Logic Programming
with Prolog

Prolog is based on three main ideas:

- **Logical Horn rules** (day before last)
- **Unification** (last day)
- **Top-down reasoning** (today)
Reasoning

• **Bottom-up** (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  *e.g.*, Suppose the fact $B$ and the rule $A \leftarrow B$ are given. Then infer that $A$ is true.

• **Top-down** (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

  *e.g.*, Suppose the query is $A$, and the rule $A \leftarrow B$ is given. Then to prove $A$, try to prove $B$. 
Bottom-up Inference

A rule base:

\[
\begin{align*}
A & \leftarrow B \quad (1) \\
B & \leftarrow C \quad (2) \\
C & \quad (3)
\end{align*}
\]

A bottom-up proof:

\[
\begin{align*}
\text{infer } A \\
\text{rule (1)} \\
\text{infer } B \\
\text{rule (2)} \\
\text{infer } C \\
\text{rule (3)} \\
\text{start}
\end{align*}
\]

So, A is proved
Top-Down Inference

A rule base:

\[
\begin{align*}
A & \leftarrow B \quad (1) \\
B & \leftarrow C \quad (2) \\
C & \quad (3)
\end{align*}
\]

A top-down proof:

goal A
    \downarrow
    rule (1)
    goal B
    \downarrow
    rule (2)
    goal C
    \downarrow
    rule (3)
    success

So, A is proved
Top-down vs Bottom-up Inference

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (*e.g.*, Coral).

- Each has its own advantages and disadvantages:
  
  - Bottom-up may generate many irrelevant facts.
  
  - Top-down may explore many lines of reasoning that fail.

- Top-down and bottom-up inference are logically equivalent.
  
  *i.e.*, they both prove the same set of facts.

- However, only top-down inference simulates program execution.
  
  *i.e.*, execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc.
Example 1
Bottom-up inference can derive many facts.

Rule base:

\[ p(X,Y,Z) \leftarrow q(X), q(Y), q(Z). \]
\[ q(a1). \]
\[ q(a2). \]
\[ \ldots \]
\[ q(an). \]

Bottom-up inference derives \( n^3 \) facts of the form \( p(a_i,a_j,a_k) \):

\[ p(a1, a1, a1) \]
\[ p(a1, a1, a2) \]
\[ p(a1, a2, a3) \]
\[ \ldots \]
Example 2
Bottom-up inference can derive **infinitely** many facts.

Rule base:

\[ p(f(x)) \leftarrow p(x). \]
\[ p(a). \]

Derived facts:

\[ p(f(a)) \]
\[ p(f(f(a))) \]
\[ p(f(f(f(a)))) \]
\[ \ldots \]

In contrast, top-down inference derives only the facts requested by the user, e.g.

who does jane love?
what is john’s telephone number?
Example 3
Top-down inference may fail.

Rule base:

\[ A \leftarrow B \quad (1) \]
\[ B \leftarrow C \quad (2) \]

Failed line of inference:

\[
\begin{align*}
\text{goal } A \\
\downarrow \text{ rule (1)} \\
\text{goal } B \\
\downarrow \text{ rule (2)} \\
\text{goal } C \\
\downarrow \text{ rule (3)} \\
\text{fail} \\
\text{(no rules infer C)}
\end{align*}
\]

So, A is not proved
Multiple Rules and Premises

A fact may be inferred by many rules. \textit{e.g.},

\begin{align*}
E & \leftarrow B \\
E & \leftarrow C \\
E & \leftarrow D
\end{align*}

A rule may have many premises. \textit{e.g.},

\begin{align*}
E & \leftarrow B \land C \land D
\end{align*}

In top-down inference, such rules give rise to

- inference trees
- backtracking
Example 1: Multiple Premises

Rule base:

(1) A ← B1 \(\land\) B2
(2) B1 ← C1 \(\land\) C2
(3) B2 ← C3 \(\land\) C4
C1 C2 C3 C4

Query: Is A true?

