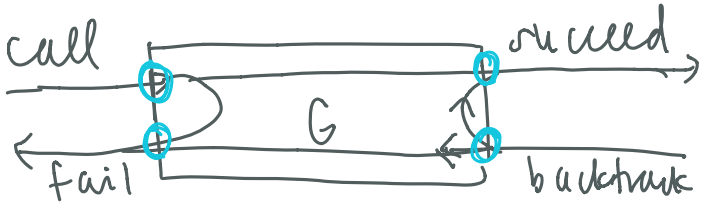
Prolog debugging

?- ask A

A = 5 ?

A = 2+7 ?

Prolog 4-port debugging model



- accessed through ?- trace
- use only at the beginning to understand how Prolog works

Terminology

clause:

a) fact

b) rule

c) query

To run, Prolog:

- starts with query
- backwards chaining

from query

↳ starts with what you're trying to prove and goes backward from it

↳ from rules to facts

↳ goal oriented

- left-to-right

AND. ↳ tries leftmost goals first within a query/clause

OR ↳ for heads of clauses

Problems with Prolog

①

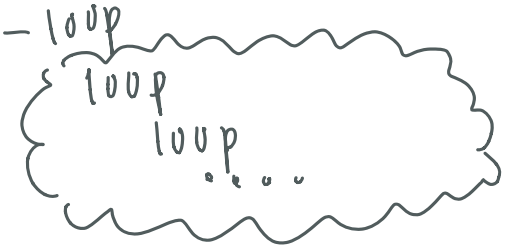
```
?- true
yes
?- fail.
no
?- precedince(A,B)
(message) // extra check
no.
```

source code for true

source code for fail (doesn't have source code)

②

```
?- loop
```




```
?- repeat
yes.
```

```
loop :- loop
```

```
repeat.
repeat :- repeat.
```

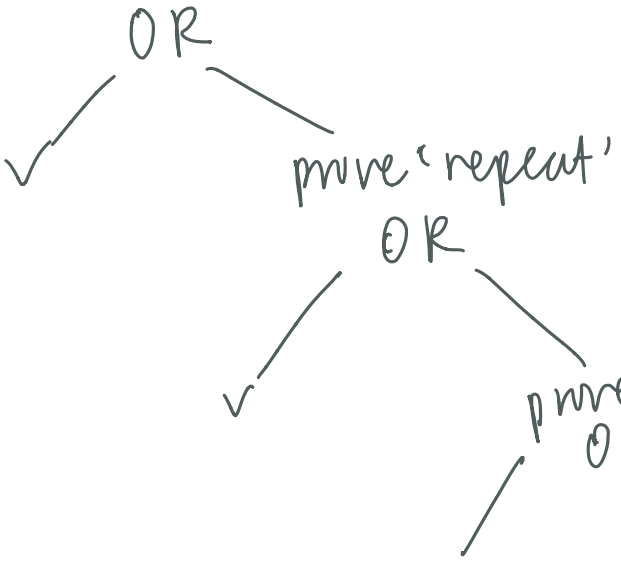
irresistable force

immovable object

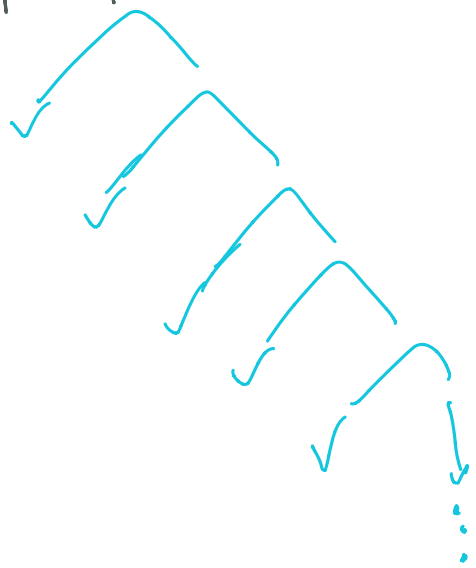
```
?- repeat, read(x), write(x), nl, fail.
```

infinite loop, continuously copies data from terminal to display.

