Stands for **Programming in Logic**.
Invented in approximately 1972.
Belongs to the logical, functional and declarative paradigms.
Based on first order predicate calculus.
Used for artificial intelligence, theorem proving, expert systems, and natural language processing.
Used as a standalone language or complements traditional languages.
Radically different than most other languages.
Each program consists of 2 components:
- database (*program*): contains facts and rules
- query: ask questions about relations
Two ways to state facts:

?- [user].
sunny.
% user://1 compiled 0.00 sec, 408 bytes
true.

(same as ?- consult(user).)

Or:

?- assert(sunny).  state the fact
true.
What facts can we describe?

1. Items
   - `assert(sunny)`.

2. Relationships between items:
   - `assert(likes(john,mary)).`

Query the database:
- `likes(john,mary).`
  - `true`.
- `likes(mary,john).`
  - `false`.
- `likes(john,sue).`
  - `false`. 
Relations

- Relations take an arbitrary number of parameters.
- Arguments can be legal Prolog terms: integer, atom, variable, structure.
- Atoms: letters, digits, underscore with lowercase characters, or anything in quotes.
  - Legal: hello, hi123, two_words, “G_1)!#)@blah”
  - Illegal: Hello, 123hi, _hello, two-words
- Variables: Any word beginning with a capital letter.
- Structures: Functors with a list of arguments.

Note: variables bind to values, not memory locations.

?- likes(john,Who).
Who = mary

Prolog will display one instantiation. Type a semicolon for more.
All satisfying \texttt{likes} relations:

\texttt{?- likes(Who1,Who2).
Who1 = john; Who2 = mary}

Constrain queries using variables:

\texttt{?- likes(Who,Who).
false.}

(People who like themselves.)

Use wild card to determine if some instantiation exists:

\texttt{?- likes(john,_).
true.}

(That is, john likes \texttt{someone}—we don’t care who.)

Wild cards can be used in conjunction with variables:

\texttt{?- likes(Who,\_).
Who = john}
Rules express conditional statements about our world. Consider the assertion: “All men are mortal.”

Expressible as modus ponens: \( \text{human} \rightarrow \text{mortal} \) (“human implies mortal.”)

mortal is a \textit{goal} (or \textit{head}), and human is a \textit{subgoal} (or \textit{body}).

In Prolog, we write it in the following form:

\[ \text{mortal} \leftarrow \text{human}. \]

Or more generally,

\[ \text{goal} \leftarrow \text{subgoal}. \]

There can be multiple subgoals. Example:

\[ \text{goal} \leftarrow \text{subgoal}_1, \ldots, \text{subgoal}_n. \]

This form is called a \textit{Horn clause}. 
?- assert(mortal(X) :- human(X)).
   true.
?- assert(human(socrates)).
   true.

Now we query:
?- mortal(socrates).
   true.

You can also ask who is mortal:
?- mortal(X).
   X = socrates
Prolog relies on everything it is told being true: both facts and rules.

e.g., if you tell Prolog the sky is green, it won’t argue with you.

?- assert(sky_is_green).
true.

This is called a closed world assumption.

For the semantics of the not goal to be correct, Prolog must assume that everything that is true has been asserted accordingly.

If only married(brian) and married(linda) are stated as facts, then only brian and linda are married as far as Prolog is concerned—nobody else.
Conjunction and Disjunction

Conjunction is expressed using commas:

?- fun(X) :- red(X), car(X).

Disjunction is expressed with semicolons or separate clauses:

?- fun(X) :- red(X); car(X).

...is the same as

?- fun(X) :- red(X).
?- fun(X) :- car(X).  Order of rules matters!
daughter(X,Y) :- mother(Y, X), female(X).

grandfather(X,Y) :- male(X), parent(X,Z), parent(Z,Y).

Quantification:
- Variables appearing in the goal are \textit{universally} quantified.
- Variables appearing only in the subgoal are \textit{existentially} quantified.

The grandfather goal reads as:
\[ \forall X, Y \exists Z : \text{grandfather}(X, Y) \leftarrow \text{male}(X), \text{parent}(X, Z), \text{parent}(Z, Y). \]
Prolog responds to queries using the *resolution principle*:

If $C_1$ and $C_2$ are rules and the head of $C_1$ matches one of the terms in the body of $C_2$, then replace the term in $C_2$ with the body of $C_1$.

