Logic Programming — Prolog

Steven Zeil
Dec. 1, 2003

1 Overview
Logic programming is characterized by

• Statements
  – facts
  – inference rules

• Queries

programmer describes world using logic

programmer/user poses queries as to what is true in that world

an inference engine tries to determine the truth/falsehood of the query from the logical description.

2 Relations
In logic programming, a programmer spends more effort on saying

• “what is true”

than on

• “how to compute it”

In effect, functions are replaced by relations.

Impreter/functional programmers see + as the name for a function: you supply its operands and it returns a result.

But go back to grade school math, and you get a different view:

• What is 3 plus 2?
• What plus 2 equals 4?
• 3 plus what equals 6?

“plus” is not viewed here as an executable operator, but as a relation among triplets of numbers.
2.1 Definition

A relation of arity $n$ is a set of $n$-tuples.

Given a relation $R$, we can ask questions like

- Is $(x_1, x_2, \ldots, x_n) \in R$?
- For what values of $x_2$ is $(x_1, x_2, \ldots, x_n) \in R$?

For example, consider the relation `prereq` defined as

$$\{(cs150, cs250), (cs150, cs281), (cs250, cs361),
\quad (cs281, cs361), (cs361, cs350), (cs250, cs355),
\quad (cs281, cs390), (mt163, cs281)\}$$

We can use this relation to answer questions like

- Is $cs281$ an (immediate) prerequisite for $cs361$?
- What are the (immediate) prerequisites for $cs281$?

Furthermore, we can, with a few appropriate rules, infer the answers to questions like

- Must $cs250$ be taken before $cs390$?
- Can $cs281$ and $cs361$ be taken within the same semester?

Questions like

- Must $cs250$ be taken before $cs390$?
- Can $cs281$ and $cs361$ be taken within the same semester?

actually introduce new relations.

The relation `mustTakeBefore` can be defined as

$$\{(cs150, cs250), (cs150, cs281), (cs150, cs350),
\quad (cs150, cs361), (cs150, cs390), (cs250, cs350),
\quad (cs250, cs361), (cs281, cs350), (cs281, cs361),
\quad (cs281, cs390), (mt163, cs281), (mt163, cs350),
\quad (mt163, cs361), (mt163, cs390), (cs361, cs350)\}$$

This relation is symmetric. If $(x, y) \in canTakeTogether$, then $(y, x) \in canTakeTogether$.

Not all relations are binary. We have already mentioned `plus`:

$$\{(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 2), \ldots\}$$

which is not only not binary, it is not finite!

We can also have unary relations, such as `requiredCSCourse`:

$$\{cs150, cs250, cs281, cs300, cs350,\quad cs355, cs361, cs390\}$$

3 Prolog Basics

1. **Data Types**
3. Data Types

Data in Prolog is a mixture of ideas familiar from SML and Scheme:

1. Variables
   - Variables in Prolog must begin with a capital letter:
     \( X, \text{AVariable,AnotherVariable} \)
   - A special exception is \( _{} ^{\prime} \), which is an anonymous variable (as in SML).
   - Each occurrence of \( _{} ^{\prime} \) denotes a separate variable.
   - Many Prolog systems generate internal variable names that begin with \( _{} ^{\prime} \).
   - Variables are not typed — they can be bound to any value.

2. Simple Terms
   - Atoms must begin with lower-case letters or non-alphanumerics, or appear within single quotes:
     \( \text{atom, aTOM, +, ++++, ’Atom’} \)
   - Numbers: \( 1997, 3.14159 \)

3. Compound Terms

   The general form of a compound term is
   \[
   \langle \text{compound-term} \rangle ::= \text{atom}(\langle \text{terms} \rangle) \\
   \langle \text{terms} \rangle ::= \langle \text{term} \rangle \\
   \mid \langle \text{term} \rangle, \langle \text{terms} \rangle
   \]

   Examples:
   \[
   f(\text{g(h), 2}) \\
   +(\text{x, y}) \\
   \text{node}(10, \text{leaf}(1), \text{node}(20, \text{empty}, \text{leaf}(25)))
   \]

   Certain compound forms have “convenient” alternate syntaxes:
   - Arithmetic operators can be written in infix form: \( 1+2 \)
   - Lists can be written as
     \( [a,b,c,d] \) or as \( [x \mid L] \) where \( x \) is the first element and \( L \) is the list of remaining elements.
   - Strings can be written in double quotes (“abc”).
     A string is really a list of characters \( * \) and characters are really integers

3.2 Facts and Rules

1. Facts
   - Facts are simple statements of relationships:
Logic Programming — Prolog

preq(c150, c250).
preq(c150, c281).
preq(c250, c361).