Goal A
Rule (1)
B1 \(\land\) B2

Goal B1
Rule (2)
C1 \(\land\) C2

Goal C1
success

Goal B2
Rule (3)
C3 \(\land\) C4

Goal C3
success

Goal C4
success

So, goal A is proved. (all paths must succeed)
**Example 2: Multiple Rules**

**Rule base:**

- $A \leftarrow B_1$
- $B_1 \leftarrow C_1$
- $B_2 \leftarrow C_3$
- $A \leftarrow B_2$
- $B_1 \leftarrow C_2$
- $B_2 \leftarrow C_4$
- $C_4$

**Query:** Is $A$ true?

So, goal $A$ is proved. (only one path must succeed)
Example 3: Variables

Rule base:

student(1234,sam).  enrolled(1234,csc324).
student(3456,joe).  enrolled(1234,csc364).
student(5678,lisa).  enrolled(1234,csc378).
student(6789,bart).  enrolled(3456,csc324).
                    enrolled(3456,csc364).
                    enrolled(5678,csc378).

takes(Name,Course) :- student(Number,Name),
                      enrolled(Number,Course).

% i.e., view course enrollment in terms of
% student names, instead of student numbers.

Query:
Find N and C such that takes(N,C) is true.

Answer:
N=sam,  C=csc324;
N=sam,  C=csc364;
N=sam,  C=csc378;
N=joe,  C=csc324;
N=joe,  C=csc364;
N=lisa, C=csc378;
no
Example 3 (continued)

Same rule base:

student(1234,sam). enrolled(1234,csc324).
student(3456,joe). enrolled(1234,csc364).
student(5678,lisa). enrolled(1234,csc378).
student(6789,bart). enrolled(3456,csc324).
enrolled(3456,csc364).
enrolled(5678,csc378).

takes(Name,Course) :- student(Number,Name), enrolled(Number,Course).

Query:
Find N such that takes(N,csc324) is true.

Answer:
N=sam;
N=joe;
no
Example 4: Backtracking

Rule base:

\[ p(X) :- q(X), r(X). \]
\[ q(d). \quad q(e). \quad q(f). \quad q(g). \]
\[ r(e). \quad r(g). \]

Query: Find \( x \) such that \( p(x) \) is true.

\[ p(X) \]
\[ \downarrow \]
\[ q(X), r(X) \]
\[ \downarrow \]
\[ X=d -> r(d) \text{ fail} \]
\[ X=e -> r(e) \text{ success (print "X=e")} \]
\[ X=f -> r(f) \text{ fail} \]
\[ X=g -> r(g) \text{ success (print "X=g")} \]
Example 5: Backtracking

Rule base:

\[ p(X) \leftarrow q(X), r(X,Y), s(Y) \].

\[ q(a). \quad r(a,b). \quad r(c,b). \quad s(c). \]

\[ q(c). \quad r(a,c). \quad r(c,c). \]

\[ r(a,d). \]

Query: Find \( x \) such that \( p(x) \) is true.

\[ p(X) \]

\[ \rightarrow q(X), r(X,Y), s(Y) \].

\[ \{ X/a \} \]

\[ \{ X/c \} \]

\[ r(a,Y), s(Y). \]

\[ \{ Y/b \} \]

\[ \{ Y/c \} \]

\[ s(b) \quad s(c) \quad s(d) \]

\[ \text{fail} \quad \text{success} \quad \text{fail} \]

\[ r(c,Y), s(Y). \]

\[ \{ Y/b \} \]

\[ \{ Y/c \} \]

\[ s(b) \quad s(c) \]

\[ \text{fail} \quad \text{success} \]
Hints on Debugging

We can follow the execution of Prolog programs with write statements. *e.g.*, 

**Rule base:**

\[
p(X) :- q(X), \text{write}(X), r(X).
\]

\[
q(a). \quad q(b). \quad q(c). \quad q(d). \quad q(e).
\]

\[
r(a). \quad r(d).
\]

**Query:** Find \( x \) such that \( p(x) \) is true.

Then Prolog prints:

\[
a
\]

\[
X = a
\]

\[
bcd
\]

\[
X = d
\]

\[
e
\]

\[
\text{no}
\]
Recursion in Prolog

If a predicate symbol occurs in both the head and body of a rule, then the rule is *recursive*.

For example,

\[
a(X) :- b(X,Y), a(Y).
\]

i.e., to prove \( a(X) \), Prolog must prove \( a(Y) \).

The predicate \( a \) acts like a recursive subroutine.

