Logic Programming

Lecture 5:
Nonlogical features, continued:
negation-as-failure, collecting solutions, assert/retract

Outline for today

- Nonlogical features continued
  - Negation-as-failure
  - Collecting solutions (findall, setof, bagof)
  - Assert/retract

Negation-as-failure

- We can use cut to define *negation-as-failure*
  - recall Tutorial #1

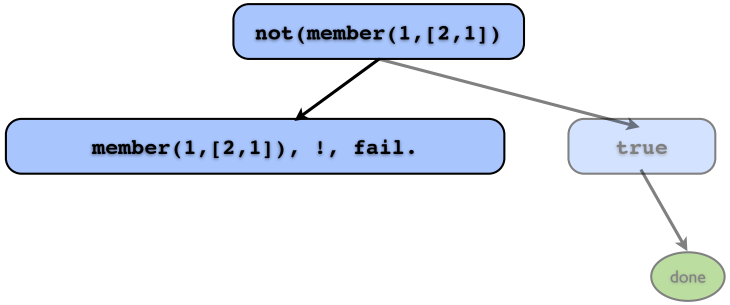
```
not(G) :- G, !, fail ; true.
```

- This tries to solve G
  - If successful, fail
  - Otherwise, succeed

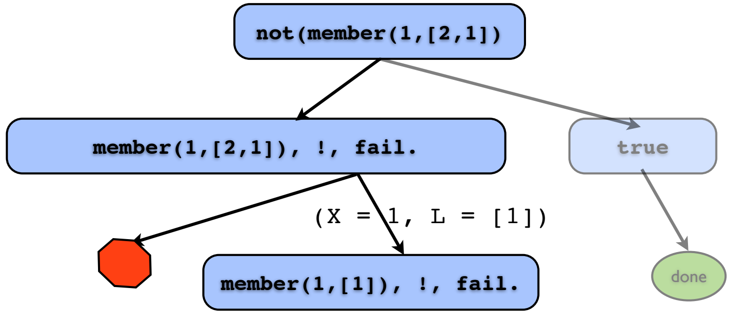
How it works

```
not(member(1,[2,1]))
```

