigger( X, Y) :- bigger( X, Y).

is_bigger( X, Y) :- bigger( X, Z), is_bigger( Z, Y).
                 ↑                ↑
                "if"            "and"
```

Now it works

```
?- is_bigger( elephant, monkey).
Yes
```

Even better, we can use the *variable* `X`:

```
?- is_bigger( X, donkey).
X = horse ;
X = elephant ;
No
```

Press ; to find alternative solutions. `No` at the end indicates there are no further solutions.

Another Example

Are there any animals which are both smaller than a donkey and bigger than a monkey?

```
?- is_bigger( donkey, X), is_bigger( X, monkey).
No
```

Terms

Prolog *terms* are either *numbers*, *atoms*, *variables*, or *compound terms*.

Atoms start with a lowercase letter or are enclosed in single quotes:

```
elephant, xYZ, a_123, 'Another pint please'
```

Variables start with a capital letter or the underscore:

```
X, Elephant, _G177, MyVariable, _
```


Terms (continued)

Compound terms have a *functor* (an atom) and a number of *arguments* (terms):

```
is_bigger( horse, X), f( g( Alpha, _), 7),
'My Functor'( dog)
```

Atoms and numbers are called *atomic terms*.

Atoms and compound terms are called *predicates*.

Terms without variables are called *ground terms*.

Facts and Rules

Facts are predicates followed by a dot. Facts are used to define something as being unconditionally true.

```
bigger( elephant, horse).
parent( john, mary).
```

Rules consist of a *head* and a *body* separated by `:-`. The head of a rule is true if all predicates in the body can be proved to be true.

```
grandfather( X, Y) :-
  father( X, Z),
  parent( Z, Y).
```


Programs and Queries

Programs. Facts and rules are called *clauses.* A Prolog program is a list of clauses.

Queries are predicates (or sequences of predicates) followed by a dot. They are typed in at the Prolog prompt and cause the system to reply.

```
?- is_bigger( horse, X), is_bigger( X, dog).
X = donkey
Yes
```

Built-in Predicates

- Compiling a program file:

```
?- consult( 'big-animals.pl').
Yes
```

- Writing terms on the screen:

```
?- write( 'Hello World!'), nl.
Hello World!
Yes
```


Matching

Two terms *match* if they are either identical or if they can be made identical by substituting their variables with suitable ground terms.

We can explicitly ask Prolog whether two given terms match by using the equality-predicate `=` (written as an infix operator).

```
?- born( mary, yorkshire) = born( mary, X).
X = yorkshire
Yes
```

The variable instantiations are reported in Prolog's answer.

Matching (continued)

```
?- f( a, g( X, Y)) = f( X, Z), Z = g( W, h( X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes

?- p( X, 2, 2) = p( 1, Y, X).
No
```


The Anonymous Variable

The variable _ (underscore) is called the *anonymous variable*. Every occurrence of _ represents a different variable (which is why instantiations are not reported).

```
?- p( _, 2, 2) = p( 1, Y, _).
Y = 2
Yes
```

Answering Queries

Answering a query means proving that the goal represented by that query can be satisfied (according to the programs currently in memory).

Recall: Programs are lists of facts and rules. A fact declares something as being true. A rule states conditions for a statement being true.

Answering Queries (continued)

- If a goal matches with a *fact*, it is satisfied.
- If a goal matches the *head of a rule*, then it is satisfied if the goal represented by the rule's body is satisfied.
- If a goal consists of several *subgoals* separated by commas, then it is satisfied if all its subgoals are satisfied.
- When trying to satisfy goals with built-in predicates like `write/1` Prolog also performs the associated action (e.g. writing on the screen).

Example: Mortal Philosophers

Consider the following argument:

All men are mortal.

Socrates is a man.

Hence, Socrates is mortal.

It has two *premises* and a *conclusion*.

Translating it into Prolog

The two premises can be expressed as a little Prolog program:

