# Artificial Intelligence: Foundations & Applications

## Introduction to Prolog Programming

Prof. Partha P. Chakrabarti & Arijit Mondal
Indian Institute of Technology Kharagpur

# What is Prolog?

- Invented early seventies by Alain Colmerauer in France and Robert Kowalski in Britain
- Prolog = Programmation en Logique (Programming in Logic).
- Prolog is a declarative programming language unlike most common programming languages.
- In a declarative language
  - The programmer specifies a goal to be achieved
  - The Prolog system works out how to achieve it
- In purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system

# Relations

- Prolog programs specify relationships among objects and properties of objects
- When we say, "Ayesha owns the pen", we are declaring the ownership relationship between two objects: Ayesha and the pen.
- When we ask, "Does Ayesha own the pen?" we are trying to find out about a relationship
- Relationships can also be rules such as:
  - Two people are sisters if – they are both female AND they have the same parents.
- In traditional programming relationship may be defined as
  - A and B are sisters if – A and B are both female AND they have the same father AND they have the same mother AND A is not the same as B
- A rule allows us to find out about a relationship even if the relationship is not explicitly stated as a fact

# Programming prolog

- Declare facts describing explicit relationships between objects and properties objects might have (e.g. Subodh likes pizza, sky has_colour blue)
- Declare rules defining implicit relationships between objects and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).
- Then the system can be used by:
  - Asking questions above relationships between objects, and/or about object properties (e.g. does Subodh like pizza? is Ayesha a parent?)

# Prolog & Predicate logic

- Prolog is a programming language based on predicate logic.
  - A Prolog program attempts to prove a goal, such as brother(Barney,x), from a set of facts and rules.
  - In the process of proving the goal to be true, using substitution and the other rules of inference, Prolog substitutes values for the variables in the goal, thereby "computing" an answer.
- How does Prolog know which facts and which rules to use in the proof?
  - Prolog uses unification to determine when two clauses can be made equivalent by a substitution of variables.
  - The unification procedure is used to instantiate the variables in a goal clause based on the facts and rules in the database.

# Tools

- **GNU prolog -** `gplc, gprolog`
- **SWI-Prolog -** `https://swi-prolog.org`
- **Online tool**
  - `https://swish.swi-prolog.org/`
  - `https://www.onlinegdb.com/online_prolog_compiler`

# A simple Prolog program

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
father(X,Y) :- parent(X,Y), male(X).       % ∀X ∀Y ((parent(X, Y) ∧ male(X)) → father(X, Y))
mother(X,Y) :- parent(X,Y), female(X).     % ∀X ∀Y ((parent(X, Y) ∧ female(X)) → mother(X, Y))
```

- A fact/rule (statement) ends with "." and white space ignored
- Read ':-' after RULE HEAD as "if"
- Read comma in body as "and"
- Comment a line with % or use /* */ for multi-line comments

# Facts & Rules

- The Prolog environment maintains a set of facts and rules in its database.
  - Facts are axioms; relations between terms that are assumed to be true.
  - Rules are theorems that allow new inferences to be made.
- Example facts & rules:
  - male(adam).
  - female(anne).
  - parent(adam,barney).
  - son(X,Y) :- parent(Y,X) , male(X)
  - daughter(X,Y) :- parent(Y,X) , female(X)
- The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male. $\forall X \forall Y((parent(Y, X) \wedge male(X)) \rightarrow son(X, Y))$
- The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female. $\forall X \forall Y((parent(Y, X) \wedge female(X)) \rightarrow daughter(X, Y))$

# Horn clauses

- To simplify the resolution process in Prolog, statements must be expressed in a simplified form, called Horn clauses.
  - Statements are constructed from terms.
  - Each statement (clause) has (at most) one term on the left hand side of an implication symbol ( :- ).
  - Each statement has a conjunction of zero or more terms on the right hand side.
- Prolog has three kinds of statements, corresponding to the structure of the Horn clause used.
  - A fact is a clause with an empty right hand side.
  - A question (or goal) is a clause with an empty left hand side.
  - A rule is a clause with terms on both sides.

# Execution of Prolog program