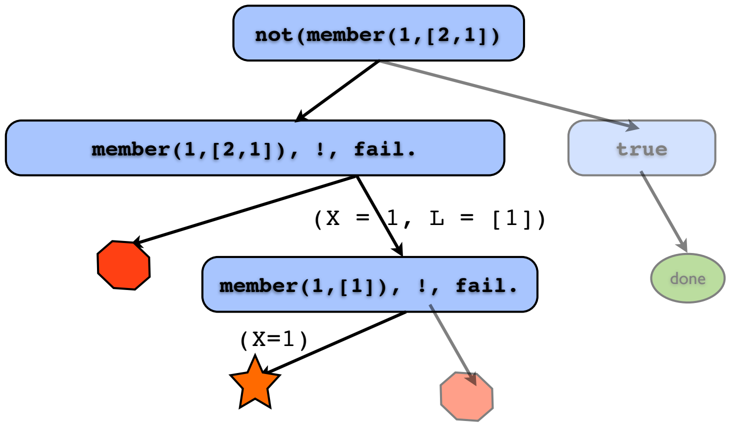
How it works



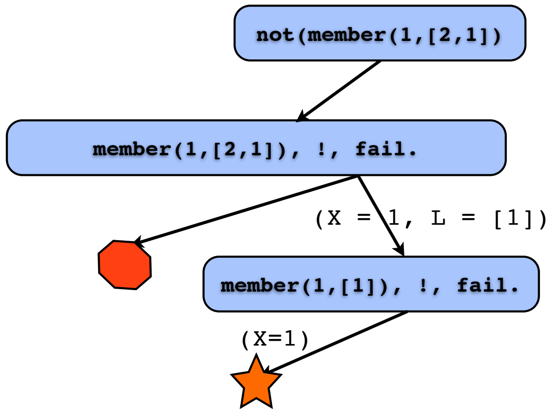
How it works



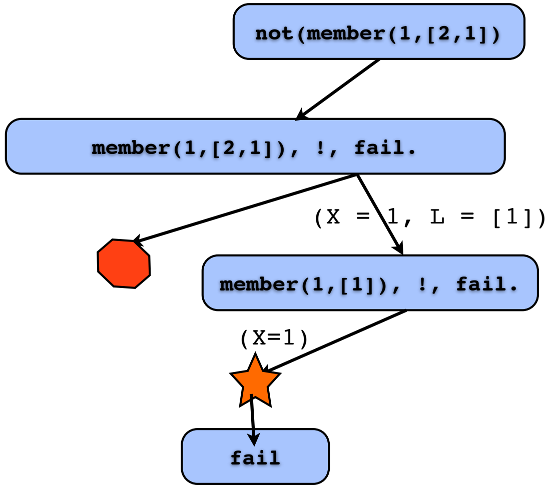
How it works



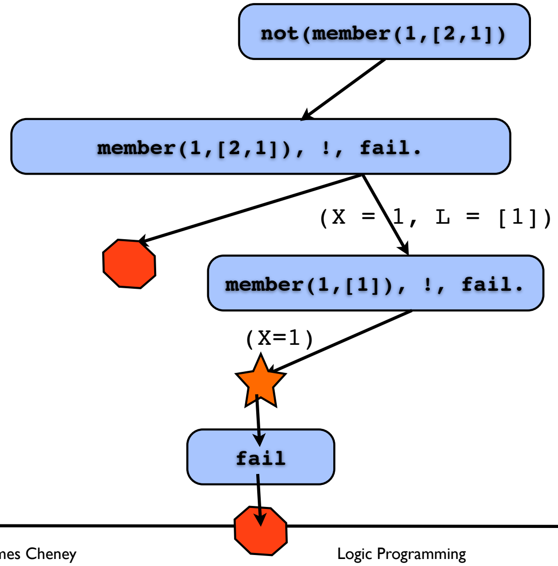
How it works



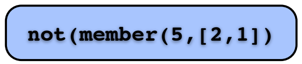
How it works



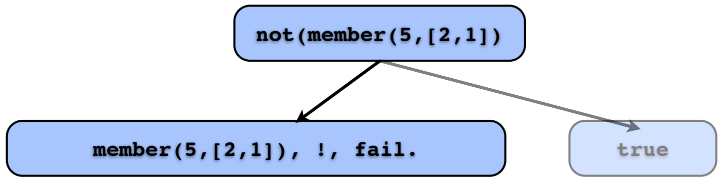
How it works



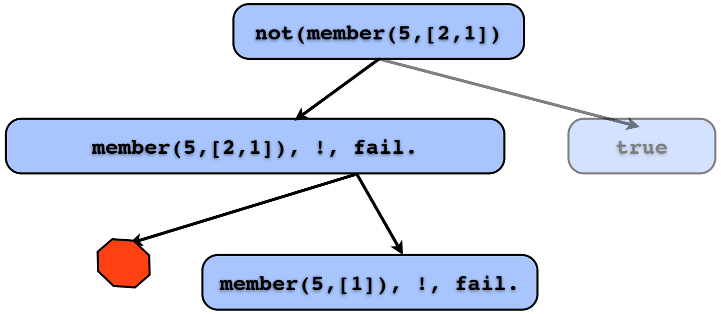
How it works



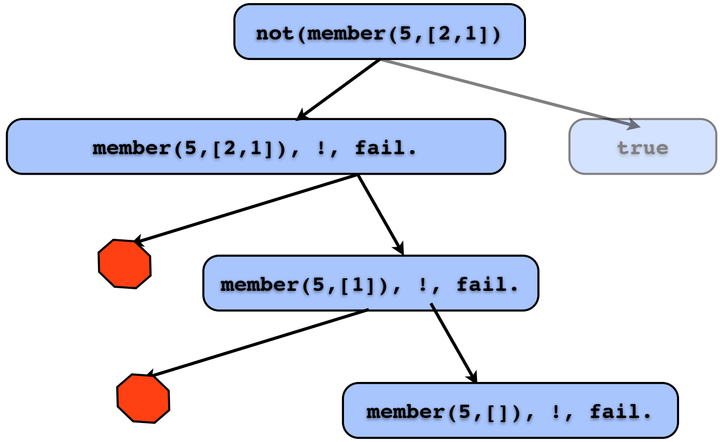
How it works



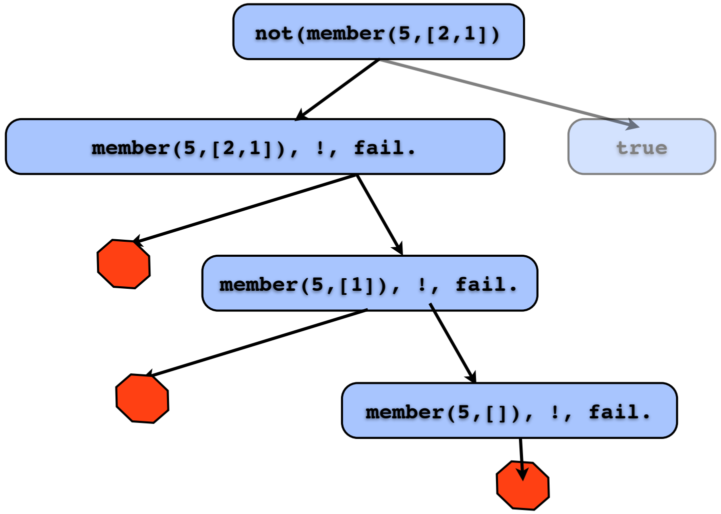
