VNU-HUS MAT1206E/3508: Introduction to AI

Logic Programming with PROLOG
In-class Discussion

Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học
Đại học KHTN, ĐHQG Hà Nội
hoanganhduc@hus.edu.vn

Contents

Additional Materials

Learn PROLOG Now!

- `https://www.let.rug.nl/bos/lpn/index.php`
- by **Patrick Blackburn**, **Joost Bos**, and **Kristina Striegnitz**.

Basic of PROLOG

- *PROLOG* = **Pro**gramming in **Log**ic
- PROLOG is used in many projects, primarily in *AI* and *computational linguistics*.
- We give a *short introduction*, show key concepts and strengths, and compare PROLOG with other programming languages and theorem provers.
- For a complete programming course, see [Bratko 2011]; [Clocksin and Mellish 2013].
- *Software note:* common systems are *SWI-Prolog* (`https://www.swi-prolog.org/`) and *GNU Prolog* (`https://www.gprolog.org/`). Here, ordinary examples use SWI-Prolog syntax; CLP uses GNU Prolog.
- Most modern PROLOG systems use an interpreter based on the *Warren Abstract Machine (WAM)*.
- PROLOG source code is compiled into *WAM code*, then interpreted by the WAM.
- *Performance:* up to 10 million logical inferences per second (*LIPS*) on a 1 Gigahertz PC

Basic of PROLOG

- PROLOG is a *declarative programming language*, i.e., the programmer declares *what* the program should accomplish without specifying *how* to achieve the result.
- PROLOG is based on *Horn clauses*.
- A PROLOG program consists of a *knowledge base* (*database*), which is simply a set of *facts* and *rules* about some problem domain.
  - A *knowledge base* KB of family relationships is coded as a PROLOG program

```
1  child(oscar, karen, frank).
2  child(mary, karen, frank).
3  child(eve, anne, oscar).
4  child(henry, anne, oscar).
5  child(isabelle, anne, oscar).
6  child(clyde, mary, oscarb).
```

Basic of PROLOG

- The execution of a PROLOG program is initiated by a *query*, which is answered by proving that the query logically follows from the facts and rules in the program.
  - Example query

```
?- child(eve, anne, oscar).
```

  - The query asks whether `eve` is a child of `anne` and `oscar`.
  - The expected answer is `true` (because it is a fact in the knowledge base KB).
  - How does PROLOG find the answer?
    - PROLOG tries to unify the query with the facts in the knowledge base KB.
    - There are six facts in the knowledge base.
    - Unification is attempted between the query and each of the complementary literals in the input data in order of occurrence. (In this example, the query unifies with the third fact.)
    - If one of the alternatives fails, this results in backtracking to the last branching point, and the next alternative is tested.

Basic of PROLOG

Capitalization changes the meaning of a term in PROLOG.

Quiz

Which term contains a *variable*?

(a) `woman(mia)`

(b) `woman(Mia)`

(c) `woman('Mia')`

Basic of PROLOG

An *atom* is a constant term. Single quotes can make one atom out of text that would otherwise not be a simple atom.

Quiz

Which one is valid PROLOG syntax for a single *atom whose name contains a space*?

(a) `intro ai`

(b) `'intro ai'`

(c) `IntroAi`

Basic of PROLOG

A predicate is identified by its *functor* and *arity* (number of arguments).

Quiz

Are `loves(john, mary)` and `loves(john, mary, secret)` clauses of the same predicate?

(a) Yes, because both use `loves`.

(b) No, they define `loves/2` and `loves/3`.

(c) No, because predicate names cannot repeat.

Basic of PROLOG

Central Ideas of PROLOG

- **SUCCESS (true) / FAILURE (false)**
  - any computation can “succeed” or “fail”, and this is used as a ‘test’ mechanism.
- **UNIFICATION (2-WAY MATCHING)**
  - any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.
- **SEARCHING**
  - the whole activity of the PROLOG system is to search through various options to find a combination that succeeds.
- **BACKTRACKING**
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.

Basic of PROLOG


```
1  teaches(mai, ai).
2  student(an).
3  can_help(T,S):- teaches(T, ai), student(S).
4  ?- can_help(mai, an).
```

Quiz

Which labeling is correct?

(a) Line 1 is a fact, line 3 is a rule, and line 4 is a query.

(b) Line 1 is a query, line 3 is a fact, and line 4 is a rule.

(c) All four lines are facts.

Basic of PROLOG


```
1  woman(mia).
2  woman(jody).
3  woman(yolanda).
4  playsAirGuitar(jody).
5  party.
```

```
?- woman(mia).
?- playsAirGuitar(mia).
?- playsAirGuitar(vincent).
?- woman(vincent).
```

Quiz

For the above queries, which option correctly describes the expected Prolog answers?

(a) All four queries succeed.

(b) Only the first query succeeds.

(c) Only the last query succeeds.

Basic of PROLOG

```
1  happy(yolanda).
2  listens2Music(mia).
3  listens2Music(yolanda):- happy(yolanda).
4  playsAirGuitar(mia):- listens2Music(mia).
5  playsAirGuitar(yolanda):- listens2Music(yolanda).
```

Quiz

Why does `?- playsAirGuitar(yolanda).` succeed?

(a) It is listed as a fact.