Example:

$C_1$: happy($X$) :- workday($Z$), day_off($X$,$Z$).

$C_2$: go_walking($X$) :- happy($X$).

1. Query: ?- go_walking(emily).
2. Instantiate the rule: go_walking(emily) :- happy(emily).
3. Apply resolution principle:
   go_walking(emily) :- workday($Z$), day_off(emily,$Z$).
Consider again:

\[ C_1: \text{happy}(X) :- \text{workday}(Z), \text{day_off}(X,Z). \]
\[ C_2: \text{go_walking}(X) :- \text{happy}(X). \]

When the user queries \( ?- \text{go_walking}(\text{emily}) \), How does Prolog make the connection? \( \text{go_walking}(\text{emily}) \)

Answer: \textit{unification}. 
Unification Algorithm

1. Constants: any constant unifies with itself.
2. Structures: same functor, same arity, arguments unify recursively.
3. Variables: unify with anything.
   (a) Value: variable takes on the value.
   (b) Another Variable: unify by reference.

Some examples:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>X</td>
<td>5</td>
<td>X=5</td>
</tr>
<tr>
<td>love(X,me)</td>
<td>love(you,Y)</td>
<td>X=you, Y=me</td>
</tr>
<tr>
<td>love(X,Y)</td>
<td>love(you,Y)</td>
<td>X=you, Y=Y</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>error</td>
</tr>
<tr>
<td>love(X,Y)</td>
<td>foobar(you,Y)</td>
<td>error</td>
</tr>
<tr>
<td>c(X,c(Y,c(Z,n)))</td>
<td>c(he, c(she, c(it,n)))</td>
<td>X=he, Y=she, Z=it</td>
</tr>
<tr>
<td>love(X,Y)</td>
<td>love(you,f(Y))</td>
<td>X=you, Y=??</td>
</tr>
</tbody>
</table>
Prolog isn’t the only language to implement unification.

We’ve already studied one other: ML.

Consider formal parameter \texttt{int * 'b} and actual parameter \texttt{'a * real list}.

ML will unify: \texttt{'a = int, 'b = real list}
Consider:

\[ \text{equal}(Y, f(Y)). \]

Let's try unifying \( Y = f(Y) \). We have:

- \( \text{equal}(Y, f(Y)) \) no match
- \( \text{equal}(f(Y), f(f(Y))) \) no match
- \( \text{equal}(f(f(Y)), f(f(f(Y)))) \) no match
- \( \text{equal}(f(f(f(Y))), f(f(f(f(Y))))) \) no match

Infinite recursion!

This situation can be caught with an *occurs check*. 
When attempting to unify variable \( v \) and structure \( s \), an *occurs check* determines whether \( v \) is contained within \( s \). If so, unification fails.

- Prevents infinite loops or unsoundness.
- Inefficient to implement (linear in the size of the largest term).
- Most implementations of Prolog (like SWI Prolog) omit it.

Therefore, in SWI Prolog:

\[
\text{?- equal}(Y, f(Y)).
\]

\[
Y = f(Y).
\]

If you insist on the occurs check, you can force it in SWI:

\[
\text{?- unify_with_occurs_check}(X,f(X)).
\]

\[
\text{false}.
\]
There are two ways to answer a query:

1. *Forward chaining*: start with facts/rules and work forward.
2. *Backward chaining*: start with goal and work backward. (Used by Prolog).

If the body of a rule unifies with the heads of other rules in some particular order, it can be expressed as a tree.

- Forward chaining: most suitable for: many rules, few facts.
- Backward chaining: most suitable for: few rules, many facts.
Consider:

rainy(seattle).
rainy(rochester).
cold(seattle).
snowy(X) :- rainy(X), cold(X).

?- snowy(X).
More than one “application” of a rule:

```prolog
connect(Node,Node).
connect(N1,N2) :- edge(N1,Link), connect(Link,N2).
```

Now add some edges:

```prolog
?- assert(edge(a,b)).  ?- assert(edge(c,d)).
?- assert(edge(a,c)).  ?- assert(edge(d,e)).
?- assert(edge(b,d)).  ?- assert(edge(f,g)).

?- connect(a,e).
true.
```

```prolog
connect(a,e) :- edge(a,b), connect(b,e)
connect(b,e) :- edge(b,d), connect(d,e)
connect(d,e) :- edge(d,e), connect(e,e)
```

```prolog
?- connect(d,f).
false.
```
Prolog maintains a list of goals to be satisfied. When a goal is queried, all *subgoals* of the goal are added to the list.