How it works



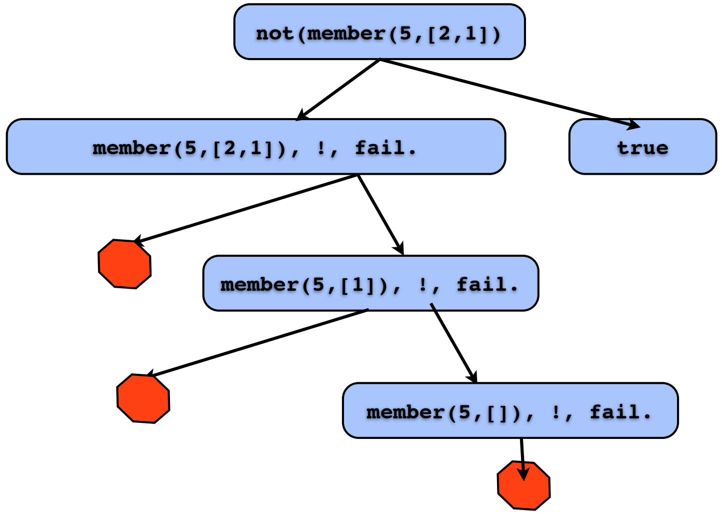
How it works



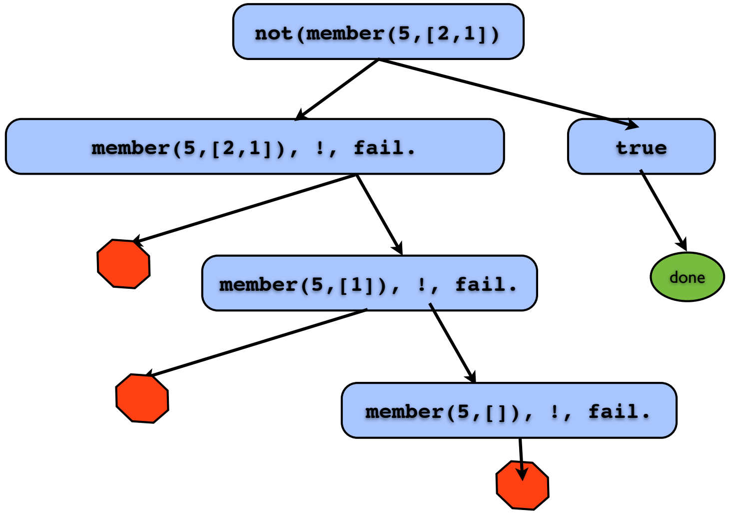
How it works



How it works



How it works

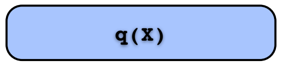


Negation-as-failure

- Built-in syntax: `\+(G)`
- Example: people that are not teachers

```
q(X) :- person(X),
        \+(teaches(X,Y)).
```

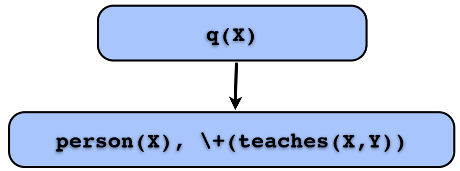

Behavior




```
  person(a).
  person(b).
teaches(a,b).
```

```