(b) It reduces to `listens2Music(yolanda)`, which follows from `happy(yolanda)`.

(c) Every happy person plays air guitar.

Basic of PROLOG

```
1  happy(vincent).
2  listens2Music(butch).
3  playsAirGuitar(vincent):- happy(vincent),
4          listens2Music(vincent).
5  playsAirGuitar(butch):- happy(butch).
6  playsAirGuitar(butch):- listens2Music(butch).
```

Quiz

For these queries, which option correctly describes the expected Prolog answers?

```
?- playsAirGuitar(vincent).
?- playsAirGuitar(butch).
```

(a) Both queries succeed.

(b) Only the second query succeeds.

(c) Only the first query succeeds.

Basic of PROLOG

```
1  woman(mia).
2  woman(jody).
3  woman(yolanda).
```

Quiz

What answers does `?- woman(X).` return?

(a) `X=mia`; then `X=jody`; then `X=yolanda`.

(b) Only `X=mia`.

(c) The query fails.

Basic of PROLOG

```
1  woman(mia).
2  woman(jody).
3  loves(marsellus,mia).
4  loves(pumpkin,honey_bunny).
```

Quiz

What does `?- loves(marsellus, X), woman(X).` return?

(a) `X=mia.`

(b) `X=pumpkin.`

(c) The query fails.

Basic of PROLOG

```
1  loves(vincent,mia).
2  loves(marsellus,mia).
3  loves(pumpkin,honey_bunny).
4  loves(honey_bunny,pumpkin).
5
6  jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

Quiz

According to this rule, what answers can `?- jealous(marsellus, W).` return?

(a) `W=vincent` only.

(b) `W=vincent`; then `W=marsellus`.

(c) The query fails.

Basic of PROLOG

- A *search tree* shows how PROLOG tries to answer a query: successful and failed clause choices are both visible.
- A successful path ending in `[]` corresponds to a proof/answer. The whole tree also shows failed alternatives explored during backtracking.
- It makes execution explicit: *unification*, *rule expansion*, and *backtracking*.

Basic of PROLOG

Follow this consistent procedure when drawing search trees:

1. Write the initial query in a box; treat it as a to-do list of goals.
2. Select the *leftmost* goal in the current box.
3. Try matching clauses for that predicate/arity from *top to bottom*; draw the branches from left to right in the same order.
4. Use a fresh copy of the selected clause each time it is tried.
5. If unification fails, mark the branch as `fail`.
6. If a fact succeeds, remove the selected goal and keep the remaining goals; write `[]` if no goals remain.
7. If a rule succeeds, replace the selected goal by the rule body and keep the remaining goals.

This procedure uses the PROLOG search order described in Learn Prolog Now! [Blackburn, Bos, and Striegnitz 2006]; the search-tree drawing notation follows Bos [Bos 2020].

Basic of PROLOG

- In the examples below, `cl. n` means clause/line `n` in the numbered knowledge base.
- Each box is the current list of remaining goals; the blue goal is the next selected goal.
- We use `fail` for a dead branch and `[]` for a completed branch with no goals left.
- Substitutions such as `X=vincent` are written on edges or at successful leaves.
- Notation may vary, but the tree must consistently follow PROLOG's left-to-right and top-to-bottom search order.

Basic of PROLOG

Example 1 (Search Tree)

Knowledge Base:

```
1  loves(vincent, mia).
2  loves(marsellus, mia).
3  jealous(A, B) :- loves(A, C), loves(B, C).
```

Query:

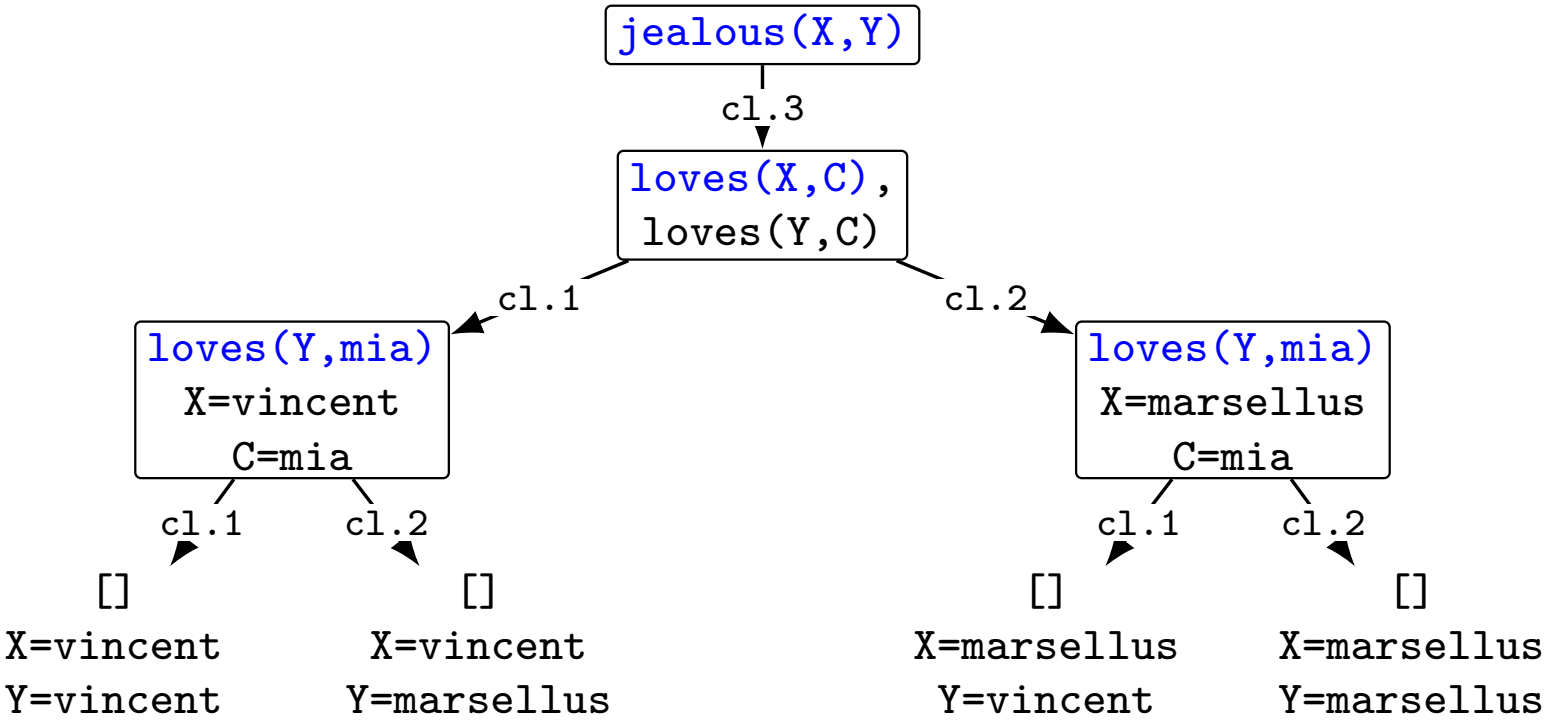
```
?- jealous(X, Y).
```

Basic of PROLOG

Search tree for the query `?- jealous(X, Y).`