It is called a recursively defined predicate, or simply a recursive predicate.
Mutual Recursion

Recursion might be indirect, involving several rules. For example,

\[
\begin{align*}
  a(X) & : = b(X,Y), c(Y). \\
  c(Y) & : = d(Y,Z), a(Z).
\end{align*}
\]

Thus, to prove \( a(X) \),

- Prolog tries to prove \( c(Y) \) (and \( b(X,Y) \))
- so it tries to prove \( a(Z) \) (and \( d(Y,Z) \)).

i.e., to prove \( a(X) \), Prolog tries to prove \( a(Z) \).

The predicates \( a \) and \( c \) are said to be mutually recursive.
Non-Linear Recursion

When the head predicate appears multiple times in the body of a rule, then the recursion is said to be \textit{non-linear}.

For example,

\[
a(X) :- b(X,Y), a(Y), c(Y,Z), a(Z).
\]

i.e., to prove \(a(X)\), Prolog tries to prove \textit{both} \(a(Y)\) and \(a(Z)\).

This generates a \textit{recursive proof tree}. 
Example (Linear Recursion)

A stack of 4 toy blocks.

```
   a
   b
   c
   d
```

Rules:

1. `above(X,Y) :- on(X,Y).`
2. `above(X,Z) :- on(X,Y), above(Y,Z).`
3. `on(a,b).`
4. `on(b,c).`
5. `on(c,d).`

Query: `?- above(a,d)`

Use top-down inference.
All leaves are true, so the root is true, i.e., above(a,d) is true.
Observation

Changing the order of rules and/or rule premises can cause problems for Prolog. Example:

(1) above(X,Z) :- above(Y,Z), on(X,Y).
(2) above(X,Y) :- on(X,Y).

Because Prolog processes premises from left to right, and rules from first to last, rule (1) causes an infinite loop.
This is a flaw in Prolog.
Beyond Horn Logic

• So far, we have studied what is known as *pure* logic programming, in which all the rules are Horn.

• For some applications, however, we need to go beyond this.

• For instance, we often need
  – Negation
  – Existential quantification
  – Arithmetic

• Fortunately, these can easily be accommodated by simple extensions to the logic-programming framework,
Negation in Prolog

- Prolog uses negation as failure.

- *I.e.*, if you cannot prove something is true, then assume it is false. *E.g.*, unless we have reason to believe otherwise, we assume the sun will rise tomorrow.

- This is NOT logical negation, but it is easy to implement, and it is typical of much common-sense reasoning.

- In Prolog, negation may appear only in queries and in rule bodies.

- For example, the rule

  \[ a \leftarrow b \land \sim c \]

  is written in Prolog as

  \[ a : - b, \text{not } c. \]

  and it means, "infer \( a \) if \( b \) can be inferred and \( c \) cannot be inferred."
Example

loves(bill,X) :- pretty(X), female(X),
             not loves(tom,X).

_i.e._, Bill loves any pretty female, unless Tom loves her.

loves(tom,X) :- famous(X), female(X),
             not dead(X).

_i.e._, Tom loves any famous living female.

female(marilyn-monroe). famous(marilyn-monroe).
female(cindy-crawford). famous(cindy-crawford).
female(martha-stewart). famous(martha-stewart).
female(girl-next-door).

pretty(marilyn-monroe). dead(marilyn-monroe).
pretty(cindy-crawford).
pretty(girl-next-door).

| ?- loves(tom,X).
  X = cindy-crawford;
  X = martha-stewart;
  no

| ?- loves(bill,X).
  X = marilyn-monroe;
  X = girl-next-door;
  no
Safety

Consider the following rule:

(*) \( \text{hates}(\text{tom}, X) :\neg \text{loves}(\text{tom}, X). \)

This may NOT be what we want, for several reasons:

- The answer is \emph{infinite}, since for any person \( p \) not mentioned in the database, we cannot infer \( \text{loves}(\text{tom}, p) \), so we must infer \( \text{hates}(\text{tom}, p) \).

Rule (*) is therefore said to be \emph{unsafe}.

- The rule does not require \( x \) to be a person. \emph{e.g.}, since we cannot infer

  \[
  \begin{align*}
  \text{loves}(\text{tom}, \text{hammer}) \\
  \text{loves}(\text{tom}, \text{verbs}) \\
  \text{loves}(\text{tom}, \text{green}) \\
  \text{loves}(\text{tom}, \text{abc})
  \end{align*}
  \]

we must infer that \text{tom} hates all these things.
Safety (Cont’d)

To avoid these problems, rules with negation should be guarded:

\[
\text{hates(tom,X)} :\text{ female(x), pretty(X), not loves(tom,X).}
\]

\text{i.e., Tom hates every pretty female whom he does not love.}

Here, female and pretty are called guard literals. They guard against safety problems by binding x to specific values in the database.
Quantified Rule Bodies

∀X [happy(X) ← ∀Y loves(Y, X)]

i.e., A person is happy if everyone loves him. This rule is not Horn.