q(X) :- person(X),

        \+(teaches(X,Y)).
```

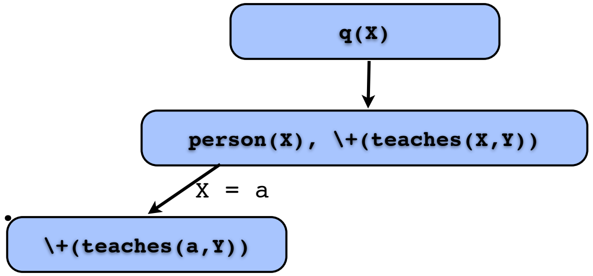

Behavior




```
  person(a).
  person(b).
teaches(a,b).
```

```
q(X) :- person(X),

        \+(teaches(X,Y)).
```

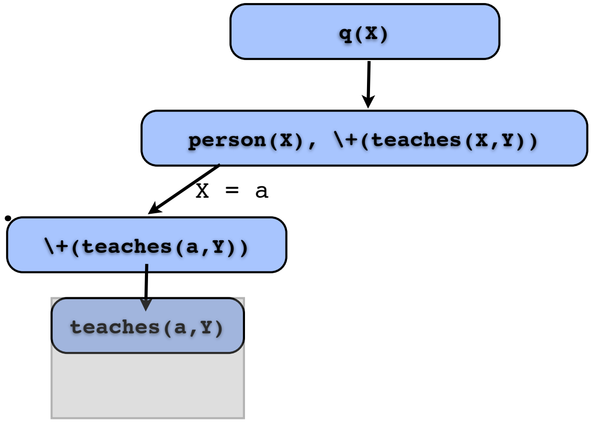
Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- person(X),
        \+(teaches(X,Y)).
```

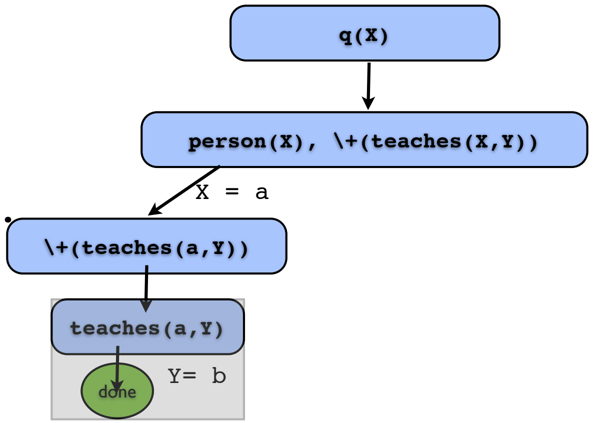

Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- person(X),
        \+(teaches(X,Y)).
```

