Prolog Mini-Examples

(to consolidate understanding)

1. Propagation of bindings

---

query ?- p(5), g(U).
clause p(X) :- q(X, Y), r(Y).

matching requires X/5 which has no impact upon the query variable U

derived query ?- q(5, Y), r(Y), g(U).

---

query ?- p(U), g(U).
clause p(X) :- q(X, Y), r(Y).

matching requires X/U which has no impact upon the query variable U

derived query ?- q(U, Y), r(Y), g(U).

Note that if we instead bound U/X we would get

derived query ?- q(X, Y), r(Y), g(X).

which is exactly the same, other than using different names for its variables

---

query ?- p(U), g(U).
clause p([3]).

matching requires U/[3] and this is
(a) propagated to all occurrences of U in the query, and
(b) recorded in the binding environment because U is a query variable

derived query ?- g([3]).

---

query ?- p([U, 2 | T], U), g(U).
clause p([1 | X], Z) :- q(X, Y, Z).

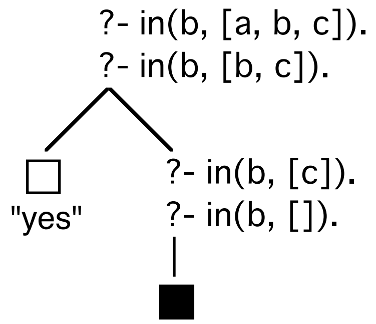
matching requires U/1, X/[2 | T], Z/1 — the binding U/1 is
(a) propagated to all occurrences of U in the query, and
(b) recorded in the binding environment because U is a query variable

derived query ?- q([2 | T], Y, 1), g(1).

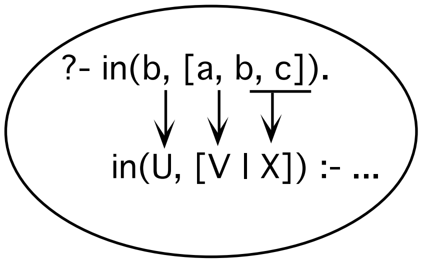
2. Sensitivity of Search Tree to Initial Query

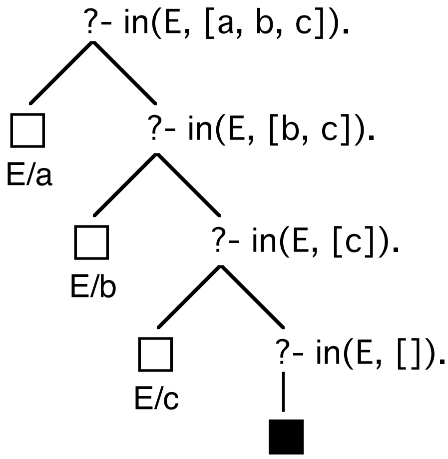
program

```
in(U, [U | X]).
in(U, [V | X]) :- in(U, X).
```

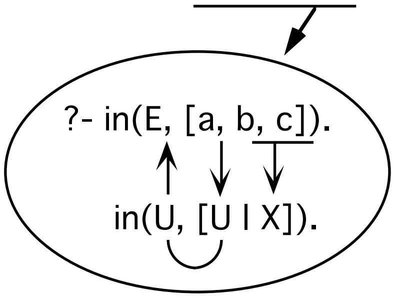


Here, the dataflow in each step is
from the selected call
into the selected clause



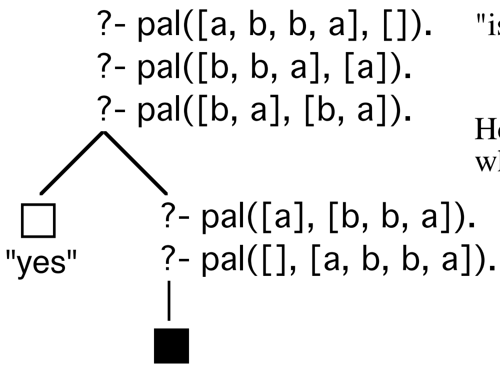


Here, the dataflow in each use of the first clause is bi-directional



program

```
pal(X, X).
pal([U | X], X).
pal([U | X], Z) :- pal(X, [U | Z]).
```

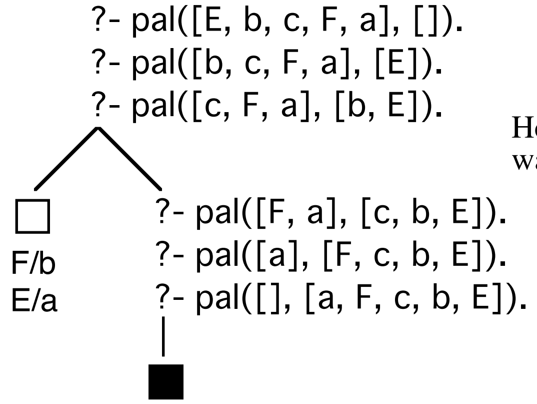


"is [a, b, b, a] a palindrome ?"

