Introduction:
Logic and Logic Programming

CONSTRAINT PROGRAMMING COURSE

M1 IDO

Origins of PROLOG

1972	Alain Colmerauer and Philippe Roussel develop Prolog at the University of Aix-Marseille, France. The first interpreter is implemented in Fortran.
1980	Prolog gains international recognition as the primary development language for Artificial Intelligence research and expert systems.
1990s	Constraint Logic Programming (CLP) extends Prolog with declarative constraint solving — enabling powerful applications in scheduling, planning, and combinatorics.

Programming Paradigms

Imperative

C, Pascal, Java, Ada

Describes HOW to solve a problem step-by-step. Explicit control flow: loops, assignments, conditionals.

Functional

Lisp, Scheme, OCaml, Haskell

Expresses computation as evaluation of mathematical functions. Avoids mutable state.

Declarative

Prolog, SQL, Datalog

Describes WHAT the problem is — not how to solve it. The runtime engine determines the solution strategy.

Propositional Logic — Syntax

Building complex statements from atomic propositions

- Propositional logic is the simplest formal logic system.
- A proposition is a statement that is either TRUE or FALSE (e.g., P, Q, R …).
- Atomic statements consist of a single propositional symbol.
- Complex statements are formed by combining atomic ones with logical connectives:

Example: $\neg P \wedge (Q \Rightarrow R)$
"Assuming P is false, if Q is true then R must hold."

Connective Symbols

$\neg E$	Negation *"not E"*
$E_1 \wedge E_2$	Conjunction *"E_1 and E_2"*
$E_1 \vee E_2$	Disjunction *"E_1 or E_2"*
$E_1 \Rightarrow E_2$	Implication *"if E_1 then E_2"*
$E_1 \Leftrightarrow E_2$	Equivalence *"E_1 iff E_2"*

Propositional Logic — Semantics

Truth tables for the standard connectives

- A model assigns a truth value (true/false) to each propositional symbol.
- For n symbols there are 2^n possible models (e.g., n = 3 → 8 models).
- An expression is evaluated recursively from its sub-expressions.
- Example: $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \neg F \wedge (F \vee T) = T \wedge T = TRUE$

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

T = True, F = False. Note: P ⇒ Q is FALSE only when P is TRUE and Q is FALSE.

First-Order Logic — Overview

Propositional logic can only represent bare facts. FOL introduces:

Objects	Relations	Functions
• People, houses, numbers • Colors, games, events • Abstract concepts	• Unary (properties): red, prime, mortal • n-ary: brother-of, larger-than • Written as predicates: P(x, y)	• Map objects to a unique value • father-of(x), square-root(n) • Written as terms: $f(x_1, \ldots, x_n)$

Syntax elements: Constants • Predicates • Functions • Variables • Connectives ($\neg \wedge \vee \Rightarrow \Leftrightarrow$) • Quantifiers ($\forall \exists$)

FOL — Terms, Atoms & Compound Statements

Terms

- A term is either: a Constant (2, Fateh, X_1), a Variable (x, y, a), or a Function applied to terms: $f(t_1, \ldots, t_n)$.
- Examples of terms: 2, Fares, Father(Rachid), SquareRoot(4)

Atomic Statements

- Form: Predicate($t_1, \ldots, t_n$) — applies a relation to terms.
- Examples: Brother(Rachid, Fares) Married(Father(Rachid), Mother(Fares))

Compound Statements

- Combine atoms with connectives: $\neg E$, $E_1 \wedge E_2$, $E_1 \vee E_2$, $E_1 \Rightarrow E_2$, $E_1 \Leftrightarrow E_2$
- Example: Brother(Salim, Rachid) $\Rightarrow$ Brother(Rachid, Salim)
- Example: $\neg(1 > 2) \Leftrightarrow (1 \leq 2)$

FOL — Quantifiers: ∀ and ∃

∀ Universal Quantification

- ∀x P is true if P holds for every object x.
- Use ⇒ as the main connective.
- Example: ∀x Student(x) ⇒ Intelligent(x)
- "Every student is intelligent."

