Logic Programming: Prolog

COMS W4115

Aristotle

Prof. Stephen A. Edwards
Spring 2007
Columbia University
Department of Computer Science
Logic

All Caltech graduates are nerds.

Stephen is a Caltech graduate.

Is Stephen a nerd?
All Caltech graduates are nerds.  \( \text{nerd}(X) :\!-\! \text{techer}(X) \).  

Stephen is a Caltech graduate.  \( \text{techer}(\text{stephen}) \).  

Is Stephen a nerd?  \( \text{?\!-\!nerd}(\text{stephen}) \).  

yes
More Logic

“My Enemy’s Enemy is My Friend.”

friend(X,Z) :-
    enemy(X,Y), enemy(Y,Z).

enemy(stephen, ryan).
enemy(ryan, jordan).
enemy(jordan, jacob).

?- friend(stephen,jordan).
yes

?- friend(stephen,X).
X = jordan

?- friend(X, Y).
X = stephen Y = jordan
X = ryan Y = jacob
The Basic Idea of Prolog

- AI programs often involve searching for the solution to a problem.

- Why not provide this search capability as the underlying idea of the language?

- Result: Prolog
**Prolog**

Mostly declarative.

Program looks like a declaration of facts plus rules for deducing things.

“Running” the program involves answering questions that refer to the facts or can be deduced from them.

More formally, you provide the axioms, and Prolog tries to prove theorems.
Prolog Execution

Facts

nerd(X) :- techer(X).
techer(stephen).

Query

?- nerd(stephen).

→ Search (Execution)

Result

yes
Simple Searching

Starts with the query:

?- nerd(stephen).

*Can we convince ourselves that *nerd*(stephen) is true given the facts we have?*

**teacher**(stephen).
nerd(X) :- teacher(X).

First says teacher(stephen) is true. Not helpful.

Second says that we can conclude nerd(X) is true if we can conclude teacher(X) is true. More promising.
Simple Searching

techer(stephen).
nerd(X) :- techer(X).
?- nerd(stephen).

*Unifying* nerd(stephen) with the head of the second rule, nerd(X), we conclude that X = stephen.

We’re not done: for the rule to be true, we must find that all its conditions are true. X = stephen, so we want techer(stephen) to hold.

This is exactly the first clause in the database; we’re satisfied. The query is simply true.
More Clever Searching

techer(stephen).
techer(todd).
nerd(X) :- techer(X).
?- nerd(X).

“Tell me about everybody who’s provably a nerd.”

As before, start with query. Rule only interesting thing.

Unifying nerd(X) with nerd(X) is vacuously true, so we need to establish techer(X).
More Clever Searching

techer(stephen).
techer(todd).
nerd(X) :- techer(X).
?- nerd(X).

Unifying techer(X) with techer(stephen) succeeds, setting $X = \text{stephen}$, but we’re not done yet.

Unifying techer(X) with techer(todd) also succeeds, setting $X = \text{todd}$, but we’re still not done.

Unifying techer(X) with nerd(X) :- fails, returning no.
More Clever Searching

> ~/tmp/beta-prolog/bp


| ?- [user].
| :techer(stephen).
| :techer(todd).
| :nerd(X) :- techer(X).
| :^D

yes
| ?- nerd(X).
X = stephen?;
X = todd?;
no
| ?-
Order Matters

> ~/tmp/beta-prolog/bp
| ?- [user].
| :techer(todd).
| :techer(stephen).
| :nerd(X) :- techer(X).
| :^D
yes
| ?- nerd(X).
X = todd?; Todd returned first
X = stephen?;
no
| ?-
Searching and Backtracking
The Prolog Environment

Database consists of clauses.

Each clause consists of terms, which may be constants, variables, or structures.

Constants: foo my_Const + 1.43

Variables: X Y Everybody My_var

Structures: rainy(rochester)
            teaches(edwards, cs4115)
Structures and Functors

A structure consists of a functor followed by an open parenthesis, a list of comma-separated terms, and a close parenthesis:

```
bin_tree( foo, bin_tree(bar, glarch) )
```

What’s a structure? Whatever you like.

A predicate `nerd(stephen)`
A relationship `teaches(edwards, cs4115)`
A data structure `bin(+, bin(-, 1, 3), 4)`
Unification

Part of the search procedure that matches patterns.

The search attempts to match a goal with a rule in the database by unifying them.