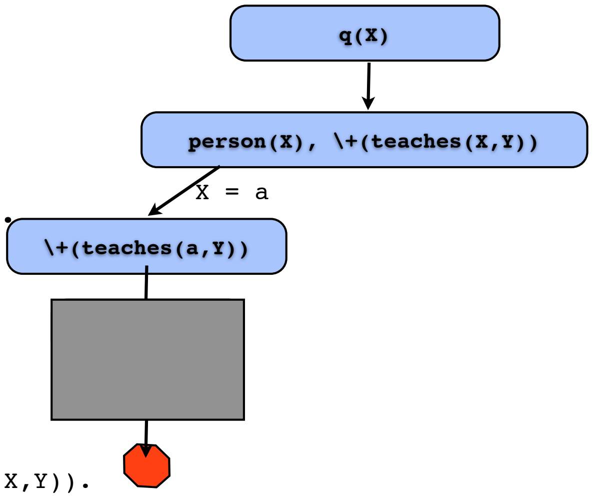

Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- person(X),
        \+(teaches(X,Y)).
```


Behavior

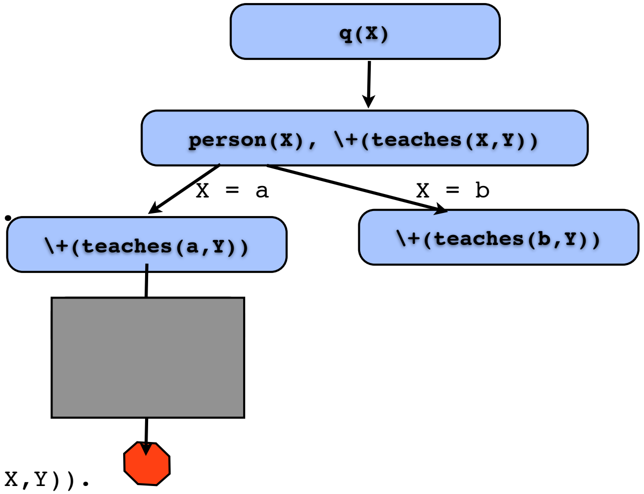



```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- person(X),
        \+(teaches(X,Y)).
```

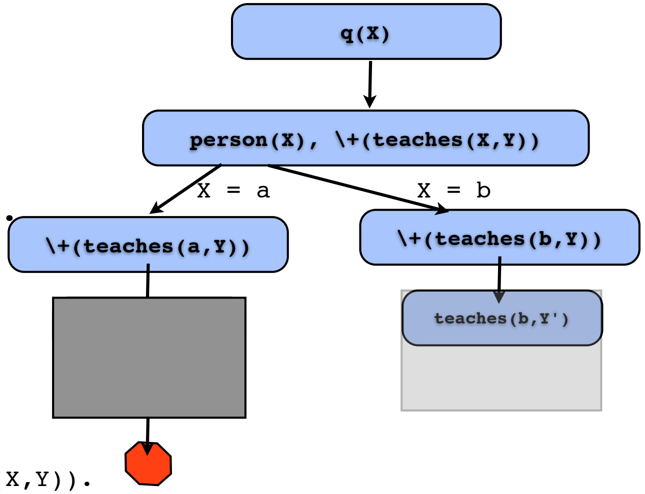
Behavior

```
person(a).
person(b).
teaches(a,b).