Here, one list is deconstructed while another is constructed

program
(again)

```
pal(X, X).
pal([U | X], X).
pal([U | X], Z) :- pal(X, [U | Z]).
```



Here, both lists have slots waiting to be determined

> **Remember that in all cases the evaluation is driven solely by the possibilities for matching**

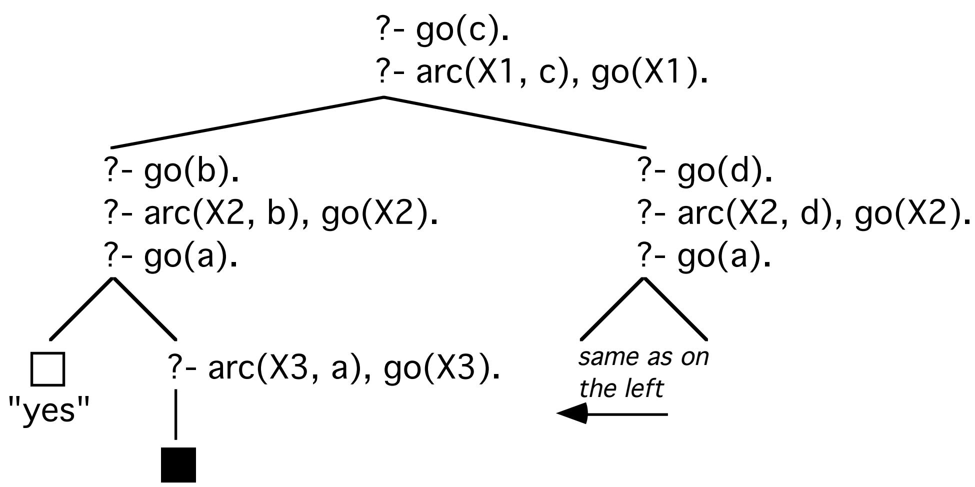
3. Varying the knowledge representation

program

```
arc(a, b).
arc(b, c).
arc(d, c).
arc(a, d).
go(Y) :- arc(X, Y), go(X).
```

The arcs represent a directed graph ...

To seek a path from a to c,
add go(a) to the program, and
pose the query ?- go(c).

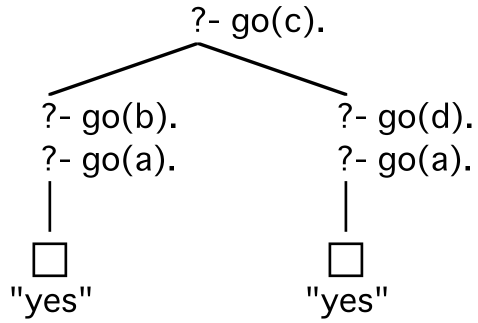


program

```
go(b) :- go(a).
go(c) :- go(b).
go(c) :- go(d).
go(d) :- go(a).
```

The same directed graph represented differently ...

Again, to seek a path from a to c,
add go(a) to the program, and
pose the query ?- go(c).



This seems simpler than the previous representation. However, it is also weaker, because without the arc predicate we cannot ask for those paths guaranteed to be of unit length.

Neither of the two representations just shown can report a path as output, and neither of them can be used when the start of a path is unknown.

The more usual way to represent path-finding is as follows:

program

```
arc(a, b).
arc(b, c).
arc(d, c).
arc(a, d).

path(X, Z) :- arc(X, Z).
path(X, Z) :- arc(X, Y), path(Y, Z).
```

The same representation as before for the graph, but a new representation for the notion of a path ...

To seek a path from a to c,
pose the query ?- path(a, c).

Again, this cannot report a path as output, but it can cope whether or not the start and end of the path are known. It works best when at least the start is known.

Yet another possibility is

```
path(X, Z) :- arc(X, Z).
path(X, Z) :- arc(Y, Z), path(X, Y).
```

This works best when at least the end is known.

And another possibility is

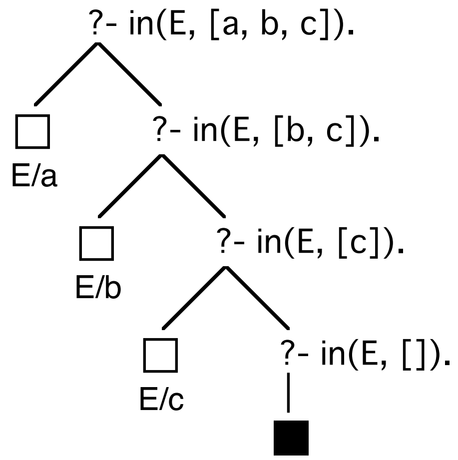
```
path(X, Z) :- arc(X, Z).
path(X, Z) :- path(X, Y), path(Y, Z).
```

This one tends to be inefficient and loop-prone.