The collection of all preq facts will define the preq relation.

Some more facts:
requiredCScourse(c150).
requiredCScourse(c250).
requiredCScourse(c281).

3.2.2 Rules

⟨rule⟩ ::= ⟨term⟩ :− ⟨terms⟩.

Rules describe relations using other relations. A term is in the relation if it satisfies any rule for that relation:
mustTakeBefore(C,D) :− preq(C,D).
mustTakeBefore(C,D) :−
    preq(C,E), mustTakeBefore(E,D).

mustTakeBefore(C,D) :− preq(C,D).
mustTakeBefore(C,D) :−
    preq(C,E), mustTakeBefore(E,D).

Read this as

C must be taken before D if C is a prerequisite of D, or
if there is some course E that C is a prerequisite for, and E
must be taken before D.

Describe a relation cannotTakeTogether(C,D), for courses C and D that must be taken in separate semesters.

3.3 Interacting with Prolog

Most Prolog systems are in one of 2 modes at any given time.

• Query mode
• Definition mode

3.3.1 Query Mode

This is the interactive mode of prolog.

Queries may be entered as

⟨query⟩ ::= ⟨terms⟩.

Sample queries:
X = abc.
f(g(h)) = f(g(h)).
f(g(h)) = f(h(g)).
X = "abc".
[X, b, c] = [a, b, Y].

• The Prolog system determines if the query is true/false by trying
to find a satisfactory assignment for each variable in the query.

• If it cannot, it responds “No” or “Fail”, indicating that the query
cannot be satisfied.

• If it can, it responds “Yes” or “Success”, and lists the variable
bindings that it found to satisfy the query.

The user can then

– hit Return to terminate the query, or
– type a semi-colon to request that the Prolog system find a
different set of variable bindings that satisfy the query.

From query mode, one can ask the system to load a file (in definition mode) via
[filename−atom].

For example:
['preq.pro'].

Alternatively, one can say:
consult('preq.pro').

3.3.2 Definition Mode

Used when reading from a file.

The system is reading facts and rules and storing them in its internal
data base.

Queries may be posted as

⟨query⟩ ::= :− ⟨terms⟩.

A more typical Prolog session might look like:
3.4 Unification

All variable binding in Prolog occurs via unification, a kind of pattern matching between terms.

- More general than SML’s pattern matching
  - SML: each variable can occur only once in a pattern
  - Prolog: variables can occur multiple times

3.4.1 Unification: Definition

A term $U$ is an instance of another term $T$ if $U$ is obtained by replacing variables $V_1, V_2, \ldots, V_k$ in $T$ by terms $u_1, u_2, \ldots, u_k$.

Terms $T_1$ and $T_2$ unify if they have a common instance $U$.

3.4.2 Unification: Examples

Unification occurs each time a fact/rule is matched against a query:

query: `prereq(cs250, Z)`.

rule 1: `prereq(cs150,cs250)`.

The two terms do not unify.

- There’s only one variable, $Z$
- and no substitution for it would make these terms identical.

query: `prereq(cs250, Z)`.

rule 2: `prereq(cs250,cs350)`.

This unifies with the query, under a substitution of $Z\rightarrow cs350$.

An easy way to experiment with unification is by using `=` in Prolog is neither an assignment nor an equality test. It means “unifies with”:

$X = 1$.

$1 = X$.

$X = y+1$.

$2+X=Y+3$.

$2+X=X+3$.

$Y+X=X+3$.

$f(X, g(h(z))) = f(x+y, Y)$.

$f(X, g(h(X))) = f(x+y, Y)$.

$f(X, g(h(Y))) = f(x+y, Y)$.

3.4.3 Most General Unifiers

In some cases, there is more than one unifier (substitution) that could be applied to unify two terms.