```
mortal( X) :- man( X).
man( socrates).
```

The conclusion can then be formulated as a query:

```
?- mortal( socrates).
Yes
```

Goal Execution

1. The query `mortal( socrates)` is made the initial goal.
2. Prolog looks for the first matching fact or head of rule and finds `mortal( X)`. Variable instantiation: `X = socrates`.
3. This variable instantiation is extended to the rule's body, i.e. `man( X)` becomes `man( socrates)`.
4. New goal: `man( socrates)`.
5. Success, because `man( socrates)` is a fact itself.
6. Therefore, also the initial goal succeeds.

Summary: Syntax

- All Prolog expression are made up from *terms* (numbers, atoms, variables, or compound terms).
- *Atoms* start with lowercase letters or are enclosed in single quotes; *variables* start with capital letters or underscore.
- Prolog programs are lists of *facts* and *rules* (*clauses*).
- *Queries* are submitted to the system to initiate a computation.
- Some *built-in predicates* have special meaning.

Summary: Answering Queries

- When answering a query Prolog tries to prove the corresponding goal's satisfiability. This is done using the rules and facts given in a program.
- A goal is executed by *matching* it with the first possible fact or head of a rule. In the latter case the rule's body becomes the new goal.
- The variable instantiations made during matching are carried along throughout the computation and reported at the end.
- Only the *anonymous variable* _ can be instantiated differently whenever it occurs.

Lists in Prolog

An example for a Prolog list:

```
[elephant, horse, donkey, dog]
```

Lists are enclosed in square brackets. Their elements could be any Prolog terms (including other lists). The empty list is `[]`.

Another example:

```
[a, X, [], f(X,y), 47, [a,b,c], bigger(cow,dog)]
```

Internal Representation

Internally, the list

```
[a, b, c]
```

corresponds to the term

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

That means, this is *just a new notation*. Internally, lists are just compound terms with the functor `.` and the special atom `[]` as an argument on the innermost level.

The Bar Notation

If a bar `|` is put just before the last term in a list, it means that this last term denotes a sub-list. Inserting the elements before the bar at the beginning of the sub-list yields the entire list.

For example, `[a, b, c, d]` is the same as `[a, b | [c, d]]`.

Examples

Extract the second element from a given list:

```
?- [a, b, c, d, e] = [_, X | _].
X = b
Yes
```

Make sure the first element is a `1` and get the sub-list after the second element:

```
?- MyList = [1, 2, 3, 4, 5], MyList = [1, _ | Rest].
MyList = [1, 2, 3, 4, 5]
Rest = [3, 4, 5]
Yes
```


Head and Tail

The first element of a list is called its *head*. The rest of the list is called its *tail*. (The empty list doesn't have a head.)

A special case of the bar notation – with exactly one element before the bar – is called the *head/tail-pattern*. It can be used to extract head and/or tail from a list. Example:

```
?- [elephant, horse, tiger, dog] = [Head | Tail].
Head = elephant
Tail = [horse, tiger, dog]
Yes
```

Head and Tail (continued)

Another example:

```
?- [elephant] = [X | Y].
X = elephant
Y = []
Yes
```

Note: The tail of a list is always a list itself. The head of a list is an element of that list. It doesn't have to be a list itself, but it could be.

Appending Lists

We want to write a predicate `concat_lists/3` to concatenate two given lists.

It should work like this:

```
?- concat_lists( [1, 2, 3, 4], [dog, cow, tiger], L).
L = [1, 2, 3, 4, dog, cow, tiger]
Yes
```

Solution

The predicate `concat_lists/3` is implemented *recursively*. The *base case* is when one of the lists is empty. In every recursion step we take off the head and use the same predicate again, with the (shorter) tail, until we reach the base case.