All of them may get into difficulty if the given graph is cyclic.

4. Sensitivity of Search to Clause Order

program | in(U, [U | X]).
(again) | in(U, [V | X]) :- in(U, X).



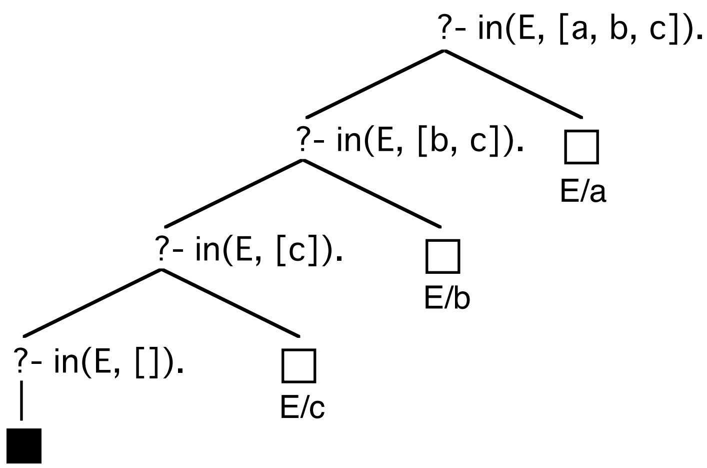
By convention, the branches are drawn from left-to-right in the order in which they are actually generated at run-time, which corresponds in turn to the text-order of the clauses in the program.

Here, we first get E/a, then E/b, then E/c and finally a finite failure.

If we reverse the clause order to get

program | in(U, [V | X]) :- in(U, X).
in(U, [U | X]).

then the tree is *structurally* unchanged but must be *drawn* differently:

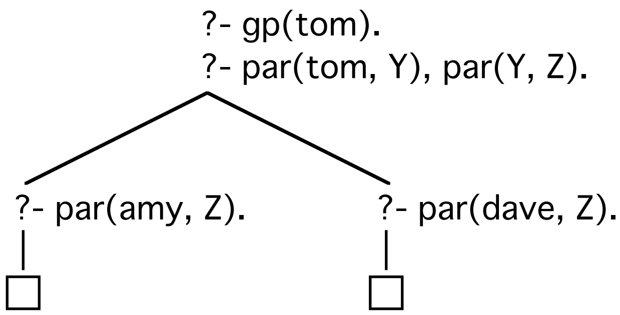


Now, we first get the finite failure, then E/c, then E/b and finally E/a.

5. Sensitivity of Efficiency to Call Order

program

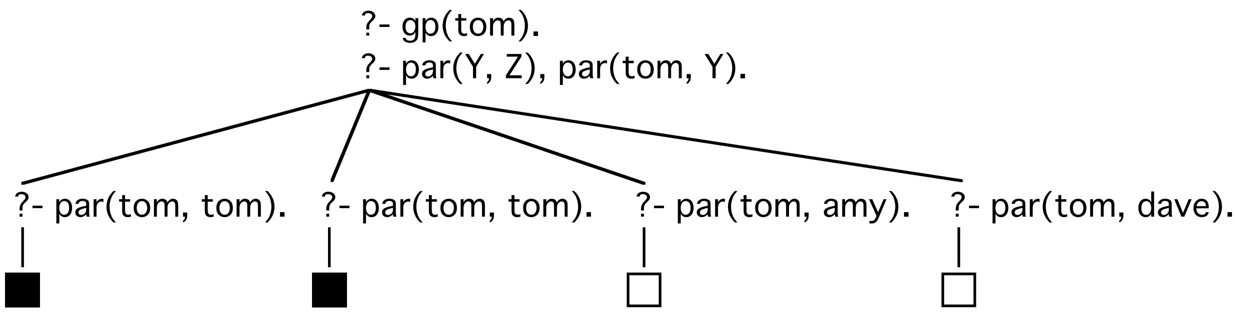
```
gp(X) :- par(X, Y), par(Y, Z).            gp denotes grandparent ...

par(tom, amy).
par(tom, dave).
par(amy, jill).
par(dave, kay).
```



If we reverse the call-order in the gp clause to get

gp(X) :- par(Y, Z), par(X, Y)

then the evaluation is much more non-deterministic:



In general, try to order the calls so that those expected to be most deterministic will be selected soonest.

So, knowing that X is going to be bound to tom by our initial query, it is better to do par(X, Y) before (Y, Z) — a par call with only one unbound variable (Y) is going to behave more deterministically than a par call with two (Y and Z), because *it is likely to match fewer program clause-heads*.