Programming Languages

Prolog

CSCI-GA.2110-001
Summer 2011
Prolog overview

- Stands for **Programming in Logic**.
- Invented in approximately 1972.
- Belongs to the logical 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 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 express conditional statements about our world. Consider the assertion: “All men are mortal.”

Expressible as modus ponens: $\text{human} \rightarrow \text{mortal}$.

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

?- 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).
$X = \text{socrates}$
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!

Consider:

?- assert(car(jeep)). assert(car(ford)). assert(red(jeep)).
?- fun(jeep).
true.
?- fun(ford).
false.
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:
\[ \forall x, y \exists z : \text{grandfather}(x, y) \iff \text{male}(x), \text{parent}(x, z), \text{parent}(z, y). \]
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:

$C_1$: \(\text{happy}(X) \leftarrow \text{workday}(Z), \text{day\_off}(X,Z).\)

$C_2$: \(\text{go\_walking}(X) \leftarrow \text{happy}(X).\)

1. Query: \(\text{?- go\_walking(emily)}.\)
2. Instantiate the rule: \(\text{go\_walking(emily)} \leftarrow \text{happy(emily)}.\)
3. Apply resolution principle:
   \(\text{go\_walking(emily)} \leftarrow \text{workday}(Z), \text{day\_off(emily,Z)}.\)
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 make the connection? \[ \text{go\_walking}(emily) \quad \text{go\_walking}(X) \]

Answer: \textit{unification}. 
Unification Algorithm

1. Constants: Any constant unifies with itself.
2. Structures: Same functor, same arity, arguments unify recursively.
   (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>
Consider:
\[
\text{equal}(Y, f(Y)).
\]

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

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

Infinite recursion!

This situation can be caught with an \textit{occurs check}. 
An *occurs check* causes unification to fail if variable is contained within a structure.

- 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.
```
There are two ways to answer a query:

1. *Forward chaining*: start with existing clauses 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.
Consider:

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

\[
\text{rainy(seattle).} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- rainy(X), cold(X).} \\
?\text{- snowy(X).}
\]
?- 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 \textit{cons pair} is denoted \([X|Y]\) where \(X\) is the \textit{head} and \(Y\) is the \textit{tail}.

Rules for testing list membership:

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

Testing membership:

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

You can also extract list membership:

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

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

Invoking the reverse rule:
?- reverse([a,b,c], X).
X = [c, b, a].
?- reverse([a,b,c], [a,c,b]).
false.
The reverse rule at work:

reverse([a,b,c],[c,b,a])
append([c,b],[a],[c,b,a])
append([b],[a],[b,a])
append([],[a],[a])
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:

\[
\text{statement} :\!- \text{ condition}, !, \text{ then_part}.
\]

\[
\text{statement} :\!- \text{ else_part}.
\]

- Cut prevents the condition from being retested.
- If condition is true, subgoal \text{ then_part} will be attempted.
- If \text{ then_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:
\[
\text{unmarried\_student}(X) :\neg \text{not(married}(X)), \text{student}(X).
\]

Definition of not (also known as \(+\)):
\[
\text{not}(\text{Goal}) :\neg \text{call}(\text{Goal}), !, \text{fail}.
\]
\[
\text{not}(\text{Goal}).
\]

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

Example:
\[
\text{single}(\text{Person}) :\neg \text{\(+\) married}(\text{Person,}_), \text{\(+\) married}(\_,\text{Person}).
\]

Note: \(+\) indicates \textit{inability to prove}—not falsehood.