∀X [happy(X) ← ∃Y loves(Y, X)]

i.e., A person is happy if someone loves him. This rule is not Horn either, but it is equivalent to the following Horn rule:

∀X ∀Y [happy(X) ← loves(Y, X)]

Why? (Left as an exercise)

Examples:

loves(bill, mary) ⇒ happy(mary)  \{X\mary, Y\bill\}
loves(bill, cindy) ⇒ happy(cindy)  \{X\cindy, Y\bill\}
loves(tom, cindy) ⇒ happy(cindy)  \{X\cindy, Y\tom\}

So, in Horn logic, existential quantifiers can appear in the premise of a rule.

They can also appear in queries, since a rule premise is just a query placed inside a rule.
Declarative Arithmetic

What we would like:

- Given a set of equations with variables, find values of the variables that satisfy the equations.

  eg., query: \( X + 3 = 5 \).  
  answer: \( X = 2 \)

  query: \( X + Y = 1, \ X - Y = 2 \).  
  answer: \( X = 3/2, \ Y = -1/2 \)

  query: \( X^2 = 4 \).  
  answers: \( X = 2 \)
                 \( X = -2 \)

  query: \( X + Y = 0, \ 2X + 2Y = 1 \).  
  answer: no  
  (no solutions since equations are contradictory)

  query: \( X = 1, \ X = 2 \).  
  answer: no
Declarative Arithmetic (Cont’d)

There are two problems with this ideal.
(1) There may be infinitely many answers
eg. query: \( x + y = 0 \).
answers: \( x = 0, y = 0 \)
\( x = 1, y = -1 \)
\( x = 2, y = -2 \)
etc.

(2) The solutions may be difficult (or impossible) to compute
eg. query: \( xy + xy^2 + y^2x = 10 \).
\( (xy)^2 + x^2 + y^2 = 6 \).
answers: ??

These are really problems in numerical analysis, not logic programming.
Dealing with These Problems

Prolog takes a simple, but practical approach (though somewhat procedural and non-logical).

- Require that queries have the form
  \[ x_1 \text{ is } \phi_1, \ x_2 \text{ is } \phi_2, \ldots \ x_n \text{ is } \phi_n, \]
  where each \( \phi_i \) is an arithmetic expression and each \( x_i \) is a variable or a constant.

This query is interpreted to mean
\[ (x_1 = \phi_1) \land (x_2 = \phi_2) \land \ldots \land (x_n = \phi_n). \]
This is processed from left to right (as usual):

First \( x_1 \) is set to the value of \( \phi_1 \)
then \( x_2 \) is set to the value of \( \phi_2 \)

... 
\( x_n \) is set to the value of \( \phi_n \).

Note: once a variable is assigned a value, it is fixed, i.e., it cannot change.
Examples

|? - X is 5+7.
  X = 12

|? - X is 5+7, Y is X-2.
  X = 12
  Y = 10
  (* left-to-right evaluation *)

|? - Y is X-2, X is 5+7.
  no
  (* X is unbound here *)

|? - 7 is 4+3.
  yes

|? - 8 is 4+3.
  no

A variable can only be given one value. e.g.,

|? - X is 4, X is 5.
  no

i.e., there is no value of X such that

\[ X = 4 \land X = 5. \]
Arithmetic in Rule Bodies

\[ \text{square}(X,Y) :- Y \text{ is } X \times X. \]

e.g. \texttt{l? - } \text{square}(5,Y).
    \begin{align*}
    &Y = 25 \\
    &\text{l? - } \text{square}(5,25).
    &\text{yes} \\
    &\text{l? - } \text{square}(5,13).
    &\text{no} \\
    &\text{l? - } \text{square}(X,25) \\
    &\text{Error: } X \text{ is unbound.}
\end{align*}

i.e., The query \texttt{square}(X,25) becomes the subquery \texttt{25 is } X \times X, in which \texttt{x} is unbound.