Logic Programming and Prolog

Logic PLs are neither procedural nor functional.

- Specify **relations** between objects
  - `larger(3,2)`
  - `father(tom,jane)`

- Separate logic from control:
  - Programmer declares **what** facts and relations are true.
  - System determines **how** to use facts to solve problems.
  - System **instantiates** variables in order to make relations true.

- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
  - Higher-level than imperative (and functional) languages.
Suppose we state these **facts**:

- male(charlie).
- female(alice).
- male(bob).
- female(eve).

We can then make **queries**:

?- male(charlie).
  true

?- male(eve).
  false

?- female(Person).
  Person = alice;
  Person = eve;
  false

?- parent(Person, bob).
  Person = charlie;
  Person = eve;
  false

?- parent(Person, bob),
   female(Person).
  Person = eve;
  false

parent(charlie,bob).
parent(eve,bob).
parent(charlie,alice).
parent(eve,alice).
We can also state rules, such as this one:

\[
\text{sibling}(X, Y) :- \text{parent}(P, X), \\
\text{parent}(P, Y).
\]

Then the queries become more interesting:

?- sibling(charlie, eve).
false

?- sibling(bob, Sib).
Sib = bob;  
Sib = alice;  
Sib = bob;  
Sib = alice;  
false
Logic vs Functional Programming

In FP, we program with **functions**.

- \( f(x, y, z) = x \times y + z - 2 \)
- Given a function, we can only ask one kind of question:
  
  *Here are the argument values; tell me what the function’s value is.*

In LP, we program with **relations**.

- \( r(x, y, z) = \{(1, 2, 3), (9, 5, 13), (0, 0, 4)\} \)
- Given a predicate, we can ask many kinds of question:
  
  *Here are some of the argument values; tell me what the others have to be in order to make a true statement.*
Logic Programming

- A program consists of facts and rules.
- Running a program means asking queries.
- The language tries to find one or more ways to prove that the query is true.
- This may have the side effect of freezing variable values.
- The language determines how to do all of this, not the program.
- How does the language do it? Using unification, resolution, and backtracking.
Prolog Syntax

**constants** and **variables**:

- Constants are:
  - Atoms: any string consisting of alphanumerics and underscore (_), starting with a lowercase letter.
    E.g.: charilie, bob, a_book, book_n10, ...
  - Numbers.
    E.g.: 1, -2, 6.02e-23, 99.1, ...

- Variables are strings beginning with `_` or uppercase letters.
  E.g., Person, _X, _, A_very_long_var_name, ...

Example: in

```
parent(Person, bob).
```

bob is a Constant, and Person is a Variable.
Prolog Syntax

<const> ::= <atom> | <number>
<var> ::= <ucase> <string> | _<string>
<atom> ::= <lcase> <string>
<letter> ::= <ucase> | <lcase>
<string> ::= epsilon | <letter><string> | <number><string> | _<string>
<ucase> ::= A | ... | Z
<lcase> ::= a | ... | z
<number> ::= ...
Prolog Syntax

terms are inductively defined:

• Constants and variables are terms.

• Compound terms – applications of any $n$-ary functor to any $n$ terms – are terms.

  author(jane_austen)
  book(persuasion,author(jane_austen))

• Nothing else is a term.

terms denote entities in the world.

Note: Prolog distinguishes numbers and variables from other terms in certain contexts, but is otherwise untyped/monotyped.
Prolog Syntax & Semantics

\[
\text{<term> ::= <const> | <var> | <functor> '('<term> { , <term> } ')'}
\]

terms denote entities in the ‘world’.
Prolog Syntax

atomic formulae consist of an $n$-ary relation (also called predicate) applied to $n$ terms (also called arguments).

Note: syntactically, formulae look like terms because of Prolog’s weak typing. But formulae are either true or false; terms denote entities in the world.