q(X) :- person(X),
        \+(teaches(X,Y)).
```



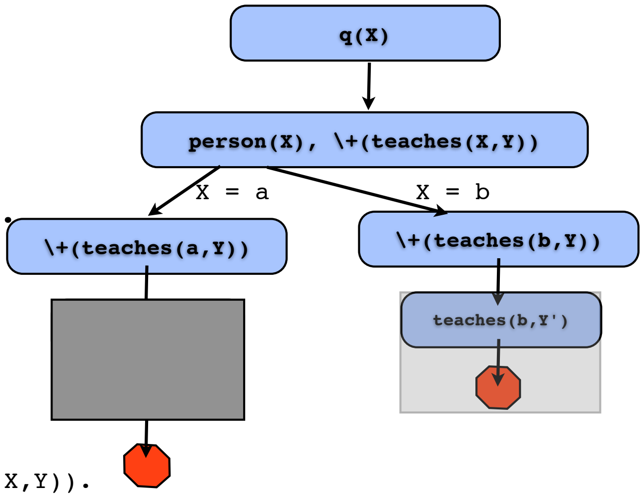
Behavior

```
person(a).
person(b).
teaches(a,b).

q(X) :- person(X),
        \+(teaches(X,Y)).
```



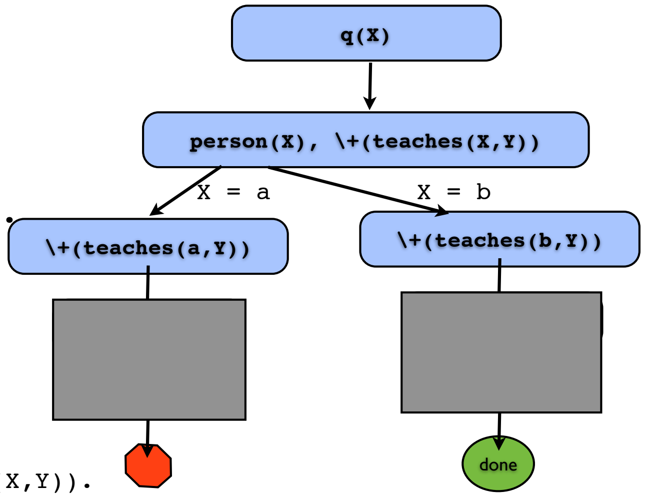
Behavior

```
person(a).
person(b).
teaches(a,b).

q(X) :- person(X),
        \+(teaches(X,Y)).
```

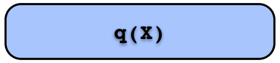


Behavior

```
person(a).
person(b).
teaches(a,b).

q(X) :- person(X),
        \+(teaches(X,Y)).
```



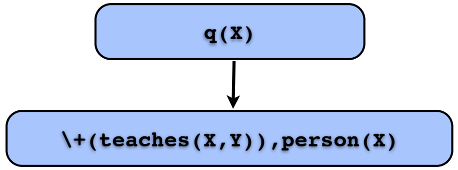
Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- \+(teaches(X,Y)),
        person(X).
```

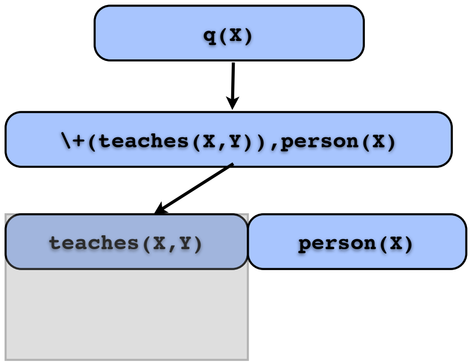

Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- \+(teaches(X,Y)),
        person(X).
```

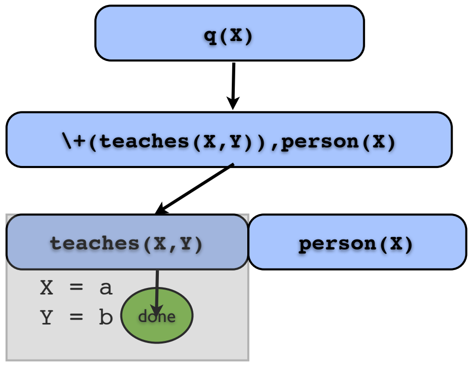

Behavior