```
cl.1: loves(vincent,mia).    cl.2: loves(marsellus,mia).
cl.3: jealous(A,B) :- loves(A,C), loves(B,C).
```

Step 6: solve `loves(Y,mia)` on this branch with `cl.1`, then `cl.2`.



Example 2 (Search Tree)

Knowledge Base:

```
1  loves(vincent,mia).
2  loves(marsellus,mia).
3  loves(pumpkin,honey_bunny).
4  loves(honey_bunny,pumpkin).
5  jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

Query:

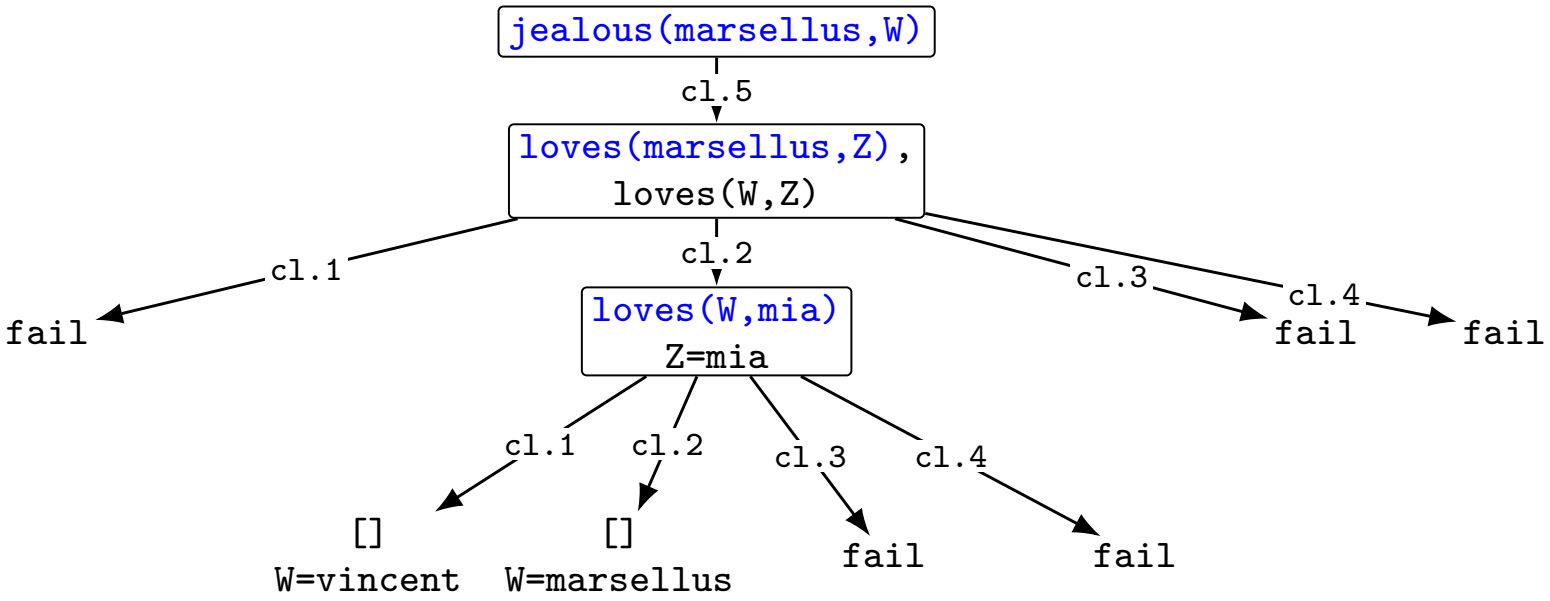
```
?- jealous(marsellus, W).
```

Basic of PROLOG

Search tree for the query `?- jealous(marsellus, W).`

```
cl.1: loves(vincent,mia).    cl.2: loves(marsellus,mia).
cl.3: loves(pumpkin,honey_bunny).    cl.4: loves(honey_bunny,pumpkin).
cl.5: jealous(X,Y) :- loves(X,Z), loves(Y,Z).
```