In Prolog, atomic formulae can be used in three ways:

- As facts: in which they assert truth,
  e.g., own(mary,book(persuasion,author(jane_austen)))
- As queries: in which they enquire about truth,
  e.g., own(mary,book(_,author(_)))
- As components of more complicated statements (see below).
Prolog Syntax

\[
\begin{align*}
\text{<term>} & ::= \text{<const>} | \text{<var>} | \\
& \quad \text{<functor>} \ ( \text{<term>} \ \{ \ , \ \text{<term>} \ \} \ )
\end{align*}
\]

\[
\begin{align*}
\text{<pred>} & ::= \text{<pname>} \ ( \text{<term>} \ \{ \ , \ \text{<term>} \ \} \ )
\end{align*}
\]

terms denote entities in the ‘world’.

predicates specify relations among entities.
Prolog Queries

A **query** is a proposed fact that is to be proven.

- If the query has no variables, returns true/false.
- If the query has variables, returns appropriate values of variables (called a substitution).

?- male(charlie).
true

?- male(eve).
false

?- female(Person).
Person = alice;
Person = eve;
false
Prolog Syntax

**complex formulae** are formed by combining simpler formulae. We only discuss conjunction and implication.

- conjunction: in Prolog, *and* looks like ’,’

?- parent(Person, bob), female(Person).
Person = eve;
false

Conjunctions can be used as queries, but not as facts. Using multiple facts is an implicit conjunction.
Prolog Syntax

- implication: in Prolog, these are very special.

They are written backwards (conclusion first), and the conclusion must be an atomic formula. This backwards implication is written as ‘:-’, and is called a rule.

```
sibling(X, Y) :- parent(P, X), parent(P, Y).
```

Rules can be used in programs to assert implications, but not in queries.
Prolog and Horn Clauses

Recall:

- Prolog program: a set of **facts** and **rules**.
- Running a program: asking **queries**.
- System/Language tries to prove that the query is true.

Note that the Prolog system neither understands the facts and rules, nor can think or reason about them.

Prolog system builds proofs by making inferences based on a resolution theorem-prover for **Horn clauses** in first-order logic.
A Horn clause is: $c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n$

- **Consequent** $c$: an atomic formula.
- **Antecedents**: zero or more atomic formulae ($h_i$) conjoined.

Meaning of a Horn clause:
- “The consequent is true if the antecedents are all true”
- $c$ is true if $h_1$, $h_2$, $h_3$, $\ldots$, and $h_n$ are all true
Horn Clause

In Prolog, a Horn clause

\[ c \leftarrow h_1 \land \ldots \land h_n \]

is written

\[ c :\ldots : h_1, \ldots, h_n. \]

- Horn clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Tail
- Horn clause with No tail = Fact
- Horn clause with tail = Rule
- Horn clause with No head = Query
Prolog Horn Clause

A Horn clause with no tail:

\[
\text{male(charlie).}
\]

i.e., a \textbf{fact}: charlie is a male dependent on no other conditions.

A Horn clause with a tail:

\[
\text{father(charlie,bob) :- male(charlie), parent(charlie,bob).}
\]

i.e., a \textbf{rule}: charlie is the father of bob if charlie is male and charlie is a parent of bob's.
A Horn clause with no head:

?- male(charlie).

i.e., a **query**: is it true that charlie is a male?

Note that this corresponds to the head-less Horn clause:

:- male(charlie).
Syntax of a Prolog Program

A logic program (a theory that needs to be proven) is a collection of Horn clauses.

A simplified Prolog grammar:

```
<clause> ::= <pred> . | <-- fact
    <pred> :- <pred> { , <pred> } . <-- rule

<pred> ::= <pname> '(' <term> { , <term> } ')'  

<term> ::= <functor> '(' <term> { , <term> } ')'  
    | <const> | <var>

<const> ::= ...
<var> ::= ...
```
Meaning of Prolog Rules

A prolog rule must have the form:

\[ c :\!\!:- a_1, a_2, a_3, \ldots, a_n. \]

which means in logic:

\[ a_1 \land a_2 \land a_3 \land \cdots \land a_n \rightarrow c \]

Restrictions

- There can be zero or more antecedents, but they are conjoined; we cannot disjoin them.
- There cannot be more than one consequent.
non-Horn clauses

Many non-Horn formulae can be converted into logically equivalent Horn-formulae, using propositional tautologies, e.g.:

\[-\neg a \iff a\]  
\[-(a \lor b) \iff \neg a \land \neg b\]  
\[a \lor (b \land c) \iff (a \lor b) \land (a \lor c)\]  
\[a \rightarrow b \iff \neg a \lor b\]  

double negation  
DeMorgan  
distributivity  
implication
Bending the Restrictions?