∃ Existential Quantification

- ∃x P is true if P holds for at least one object x.
- Use ∧ as the main connective.
- Example: ∃x Student(x) ∧ Intelligent(x)
- "Some student is intelligent."

⚠ Common Error — ∀

∀x Student(x) ∧ Intelligent(x) means
'Everyone is a student AND intelligent' — WRONG!

⚠ Common Error — ∃

∃x Student(x) ⇒ Intelligent(x) is TRUE if there is anyone who is NOT a student — WRONG!

`Quantifier Duality: ∀x P(x) ≡ ¬∃x ¬P(x)  ∃x P(x) ≡ ¬∀x ¬P(x) | Order matters: ∀x ∃y ≠ ∃y ∀x`

Horn Clauses

- A Horn clause is a disjunction of literals with at most one positive literal.
- Every Horn clause can be written as an implication: $\text{Premise}_1 \wedge \ldots \wedge \text{Premise}_n \Rightarrow$ Conclusion.
- Definite clause: exactly one positive literal (the head); the rest form the body.
- Fact: a Horn clause with no negative literals (empty body).

Predicate Logic	Horn Clause	Prolog	Type
P(x, y)	P(x, y)	p(X, Y).	Fact
Q(x) ⇒ P(x, y)	P(x, y) ← Q(x)	p(X,Y) :- q(X).	Rule
Q(x) ∧ Q(y) ⇒ P(x,y)	P(x,y) ← Q(x) ∧ Q(y)	p(X,Y) :- q(X), q(Y).	Rule
?- Q(x) ∧ Q(y)	← Q(x) ∧ Q(y)	?- q(X), q(Y).	Query

Prolog — Syntax: Constants, Variables, Facts & Rules

Constants

- Numbers: 128, 4.5
- Atoms (lowercase start): fateh, rachid, paris
- Quoted strings: "Fateh", "Hello World"

Variables

- Uppercase start: X, House, Father
- Anonymous variable: _ (matches anything, not bound)
- Prefixed: _name, _001

Facts & Rules

```
% Facts
pere(fateh, tarek).    % F1
pere(fateh, fares).    % F2
pere(fares, zohir).    % F3
mere(khadidja, zohir).% F4
```

```
% Rules
papy(X, Y) :-
   pere(X, Z),
   pere(Z, Y).  % R1
papy(X, Y) :-
   pere(X, Z),
   mere(Z, Y).  % R2
```

Syntax: facts end with '.' Rules use ':-' to separate head from body. ',' in body means AND.

Prolog — Querying the Knowledge Base

Example facts and queries

```
pere(fateh, tarek).  % F1
pere(fateh, fares).  % F2
pere(fares, zohir).  % F3
mere(khadidja, zohir).%F4
mere(khadidja, luc). % F5
pere(salim, khadidja).%F6
```

Query	Answer
?- pere(fateh, tarek).	yes
?- pere(fateh, zohir).	no
?- pere(fateh, X).	X = tarek ; X = fares
?- mere(X, zohir).	X = khadidja
?- pere(fateh,X), pere(X,zohir).	X = fares (grandfather query)

How Prolog answers queries:

- Prolog searches the knowledge base for facts/rules that unify with the query.
- Variables in queries are bound to values that satisfy the goal (pattern matching).
- The ';' separator requests additional solutions (backtracking).
- Conjunctive queries (Q_1, Q_2) require both sub-goals to succeed simultaneously.

Unification — Definition & Algorithm

The core inference mechanism of Prolog

Unification is the process of making two terms syntactically identical by finding a consistent substitution (called the Most General Unifier, or MGU) for the variables they contain.

Unification Algorithm (Martelli-Montanari)

- 1. If equation is t = t (identical), delete it.
- 2. If f(…) = g(…) with f ≠ g (different functors or arities), → FAIL.
- 3. If $f(t_1,\ldots,t_n) = f(t_1',\ldots,t_n')$, decompose into n equations $t_i = t_i'$.
- 4. If t = x and t is not a variable, swap to x = t (orient).
- 5. If x = t and x occurs in t (occurs check), → FAIL (prevents circular terms).
- 6. If x = t and x ∉ t, substitute x → t everywhere else in the equation set.