Step 6: first try `cl.3` and `cl.4` for `loves(W,mia)`; both fail. Then backtrack to `loves(marsellus,Z)` and try `cl.3` and `cl.4`; both fail.



Recursive Definitions
Introduction

- In PROLOG, we can define predicates recursively.
- A recursive definition usually has:
  - At least one *base clause* (non-recursive)
  - At least one *recursive clause*
- For execution, the base clause must be *reachable* under PROLOG's clause order and subgoal order.
- We first use a small eating-and-digestion example, then return to Ertel's family-relationship example.

Recursive Definitions
Example: Eating

Consider the following knowledge base:

```
1  is_digesting(X,Y)  :-  just_ate(X,Y).
2  is_digesting(X,Y)  :-
3          just_ate(X,Z),
4          is_digesting(Z,Y).
5
6  just_ate(mosquito,blood(john)).
7  just_ate(frog,mosquito).
8  just_ate(stork,frog).
```

- Line 1 is the *base clause*: direct eating is enough.
- Lines 2–4 are the *recursive clause*: one direct eating step, then the same question again.

Recursive Definitions
Declarative Meaning

- *Declarative meaning*: The logical meaning of the PROLOG knowledge base
- Base clause (non-recursive):
  - "If `X` has just eaten `Y`, then X is now digesting Y"
- Recursive clause:
  - "If `X` has just eaten `Z` and `Z` is digesting `Y`, then `X` is digesting `Y` too"
- This captures the intuition of indirect digestion through food chains

Recursive Definitions
Procedural Meaning

- *Procedural meaning*: How PROLOG actually executes the queries
- For a ground query such as `is_digesting(stork,mosquito)`, PROLOG:
  - First tries the base rule: "Has `X` just eaten `Y`?"
  - If that fails, tries the recursive rule by finding some `Z` where:
    - `X` has just eaten `Z`, AND
    - `Z` is digesting `Y` (recursive subgoal)
- For queries with variables, PROLOG may also *backtrack* after a success to look for more answers.

Recursive Definitions

Quiz: Answer Order

```
1  is_digesting(X,Y) :- just_ate(X,Y).
2  is_digesting(X,Y) :- just_ate(X,Z), is_digesting(Z,Y).
3  just_ate(mosquito,blood(john)).
4  just_ate(frog,mosquito).
5  just_ate(stork,frog).
```

Quiz

What answers does `?- is_digesting(stork,Y).` return, in order?

(a) `Y=frog`; then `Y=mosquito`; then `Y=blood(john)`.

(b) `Y=blood(john)`; then `Y=mosquito`; then `Y=frog`.

(c) Only `Y=frog`.

Recursive Definitions
The Importance of Base Cases

Warning

A recursive definition needs a *reachable* stopping path.

Consider this dangerous rule:

```
p :- p.
```

- Declaratively: "If property `p` holds, then property `p` holds" (logical)
- Procedurally: Creates an *infinite loop*
  - To prove `p`, I need to prove `p`
  - To prove `p`, I need to prove `p`
  - ...and so on, forever
- A base clause helps only if PROLOG can actually reach it under its search order.

Recursive Definitions

Family Relationships: A Dangerous Symmetry Rule

Consider the following knowledge base, adapted from Ertel's family-relationship example [Ertel 2025], Sec. 5.2.

```
1   child(oscar, karen, frank).
2   child(mary, karen, frank).
3   child(eve, anne, oscar).
4   child(henry, anne, oscar).
5   child(isabelle, anne, oscar).
6   child(clyde, mary, oscarb).
7
8   child(X,Z,Y) :- child(X,Y,Z).
9
10  descendant(X,Y) :- child(X,Y,Z).
11  descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

Now consider the query:

```
?- descendant(clyde,karen).
```

Quiz

What happens with the query `?- descendant(clyde,karen).` in the program above?

(a) PROLOG answers `false`.

(b) PROLOG answers `true` immediately.

(c) PROLOG follows an infinite branch before reaching the useful recursive descendant rule.

Recursive Definitions

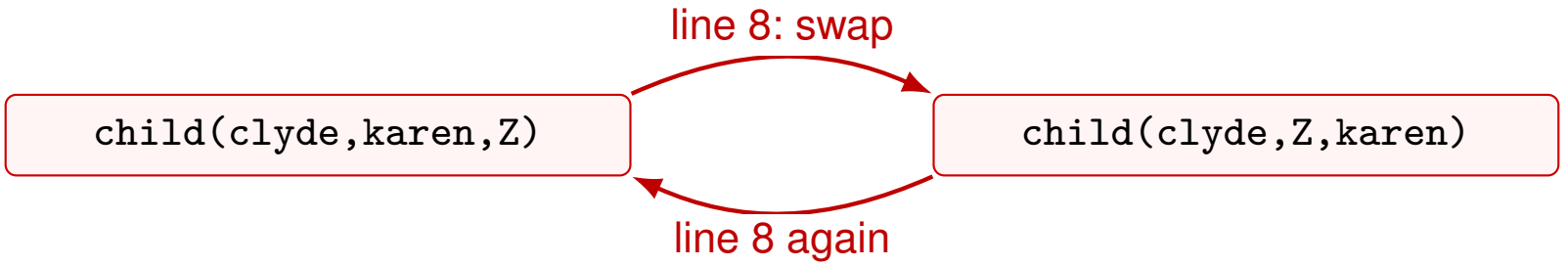
Family Relationships: Where the Loop Comes From

Important distinction

Not answered here means nontermination. It is *not* the answer `false`.

For `?- descendant(clyde,karen).`, PROLOG first tries line 10. If facts do not match the resulting `child` goal, line 8 can fire:

```
...                                    % omitted
 8  child(X,Z,Y) :- child(X,Y,Z).      % swap
...                                    % omitted
10  descendant(X,Y) :- child(X,Y,Z).   % base
...                                    % omitted
```



Recursive Definitions

Family Relationships: First Repair and Regression

The following program is a first workaround for the nonterminating query.

rel01.pl

```
1   child(oscar, karen, frank).
2   child(mary, karen, frank).
3   child(eve, anne, oscar).
4   child(henry, anne, oscar).
5   child(isabelle, anne, oscar).
6   child(clyde, mary, oscarb).
7
8   descendant(X,Y) :- child(X,Y,Z).
9   descendant(X,Y) :- child(X,Z,Y).
10  descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

- The workaround: remove the recursive `child/3` symmetry rule; lines 8–9 test both parent positions directly in `descendant/2`.
- It answers `?- descendant(clyde,karen).` with `true`.
- Remaining problem: `?- child(eve,oscar,anne).` now fails, so this is only a *partial workaround*.

Recursive Definitions

Family Relationships: Separating Facts from Rules

For this finite family example, both shown problems are avoided by separating stored facts from the derived predicate.

rel02.pl

```
1   child_fact(oscar, karen, frank).
2   child_fact(mary, karen, frank).
3   child_fact(eve, anne, oscar).
4   child_fact(henry, anne, oscar).
5   child_fact(isabelle, anne, oscar).
6   child_fact(clyde, mary, oscarb).
7
8   child(X,Z,Y) :- child_fact(X,Y,Z).
9   child(X,Z,Y) :- child_fact(X,Z,Y).
10
11  descendant(X,Y) :- child(X,Y,Z).
12  descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

- `child_fact/3` stores the original facts.
- `child/3` derives both parent orders *without recursively calling itself*.

Quiz

Why does the program with `child_fact/3` avoid the previous `child/3` loop?

(a) Stored facts are separated from the derived symmetric `child/3` predicate.

(b) Recursion is removed from `descendant/2`.

(c) PROLOG stops using depth-first search.

- This repair fixes only this loop: the old symmetric `child/3` rule kept recreating a `child/3` subgoal.
- Other recursive rules can still loop, so a PROLOG programmer must *pay attention to processing* and *avoid infinite loops*.

Execution Control and Procedural Elements

Note

As we have seen in the previous examples, *it is important to control the execution of PROLOG*.

- *Avoiding unnecessary backtracking* especially can lead to large increases in efficiency. One means to this end is the *cut* operator.
- The cut `!` itself does *not* fail: when PROLOG reaches it, `!` succeeds and commits to the choices already made.

```
p(X) :- q(X), !, r(X).
p(X) :- s(X).
```

- If `q(X)` fails, PROLOG never reaches `!`; it can still try the second clause `p(X) :- s(X).`
- If `q(X)` succeeds and PROLOG reaches `!`, it commits to the first clause. If `r(X)` then fails, the second clause is not tried for this call. Thus the *position* of `!` matters [Ertel 2025], Sec. 5.3; [Bratko 2011], Sec. 5.1.

Execution Control and Procedural Elements

Example 3 (Cut operator in PROLOG)

max(X, Y, Max) means "the maximum of two numbers X and Y is *Max*"

max.pl

```
1  max(X,Y,Max) :-
2       X >= Y,
3       Max = X.
4  max(X,Y,Max) :-
5       X < Y,
6       Max = Y.
```

- Without cut.
- Query `?- max(3,2,Z), Z > 10.` first binds `Z=3` for `max(3,2,Z)`. Then `Z > 10` fails; backtracking reaches lines 4–6, where `3 < 2` fails.

maxwCut.pl

```
1  max(X,Y,Max) :-
2       X >= Y,
3       !,
4       Max = X.
5  max(_,Y,Y).
```

- With cut.
- Line 5 (`max(_,Y,Y)`) is tried only if line 2 fails before `!`. For `max(2,3,Z)`, line 2 tests `2 >= 3`; line 5 then gives `Z=3`.
- Cut can make programs harder to read.

Quiz

Consider this cut variant:

```
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).
?- max(3,2,2).
```

What answer does PROLOG give?

(a) It returns `false`.

(b) It returns `true`.

(c) It loops.

Quiz

Consider this incorrect variant, where the cut is before the test:

```
max(X,Y,X) :- !, X >= Y.
max(X,Y,Y).
?- max(2,3,Z).
```

What happens?

(a) `Z=3`, because PROLOG still tries clause 2.

(b) `false`, because `!` commits before `2 >= 3` is tested.

(c) `Z=2`, because `!` assigns `X` to `Z`.

Execution Control and Procedural Elements

Predicate `fail`

- The built-in predicate `fail` *always fails*.
- It is used to deliberately reject the current proof path.
- Common uses:
  - after printing one solution, to force PROLOG to backtrack and print the next one;
  - together with cut, to exclude a special case.
- See [Ertel 2025], Sec. 5.3; [Bratko 2011], Sec. 5.3.

Execution Control and Procedural Elements

Example 4 (Predicate `fail` in PROLOG)

- In the family relationship example we can quite simply print out all children and their parents with the query

```
?- child_fact(X,Y,Z), write(X),
write(' is a child of '), write(Y),
write(' and '), write(Z), write('.'),
nl, fail.
```

- The corresponding output is

```
oscar is a child of karen and frank.
mary is a child of karen and frank.
eve is a child of anne and oscar.
henry is a child of anne and oscar.
isabelle is a child of anne and oscar.
clyde is a child of mary and oscarb.
false.
```

where the predicate `nl` causes a line break in the output. What would be the output in the end without use of the `fail` predicate?

Quiz

Consider this program:

```
snake(cobra).
animal(cobra).

likes(mary,X) :- snake(X), !, fail.
likes(mary,X) :- animal(X).
?- likes(mary,cobra).
```

What happens?

(a) It succeeds because `animal(cobra)` is a fact.

(b) It fails: cut commits to the snake case; `fail` rejects it.

(c) It succeeds, then fails, because both rules are used.

Execution Control and Procedural Elements

Example 5 (Negation as Failure)

- In the family relationship example, the query

```
?- child_fact(ulla,X,Y).
```

  would result `false.` because there are no facts about `ulla`.
- This answer is *not logically correct*. Specifically, *it is not possible to prove that there is no object with the name* `ulla`. Here the prover E would correctly answer "`No proof found.`"
- Thus if *PROLOG answers* `false.`, this only means that *the query* Q *cannot be proved*. For this, however, $\neg Q$ *must not necessarily be proved*.

Lists

- A collection of *ordered data*.
- Has zero or more elements enclosed by square brackets and separated by commas (‘,’).

Example	Description
`[A]`	A list with one element
`[]`	An empty list
`[34,tom,[2,3]]`	A list with three elements where the third element is a list of two elements
`[mia, love(honey), mia]`	A list with three elements where the first and last elements are identical

- Like any object, a list can be unified with a variable

```
?- X = [Any, list, 'of elements'].
X = [Any, list, 'of elements'].
```

Lists

- A list can be decomposed into its *head* (first element) and *tail* (remaining elements) using the vertical bar operator ('|').
- For example, the list `[A, B, C]` can be decomposed as follows:

```
?- [Head|Tail] = [A, B, C].
Head = A,
Tail = [B, C].
```

- What are the head and the tail of the list `[dead(z)]`?
- **Note:** The empty list has neither a head nor a tail. That is, the empty list has no internal structure; for PROLOG, `[]` is a special, particularly simple, list.
  - What is the output of the following query?

```
?- [H|T] = [].
```

Quiz

What are the bindings for this query?

```
?- [X|Y] = [[], dead(z), [2,[b,c]], [], Z].
```

(a) `X=[], Y=[dead(z),[2,[b,c]],[],Z].`

(b) `X=dead(z), Y=[[2,[b,c]],[],Z].`

(c) The query fails.

Quiz

What are the bindings for this query?

```
?- [X,Y|W] = [[], dead(z), [2,[b,c]], [], Z].
```

(a) `X=[], Y=dead(z), W=[[2,[b,c]],[],Z].`

(b) `X=[], Y=[dead(z),[2,[b,c]],[],Z], W=[].`

(c) The query fails.

Quiz

What are the bindings for this query?

```
?- [X1,X2,X3,X4|Tail] =
   [[], dead(z), [2,[b,c]], [], Z].
```

(a) `X1=[], X2=dead(z), X3=[2,[b,c]], X4=[], Tail=[Z].`

(b) `X1=[], X2=dead(z), X3=[2,[b,c]], X4=[], Tail=Z.`

(c) The query fails.

Quiz

What are the bindings for this query?

```
?- [_,X,_,Y|_] =
   [[], dead(z), [2,[b,c]], [], Z].
```

(a) `X=[], Y=dead(z).`

(b) `X=dead(z), Y=[].`

(c) `X=[2,[b,c]], Y=Z.`

Lists
Quiz: A Nested List

Quiz

What is the binding for `X`?