```
$> gplc family.pl
$> ./family
?- male(albert).
yes
?- male(victoria).
no
?- male(subodh).
no
?- male(X).
X = albert ?  ;
X = edward
?- father(F,C).
F=albert, C=edward ?  ;
no
```

**male(albert).**
**male(edward).**
**female(alice).**
**female(victoria).**
**parent(albert,edward).**
**parent(victoria,edward).**
**father(X,Y) :- parent(X,Y), male(X).**
**mother(X,Y) :- parent(X,Y), female(X).**

**Query:**
**male(X).** **%** $\exists X\, male(X)$
**father(F,edward).** **%** $\exists F\, father(F, edward)$
**father(F,C).** **%** $\exists F\, \exists C\, father(F, C)$

# Observation about Prolog rules

- **The implication is from right to left**
- **The scope of a variable is the clause in which it appears.**
- **Variables whose first appearance is on the left hand side of the clause have implicit universal quantifiers.**
- **Variables whose first appearance is on the right hand side of the clause have implicit existential quantifiers.**

# Basic syntax of Prolog: Terms

- Constants:
  - Identifiers - sequences of letters, digits, or underscore "_" that start with lower case letters.
  - Numbers - 1.001
  - Strings enclosed in single quotes
    - Can start with upper case letter or can be a number now treated as a string
- Variables:
  - Sequence of letters digits or underscore that start with an upper case letter or the underscore
    - Undescore by itself is the special "anonymous" variable
- Structures (like function applications)
  - <identifier>(Term-1,. . . ,Term-k)
    - date(20,April,2020), point(X,Y,Z)
  - Definition can be recursive, so each term can itself be a structure
    - date(+(5,15),April,+(2000,−(140,120)))
  - Structures can be represented as tree

# Syntax of Prolog: Lists

- Lists are a very useful data structure in Prolog
- Lists are structured terms represented in a special way - [a, b, c, d]
  - This is actually structured term - [ a | [ b | [ c | [ d | []]]]]
  - In the above [] denotes empty list
  - Each list is thus of the form [ <head> | <tail> ]
  - <head> is an element of the list (not necessary a list itself)
  - <tail> is a list / sublist
  - Also, [a,b,c,d] = [a | [b,c,d]] = [a,b | [c,d]] = [a,b,c | [d]]
- This structure has important implications when it comes to matching variables against lists!

# Syntax of Prolog: Predicates

- Predicates are syntactically identical to structured items – <identifier>(Term-1,. . . ,Term-k)
  - male(edward)
  - parent(edward,albert)
  - taller_than(subodh,shyam)
  - likes(X)
    - Note that X is a variable. X can take on any term as value so that this fact asserts
- Facts make assertion

# Syntax of Prolog: Facts and Rules

- Rules: PredicateH :- predicate-1, . . . , predicate-k.
  - First predicate is *rule head*. Terminated by a period
  - Rules encode ways of deriving or computing a new fact
  - animal(X) :- elephant(X). - X can be concluded to be animal if it shown that X is elephant
  - taller_than(X,Y) :- height(X,H1), height(Y,H2), H1 > H2.
  - father(X,Y) :- parent(X,Y), male(X).

# Operation of Prolog

- A query is a sequence of predicates: predicate-1, predicate-2, . . . , predicate-k
- Prolog tries to prove that this sequence of predicates is true using the facts and rules in the Prolog Program.
- In proving the sequence it performs the computation you want.
- Example:
  - elephant(fred).
  - elephant(mary).
  - elephant(joe).
  - animal(fred) :- elephant(fred).
  - animal(mary) :- elephant(mary).
  - animal(joe) :- elephant(joe).
  - QUERY: animal(fred), animal(mary), animal(joe).

# Operation

- Starting with the first predicate P1 of the query Prolog examines the program from TOP to BOTTOM.
- It finds the first RULE HEAD or FACT that matches P1
- Then it replaces P1 with the RULE BODY.
- If P1 is matched a FACT, we can think of FACTs as having empty bodies (so P1 is simply removed).
- The result is a new query.
- Example
  - P1 :- Q1, Q2, Q3.
  - QUERY: P1, P2, P3.
  - P1 matches with the rule, therefore, new QUERY: Q1, Q2, Q3, P2, P3.

# Execution of Prolog program