The resulting substitution is the Most General Unifier (MGU). Unification can succeed or fail; if it fails, Prolog backtracks.

Unification — Worked Examples

Example 1

Input: { X = Y, f(Y) = U }

1. Invert f(Y) = U → U = f(Y)
2. MGU: { X/Y, U/f(Y) }

Example 2

Input: { X = g(Y), f(X) = Z, Y = a }

1. Y/a → X = g(a)
2. X/g(a) → f(g(a)) = Z → Z = f(g(a))
3. MGU: { X/g(a), Z/f(g(a)), Y/a }

Example 3

Input: { g(Y) = g(Z), X = f(Y) }

1. Decompose g(Y) = g(Z) → Y = Z
2. Substitute Y/Z → X = f(Z)
3. MGU: { Y/Z, X/f(Z) }

Example 4 ✗

Input: { X = Y, X = f(Y) }

1. Substitute X/Y → Y = f(Y)
2. FAIL: occurs check — Y appears inside f(Y)

Prolog — Resolution by Refutation

How Prolog proves goals

- Prolog proves a query Q by attempting to derive a contradiction from ¬Q (proof by refutation).
- The interpreter selects clauses from the knowledge base, unifies them with goals, and resolves sub-goals left-to-right.
- If a sub-goal fails, Prolog backtracks to the most recent choice point and tries the next alternative.

```
% Knowledge base
pere(fateh, hamza). % F1
pere(aziz, fateh).  % F2

papy(X,Y) :-        % R1
  pere(X,Z), pere(Z,Y).

% Query
?- papy(X, Y).
```

Resolution steps:

- 1. Unify papy(X,Y) with R1: X_1=X, Y_1=Y
- 2. New goals: pere(X,Z_1), pere(Z_1,Y)
- 3. Unify pere(X,Z_1) with F2: X=aziz, Z_1=fateh
- 4. Unify pere(fateh,Y) with F1: Y=hamza
- 5. ✓ SUCCESS — papy(aziz, hamza)
- Backtrack path tried aziz+fateh pair after failing hamza branch.

Lists — Structure and Notation

The most fundamental data structure in Prolog

- Lists can hold any type of term: atoms, numbers, variables, compound terms, or nested lists.
- Written with square brackets, elements separated by commas: [a, b, c]
- The empty list [] is a special atom; it cannot be decomposed into head and tail.

Enumerated Notation

```
[a, b, c]
[1, 2, 3, 4]
[]
[Y, b, [1,2,3]]
[le, chat, mange]
```

Head | Tail Notation

```
[a|[b,c]]       = [a,b,c]
[a|[b|[c|[]]]]  = [a,b,c]
[H|T] = [1,2,3]
  → H=1, T=[2,3]
[] ≠ [[]|[]]    (fails!)
```

List Unification

Matching lists with pattern variables

Query	Result
`[T|Q] = [1, 2, 3, 4]`	`T = 1,  Q = [2, 3, 4]`
`[X, Y] = [1, 2]`	`X = 1,  Y = 2`
`[X, Y] = [1, 2, 3]`	`No  (length mismatch)`
`[X, Y | Z] = [1, 2, 3]`	`X = 1,  Y = 2,  Z = [3]`
`[X, Y | Z] = [1, 2]`	`X = 1,  Y = 2,  Z = []`
`[a, [2,2], c] = [a, Y, c]`	`Y = [2, 2]`
`[a, b, [c, d]] = [a, b, [c, X]]`	`X = d`

Key insight: [H|T] binds H to the first element and T to the rest of the list. This is the basis for all recursive list processing.