```
?- [_,_,[_|X]|_] =
   [[], dead(z), [2,[b,c]], [], Z].
```

(a) `X=[2,[b,c]].`

(b) `X=[[b,c]].`

(c) `X=[b,c].`

Lists
First recursive list program

It's time to look at an example (from "Learn PROLOG Now!") of a recursive PROLOG program for lists: the predicate `member/2`.

- Goal: given an object `X` and a list `L`, decide whether `X` belongs to `L`.
- The standard definition (one fact and one recursive rule):

```
1  member(X,[X|T]).
2  member(X,[H|T]) :- member(X,T).
```

  - First clause (fact): "`X` is a member of a list if `X` is the head of that list." (uses the `|` operator)
  - Second clause (recursive rule): "`X` is a member of a list if `X` is a member of the tail of the list."
  - Declaratively this is straightforward: the two clauses capture membership directly from the structure of lists.

Lists
Procedural behaviour — examples

Consider how PROLOG answers queries.

- Immediate success:

```
?- member(yolanda,[yolanda,trudy,vincent,jules])
```

PROLOG succeeds immediately by unifying with the first clause.

- Requires recursion/backtracking:

```
?- member(vincent,[yolanda,trudy,vincent,jules])
```

PROLOG tries the first clause (fails), uses the recursive clause repeatedly until the subgoal

```
member(vincent,[vincent,jules])
```

unifies with the first clause and succeeds.

Lists
Failure and termination

- If the queried element is not in the list, recursion eventually reaches the empty list and cannot proceed:

```
?- member(zed,[yolanda,trudy,vincent,jules]).
```

- PROLOG will derive successive goals

```
member(zed,[trudy,vincent,jules])
member(zed,[vincent,jules])
member(zed,[jules])
member(zed,[])
```

and at `member(zed,[])`, neither clause applies (empty list cannot be split), so search stops and the answer is no.

Lists

Enumerating members & a small improvement

- `member/2` can be used with variables to enumerate elements:

```
?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda ;
X = trudy ;
X = vincent ;
X = jules ;
no
```

- Small stylistic improvement: use anonymous variables for irrelevant parts

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

Semantically identical, but clearer because each clause names only what matters.

Self-modifying Programs

- PROLOG programs are not fully compiled, rather, they are interpreted by the WAM. Therefore *it is possible to modify programs at runtime*. A program can even modify itself.
- With commands such as `assert` and `retract`, facts and rules can be added to the knowledge base or taken out of it.
- Assert predicates
  - `assert(X)`: Adds a new fact or clause to the database. Term is asserted as the last fact or clause with the same key predicate.
  - `asserta(X)`: Same as assert, but adds a clause at the beginning of the database.
  - `assertz(X)`: Exactly same as `assert(X)`.
- Retract predicates
  - `retract(X)`: Removes fact or clause `X` from the database.
  - `retractall(X)`: Removes all facts or clauses from the database for which the head unifies with `X`.

Self-modifying Programs

A simple application of `asserta` is the addition of derived facts to the beginning of the knowledge base with the goal of avoiding a repeated, potentially time-expensive derivation.

Example 6 (Family Relationship)

dynamic_rel.pl

```
1   child_fact(oscar, karen, frank).
2   child_fact(mary, karen, frank).
3   child_fact(eve, anne, oscar).
4   child_fact(henry, anne, oscar).
5   child_fact(isabelle, anne, oscar).
6   child_fact(clyde, mary, oscarb).
7
8   child(X,Z,Y) :- child_fact(X,Y,Z).
9   child(X,Z,Y) :- child_fact(X,Z,Y).
10
11  :- dynamic descendant/2.
12  descendant(X,Y) :- child(X,Y,Z), asserta(descendant(X,Y)).
13  descendant(X,Y) :- child(X,U,V), descendant(U,Y),
14                     asserta(descendant(X,Y)).
```

Self-modifying Programs

```
?- [dynamic_rel].
true.

?- descendant(clyde, karen).
true .

?- listing(descendant).
:- dynamic descendant/2.

descendant(clyde, karen).
descendant(mary, karen).
descendant(X, Y) :-
child(X, Y, Z),
asserta(descendant(X, Y)).
descendant(X, Y) :-
child(X, U, V),
descendant(U, Y),
asserta(descendant(X, Y)).