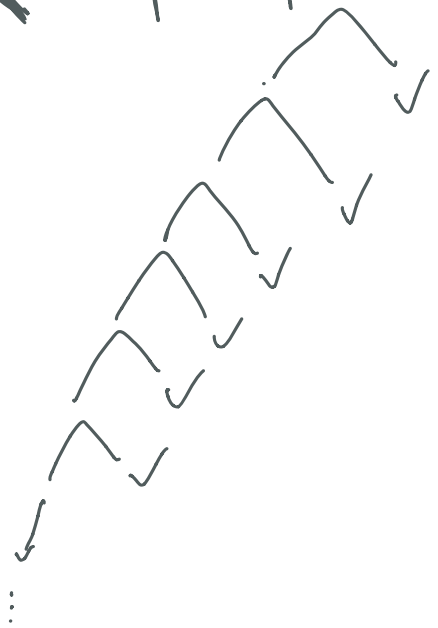
How to prove repeat



repeat proof tree



✗ crepeat proof tree



```
crepeat :- crepeat
crepeat.
```

Moral of the story: order matters

Prolog unification

= Two-way pattern matching

```
?- p(X, f(X))
```

data structure

$f(X) \neq g(B)$

```
p(A, g(B)) :- q(B, A).
```

p(A, A(C)) :- q(g(C, A))

p(h(I), J) :- q(J, I)

X = h(I), f(y) = J

Unification

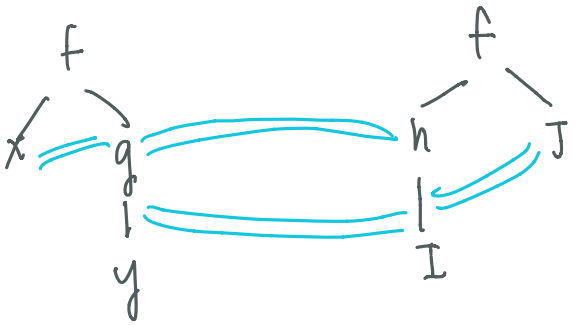
- make 2 terms identical by operationalizing them/ by substituting values for variables uniformly

f(x, g(x)) f(h(J), I)

↓ {x : h(J), I : g(y)}

↳ maps logical variables to terms (partial function)

f(h(J), g(y)) f(h(J), g(y))



p(x, x) ?- p(f(g(A), h(J)), f(x, h(g(z))))

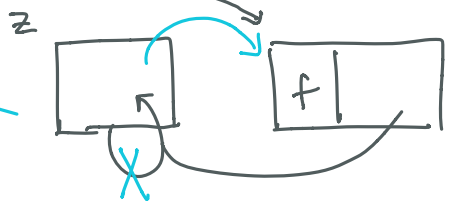
works! unification

p(x, x) ?- p(f(g(A), h(J)), f(x, i(g(z))))

wouldn't work

p(X, X)

?- p(Z, f(Z)).



- will give unbound variable a pointer to bound value
- f(f(f(... goes on forever
  ↳ infinite term, a common problem in Prolog

```
?- p(L, [_|L]),
?- L = [_|L],
'='(X, X).
X = X.
```

↗ infinite loop

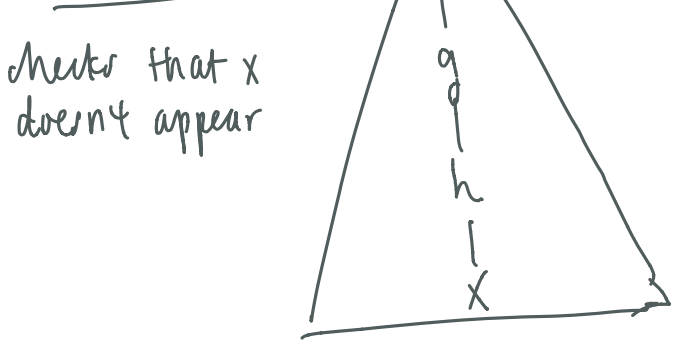
addition in Prolog from last week error:

```
? - add(A, A).
A = succ(succ(succ(...)...   infinity
```