```
concat_lists( [], List, List).

concat_lists( [Elem|List1], List2, [Elem|List3]) :-
  concat_lists( List1, List2, List3).
```


Do More

Among other things, `concat_lists/3` can also be used for decomposing lists:

```
?- concat_lists( Begin, End, [1, 2, 3]).
Begin = []
End = [1, 2, 3] ;
Begin = [1]
End = [2, 3] ;
Begin = [1, 2]
End = [3] ;
Begin = [1, 2, 3]
End = [] ;
No
```

Built-in Predicates for List Manipulation

`append/3`: Append two lists (same as `concat_lists/3`).

```
?- append( [1, 2, 3], List, [1, 2, 3, 4, 5]).
List = [4, 5]
Yes
```

`length/2`: Get the length of a list.

```
?- length( [tiger, donkey, cow, tiger], N).
N = 4
Yes
```


Membership

`member/2`: Test for membership.

```
?- member( tiger, [dog, tiger, elephant, horse]).
Yes
```

Backtracking into `member/2`:

```
?- member( X, [dog, tiger, elephant]).
X = dog ;
X = tiger ;
X = elephant ;
No
```

Example

Consider the following program:

```
show( List) :-
  member( Element, List),
  write( Element),
  nl,
  fail.
```

Note: `fail` is a built-in predicate that always fails.

What happens when you submit a query like the following one?

```
?- show( [elephant, horse, donkey, dog]).
```


Example (continued)

```
?- show( [elephant, horse, donkey, dog]).
elephant
horse
donkey
dog
No
```

The `fail` at the end of the rule causes Prolog to backtrack. The subgoal `member( Element, List)` is the only choicepoint. In every backtracking-cycle a new element of `List` is matched with the variable `Element`. Eventually, the query fails (`No`).

More Built-in Predicates

`reverse/2`: Reverse the order of elements in a list.

```
?- reverse( [1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

More built-in predicates can be found in the reference manual.

Summary: List Manipulation

- List notation:
  - normal: `[Element1, Element2, Element3]` (empty list: `[]`)
  - internal: `.( Element1, .( Element2, .( Element3, [])))`
  - bar notation: `[Element1, Element2 | Rest]`
  - head/tail-pattern: `[Head | Tail]`
- Many predicates can be implemented recursively, exploiting the head/tail-pattern.
- Built-in predicates: `append/3`, `member/2`, `length/2`, ...

Arithmetic Expressions in Prolog

Prolog comes with a range of predefined arithmetic functions and operators. Something like `3 + 5`, for example, is a valid Prolog term.

So, what's happening here?

```
?- 3 + 5 = 8.
No
```


Matching v Arithmetic Evaluation

The terms `3 + 5` and `8` *do not match.* In fact, when we are interested in the sum of the numbers 3 and 5, we can't get it through matching, but through *arithmetic evaluation.*

We have to use the `is`-operator:

```
?- X is 3 + 5.
X = 8
Yes
```

The is-Operator

The is-operator causes the term to its right to be evaluated as an arithmetic expressions and matches the result of that evaluation with the term on the operator's left. (The term on the left should usually be a variable.)

Example:

```
?- Value is 3 * 4 + 5 * 6, OtherValue is Value / 11.
Value = 42
OtherValue = 3.81818
Yes
```


The is-Operator (continued)

Note that the term to the right will be evaluated to an integer (i.e. not a float) whenever possible:

```
?- X is 3.5 + 4.5.
X = 8
Yes
```

That means, a further subgoal like `X = 8.0` would not succeed.

Example: Length of a List

Instead of using `length/2` we can now write our own predicate to compute the length of a list:

```
len( [], 0).

len( [_ | Tail], N) :-
  len( Tail, N1),
  N is N1 + 1.
```


Functions

Prolog provides a number of built-in *arithmetic functions* that can be used with the `is`-operator. See manual for details.