true.
```

Self-modifying Programs

- By manipulating rules with `assert` and `retract`, even programs that change their own rules can be written. A related machine-learning idea is *genetic programming*, where programs or rules are changed automatically.
- In practice, however, it turns out that, due to the *huge number of senseless possible changes*, *changing the code by trial and error rarely leads to a performance increase*.
- *Systematic changing of rules*, on the other hand, *makes programming so much more complex* that, so far, such programs that extensively modify their own code have not been successful.
- *Machine learning* has been quite successful. However, only *very limited modifications* of the program code are being conducted here.

A Planning Example

Exercise 1

Understand how to solve this problem using PROLOG.

A farmer wants to bring a cabbage, a goat, and a wolf across a river, but his boat is so small that he can only take them across one at a time. The farmer thought it over and then said to himself: “If I first bring the wolf to the other side, then the goat will eat the cabbage. If I transport the cabbage first, then the goat will be eaten by the wolf. What should I do?”

Constraint Logic Programming

- The *programming of scheduling systems*, in which many (sometimes complex) logical and numerical conditions must be fulfilled, *can be very expensive and difficult with conventional programming languages*.
- This is precisely where *logic could be useful*.
- An approach is to simply *write all logical conditions in PL1 and then enter a query*. *Usually this direct PL1 approach is not convenient*. As the penguin example showed, PL1 does not guess facts that we did not write down. For example, from `penguin(tweety)` it does not guess that Tweety is not a raven; such exclusions must be stated explicitly.
- For scheduling problems, the relevant restrictions can be stated explicitly as constraints. *Constraint Logic Programming (CLP)* [Jaffar and Lassez 1987], which allows the *explicit formulation of constraints for variables*, offers an elegant and efficient mechanism for such constraint-heavy search problems.
  - The *interpreter constantly monitors the execution of the program for adherence to all of its constraints*.
  - The programmer is fully relieved of the task of controlling the constraints, which in many cases can greatly simplify programming.

Constraint Logic Programming

Example 7 (Applying the CLP mechanism of GNU Prolog (The finite domain (FD) constraint solver))

The secretary of Albert Einstein High School has to come up with a plan for allocating rooms for final exams. He has the following information: the four teachers Mayer, Hoover, Miller and Smith give tests for the subjects German, English, Math, and Physics in the ascendingly numbered rooms 1, 2, 3 and 4. Every teacher gives a test for exactly one subject in exactly one room. Besides that, he knows the following about the teachers and their subjects.

(1) Mr. Mayer never tests in room 4.
(2) Mr. Miller always tests German.
(3) Mr. Smith and Mr. Miller do not give tests in neighboring rooms.
(4) Mrs. Hoover tests Mathematics.
(5) Physics is always tested in room number 4.
(6) German and English are not tested in room 1.

Who gives a test in which room?

Constraint Logic Programming

raumplan.pl

```
1   %%% Run in GNU Prolog
2   start :-
3       fd_domain([Mayer, Hoover, Miller, Smith],1,4),
4       fd_all_different([Mayer, Miller, Hoover, Smith]),
5
6       fd_domain([German, English, Math, Physics],1,4),
7       fd_all_different([German, English, Math, Physics]),
8
9       fd_labeling([Mayer, Hoover, Miller, Smith]),
10
11      Mayer #\=4,                    % Mayer not in room 4
12      Miller #= German,              % Miller tests German
13      dist(Miller, Smith) #>= 2,     % Distance Miller/Smith >= 2
14      Hoover #= Math,                % Hoover tests mathematics
15      Physics #= 4,                  % Physics in room 4
16      German #\= 1,                  % German not in room 1
17      English #\= 1,                 % English not in room 1
18      nl,
19      write([Mayer, Hoover, Miller, Smith]), nl,
20      write([German, English, Math, Physics]), nl.
```

Constraint Logic Programming
GNU Prolog predicates

- GNU Prolog built-in predicates:
  - `fd_domain(Vars, Lower, Upper)` constrains each element `X` of `Vars` to take a value in `Lower..Upper`.
  - `fd_all_different(List)` constrains all variables in `List` to take distinct values.
  - `fd_labeling(Vars, Options)` assigns a value to each variable `X` of the list `Vars` according to the list of labeling options given by `Options`. This predicate is re-executable on backtracking. `fd_labeling(Vars)` is equivalent to `fd_labeling(Vars, [])`.

Constraint Logic Programming
Reading the program

- The variables `Mayer,Hoover,Miller,Smith` as well as `German,English,Math,Physics` can each take on an integer value from 1 to 4 as the room number. (Lines 3–6.)
- A binding `Mayer = 1` and `German = 1` means that Mr. Mayer gives the German test in room 1.
- Lines 4 and 7 ensure that the four particular variables take on different values.
- Line 9 starts the finite-domain search: GNU Prolog chooses concrete room numbers for the teacher variables. If a later constraint fails, PROLOG backtracks to line 9 and tries another assignment. The subject rooms are then determined by the constraints below. Without this labeling step, GNU Prolog may print finite-domain intervals instead of concrete room numbers.
- In lines 11–17 the constraints are given, and the remaining lines output the room numbers for all teachers and all subjects in a simple format.

Constraint Logic Programming

The program is loaded into GNU Prolog with `['raumplan.pl'].`, and with `start.` we obtain the output

```
[3,1,2,4]
[2,3,1,4]

true ?

yes
```

This output corresponds to the plan

Room num.	1	2	3	4
Teacher	Hoover	Miller	Mayer	Smith
Subject	Math	German	English	Physics

References

- Ertel, Wolfgang (2025). *Introduction to Artificial Intelligence*. 3rd. Springer. DOI: 10.1007/978-3-658-43102-0.
- Bos, Johan (2020). *Drawing Prolog Search Trees: A Manual for Teachers and Students of Logic Programming*. arXiv:2001.08133. URL: https://arxiv.org/abs/2001.08133 (visited on 07/05/2026).
- Clocksin, William F. and Christopher S. Mellish (2013). *Programming in PROLOG Using the ISO Standard*. 3rd. Springer Science & Business Media. DOI: 10.1007/978-3-642-97005-4.
- Bratko, Ivan (2011). *Prolog programming for artificial intelligence*. 4th. Addison-Wesley.

References (cont.)

Blackburn, Patrick, Johan Bos, and Kristina Striegnitz (2006). *Learn Prolog Now!* URL: https://www.let.rug.nl/bos/lpn/ (visited on 07/05/2026).

Jaffar, Joxan and Jean-Louis Lassez (1987). "Constraint Logic Programming." In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 111–119. DOI: 10.1145/41625.41635.