```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- \+(teaches(X,Y)),
        person(X).
```


Behavior

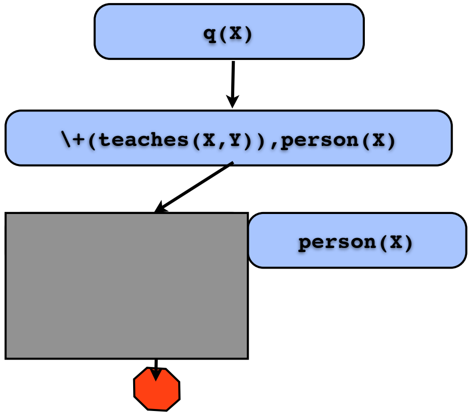



```
person(a).
person(b).
teaches(a,b).
```

```
q(X) :- \+(teaches(X,Y)),
        person(X).
```

Behavior

```
person(a).
person(b).
teaches(a,b).
```




```
q(X) :- \+(teaches(X,Y)),
        person(X).
```


Searching a graph

```
find(X,X).
find(X,Z) :- edge(X,Y),
             find(Y,Z).
```

- **Problem:** Loops easily if graph is cyclic:

```
edge(a,b).
edge(b,c).
edge(c,a).
```


Searching a graph (2)

- To avoid looping:
  1. Remember where we have been
  2. Stop if we try to visit a node we've already seen.

```
find2(X,X,_).
find2(X,Z,P) :- \+(member(X,P)),
                edge(X,Y),
                find2(Y,Z,[X|P]).
```

Note: Needs mode `(+,?,+)`.

Safe use of negation-as-failure

- As with cut, negation-as-failure can have non-logical behavior
  - **Goal order matters**
- `\+(X = Y), X = a, Y = b`
  - fails
- `X = a, Y = b, \+(X = Y)`
  - succeeds

Safe use of negation as failure (2)

- Can read `\+(G)` as "not `G`" only if `G` is **ground when we start solving it**
- Any free variables "existentially quantified"
  - `\+(1=2) == 1 ≠ 2`
  - `\+(X=Y) == ∃ X, Y. X ≠ Y`
- General heuristic: delay negation after other goals to make negated goals ground

Collecting solutions

- We'd like to find **all solutions**
  - collected as an explicit list
- `alist(bart,X)` = "`X` lists the ancestors of `bart`"
- Can't do this in pure Prolog
  - cut doesn't help
- Technically possible (but painful) using assert/ retract

Collecting solutions, declaratively

Collecting solutions, declaratively

- Built-in predicate to do same thing:
  - `findall/3` - list of solutions

Collecting solutions, declaratively

- Built-in predicate to do same thing:
  - `findall/3` - list of solutions

```
?- findall(Y,ancestor(Y,bart),L).
L = [homer,marge,abe,jacqueline]
```


Collecting solutions, declaratively

- Built-in predicate to do same thing:
  - `findall/3` - list of solutions

```
?- findall(Y,ancestor(Y,bart),L).
L = [homer,marge,abe,jacqueline]
?- findall((X,Y),ancestor(X,Y),L).
L = [(abe,homer),(homer,bart),
(homer,lisa),(homer,maggie)|...]
```


`findall/3`

- Usage:

```
findall(?X, ?Goal, ?List)
```

- On success, `List` is list of all substitutions for `X` that make `Goal` succeed.
- Goal can have free variables!
  - `X` treated as "bound" in `G`
- (`X` could also be a "pattern"...)

`bagof/3`

bagof/3

- `bagof/3` - list of solutions

```
?- bagof(Y,ancestor(Y,bart),L).
L = [homer,marge,abe,jacqueline]
```


bagof/3

- `bagof/3` - list of solutions

```
?- bagof(Y,ancestor(Y,bart),L).
L = [homer,marge,abe,jacqueline]
```

- different instantiations of free variables lead to different answers

```
| ?- bagof(Y,ancestor(Y,X),L).
L = [homer,marge,abe,jacqueline],
X = bart ? ;
```


bagof/3

- `bagof/3` - list of solutions

```
?- bagof(Y,ancestor(Y,bart),L).
L = [homer,marge,abe,jacqueline]
```

- different instantiations of free variables lead to different answers

```
| ?- bagof(Y,ancestor(Y,X),L).
L = [homer,marge,abe,jacqueline],
X = bart ? ;
L = [abe],
X = homer ? ...
```


Quantification

- In goal part of a `bagof/3`, we can write:

  `X^G(X)`

  to "hide" (existentially quantify) `X`.

