

Prolog (warts and wrinkles)

The Closed World Assumption

What does it mean when a Prolog program comes back with "no" (failure)?

- Means that the system could not deduce the truth of the query proposition from the facts and rules it was given
- Search process failed to find a set of instantiations that will make the prop. true

Notice that this is somewhat different from "the proposition is false"

- There may be facts or rules the system does not know about that could be used to show the proposition to be true
- Failure is the same as proving the proposition false only if this can't happen

=> "The closed world assumption"

Prolog includes a "not" but its problematical

Can say not(P) for any term P

- in the body only, not the head
- can't declare a negative fact

However, this is not standard logical negation

"not" means the system failed to find/prove P
the term not(P) succeeds iff P fails
=> failure as negation

Fundamental reason why we don't have a "real not" is a limitation in what can be expressed with Horn clauses

$A :- B_1, B_2, \dots, B_n$

means:

$B_1 \text{ and } B_2 \text{ and } \dots \text{ and } B_n \Rightarrow A$

The implication goes only one way:

if all the B's are true then A is true

however, if some B's are false we know
nothing about A

If the closed world assumption holds true, then we are ok. However, if not then "weird things" can happen...

```
mother(mary, tyler).
father(scott, tyler).
non_parent(X, Y) :- not(father(X, Y)),
                    not(mother(X, Y)).
```

Works ok sometimes ...

```
?- non_parent(joe, tyler).
yes
?- non_parent(scott, tyler).
no
```

but...

```
?- non_parent(lloyd, scott).
yes          - wrong in the real world
```

We can probably live with this sort of simple case if we keep the meaning of not in mind, but...

.... It can get worse.

```
?- non_parent(X,Y).  
no
```

Ooops...

"negated" term/clause never returns
instantiations of its variables!
<Why?>

Gets even worse/compounded:

```
mother(mary, tyler).  
father(scott, tyler).  
blue_eyed(tyler).  
non_parent(X, Y) :- not(father(X,Y)),  
                    not(mother(X,Y)).
```

```
?- blue_eyed(X), non_parent(X,Y).  
X = tyler  
Y = <non-instantiated>  
yes  
?- non_parent(X,Y), blue_eyed(X).  
no
```

Reordering terms changes success/failure!
<What happened?>

Other cases where order matters:

```
ancestor(X, X).  
ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).
```

Is always infinite recursive, whereas

```
ancestor(X, X).  
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Is just fine (obvious from procedural view, but not from declarative semantics).

=> typically want base clauses first, and not recursive terms first

Order can also have a huge impact on efficiency (infinite recursion is ok, its just inefficient :-)

Top to bottom, left to right search

want most likely clauses first

want most restrictive terms first

want recursive terms last

- tail recursion

- can be transformed into iteration

Can also restrict backtracking for efficiency

Special construct: "cut"

Expressed as a special term: "!"

Always succeeds

Has side effect of stopping backtracking

Specifically:

system won't backtrack through a cut

Assume a goal: $a, b, ! c, d$

Once the system finds succeeding rules for a and b (i.e., once it gets to the cut the first time) it won't backtrack and look for others should c or d fail (but it will consider different alternatives for c, d)

- can think of cut as a diode, control can pass through it one way (the call direction), but not the other (backtrack direction)

This can change the results in fairly arbitrary ways

- "red" cuts
- "cut is considered the goto of Prolog"

Or with care, just avoid wasted computations

- "green" cuts

Using cut

Cut can indicate that we have found the correct answer (when we know there is exactly one) and we should never back out of it and try something else

Consider mutually exclusive rules:

A :- B, C.

A :- not(B), D.

End up testing B twice

- if B is expensive, this could double run time

Rewrite as:

A :- B, !, C.

A :- D.

More subtle example:

Suppose we have sets stored in lists (no repeats)

```
member(Element, [Element | _]).  
member(Element, [_ | Rest]) :-  
    member(Element, Rest).
```

Works fine, but no reason to ever backtrack into second rule if the first one succeeds

Better:

```
member(Element, [Element | _]) :- ! .
```

More egregious example:

```
sum_to(1,1) :- !.  
sum_to(N, Res) :- N1 is N-1,  
    sum_to(N1, TRes), Res is TRes + N.
```

Consider: "sum_to(1,X), *something_that_fails*."
Without the cut we have infinite recursion

Cut can also be used with "fail" to eliminate a class of "exceptions" from further consideration

```
average_taxpayer(X) :- foreigner(X), fail
average_taxpayer(X) :- ...
```

Doesn't work...

If we did that to eliminate foreigners from further consideration, backtracking would still consider the later rules anyway and we wouldn't be eliminating foreigners from the relation

Instead use cut/fail:

```
average_taxpayer(X) :- foreigner(X), !, fail
average_taxpayer(X) :- ...
```

This prevents testing any further rules once we know X is a foreigner.

(This can always be done with not)

Using cut for generate & test

"Generate and test" is a common programming idiom in Prolog

Early subgoal(s) generate a potential solution
Later subgoal(s) test that its valid

Rejected tests cause backtracking into generator to produce a new potential solution

=> enumerate the search space

If we know there is a unique solution, we want to stop when we find it and never backtrack to try more.

Can place a cut at the end of the test portion.

Assume `is_integer(X)` succeeds as long as its parameter can be instantiated to a non-negative integer -- backtracks to increasing values

```
divide(N1, N2, Result) :-  
    is_integer(Result),  
    P1 is Result * N2,  
    P2 is (Result + 1) * N2,  
    P1 =< N1, P2 > N1,  
    !.
```

`is_integer` is the generator
(generates all non-negative integers)

Test is the test

Cut stops once we find the one and only answer

?- `divide(27, 6, X)`

```
is_integer(1), P1 is 1*6, P2 is 2*6, P1 =<27,  
P2 > 27 *** fails ****
```

... backtrack and try $X = 2$, etc...

```
is_integer(4), P1 is 4*6, P2 is 5*6, P1 =< 27,  
P2 > 27, ! *** success ***
```

One last item the "occurs check"

What if we unify X with a term containing X

$$X = p(X)$$

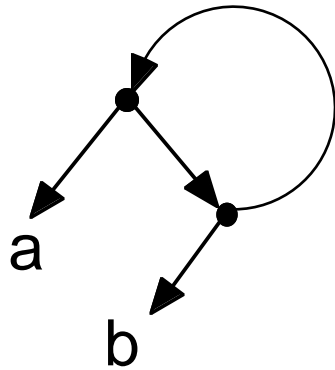
This can cause problems...

`append([], L, L).`

`append([H | T], L, [H | Res]) :- append(T,L,Res).`

`?- append([], E, [a, b | E]).`

End up unifying E with $[a, b | E]$ and get a self referent list



Works fine, until you try to print it...

For efficiency, Prolog does not do "occurs check"

- unification is "the main loop" efficiency-wise
- can force the check

- Cuts don't really relate to logic and understanding their effect requires understanding the underlying control mechanisms
- We have also seen that order of clauses and terms can effect termination (which is not an issue in logic)
- In practice one spends a good bit of time rearranging things to make the computation practical / efficient

So the goal of truly declarative semantics is not really realized

- can't just write down logical expression and then just execute it

However, Prolog comes a lot closer than lots of other languages to this ideal

- mostly declarative in nature
can think (at least initially) in declarative terms
- very well suited for some classes of problems
 - where logic and "relations" are a good way to think about the problem
 - where theorem proving / search is good