Programming Languages

Prolog

CSCI-GA.2110-001
Fall 2012
Prolog overview

- Stands for **Programming in Logic**.
- Invented in approximately 1972.
- Belongs to the logical & 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
Stating Facts

Two ways to state facts:

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

(same as ?- consult(user).)

Or:

?- assert(sunny). state the fact
true.
Stating Facts 2

What facts can we describe?

1. Items
   ?- assert(sunny).

2. Relationships between atoms:
   ?- assert(likes(john,mary)).

Query the database:
?- likes(john,mary).
true.
?- likes(mary,john).
false.
?- likes(john,sue).
false.
Prolog Terminology

- Functors: an atom (defined below) with arguments.
- Arguments can be legal Prolog terms: integer, atom, variable, structure.
- Atoms: lowercase characters, digits, underscore (not first), graphic characters (e.g. #,&,@) or anything in quotes.
  - Legal: hello, hi123, two_words, @pl, “G_1)!#)@blah”
  - Illegal: Hello, 123hi, _hello, two-words
- Variables: Any word beginning with a capital letter.
- Structures: Functors with a list of arguments.

Structures are also known as relations, compound terms, and predicates.

Like functional languages, variables bind to values (not memory locations). Unlike functional languages, there is no clear notion of input and output.

?- likes(john,Who).

Who = mary

Prolog will display one instantiation at a time. Type a semicolon for more.
More Relations

All satisfying likes relations:
?- likes(Who1,Who2).
Who1 = john; Who2 = mary

Constrain queries using variables:
?- likes(Who,Who).
false.
(People who like themselves.)

Use wild card to determine if some instantiation exists:
?- likes(john,_).
true.
(That is, john likes someone—we don’t care who.)

Wild cards can be used in conjunction with variables:
?- likes(Who,_).
Who = john
Rules

Rules express conditional statements about our world. Consider the assertion: “All men are mortal.”

Expressible as modus ponens: human → mortal (“human implies mortal.”)
mortal is a goal (or head), and human is a subgoal (or body).

In Prolog, we write it in the following form:
mortal ← human.

Or more generally,
goal ← subgoal.

There can be multiple subgoals. Example:
goal ← subgoal₁, ..., subgoalₙ.

This form is called a Horn clause.
Rules Example

?- 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
Closed World Assumption

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_color(green)).
  true.

This is called a closed world assumption.

For the semantics of the not goal to be correct, the universe of facts must be complete (everything that is true has been asserted accordingly.)

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

Conjunction is expressed using commas:

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

A red car.

Disjunction is expressed with semicolons or separate clauses:

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

Something red or a car.

...is the same as

?- fun(X) :- red(X).
?- fun(X) :- car(X).  Order of rules matters!

Subgoal red(X) will be attempted first, then car(X).
Multi-Variable Rules

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 *uni-versally* quantified.
- Variables appearing only in the subgoal are *exis-tentially* quantified.

The grandfather goal reads as:

\[ \forall X, Y \exists Z : \text{grandfather}(X, Y) \leftarrow \text{male}(X), \text{parent}(X, Z), \text{parent}(Z, Y). \]
Resolution Principle

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}(emily), How does Prolog connect the rules? \text{go\_walking}(emily) \text{happy}(X)

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>love(you,Y)</td>
</tr>
<tr>
<td>love(X,Y)</td>
<td>love(you,Y)</td>
<td>X=you,Y=me</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 Unification

?- a=a.
true.
?- a=b.
false.
?- foo(a,b) = foo(a,b).
true.
?- foo(a,X) = foo(a,b).
X=b.
?- X=a.
X=a.
?- A=B
A=B.
?- A=B, A=a, B=Y
A=a; B=a; Y=a.
Unification in ML

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}
Occurs Check

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

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

\[
\begin{align*}
\text{equal}(Y, f(Y)) & \quad \text{no match} \\
\text{equal}(f(Y), f(f(Y))) & \quad \text{no match} \\
\text{equal}(f(f(Y)), f(f(f(Y)))) & \quad \text{no match} \\
\text{equal}(f(f(f(Y))), f(f(f(f(Y))))) & \quad \text{no match} \\
\text{Infinite recursion!}
\end{align*}
\]

This situation can be caught with an *occurs check*. 
More on 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:

```prolog
?- equal(Y, f(Y)).
Y = f(Y).
```

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

```prolog
?- unify_with_occurs_check(X,f(X)).
false.
```
Execution Order

There are two ways to answer a query:

1. **Forward chaining**: start with facts/rules and work forward toward goal.
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. (Why?)
- Backward chaining: most suitable for: few rules, many facts. (Why?)
Consider:
rainy(seattle).
rainy(rochester).
cold(seattle).
snowy(X) :- rainy(X), cold(X).
?- snowy(X).
Reflexive Transitive Closure

More than one “application” of a rule:
connect(Node,Node).
connect(N1,N2) :- edge(N1,Link), connect(Link,N2).

Now add some edges:
?- 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.

    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)

?- connect(d,f).
false.
Backtracking

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

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

Print the backtrace by invoking \texttt{trace}., then \texttt{snowy}(X).

\begin{itemize}
  \item \texttt{Call: (6) snowy(_G466) ? creep}
  \item \texttt{Call: (7) rainy(_G466) ? creep}
  \item \texttt{Exit: (7) rainy(seattle) ? creep}
  \item \texttt{Call: (7) cold(seattle) ? creep}
  \item \texttt{Fail: (7) cold(seattle) ? creep}
  \item \texttt{Redo: (7) rainy(_G466) ? creep}
  \item \texttt{Exit: (7) rainy(rochester) ? creep}
  \item \texttt{Call: (7) cold(rochester) ? creep}
  \item \texttt{Exit: (7) cold(rochester) ? creep}
  \item \texttt{Exit: (6) snowy(rochester) ? creep}
\end{itemize}

\[X = \text{rochester}\]
Lists

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:

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

Testing membership:

\[
\begin{align*}
?-& \text{member}(b, [a,b,c]). \\
& \text{true}.
\end{align*}
\]

\[
\begin{align*}
?-& \text{member}(b, [a,c]). \\
& \text{false}.
\end{align*}
\]

You can also extract list membership:

\[
\begin{align*}
?-& \text{member}(X, [a,b,c]). \\
& X = a; X = b; X = c.
\end{align*}
\]
Reversing Lists

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:

? predicate reverse([a,b,c], X).
\[
X = [c, b, a].
\]
?- predicate reverse([a,b,c], [a,c,b]).
\[
\text{false.}
\]
The reverse rule at work:
Cut Operator

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:

```
prime_candidate(X,Candidates) :- 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).
```
More on Cut

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 then\_part will be attempted.
- If then\_part fails, the system will not backtrack into the condition.
- Because it will not backtrack into the condition, it also will not attempt to try the other subgoal, else\_part.
- If first goal fails (meaning the condition failed), else\_part will be tried.
Negation

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—\textbf{not} falsehood.