Logic Programming: Prolog

COMS W4115

Prof. Stephen A. Edwards
Spring 2003
Columbia University
Department of Computer Science

More Logic

“All Caltech graduates are nerds.
Stephen is a Caltech graduate.
Is Stephen a nerd?”

Prolog

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

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

The Basic Idea of 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.

Simple Searching

Facts

\( \text{nerd}(X) :\!\!: \text{techer}(X) . \)
\( \text{techer}(\text{stephen}) . \)

Query

\( \neg \text{nerd}(\text{stephen}) . \)

Simple Searching

- Starts with the query:
- Can we convince ourselves that \( \text{nerd}(\text{stephen}) \) is true given the facts we have?

First says \( \text{techer}(\text{stephen}) \) is true. Not helpful.
Second says that we can conclude \( \text{nerd}(X) \) is true if we can conclude \( \text{techer}(X) \) is true. More promising.
More Clever Searching

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

Order Matters

> `/tmp/beta-prolog/bp`
| ?- [user].
| :teacher(todd).
| :teacher(stephen).
| :nerd(X) :- teacher(X).
| :

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

Searching and Backtracking

Unifying teacher(X) with teacher(X) succeeds, setting X = stephen, but we're not done yet.
Unifying teacher(X) with teacher(todd) also succeeds, setting X = todd, but we're still not done.
Unifying teacher(X) with nerd(X) : fails, returning no.

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:

| ?- a = a. | % Constant unifies with itself
| yes | a = b. |
| no | ?- 5.3 = a. | % Mismatched constants
| yes | ?- 5.3 = X. | % Variables unify
| X = 5.3; |
| no | ?- foo(a, X) = foo(X, b). | % X=a required, but inconsistent
| no | ?- foo(a, X) = foo(X, a). |
| yes | ?- foo(a, X) = foo(a, Y). | % X=a is consistent
| X = a; |
| no | ?- foo(X, b) = foo(a, Y). |
| X = a; |
| no | ?- foo(X, b) = foo(a, Y). |
| X = b; |
| no | ?- foo(b, a, c). |
| no | ?- foo(X, a, X) = foo(b, a, c). | X=b required, but inconsistent
The Searching Algorithm

```
search(goal g, variables e)
for each clause h :- t₁, ..., tₙ in the database
e = unify(g, h, e)
if successful,
  for each term t₁, ..., tₙ,
e = search(tᵢ, e)
if all successful, return e
return no
```

Order matters

```
search(goal g, variables e) In the order they appear
for each clause h :- t₁, ..., tₙ in the database
e = unify(g, h, e)
if successful, In the order they appear
  for each term t₁, ..., tₙ,
e = search(tᵢ, e)
if all successful, return e
return no
```

Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
path(X, X).
Consider the query
?- path(a, a).
```

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

Order Affect Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
path(X, X).
Consider the query
?- path(a, a).
```

Will eventually produce the right answer, but will spend much more time doing so.

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.
techer(stephen) :- !.
techer(todd).
nerd(X) :- techer(X).
?- nerd(X).
X= stephen?
  no
```

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 example.
path(X, Y) :- path(X, Z), edge(Z, Y).
"To solve P, solve P..."
```

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.
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:

```prolog
sort(L1, L2) :- permute(L1, L2), sorted(L2).
permute([], []).
permute([H|T], [H|S]) :- append(P, [H|S], L), append(P, S, W), permute(W, T).
```

Instead, need to make algorithm more explicit:

```prolog
qsort([], []).
qsort([A|L1, L2] :- part(A, L1, P1, S1),
      qsort(P1, P2), qsort(S1, S2), append(P2, [A|S2], L2).
part(A, [], [], []).
part(A, [H|T], [H|P], S) :- A >= H, 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.