Examples:

```
?- X is max( 8, 6) - sqrt( 2.25) * 2.
X = 5
Yes

?- X is (47 mod 7) ** 3.
X = 125
Yes
```

Relations

Arithmetic relations are used to compare two arithmetic values. Example:

```
?- 2 * 3 > sqrt( 30).
Yes
```

The following relations are available:

`=:=`	arithmetic equality	`=\=`	arithmetic inequality
`>`	greater	`>=`	greater or equal
`<`	lower	`=<`	lower or equal

Examples

Recall the difference between *matching* and *arithmetic evaluation*:

```
?- 3 + 5 = 5 + 3.
No
?- 3 + 5 =:= 5 + 3.
Yes
```

Recall the *operator precedence* of arithmetics:

```
?- 2 + 3 * 4 =:= (2 + 3) * 4.
No
?- 2 + 3 * 4 =:= 2 + (3 * 4).
Yes
```

Summary: Arithmetics in Prolog

- For logical pattern matching use `=`, for arithmetic evaluation use the `is`-operator.
- A range of built-in arithmetic functions is available (some are written as operators, e.g. `+`).
- Arithmetic expressions can be compared using arithmetic relations like `<` or `=:=` (without `is`-operator).

Operators in Prolog

Operators provide a more convenient way of writing certain terms in Prolog. For example, we can write `3 * 155` instead of `*( 3, 155)` or `N is M + 1` instead of `is( N, +( M, 1))`.

Both notations are considered to be equivalent, i.e. matching works:

```
?- +( 1000, 1) = 1000 + 1.
Yes
```

The objective of this lecture is to show you how you can define your own operators in Prolog.

Operator Precedence

Some operators bind stronger than others. In mathematics, for example, `*` binds stronger than `+`. We also say, `*` has a lower *precedence* than `+`.

In Prolog, operator precedences are numbers (in SWI-Prolog between 0 and 1200). The arithmetical operator `*`, for example, has precedence 400, `+` has precedence 500.

This is why Prolog is able to compute the correct result in the following example (i.e. not 25):

```
?- X is 2 + 3 * 5.
X = 17
Yes
```


Precedence of Terms

The precedence of a term is defined as the precedence of its *principal operator*. If the principal functor isn't (written as) an operator or the term is enclosed in parentheses then the precedence is defined as 0.

Examples:

- The precedence of `3 + 5` is 500.
- The precedence of `3 * 3 + 5 * 5` is also 500.
- The precedence of `sqrt( 3 + 5)` is 0.
- The precedence of `elephant` is 0.
- The precedence of `(3 + 5)` is 0.
- The precedence of `3 * +( 5, 6)` is 400.

Operator Types

Operators can be divided into three groups:

- *infix operators*, like `+` in Prolog
- *prefix operators*, like $\neg$ in logic
- *postfix operators*, like ! in math

Is giving the type of an operator and its precedence already enough for Prolog to fully 'understand' the structure of a term containing that operator?

Example

Consider the following example:

```
?- X is 25 - 10 - 3.
X = 12
Yes
```

Why not 18?

Obviously, precedence and type alone are *not* enough to fully specify the structural properties of an operator.

Operator Associativity

We also have to specify the *associativity* of an operator: `-`, for example, is left-associative. This is why `20 - 10 - 3` is interpreted as `(20 - 10) - 3`.

In Prolog, associativity is represented by atoms like `yfx`. Here `f` indicates the position of the operator (i.e. `yfx` denotes an infix operator) and `x` and `y` indicate the positions of the arguments. A `y` should be read as *on this position a term with a precedence lower or equal to that of the operator has to occur*, whereas `x` means that *on this position a term with a precedence strictly lower to that of the operator has to occur.*

Associativity Patterns

Pattern	**Associativity**		**Examples**
`yfx`	infix	left-associative	`+, -, *`
`xfy`	infix	right-associative	`,` (for subgoals)
`xfx`	infix	non-associative	`=, is, <` (i.e. no nesting)
`yfy`	makes no sense, structuring would be impossible		
`fy`	prefix	associative	
`fx`	prefix	non-associative	`-` (i.e. `--5` not possible)
`yf`	postfix	associative	
`xf`	postfix	non-associative	

Checking Precedence and Associativity

You can use the built-in predicate `current_op/3` to check precedence and associativity of currently defined operators.

```
?- current_op( Prec, Assoc, *).
Prec = 400
Assoc = yfx
Yes

?- current_op( Prec, Assoc, is).
Prec = 700
Assoc = xfx
Yes
```


Defining Operators

New operators are defined using the `op/3`-predicate. This can be done by submitting the operator definition as a query. Terms using the new operator will then be equivalent to terms using the operator as a normal functor, i.e. predicate definitions will work.