Lists — Recursive Predicates

member/2, append/3, reverse/2

```
% member/2 – element membership
member(X, [X|_]).        % R1
member(X, [_|T]) :-      % R2
  member(X, T).

% R1: X is the head – found!
% R2: search recursively in tail
```

```
% append/3 – list concatenation
append([], L, L).          % base
append([X|L], M, [X|N]) :-
  append(L, M, N).

?- append([a,b],[c,d], L).
%  L = [a, b, c, d]
```

```
% Efficient list reversal
rev(L, R) :- rev(L, [], R).
rev([], Acc, Acc).          % base
rev([H|T], Acc, R) :-
  rev(T, [H|Acc], R).

?- rev([a,b,c], R). % R=[c,b,a]
```

```
% List sum (traversal pattern)
sum([], 0).                  % base
sum([H|T], S) :-
  sum(T, S1),
  S is H + S1.

?- sum([1,2,3,4], S). % S=10
```

Pattern: recursion always reduces the list size by removing the head. Base case is the empty list []. Accumulators enable tail recursion (efficient).

Arithmetic — The is/2 Predicate & Comparisons

The is/2 predicate forces arithmetic evaluation. Without it, expressions remain as symbolic terms:

```
% Unification only – no evaluation
?- X = 4 + 5.
   X = 4+5   (a term, not 9!)

% Force evaluation with is/2
?- X is 4 + 5.
   X = 9   ✓
```

```
% Arithmetic operations
5  is 4 + 1.
3  is 5 - 2.
-3 is 2 - 5.
2  is 4 / 2.
12 is 2 * 6.
1  is mod(5, 2).
```

Math	Prolog	Math	Prolog
X < Y	X < Y.	X ≥ Y	X >= Y.
X ≤ Y	X =< Y.	X > Y	X > Y.
X = Y	X =:= Y.	X ≠ Y	X =\= Y.

Note: =:= compares arithmetic values; = unifies terms. X=:=Y requires both sides to be fully instantiated numbers.

The Cut Predicate — !

Pruning the search space

- The cut (!) is a built-in predicate that always succeeds but prevents backtracking past the point where it appears.
- When Prolog executes !, it commits to all choices made since the head of the current clause was selected.
- Effect: all choice-points created for sub-goals to the left of ! and for the clause itself are discarded.

Use cases:

Deterministic choice

```
human(X) :- man(X),!.
human(X) :- woman(X),!.
```

If-then-else

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

Infinite generator cut

```
nat(0).
nat(N) :- nat(N1), N is N1+1.
sqrt_int(N,R) :-
  nat(K), K*K > N, !,
  R is K-1.
```

Rule: Cut suppresses choice points for ALL child sub-goals and for alternative clauses of the predicate that called it.

Negation as Failure (NAF)

not/1 and the Closed-World Assumption

- Prolog cannot directly express classical negation (¬P is false vs. unprovable).
- Instead, Prolog uses Negation as Failure (NAF): not(F) succeeds if and only if F fails.
- Closed-World Assumption (CWA): anything not provable from the knowledge base is assumed false.

```
% Implementation of not/1 via cut+fail
not(P) :- P, !, fail.
not(_).

% Reading:
% If P succeeds → commit and fail.
% If P fails    → succeed (2nd clause).
```

Key Restrictions

- Use not/1 only for verification (arguments must be fully bound/instantiated).
- not/1 cannot be used to determine variable values.
- not(bird(X)) does not find animals that are not birds — it only checks if a specific X is not a bird.

⚠ **Important:** not(F) means 'F is not provable,' NOT 'F is logically false.' These are different in open-world environments. The CWA is fundamental to Prolog's operational semantics.

Declarative vs. Procedural Semantics

Two complementary readings of Prolog clauses

The same Prolog clause admits two interpretations:

```
p :- q, r.
```

Declarative Reading

- "p is true whenever both q and r are true."
- The programmer states what is true — not how to compute it.
- Clause order is irrelevant in pure declarative Prolog.
- Corresponds directly to the logical formula: $q \land r \Rightarrow p$.

Procedural Reading

- "To solve p, first solve q, then solve r."
- The interpreter processes goals left-to-right.
- Clause order MATTERS for efficiency and termination.
- Base cases should appear before recursive rules.

Best practice: write clauses that are correct declaratively, then tune their order for procedural efficiency.