$f(g(X)) = f(Y)$.

This could be unified by the substitution $\{Y\rightarrow g(X)\}$, or by $\{X\rightarrow foo(z), \ Y\rightarrow g(foo(z))\}$, or by ...

Prolog always uses the most general unifier, the one that binds the fewest variables.

You can find the most general unifier by matching from the outermost operators:

Do $f(A, g(h(B)))$ and $f(h(c), g(A))$ unify?

- Outermost operators $f$ match, so they unify if
  - $A$ and $h(c)$ unify, and
  - $g(h(B))$ and $g(A)$ unify

  with the same substitution.

- Do $A$ and $h(c)$ unify?
  - Yes, under substitution $\{A\rightarrow h(c)\}$

- Do $g(h(B))$ and $g(A)$ unify under $\{A\rightarrow h(c)\}$?
  - Apply substitution to get $g(h(B))$ and $g(h(c))$
  - Outer operators ($g$) match, so they unify if
    * $h(B)$ and $h(c)$ unify.
    * Outer operators ($h$) match, so they unify if $B$ and $c$
      unify.
    - They do, under substitution $\{B\rightarrow c\}$

Summary:

Do $f(A, g(h(B)))$ and $f(h(c), g(A))$ unify?

- Yes, under $\{A\rightarrow h(c), \ B\rightarrow c\}$. 

---

[ preq.pro ].

prereq(X,cs361).
mustTakeBefore(X,cs361).
3.5 Lists

Unification applies to lists as well:

- \[[a, b, c] = L\].
- \[[a, b, c] = [a|L]\].
- \[[a, b, c] = [x|L]\].
- \[[a, b, c] = [X|L]\].
- \[[a] = [X|L]\].
- \[[] = [X|L]\].
- \[[a, b, c] = [X, Y |L]\].
- \[[a, b, c] = [X, Y, Z |L]\].
- \[[a, b(X), c] = [X, Y |L]\].

3.5.1 Common List Utilities

We can use our new knowledge of unification to set up some list relations:

- \(\text{member}(X, L)\) is satisfied if \(X\) is an element of the list \(L\).
- \(\text{append}(A, B, C)\) is satisfied if \(C\) is a list formed by appending list \(B\) onto the end of list \(A\).

Example:

\(\text{member}(X, L)\):

- \(\text{member}(X, [X|\_])\).
- \(\text{member}(X, [\_|L]) : - \text{member}(X, L)\).

Try this for queries:

- \(\text{member}(2, [1,2,3])\).
- \(\text{member}(4, [1,2,3])\).
- \(\text{member}(100, "zeil")\).
- \(\text{member}(A, [a,b,c])\).
- \(\text{member}(14, [a,b,c])\).
- \(\text{member}(14, [A,A,A])\).
- \(\text{member}(100, L)\).
- \(\text{member}("zeil", [X|L])\).

Use the ';' response to try and force multiple solutions.

Example:

\(\text{append}([| R|], S, [T | [d]])\):

- Compare to the two append rules:
  - \(\text{append}([-], B, B)\).
  - \(\text{append}([X|A], B, [X|C]) : - \text{append}(A, B, C)\).

- The first rule’s head does not unify, so we try the second rule.

- \(\text{append}([a | R], S, [T | [d]])\) unifies with \(\text{append}([X|A], B, [X|C])\) under unifier \(\{X\rightarrow a, A\rightarrow R, B\rightarrow S, T\rightarrow a, C\rightarrow [d]\}\).

- Apply that substitution to the right-hand-side of the rule. New goal is \(\text{append}(R, S, [d])\).

3.6 Unification and Rules

When a rule \(A : - B_1, \ldots, B_j\) is applied to a goal (query) \(G\):

- The rule’s “head” \(A\) is unified with \(G\).
  - Let \(\sigma\) be the most general unifier of \(A\) and \(G\).
  - \(\ldots\)

- The substitution \(\sigma\) is applied to the rule body \(B_1, \ldots, B_j\) to yield new subgoals \(B_1\sigma, \ldots, B_j\sigma\).
- The Prolog system attempts to prove the subgoals.