For the following example assume the big animals program has previously been compiled:

```
?- op( 400, xfx, is_bigger).
Yes

?- elephant is_bigger dog.
Yes
```

Query Execution at Compilation Time

It is possible to write queries into a program file (using `:-` as a prefix operator). They will be executed whenever the program is compiled.

If for example the file `my-file.pl` contains the line

```
:- write( 'Hello, have a beautiful day!').
```

this will have the following effect:

```
?- consult( 'my-file.pl').
Hello, have a beautiful day!
my-file.pl compiled, 0.00 sec, 224 bytes.
Yes
?-
```


Operator Definition at Compilation Time

You can do the same for operator definitions. For example, the line

```
:- op( 200, fy, small).
```

inside a program file will cause a prefix operator called `small` to be declared whenever the file is compiled. It can be used inside the program itself, in other programs, and in user queries.

Summary: Operators

- The structural properties of an operator are determined by its precedence (a number) and its associativity pattern (like e.g. `yfx`).
- Use `current_op/3` to check operator definitions.
- Use `op/3` to make your own operator definitions.
- Operator definitions are usually included inside a program file as queries (using `:-`, i.e. like a rule without a head).

Backtracking

Choicepoints. Subgoals that can be satisfied in more than one way provide *choicepoints*. Example:

```
..., member( X, [a, b, c]), ...
```

This is a choicepoint, because the variable `X` could be matched with either `a`, `b`, or `c`.

Backtracking. During goal execution Prolog keeps track of choicepoints. If a particular path turns out to be a failure, it jumps back to the most recent choicepoint and tries the next alternative. This process is known as *backtracking*.

Example

Given a list in the first argument, the predicate `permutation/2` generates all possible permutations of that list in the second argument through backtracking (if the user presses ; after every solution):

```
permutation( [], []).

permutation( List, [Element | Permutation]) :-
  select( List, Element, Rest),
  permutation( Rest, Permutation).
```


Example (continued)

```
?- permutation( [1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
No
```

Problems with Backtracking

Asking for alternative solutions generates wrong answers for this predicate definition:

```
remove_duplicates( [], []).

remove_duplicates( [Head | Tail], Result) :-
  member( Head, Tail),
  remove_duplicates( Tail, Result).

remove_duplicates( [Head | Tail], [Head | Result]) :-
  remove_duplicates( Tail, Result).
```


Problems with Backtracking (continued)

Example:

```
?- remove_duplicates( [a, b, b, c, a], List).

List = [b, c, a] ;

List = [b, b, c, a] ;

List = [a, b, c, a] ;

List = [a, b, b, c, a] ;

No
```

Introducing Cuts

Sometimes we want to prevent Prolog from backtracking into certain choicepoints, either because the alternatives would yield wrong solutions (like in the previous example) or for efficiency reasons.

This is possible by using a cut, written as `!`. This predefined predicate always succeeds and prevents Prolog from backtracking into subgoals placed *before* the cut inside the same rule body.

Example

The correct program for removing duplicates from a list:

```
remove_duplicates( [], []).

remove_duplicates( [Head | Tail], Result) :-
  member( Head, Tail), !,
  remove_duplicates( Tail, Result).

remove_duplicates( [Head | Tail], [Head | Result]) :-
  remove_duplicates( Tail, Result).
```

Cuts

Parent goal. When executing the subgoals in a rule's body the term *parent goal* refers to the goal that caused the matching of the head of the current rule.