```
| ?- bagof(Y,X^ancestor(X,Y),L).
L = [homer,bart,lisa,maggie,rod,
    todd,ralph,bart,lisa,maggie|...]
```

  - This also works for `findall/3`, but is redundant

setof/3

setof/3

- Similar to bagof/3, but **sorts** and **eliminates duplicates**

```
| ?- bagof(Y,X^ancestor(X,Y),L).
L = [homer,bart,lisa,maggie,rod,
     todd,ralph,bart,lisa,maggie|...]
```


setof/3

- Similar to bagof/3, but **sorts** and **eliminates duplicates**

```
| ?- bagof(Y,X^ancestor(X,Y),L).
L = [homer,bart,lisa,maggie,rod,
     todd,ralph,bart,lisa,maggie|...]
| ?- setof(Y,X^ancestor(X,Y),L).
L = [bart,homer,lisa,maggie,marge,
     patty,ralph,rod,selma,todd]
```


Assert and retract

- So far we have **statically** defined facts and rules
  - usually in separate file
- We can also **dynamically** add and remove clauses

assert/1

assert/1

```
?- assert(p).
yes
```


assert/1

```
?- assert(p).
yes
?- p.
yes
```


assert/1

```
?- assert(p).
yes
?- p.
yes
?- assert(q(1)).
yes
```

assert/1

```
?- assert(p).
yes
?- p.
yes
?- assert(q(1)).
yes
?- q(X).
X = 1.
```


Searching a graph using assert/1

```
:- dynamic v/1.
find3(X,X).
find3(X,Z) :- \+(v(X)),
              assert(v(X)),
              edge(X,Y),
              find3(Y,Z).
```

- Mode (+,?).
- **Problem: Need to clean up afterwards.**

Fibonacci

```
fib(0,0).
fib(1,1).
fib(N,K) :- N >= 2,
            M is N-1, fib(M,F),
            P is M-1, fib(P,G),
            K is F+G.
```


Fibonacci, memoized

```
:- dynamic memofib/2.
fib(N,K) :- memofib(N,K), !.
...
fib(N,K) :- N >= 2,
            M is N-1, fib(M,F),
            P is M-1, fib(P,G),
            K is F+G,
            assert(memofib(N,K)).
```

`asserta/1` and `assertz/1`

- Provide limited control over clause order.
- `asserta/1` adds to beginning of KB
- `assertz/1` adds to end of KB

`retract/1`

```
?- retract(p).
yes
?- p.
no
?- retract(q(1)).
yes
?- q(X).
no
```


Warning

- If you assert or retract an unused predicate interactively, Sicstus Prolog **assumes** it is dynamic
- But if you want to use assert/retract in programs, you should **declare** as dynamic in the program
  - for example:

    ```
    :- dynamic memofig/2.
    ```
- Generally wise to avoid assert/retract without good reason

Collecting solutions using assert, retract

- Here's a way to calculate list of all ancestors using assert/retract:

```
:- dynamic p/1.
alist(X,L) :- assert(p([])),
              collect(X);
              p(L),
              retract(p(L)).
collect(X) :- ancestor(Y,X),
              retract(p(L)),
              assert(p([Y|L])),
              fail.
```

- **Kind of a hack! (also need to clean up afterwards).**

- Next time: Definite Clause Grammars
- Further reading: LPN, ch. 10 & 11