Logic Programming: Prolog

COMS W4115

Logic

All Caltech graduates are nerds.

Stephen is a Caltech graduate.

Is Stephen a nerd?

Prolog

All Caltech graduates are nerds.  nerd(X) :- techer(X).

Stephen is a Caltech graduate.  techer(stephen).

Is Stephen a nerd?  ?- nerd(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 deducting 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).

nerd(stephen) :- 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?

techer(stephen).

nerd(X) :- techer(X).

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

Second says that we can conclude nerd(X) is true if we can conclude techer(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**

```prolog
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)`.

Unifying `teacher(X)` with `teacher(stephen)` 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) :- teacher(X).` fails, returning no.

---

**Order Matters**

```prolog
> ~/tmp/beta-prolog/bp
| ?- [user].
| :teacher(todd).
| :teacher(stephen).
| :nerd(X) :- teacher(X).
| :^D
yes
| ?- nerd(X). Todd returned first
X = todd?;
X = stephen?;
no
| ?-
```

---

**Searching and Backtracking**

```prolog
> ~/tmp/beta-prolog/bp
| ?- [user].
| :teacher(todd).
| :teacher(stephen).
| :nerd(X) :- teacher(X).
| :^D
yes
| ?- nerd(X).
X = stephen?;
X = todd?;
no
| ?-
```

---

**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:

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

\[ \text{search}(\text{goal } g, \text{variables } e) \]

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 \( \text{no} \)

This is very abstract; real algorithm includes backtracking

Order Affects Efficiency

Consider the query
\[ \text{?– path}(a, a). \]

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

Order can cause Infinite Recursion

Consider the query
\[ \text{?– path}(a, a). \]

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
\[ P \text{ if } Q \text{ and } R \text{ and } S \]
can also be interpreted procedurally as
To solve \( P \), solve \( Q \), then \( R \), then \( S \).

This is the problem with the last path example.
\[ \text{path}(X, Y) :- \text{path}(X, Z), \text{edge}(Z, Y). \]

“To solve \( P \), solve \( P \ldots \)“

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.

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

?- 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:

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

Instead, need to make algorithm more explicit:

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|F], 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.