```
elephant(fred).
elephant(mary).
elephant(joe).
animal(fred) :- elephant(fred).
animal(mary) :- elephant(mary).
animal(joe) :- elephant(joe).
QUERY: animal(fred), animal(mary), animal(joe).
```

```
1.  elephant(fred), animal(mary), animal(joe).
2.  animal(mary), animal(joe).
3.  elephant(mary), animal(joe).
4.  animal(joe).
5.  elephant(joe).
6.  EMPTY QUERY
```

# Operation

- **If this process reduces the query to the empty query, Prolog returns "yes".**
- **However, during this process each predicate in the query might match more than one fact or rule head**
  - **Prolog always choose the first match it finds. Then if the resulting query reduction did not succeed (i.e., we hit a predicate in the query that does not match any rule head of fact), Prolog backtracks and tries a new match.**

# Execution of Prolog program

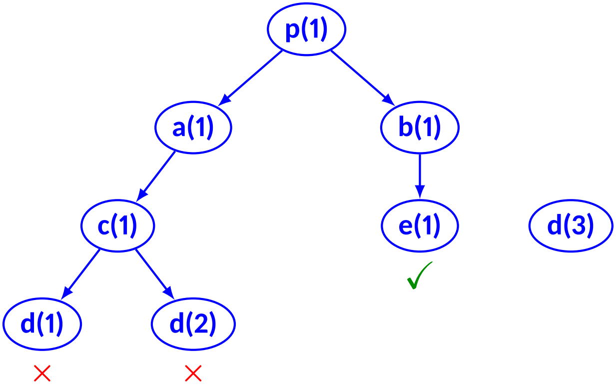
**ant_eater(fred).**
**animal(fred) :- elephant(fred).**
**animal(fred) :- ant_eater(fred).**
**QUERY: animal(fred)**

```
1.  elephant(fred).
2.  FAIL BACKTRACK
3.  ant_eater(fred).
4.  EMPTY QUERY
```

# Operation

- **Backtracking can occur at every stage as the query is processed**

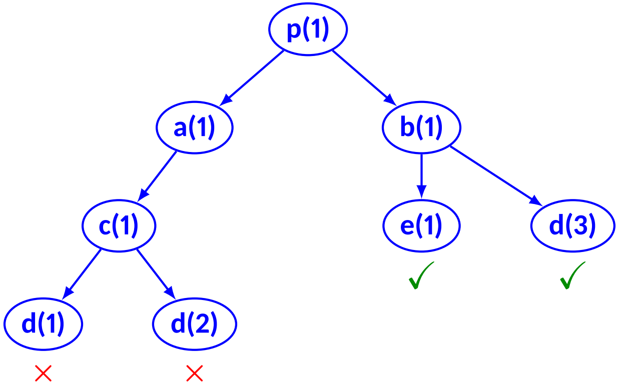
p(1) :- a(1).
p(1) :- b(1).
a(1) :- c(1).
c(1) :- d(1).
c(1) :- d(2).
b(1) :- e(1).
e(1).
d(3).
QUERY: p(1)



# Operation

- With backtracking we can get more answers by using ";"

p(1) :- a(1).
p(1) :- b(1).
a(1) :- c(1).
c(1) :- d(1).
c(1) :- d(2).
b(1) :- e(1).
b(1) :- d(3).
e(1).
d(3).
QUERY: p(1)



# Variables and Matching

- Variables allow us to
  - Compute more than yes/no answer, compress the program
- Example:
  - elephant(fred).
  - elephant(mary).
  - elephant(joe).
  - animal(fred) :- elephant(fred).
  - animal(mary) :- elephant(mary).
  - animal(joe) :- elephant(joe).
- The three rules can be replaced by the single rule animal(X) :- elephant(X).
- When matching queries against rule heads (of facts) variables allow many additional matches.

# Example

