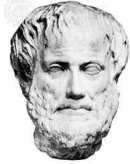


Logic Programming: Prolog

COMS W4115



Aristotle

Prof. Stephen A. Edwards

Fall 2007

Columbia University

Department of Computer Science

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).    ?- friend(stephen,jordan).
                                yes
                                ?- friend(stephen,X).
                                X = jordan
enemy(stephen, ryan).          ?- friend(X, Y).
enemy(ryan, jordan).           ?- friend(X, Y).
                                X = stephen Y = jordan
enemy(jordan, jacob).          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

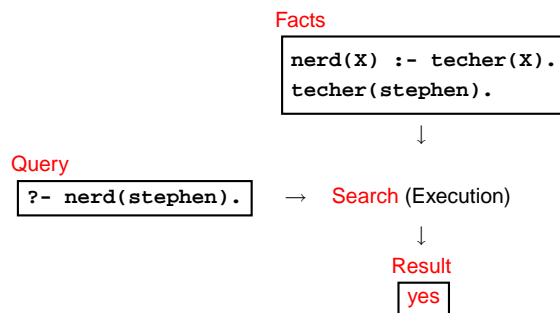
Mostly declarative.

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



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.

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(tk, e)
      if all successful, return e
  return no
  
```



FIXME: this is really broken: backtracking?

Order matters

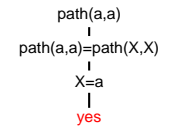
```

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



Consider the query

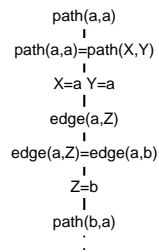
```
?- path(a, a).
```

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

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

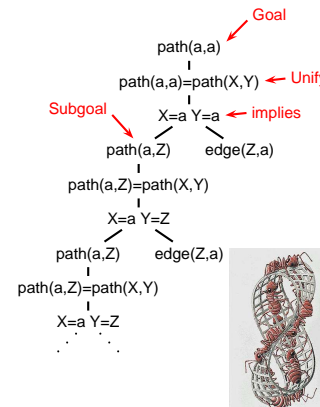
```
?- path(a, a).
```

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

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_stallions).
```

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.

```
techer(stephen) :- !.  
techer(todd).  
nerd(X) :- techer(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(L, [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|P], S) :- A >= H, part(A, T, P S).  
part(A, [H|T], P, [H|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.