Recursive rules:

- A constant only unifies with itself
- Two structures unify if they have the same functor, the same number of arguments, and the corresponding arguments unify
- A variable unifies with anything but forces an equivalence
Unification Examples

The = operator checks whether two structures unify:

<table>
<thead>
<tr>
<th>?- a = a.</th>
<th>yes</th>
<th>% Constant unifies with itself</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no</td>
<td>% Mismatched constants</td>
</tr>
<tr>
<td>?- a = b.</td>
<td>no</td>
<td>% Mismatched constants</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>% Mismatched constants</td>
</tr>
<tr>
<td>?- 5.3 = a.</td>
<td>no</td>
<td>% Mismatched constants</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>% Variables unify</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>% Variables unify</td>
</tr>
<tr>
<td>?- foo(a,X) = foo(X,b).</td>
<td>no</td>
<td>% X=a required, but inconsistent</td>
</tr>
<tr>
<td>?- foo(a,X) = foo(X,a).</td>
<td>no</td>
<td>% X=a is consistent</td>
</tr>
<tr>
<td>X = a?;</td>
<td>no</td>
<td>% X=a is consistent</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>% X=a, then b=Y</td>
</tr>
<tr>
<td>?- foo(X,b) = foo(a,Y).</td>
<td>no</td>
<td>% X=a, then b=Y</td>
</tr>
<tr>
<td>X = a a</td>
<td>Y = b?;</td>
<td>% X=a, then b=Y</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>% X=b required, but inconsistent</td>
</tr>
<tr>
<td>?- foo(X,a,X) = foo(b,a,c).</td>
<td>no</td>
<td>% X=b required, but inconsistent</td>
</tr>
</tbody>
</table>
The Searching Algorithm

\begin{verbatim}
search(goal g, variables e)
    for each clause h : t_1, \ldots, t_n in the database
        e = unify(g, h, e)
        if successful,
            for each term t_1, \ldots, t_n,
                e = search(t_k, e)
                if all successful, return e
        return no
\end{verbatim}

This is very abstract; real algorithm includes backtracking
Order matters

search(goal \( g \), variables \( e \)) for each clause \( h : - t_1, \ldots, t_n \) in the database

\[ e = \text{unify}(g, h, e) \]

if successful,

for each term \( t_1, \ldots, t_n \),

\[ e = \text{search}(t_k, e) \]

if all successful, return \( e \)

return \text{no}
Order Affects Efficiency

edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :-
    edge(X, Z), path(Z, Y).

Consider the query

?- path(a, a).

Good programming practice: Put the easily-satisfied clauses first.
Consider the query

?- path(a, a).

Will eventually produce the right answer, but will spend much more time doing so.
Order can cause Infinite Recursion

edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    path(X, Z), edge(Z, Y).
path(X, X).

Consider the query

?- path(a, a).

Like LL(k) grammars.
Bill and Ted in Prolog

super_band(X) :- on_guitar(X, eddie_van_halen).
on_guitar(X, eddie_van_halen) :- triumphant_video(X).
triumphant_video(X) :- decent_instruments(X).
decent_instruments(X) :- know_how_to_play(X).
know_how_to_play(X) :- on_guitar(X, eddie_van_halen).

super_band(wyld_stallyns).

What will Bill and Ted do?
Prolog as an Imperative Language

A declarative statement such as

P if Q and R and S

can also be interpreted procedurally as

To solve P, solve Q, then R, then S.

This is the problem with the last path example.

path(X, Y) :- path(X, Z), edge(Z, Y).

“To solve P, solve P…”
Prolog as an Imperative Language

go :- print(hello_), print(world).

?- go.
hello_world
yes
Cuts

Ways to shape the behavior of the search:

• Modify clause and term order.
  Can affect efficiency, termination.

• “Cuts”
  Explicitly forbidding further backtracking.
Cuts

When the search reaches a cut (!), it does no more backtracking.

\texttt{techer(stephen)} :- !. 
\texttt{techer(todd)}.
\texttt{nerd(X)} :- \texttt{techer(X)}.

?- \texttt{nerd(X)}.
X= stephen?;
no
Controlling Search Order

Prolog’s ability to control search order is crude, yet often critical for both efficiency and termination.

- Clause order
- Term order
- Cuts

Often very difficult to force the search algorithm to do what you want.
Elegant Solution Often Less Efficient

Natural definition of sorting is inefficient:

\[
\text{sort}(L1, L2) :\text{-} \text{permute}(L1, L2), \text{sorted}(L2).
\]
\[
\text{permute}([], []).\]
\[
\text{permute}(L, [H|T]) :\text{-}
\]
\[
\text{append}(P, [H|S], L), \text{append}(P, S, W), \text{permute}(W, T).
\]

Instead, need to make algorithm more explicit:

\[
\text{qsort}([], []).\]
\[
\text{qsort}([A|L1, L2) :\text{-} \text{part}(A, L1, P1, S1),
\]
\[
\text{qsort}(P1, P2), \text{qsort}(S1, S2), \text{append}(P2, [A|S2], L2).
\]
\[
\text{part}(A, [], [], []).\]
\[
\text{part}(A, [H|T], [H|P], S) :\text{-} A \geq H, \text{part}(A, T, P S).
\]
\[
\text{part}(A, [H|T], P, [H|S]) :\text{-} A < H, \text{part}(A, T, P S).
\]
Prolog’s Failings

Interesting experiment, and probably perfectly-suited if your problem happens to require an AI-style search.

Problem is that if your peg is round, Prolog’s square hole is difficult to shape.

No known algorithm is sufficiently clever to do smart searches in all cases.

Devising clever search algorithms is hardly automated: people get PhDs for it.