**Getting disjoined antecedents**
Example:  \( a_1 \lor a_2 \lor a_3 \rightarrow c \).
Solution?

Syntactic sugar: ;

**Getting more than 1 consequent, conjoined**
Example:  \( a_1 \land a_2 \land a_3 \rightarrow c_1 \land c_2 \).
Solution?

**Getting more than 1 consequent, disjoined**
Example:  \( a_1 \land a_2 \land a_3 \rightarrow c_1 \lor c_2 \).
Solution?
Variables

Variables may appear in antecedents & consequent of Horn clauses.

Prolog interprets free variables of a rule *universally*.

\[ c(X_1, \ldots, X_n) :- f(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \]

is, in first-order logic:

\[ \forall X_1, \ldots, X_n, Y_1, \ldots, Y_m \cdot c(X_1, \ldots, X_n) \leftarrow f(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \]

or, equivalently:

\[ \forall X_1, \ldots, X_n \cdot c(X_1, \ldots, X_n) \leftarrow \exists Y_1, \ldots, Y_m \cdot f(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \]
Variables

Because of this equivalence, we sometimes say that free variables that do not appear in the head are quantified *existentially*.

Recall that a Prolog query corresponds to a head-less Horn clause.

Thus, the Prolog query \(?- q(X_1, \ldots, X_n)\) means

$$\exists X_1, \ldots, X_n \cdot q(X_1, \ldots, X_n)$$

That is, the Prolog system tries to find whether there exist values for the variables $X_1 \ldots X_n$ such that $q(X_1, \ldots, X_n)$ is true.
Horn Clauses with Variables

\( \text{isaMother}(X) \) :- female(X), parent(X, Y).

In first-order logic:

\[ \forall X \cdot \text{isaMother}(X) \leftrightarrow \exists Y \cdot \text{parent}(X, Y) \land \text{female}(X). \]
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.4)
...
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- ['family'].  
% family compiled 0.00 sec, 2,856 bytes
true.

?- parent(Person, bob).
Person = charlie ;  
% can only ask quires
% no facts/rules can be added
Person = eve ;
false.

?-
swi-prolog interface on CDF

?- [user].

\[\text{\texttt{\% user://1 compiled 0.00 sec, 652 bytes}}\]
Horn Clauses with Variables

% swipl
Welcome to SWI-Prolog ...

?- ['family2'].
Warning: /u/afsaneh/prolog/family2.pl:24:
    Singleton variables: [Y]
% family2 compiled 0.00 sec, 3,032 bytes
true.

?- isaMother(X).
X = eve;
X = eve;
false

No singleton variables:

    isaMother(X) :- female(X), parent(X, _).
Execution of Prolog Programs

- **Unification**: variable bindings.

- **Backward Chaining/Top-Down Reasoning/Goal-Directed Reasoning**: Reduces a goal to one or more subgoals.

- **Backtracking**: Systematically searches for all possible solutions that can be obtained via unification and backchaining.
Unification

Two atomic formulae unify if and only if they can be made syntactically identical by replacing their variables by other terms.

- `parent(bob,Y)` unifies with `parent(bob,sue)` by replacing Y by sue.
  
  ```prolog
  ?- parent(bob,Y)=parent(bob,sue).
  Y = sue.
  ```

- `parent(bob,Y)` unifies with `parent(X,sue)` by replacing Y by sue and X by bob.
  
  ```prolog
  ?- parent(bob,Y)=parent(X,sue).
  Y = sue
  X = bob.
  ```
Unification

- A **substitution** is a function that maps variables to Prolog terms, e.g., \{ X\textbackslash sue, Y\textbackslash bob \}

- An **instantiation** is an application of a substitution to all of the free variables in a formula or term.

