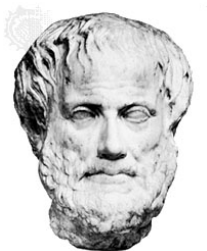# Logic Programming: The Prolog Language

Stephen A. Edwards

Columbia University

Fall 2013

All Caltech graduates are nerds.

Stephen is a Caltech graduate.

Is Stephen a nerd?

Logic

All Caltech graduates are nerds.

Stephen is a Caltech graduate.

Is Stephen a nerd?

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

techer(stephen).
```

```
?- nerd(stephen).
yes
```

More Logic

```
witch(X)  :- burns(X), female(X).
burns(X)  :- wooden(X).
wooden(X) :- floats(X).
floats(X) :- sameweight(duck, X).

female(girl).          /* By observation */
sameweight(duck,girl). /* By experiment */
```

```
? witch(girl).
yes
```

Still 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

```
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?

```
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

```
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).

Unifying techer(X) with techer(stephen) succeeds, setting X = stephen, but we're not done yet.

Unifying techer(X) with techer(todd) also succeeds, setting X = todd, but we're still not done.

Unifying techer(X) with nerd(X) fails, returning no.

More Clever Searching

```
$ prolog
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- [user].
compiling user for byte code...
techer(stephen).
techer(todd).
nerd(X) :- techer(X).
^D
user compiled, 4 lines read - 400 bytes written, 14260 ms

yes
| ?- nerd(X).

X = stephen ? ;

X = todd

yes

| ?-
```
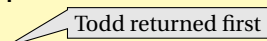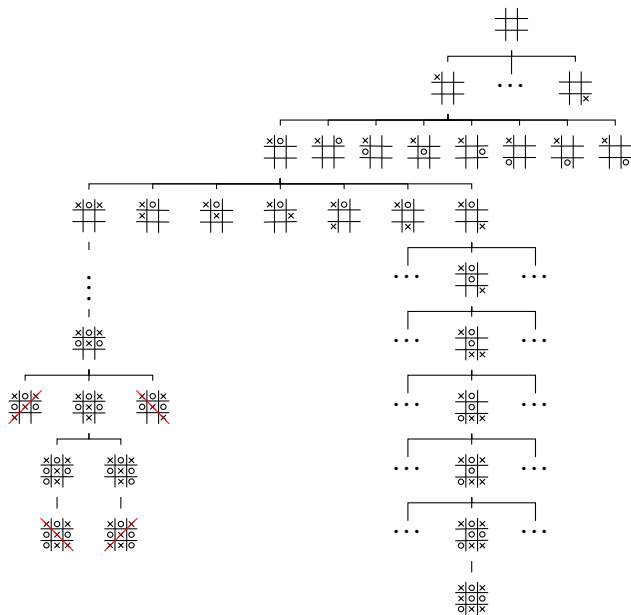
Order Matters

```
$ prolog
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- [user].
compiling user for byte code...
techer(todd).
techer(stephen).
nerd(X) :- techer(X).
^D
user compiled, 4 lines read - 399 bytes written, 14027 ms

yes
| ?- nerd(X).

X = todd ? ;          Todd returned first

X = stephen

yes

| ?-
```

Searching and Backtracking

The Prolog Environment

Database consists of Horn clauses. ("If a is true and b is true and ... and y is true then z is true".)

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:

"Functor"

paren must follow immediately

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

The Searching Algorithm

search(goal g, variables e)
    for each clause $h :- t_1, \ldots, t_n$ in the database   *in the order they appear*
        e = unify(g, h, e)
        if successful,   *in the order they appear*
            for each term $t_1, \ldots, t_n$,
                e = search(t_k, e)
        if all successful, return e
    return no

Note: This pseudo-code ignores one very important part of the searching process!

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).
```

```
        path(a,a)
            |
    path(a,a)=path(X,X)
            |
          X=a
            |
          yes
```

Consider the query

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

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

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).
```

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

path(a,a)
|
path(a,a)=path(X,Y)
|
X=a Y=a
|
edge(a,Z)
|
edge(a,Z) = edge(a,b)
|
Z=b
|
path(b,a)
⋮

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).
```
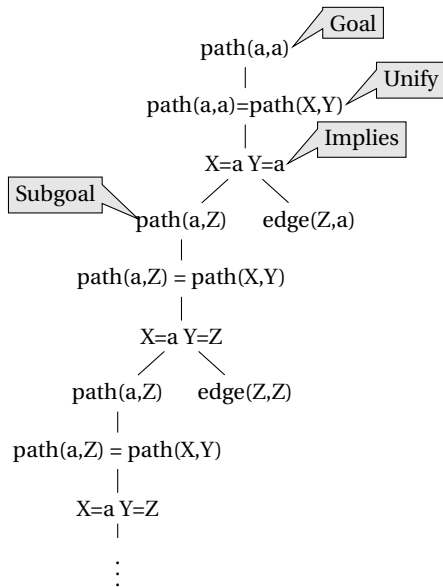
Consider the query

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


path(a,a) — Goal
|
path(a,a)=path(X,Y) — Unify
|
X=a Y=a — Implies
|
path(a,Z) — Subgoal   edge(Z,a)
|
path(a,Z) = path(X,Y)
|
X=a Y=Z
|
path(a,Z)   edge(Z,Z)
|
path(a,Z) = path(X,Y)
|
X=a Y=Z
|
⋮

Bill and Ted in Prolog


```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).
```

```prolog
| ?- 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..."

```
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.

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

```
techer(stephen) :- !.
techer(todd).
nerd(X) :- techer(X).
```

```
| ?- nerd(X).

X = stephen

yes
```

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.