★ don't write solution that leads to infinite terms

unify_with_occurs_check(X, f(g(h, X)))
A B

A = B, except it refuses to create infinite terms



unify_with_occurs_check(A, B) = O(max(|A|, |B|))
A = B = O(min(|A|, |B|))

- solution: never unify (X, X), always use unique variables
- very expensive to implement checking

Transitive closure of a relation

```
pr(cs32, cs131)
pr(c33, cs131)
pr(cs35L, cs131)
      ⋮
X 30,000
```

```
ipr(A, C) :- pr(A, B), pr(B, C).
ipr(A, Z) :- pr(A, B), ipr(B, Z).
```

Even more efficiently:

```
ipr(A, Z) :- pr(A, B), (pr(B, Z); ipr(B, Z)).
```

Equivalent without semi-colon:

```
ipr(A, Z) :- pr(A, B), prh(B, Z).
```

```
prh(A,B) :- pr(A,B).
prh(A,B) :- ipr(A,B).
```

```
?- ipr(cs31, Z)
Z = cs131;
Z = cs111;
  :
```

```
?- ipr(A, cs231).
   pr(A,B)      ,   prh(B, cs231)
```

runs really slowly
should not issue this goal

can you prove this is false

```
?-  ⊖ pr(dance 101, cs131).-
yes.
```

- no way in Prolog to say "not"
- can only tell Prolog true things, not false

CWA = closed world assumption
- "everything that I don't tell you is true must be false"
- can't make this assumption in Prolog

How to make prolog program go faster

```
p(X,Y,Z,W) :- gen(X,Z,W,Q), member(X,W),
              test(Y,Z,Q,X)
```

member(X,W): filter

test(Y,Z,Q,X): expensive

always ground terms

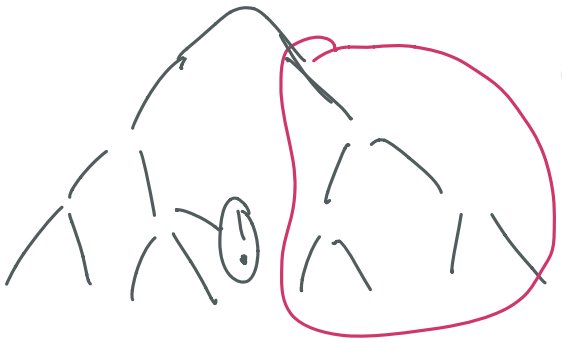
```
W = [a,b,c,d,e]
X = b
```

- still not as fast as possible
- member only needs to be run once

```
member(X, [X|_])
member(X, [_|L]):- member(X, L).
```

```
memberchk(X, [X|_]):- !.
memberchk(X, [_|L]):- memberchk(X, L).
```

cut - if we backtrack to cut, we immediately skip the predicate



- now memberchk only runs once, if we backtrack to it, we skip the calling predicate
- go to caller's caller predicate

alternative implementation to memberchk

```
once(P):- P, !.
memberchk(X, L):- once(member(X, L)).
```

meta-predicate: fails when P succeeds, succeeds when P fails

```
not(P) :- P, !, fail
not(_).
```

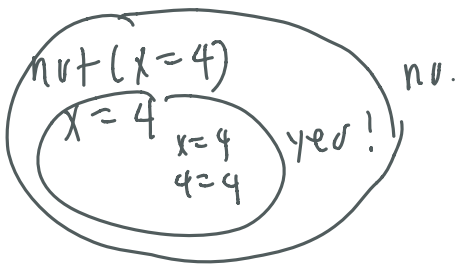
```
?- not(3 is 4 + 7)
yes.
```
fail

```
?- not(pr(dance101, cs131)).
yes.
```

* this implementation of not doesn't always work

```
?- X=3, not(x=4)
yes.
?- not(X=4), X=3
```
no

no.



- not ≠ not
- \+ P :- P, !, fail is built into prolog

Mathematical philosophy: $\vdash \equiv \models$

$\models P$ ≡ "P is true"

$\vdash P$ ≡ "P is provable"

$X \equiv Y$ ≡ "X is the same as Y"

$\not\vdash P$ ≡ "P is not provable"

$\star$ \+ (_) $\nvdash$ (P) \+P