If $S$ is a substitution and $T$ is a formula or term, then $ST$ (or $S(T)$) denotes the instantiation of $T$ by $S$.

\[
T = \text{parent}(X,Y) \\
S = \{ X\textbackslash sue, Y\textbackslash bob \} \\
ST = \text{parent}(\text{sue}, \text{bob})
\]

\[
T = \text{likes}(X,X) \\
S = \{ X\textbackslash bob \} \\
ST = \text{likes}(\text{bob}, \text{bob})
\]
Unification

- C is a **common instance** of formulae/terms A and B, if there exist substitutions S1 and S2 such that $C = S1(A) = S2(B)$.

  \[
  A = \text{parent}(\text{bob}, X), \quad B = \text{parent}(Y, \text{sue}) \\
  S1 = \{ X \backslash \text{sue} \}, \quad S2 = \{ Y \backslash \text{bob} \} \\
  S1(A) = S2(B) = \text{parent}(\text{bob}, \text{sue}) = C
  \]
Unification

• C is a **common instance** of formulae/terms A and B, if there exist substitutions S1 and S2 such that C = S1(A) = S2(B).

  \[ A = \text{parent}(bob,X), \quad B = \text{parent}(Y,sue) \]
  \[ S1 = \{ X\backslash\text{sue} \}, \quad S2 = \{ Y\backslash\text{bob} \} \]
  \[ S1(A) = S2(B) = \text{parent}(bob,sue) = C \]

• A and B are **unifiable** if they have a common instance C. A substitution that produces a common instance is called a **unifier** of A and B.

  \[ \text{parent}(bob,X) \text{ and } \text{parent}(Y,sue) \text{ are unifiable:} \]
  \[ \{ X\backslash\text{sue}, \ Y\backslash\text{bob} \} \text{ is the unifier} \]
Unification

Examples:
\[ p(a,a) \ ? \ p(a,a) \]
\[ p(a,b) \ ? \ p(a,a) \]
\[ p(X,X) \ ? \ p(b,b) \]
\[ p(X,X) \ ? \ p(c,c) \]
\[ p(X,X) \ ? \ p(b,c) \]
\[ p(X,b) \ ? \ p(Y,Y) \]
\[ p(X,Z,Z) \ ? \ p(Y,Y,b) \]
\[ p(X,b,X) \ ? \ p(Y,Y,c) \]
Unification

Examples:

\( p(a,a) \) unifies with \( p(a,a) \).

\( p(a,b) \) does not unify with \( p(a,a) \).

\( p(X,X) \) unifies with \( p(b,b) \) and with \( p(c,c) \), but not with \( p(b,c) \).

\( p(X,b) \) unifies with \( p(Y,Y) \) with unifier \( \{ X\b, Y\b \} \) to become \( p(b,b) \).

\( p(X,Z,Z) \) unifies with \( p(Y,Y,b) \) with unifier \( \{ X\b, Y\b, Z\b \} \) to become \( p(b,b,b) \).

\( p(X,b,X) \) does not unify with \( p(Y,Y,c) \).
Unification

Examples:

- \( p(f(X),X) \) unifies with \( p(Y,b) \) with unifier \( \{X\backslash b, \ Y\backslash f(b)\} \) to become \( p(f(b),b) \).

- \( p(b,f(X,Y),c) \) unifies with \( p(U,f(U,V),V) \) with unifier \( \{X\backslash b, \ Y\backslash c, \ U\backslash b, \ V\backslash c\} \) to become \( p(b,f(b,c),c) \).
Unification

p(b, f(X,X), c) does not unify with p(U, f(U, V), V).
Reason:

- To make the first arguments equal, we must replace U by b.
- To make the third arguments equal, we must replace V by c.
- These substitutions convert p(U, f(U, V), V) into p(b, f(b, c), c).
- However, no substitution for X will convert p(b, f(X,X), c) into p(b, f(b, c), c).
Unification

\[ p(f(X), X) \] should *not* unify with \[ p(Y, Y) \].

Reason:

- Any unification would require that
  \[ f(X) = Y \] and \[ Y = X \]
- But no substitution can make
  \[ f(X) = X \]

However, Prolog claims they unify:

\[
?- \quad p(f(X), X) = p(Y, Y).
X = Y, \quad Y = f(Y).
\]

?-
Unification

If a rule has a variable that appears only once, that variable is called a “singleton variable”.

Its value doesn’t matter — it doesn’t have to match anything elsewhere in the rule.

\[
\text{isaMother}(X) :- \text{female}(X), \text{parent}(X, Y).
\]

Such a variable consumes resources at run time.

We can replace it with \( _ \), the **anonymous variable**. It matches anything. If we don’t, Prolog will warn us.