- `goal(X,Y) :- subgoal1(X), subgoal2(Y).`

Prolog will try to satisfy *all* subgoals. If a subgoal cannot be satisfied, Prolog will try another way.

- `subgoal1(X) :- subsubgoal1(X).`
- `subgoal1(X) :- subsubgoal2(X), subsubgoal3(X).`

This is called *backtracking*.

Carried out through a tree data structure:

- Goal is a node.
- Subgoals are children of the node.
Consider:

\[
\text{rainy(seattle).} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- rainy(X), cold(X).} \\
\text{?- snowy(X).}
\]
?- rainy(seattle).  
?- rainy(rochester).  
?- cold(rochester).  
?- snowy(X) :- rainy(X), cold(X).

Print the backtrace by invoking `trace`, then `snowy(X)`.

Call: (6) snowy(_G466) ? creep
Call: (7) rainy(_G466) ? creep
Exit: (7) rainy(seattle) ? creep
Call: (7) cold(seattle) ? creep
Fail: (7) cold(seattle) ? creep
Redo: (7) rainy(_G466) ? creep
Exit: (7) rainy(rochester) ? creep
Call: (7) cold(rochester) ? creep
Exit: (7) cold(rochester) ? creep
Exit: (6) snowy(rochester) ? creep
X = rochester
Lists are denoted by \([ a, b, c ]\).

A *cons pair* is denoted \([X|Y]\) where \(X\) is the *head* and \(Y\) is the *tail*.

Rules for testing list membership:

?- \(\text{assert} (\text{member}(X, [X|Xs]))\).
?- \(\text{assert} (\text{member}(X, [Y|Ys]) :- \text{member}(X,Ys))\).

Testing membership:

?- \(\text{member}(b, [a,b,c])\).
true.
?- \(\text{member}(b, [a,c])\).
false.

You can also extract list membership:

?- \(\text{member}(X, [a,b,c])\).
\(X = a\); \(X = b\); \(X = c\).
Consider a list reverse rule:
\[
\text{reverse([],[]).}
\]
\[
\text{reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).}
\]

Reverse-accumulate:
\[
\text{reverse(Xs,Ys) :- reverse(Xs,[],Ys).}
\]
\[
\text{reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).}
\]
\[
\text{reverse([],Ys,Ys).}
\]

Invoking the reverse rule:
?- reverse([a,b,c], X).
\[
X = [c, b, a].
\]
?- reverse([a,b,c], [a,c,b]).
\text{false.}
The reverse rule at work:
You can tell Prolog to stop backtracking using the cut operator, !.

- Used to “commit” all unifications up to the point of the !
- Will never backtrack through any subgoal to the left of !
- Done to optimize performance.
- Generally requires intuition about the program.

Consider:

```prolog
prime_candidate(X) :- member(X, candidates), prime(X).
```

- Variable $X$ may appear several times in candidates.
- Once $X$ is found to be in candidates, no need to try other possibilities.
- Solution: use the cut operator.
  
  - `member(X, [X|_]) :- !.`
  - `member(X, [_|T]) :- member(X, T).`
The cut operator can also serve as an if-then-else construct:

\[
\text{statement} :\!\!\!\!\!\!:\:\text{condition}, \!, \!, \text{then}_\text{part}.
\]
\[
\text{statement} :\!\!\!\!\!\!:\:\text{else}_\text{part}.
\]

- Cut prevents the condition from being retested.
- If condition is true, subgoal \text{then}_\text{part} will be attempted.
- If \text{then}_\text{part} fails, the system will not backtrack into the condition.
- If first goal fails, the second goal will be tried.
One way to negate a subgoal is using predicate not:
unmarried_student(X) :- not(married(X)), student(X).

Definition of not (also known as \+):
not(Goal) :- call(Goal), !, fail.
not(Goal).

- Predicate fail unconditionally fails.
- Predicate call treats the input term as a goal and attempts to satisfy it.

Example:
single(Person) :- \+ married(Person, _), \+ married(_, Person).

Note: \+ indicates *inability to prove*—not falsehood.