```
elephant(fred).
elephant(mary).
elephant(joe).
animal(X) :- elephant(X).
QUERY: animal(fred), animal(mary), animal(joe)
```

1. X=fred, elephant(X), animal(mary), animal(joe)
2. animal(mary), animal(joe)
3. X=mary, elephant(X), animal(joe)
4. animal(joe)
5. X=joe, elephant(X)
6. EMPTY QUERY

# Operation with Variables

- Queries are processed as before (via rule and fact matching and backtracking), but now we can use variables to help us match rule heads or facts.
- A query predicate matches a rule head or fact (either one with variables) if
  - The predicate name must match. So elephant(X) can match elephant(joe), but can never match ant_eater(joe).
  - Once the predicates names the arity of the predicates match (number of terms). So foo(X,Y) can match foo(joe,mary), but cannot match foo(joe) or foo(joe,mary,fred).
  - If the predicate names and arities match then each of the k-terms match. So for foo(t1, t2, t3) to match foo(s1, s2, s3) we must have that t1 matches s1, t2 matches s2, and t3 matches t3.
  - During this matching process we might have to "bind" some of the variables to make the terms match.
  - These bindings are then passed on into the new query (consisting of the rule body and the left over query predicates).

# Variable matching (Unification)

- Two terms with variables match if :
  - If both are constants (identifiers, numbers, or strings) and are identical
  - If one or both are bound variables then they match if what the variables are bound to match
    - X and mary where X is bound to the value mary will match
    - X and Y where X is bound to mary and Y is bound to mary will match
    - X and ann where X is bound to mary will not match
  - If one of the terms is an unbound variable then they match and we bind the variable to the term
    - X and mary where X is unbound match and make X bound to mary.
    - X and Y where X is unbound and Y is bound to mary match and make X bound to mary.
    - X and Y where both X and Y are unbound match and make X bound to Y (or vice versa).

# Solving queries

- Prolog work as follows
  - Unification
  - Goal directed reasoning
  - Rule ordering
  - DFS and backtracking

# List processing in Prolog

- Much of prolog's computation is organized around lists. Two key things we do with a list is iterate over them and build new ones.
- Checking membership: member(X,Y) - X is a member of list Y
  - member(X,[X|_]).
  - member(X,[_|T]) :- member(X,T).
- Building a list of integers in range [i,j] (build(from, to, NewList))
  - build(I,J,[]) :- I>J.
  - build(I,J,[I | Rest]) :- I =< J, N is I+1, build(N,J,Rest).

# List examples

- Concatenation:
  - concatenation([], L, L).
  - concatenation([X|L1], L2, [X|L3]) :- concatenation(L1, L2, L3).
- Example:
  - concatenation([a,b],[c,d],Y).

  - X=a, concatenation([X|b],[c,d],[X|Y1]).
  - concatenation([b],[c,d], Y1).
  - X=b, concatenation([X|[]],[c,d],[X|Y2]).
  - concatenation([],[c,d], Y2).

# List examples

- Adding in front:
  - add(X, L, [X|L]).

- Deletion:
  - del(X, [X|Tail], Tail).
  - del(X, [Y|Tail], [Y|Tail1]) :- del(X, Tail, Tail1).

- Sublist:
  - sublist(S,L) :- concatenation(L1, L2, L), concatenation(S, L3, L2).

# List examples

- **Permutation:**
  - **permutation([], []).**
  - **permutation([X|L], P) :- permutation(L, L1), insert(X, L1, P).**

# Cuts

- del_duplicates([], []).
- del_duplicates([Head | Tail], Result) :- member(Head, Tail), !, del_duplicates(Tail, Result). % R1
- del_duplicates([Head | Tail], [Head | Result]) :- del_duplicates(Tail, Result). % R2

*Thank you!*