Note that \( p(_, _) \) unifies with \( p(b, c) \). Every instance of the anonymous variable refers to a different, **unique** variable.
Most General Unifier (MGU)

The atomic formulas $p(X,f(Y))$ and $p(g(U),V)$ have infinitely many unifiers.

- $\{X\leftarrow g(a), \ Y\leftarrow b, \ U\leftarrow a, \ V\leftarrow f(b)\}$ unifies them to give $p(g(a),f(b))$.

- $\{X\leftarrow g(c), \ Y\leftarrow d, \ U\leftarrow c, \ V\leftarrow f(d)\}$ unifies them to give $p(g(c),f(d))$. 
Most General Unifier (MGU)

The atomic formulas $p(X,f(Y))$ and $p(g(U),V)$ have infinitely many unifiers.

- $\{X\backslash g(a), \ Y\backslash b, \ U\backslash a, \ V\backslash f(b)\}$
  unifies them to give $p(g(a),f(b))$.

- $\{X\backslash g(c), \ Y\backslash d, \ U\backslash c, \ V\backslash f(d)\}$
  unifies them to give $p(g(c),f(d))$.

However, these unifiers are more specific than necessary.

Their most general unifier (MGU) is $\{X\backslash g(U), \ V\backslash f(Y)\}$ that unifies the two atomic formulas to give $p(g(U),f(Y))$.

Every other unifier results in an atomic formula of this form.

The MGU uses variables to fill in as few details as possible.
Most General Unifier

Example:

\[ f(W, g(Z), Z) \]

\[ f(X, Y, h(X)) \]

To unify these two formulae, we need

\[ Y = g(Z) \]
\[ Z = h(X) \]
\[ X = W \]

Working backwards from \( W \), we get

\[ Y = g(Z) = g(h(W)) \]
\[ Z = h(X) = h(W) \]
\[ X = W \]

So, the MGU is

\[ \{ X \backslash W, \ Y \backslash g(h(W)), \ Z \backslash h(W) \} \]
Most General Unifier

The substitution that results in the most general instance is called the **most general unifier** (MGU).

It is **unique**, up to consistent renaming of variables.

MGU is the one that Prolog computes.

A substitution $\sigma$ is the MGU of a set of expressions $E$ if:

- it unifies $E$,
  and
- for any unifier $\omega$ of $E$, there is a unifier $\lambda$, such that $\omega(E) = \lambda \circ \sigma(E)$. 
Most General Unifier

E.g., given $p(X,f(Y))$ and $p(g(U),V)$, an MGU $\sigma = \{ X\leftarrow g(U), V\leftarrow f(Y) \}$, and a unifier $\omega = \{ X\leftarrow g(a), Y\leftarrow b, U\leftarrow a, V\leftarrow f(b) \}$,

we have $\omega(p(X,f(Y))) = p(g(a),f(b))$,

$\sigma(p(X,f(Y))) = p(g(U),f(Y))$,
so exists $\lambda = \{ U\leftarrow a, Y\leftarrow b \}$,
so that $\omega(p(X,f(Y))) = \lambda(\sigma(p(X,f(Y))))$

also $\omega(p(g(U),V)) = p(g(a),f(b))$,

$\sigma(p(g(U),V)) = p(g(U),f(Y))$,
so for $\lambda = \{ U\leftarrow a, Y\leftarrow b \}$,
we have $\omega(p(g(U),V)) = \lambda(\sigma(p(g(U),V)))$
## Most General Unifier

Examples:

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>MGU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X,a)$</td>
<td>$f(a,Y)$</td>
<td></td>
</tr>
<tr>
<td>$f(h(X,a),b)$</td>
<td>$f(h(g(a,b),Y),b)$</td>
<td></td>
</tr>
<tr>
<td>$g(a,W,h(X))$</td>
<td>$g(Y,f(Y,Z),Z)$</td>
<td></td>
</tr>
<tr>
<td>$f(X,g(X),Z)$</td>
<td>$f(Z,Y,h(Y))$</td>
<td></td>
</tr>
<tr>
<td>$f(X,h(b,X))$</td>
<td>$f(g(P,a),h(b,g(Q,Q)))$</td>
<td></td>
</tr>
</tbody>
</table>