Example:

\(\text{append}([a | R], S, [T | [d]])\):

- Compare to the two append rules:
  - \(\text{append}([-], B, B)\).
  - \(\text{append}([X|A], B, [X|C]) : - \text{append}(A, B, C)\).

- The first rule’s head does not unify, so we try the second rule.

- \(\text{append}([a | R], S, [T | [d]])\) unifies with \(\text{append}([X|A], B, [X|C])\) under unifier \(\{X\rightarrow a, A\rightarrow R, B\rightarrow S, T\rightarrow a, C\rightarrow [d]\}\).

- Apply that substitution to the right-hand-side of the rule. New goal is \(\text{append}(R, S, [d])\).

Goal: \(\text{append}(R, S, [d])\):

- Compare to the two append rules:
  - \(\text{append}([-], B, B)\).
  - \(\text{append}([X|A], B, [X|C]) : - \text{append}(A, B, C)\).

- The first rule’s head unifies with \(\text{append}(R, S, [d])\) under unifier \(\{R\rightarrow [-], S\rightarrow [d], B\rightarrow [d]\}\).
4 Implementing LP

A query is just a list of terms. Each term is a goal.

- Goals are satisfied by unification with the heads of rules,
- but then the terms on the right-hand-side are added as additional goals.

The heart of a Prolog engine is the procedure:

```prolog
proc visit (goals: list<term>, s: unifier)
begin
  if goals == [] then
    succeed(s);
  else
    g = hd(goals);
    rest = tl(goals);
    solve(g, rest, s);
  end if;
end;

proc succeed(s: unifier)
begin
  print s;
  do
    get response;
    until (response == '\n')
    or (response == ';');
  if (response == '\n')
    abort; // and be happy
  end;
end;

proc solve(g: term, goals: list<term>, s: unifier)
begin
  for (i = 0; i < numRules; ++i)
    let rule[i] be A :- List;
  if (A unifies with g) then
    let σ be the m.g. unifier;
    newGoals = Listσ ∪ goalsσ;
    visit(newGoals, sσ);
  end if;
end loop;
end;
```

A common visualization of the Prolog inference process is:

```
<table>
<thead>
<tr>
<th>call</th>
<th>goal</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail</td>
<td>redo</td>
<td></td>
</tr>
</tbody>
</table>
```

Goals can chain together:

```
<table>
<thead>
<tr>
<th>call</th>
<th>goal</th>
<th>exit</th>
<th>call</th>
<th>goal</th>
<th>exit</th>
<th>redo</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail</td>
<td>redo</td>
<td>fail</td>
<td>redo</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

4.1 Example with Backtracking

Query: mustTakeBefore(X, cs361)

(Abbreviate mustTakeBefore as mtb)

```prolog
visit([mtb(X, cs361)], { })

visit([mtb(X, cs361)], { })
solve(mtb(X, cs361), [], { })
```

Term unifies with rule:

mustTakeBefore(C1, D1) :- prereq(C1, D1).

Unifier is \{X→C1, D1→cs361\}

```prolog
visit([mtb(X, cs361)], { })
solve(mtb(X, cs361), [], { })
visit([prereq(C1, cs361)],

\{X→C1, D1→cs361\})
```

Visit again:

```prolog
visit([mtb(X, cs361)], { })
solve(mtb(X, cs361), [], { })
visit([prereq(C1, cs361)],

\{X→C1, D1→cs361\})
solve(prereq(C1, cs361), [],

\{X→C1, D1→cs361\})
```
Several `prereq` facts fail to unify, e.g. `prereq(cs150,cs250)`
Eventually, we get to `prereq(cs250,cs361)`

Prints X=cs250
Suppose we hit ; to force a new solution.

Prints X=cs361
Suppose we hit ; again to force a new solution.
We'll loop through the rest of the rules, but none will unify with this term.

\[
\text{mustTakeBefore} \ (C2, D2)
\]: \ (- \ \text{prereq} \ (C2, E2), \ \text{mustTakeBefore} \ (E2, D2).
\]

Unifier is \(\{X \rightarrow C2, D2 \rightarrow cs361\}\)
This eventually unifies with a \texttt{prereq} fact and we succeed again, printing "\texttt{X=cs150}"

5 Impurities in Prolog

...