> Whenever a cut is encountered in a rule's body, all choices made between the time that rule's head has been matched with the parent goal and the time the cut is passed are final, i.e. any choicepoints are being discarded.

Exercise

Using cuts (but without using negation), implement a predicate `add/3` to insert an element into a list, if that element isn't already a member of the list. Make sure there are no wrong alternative solutions. Examples:

```
?- add( elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit] ;
No

?- add( donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit] ;
No
```

Solution

```
add( Element, List, List) :-
  member( Element, List), !.

add( Element, List, [Element | List]).
```


Problems with Cuts

The predicate `add/3` does not work as intended when the last argument is already instantiated! Example:

```
?- add( dog, [dog, cat, bird], [dog, dog, cat, bird]).
Yes
```

Summary: Backtracking and Cuts

- *Backtracking* allows Prolog to find all alternative solutions to a given query.
- That is: Prolog provides the search strategy, not the programmer! This is why Prolog is called a *declarative* language.
- Carefully placed *cuts* (`!`) can be used to prevent Prolog from backtracking into certain subgoals. This may make a program more efficient and/or avoid the generation of (wrong) alternative.

Prolog's Answers

Consider the following Prolog program:

```
animal( elephant).
animal( donkey).
animal( tiger).
```

... and the system's reaction to the following queries:

```
?- animal( donkey).
Yes

?- animal( duckbill).
No
```

The Closed World Assumption

In Prolog, `Yes` means a statement is *provably true*. Consequently, `No` means a statement is *not provably true*. This only means that such a statement is *false*, if we assume that all relevant information is present in the respective Prolog program.

For the semantics of Prolog programs we usually do make this assumption. It is called the *Closed World Assumption*: we assume that nothing outside the world described by a particular Prolog program exists (is true).

The \+-Operator

If we are not interested whether a certain goal succeeds, but rather whether it fails, we can use the \+-operator (negation). `\+ Goal` succeeds, if `Goal` fails (and vice versa). Example:

```
?- \+ member( 17, [1, 2, 3, 4, 5]).
Yes
```

This is known as *negation as failure*: Prolog's negation is defined as the failure to provide a proof.

Negation as Failure: Example

Consider the following program:

```
married( peter, lucy).
married( paul, mary).
married( bob, juliet).
married( harry, geraldine).

single( Person) :-
  \+ married( Person, _),
  \+ married( _, Person).
```


Example (continued)

After compilation Prolog reacts as follows:

```
?- single( mary).
No

?- single( claudia).
Yes
```

In the closed world described by our Prolog program Claudia has to be single, because she is not known to be married.

Where to use \+

Note that the \+-operator can only be used to negate *goals*. These are either (sub)goals in the *body of a rule* or (sub)goals of a *query*. We cannot negate facts or the heads of rules, because this would actually constitute a redefinition of the \+-operator (in other words an explicit definition of Prolog's negation, which wouldn't be compatible with the closed world assumption).

Disjunction

We already now know *conjunction* (comma) and *negation* (\+). We also know *disjunction*, because several rules with the same head correspond to a disjunction.

Disjunction can also be implemented directly within one rule by using ; (semicolon). Example:

```
parent( X, Y) :- father( X, Y); mother( X, Y).
```

This is equivalent to the following program:

```
parent( X, Y) :- father( X, Y).

parent( X, Y) :- mother( X, Y).
```

Example

Write a Prolog program to evaluate a row of a truth table. (Assume appropriate operator definitions have been made before.)

Examples:

```
?- true and false.
No

?- true and (true and false implies true) and neg false.
Yes
```


Solution

```
% Falsity
false :- fail.

% Conjunction
and( A, B) :- A, B.

% Disjunction
or( A, B) :- A; B.
```

Solution (continued)

```
% Negation
neg( A) :- \+ A.

% Implication
implies( A, B) :- A, !, B.
implies( _, _).
```


Note

We know that in classical logic $\neg A$ is equivalent to $A \Rightarrow \bot$. Similarly, instead of using `\+` in Prolog we could define our own negation operator as follows:

```
neg( A) :- A, !, fail.
neg( _).
```

Summary: Negation and Disjunction

- *Closed World Assumption*: In Prolog everything that cannot be proven from the given facts and rules is considered false.
- *Negation as Failure*: Prolog's negation is implemented as the failure to provide a proof for a statement.
- Goals can be negated using the `\+`-operator.
- A disjunction of goals can be written using `;` (semicolon).
- (The comma between two subgoals denotes a conjunction.)

Logic and Prolog

Today we shall see how Prolog programs can be interpreted as sets of logic formulas. In fact, when processing a query, Prolog is actually applying the rules of a logical deduction system similar to the goal-directed calculus.

Correspondence

Prolog	**First-order Logic**
predicate	predicate
argument	term
variable	universally quantified variable
atom	constant/function/predicate symbol
sequence of subgoals	conjunction
`:-`	implication (other way round)

Question

What is the logical meaning of this program?

```
bigger( elephant, horse).
bigger( horse, donkey).
is_bigger( X, Y) :- bigger( X, Y).
is_bigger( X, Y) :- bigger( X, Z), is_bigger( Z, Y).
```

Answer

$$\{ \quad bigger(elephant, horse),$$
$$bigger(horse, donkey),$$
$$\forall x.\forall y.(bigger(x,y) \Rightarrow is\_bigger(x,y)),$$
$$\forall x.\forall y.\forall z.(bigger(x,z) \wedge is\_bigger(z,y) \Rightarrow is\_bigger(x,y)) \quad \}$$

Translation of Programs

- Predicates remain the same (syntactically).
- Commas separating subgoals become $\wedge$.
- `:-` becomes $\Rightarrow$ and the order of head and body is changed.
- Every variable is bound to a universal quantifier ($\forall$).

Translation of Queries

Queries are translated like rules; the 'empty head' is translated as $\bot$. This corresponds to the negation of the goal whose provability we try to test when submitting a query to Prolog.

Logically speaking, instead of deriving the goal itself, we try to prove that adding the negation of the goal to the program would make it inconsistent:

$$\mathcal{P}, A \Rightarrow \bot \ \vdash \ \bot \quad \text{iff} \quad \mathcal{P} \ \vdash \ A$$

Example

The query

```
?- is_bigger( elephant, X), is_bigger( X, donkey).
```

corresponds to the following first-order formula:

$$\forall x.(is\_bigger(elephant, x) \land is\_bigger(x, donkey) \Rightarrow \bot)$$

Horn Formulas

The formulas we get when translating all have the same structure:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B$$

Such a formula can be rewritten as follows:

$$\begin{array}{ll} A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B & \equiv \\ \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \ \vee \ B & \equiv \\ \neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_n \vee B & \end{array}$$

Hence, formulas obtained from translating Prolog clauses can always be rewritten as equivalent *Horn formulas* (disjunctions of literals with at most one positive literal).

Resolution

The search tree built up by Prolog when trying to answer a query corresponds to a logic proof using *resolution*, which is a very efficient deduction system for Horn formulas. A short introduction can be found in the notes; for more details refer to theoretically oriented books on logic programming.

It is also possible to think of a Prolog goal execution as a goal-directed proof in first-order logic. The data formulas in such a proof would represent a list of clauses and facts, and a goal formula would correspond to a Prolog query.

Summary: Logic Foundations

- Prolog programs correspond to sets of first-order logic (Horn) formulas.
- During translation, `:-` becomes an implication (from right to left), commas between subgoals correspond to conjunctions, and all variables need to be universally quantified. Queries become (universally quantified) implications with $\perp$ in the consequent.
- Prolog's search to satisfy a query corresponds to a logical proof. In principle, any deduction calculus could be used. Historically, Prolog is based on resolution, which is particularly suited as it is tailored for Horn formulas.