# Logic Programming
## *View Evaluation*

Michael Genesereth
Computer Science Department
Stanford University

# Bottom-Up Evaluation

**Method**

Start with dataset
Apply rules repeatedly to produce closure
Repeat up the stratum hierarchy
Check whether query is in the resulting dataset

**Disadvantages**

Generates large numbers of irrelevant conclusions
Does not work with infinite extensions

# Top-Down Evaluation

**Method**

Start with query to be answered
Apply rules repeatedly to reduce to subqueries
Continue until reaches data level
Match base level subgoals against dataset

**Disadvantages**

Slightly harder to understand
Sometimes recomputes subgoals
Susceptible to *avoidable* infinite loops

# Programme

Top-Down Processing of Ground Goals and Rules

Unification

Top-Down Processing of Goals and Rules with Variables

# Ground Goals and Rules

# Sketch of Procedure for Ground Case

Given a query, a dataset, and a ruleset, do the following.

(1) If the predicate in the query is a **base predicate**, succeed if and only if query is in dataset.

(2) If the query is a **negation**, evaluate target and succeed if and only if fail to prove.

(3) If the query is a **conjunction**, succeed iff succeed on all conjuncts.

(4) If the predicate in the query is a **view predicate**, evaluate the body of each rule defining that predicate and succeed if and only if succeed on at least one rule.

# Example

**Dataset**

```
p(a)
p(b)
p(c)
q(d)
```

**Ruleset**

```
s(c) :- p(a) & q(b)
s(c) :- p(b) & t(c)
s(c) :- p(c) & ~q(c)
t(c) :- p(a) & p(d)
```

# Example

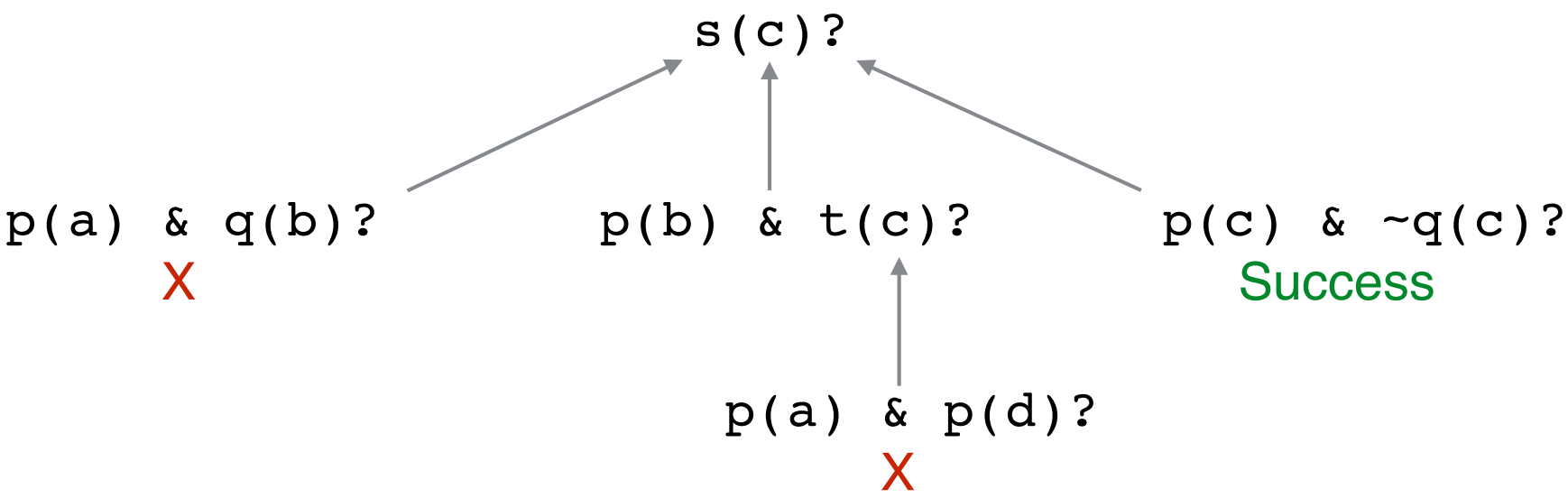
**Dataset**

```
p(a)
p(b)
p(c)
q(d)
```

**Ruleset**

```
s(c) :- p(a) & q(b)
s(c) :- p(b) & t(c)
s(c) :- p(c) & ~q(c)
t(c) :- p(a) & p(d)
```

**Top Down Evaluation**



# Unification

# Unification

*Unification* is the process of determining whether two expressions can be *unified*, i.e. made identical by appropriate substitutions for their variables.

Example: `p(a,Y)` and `p(X,b)` can be unified. If we replace `X` by `a` and `Y` by `b`, we end up with `p(a,b)` in both cases.

# Substitutions

A *substitution* is a finite set of pairs of variables and terms, called *replacements*.

$$\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(b)}, \texttt{Z}\leftarrow\texttt{V}\}$$

Domain: $\{\texttt{X}, \texttt{Y}, \texttt{Z}\}$
Range: $\{\texttt{a}, \texttt{f(b)}, \texttt{V}\}$

NB: Domain elements *must* be variables.
NB: Replacements *may* contain variables.

# Application

The result of applying a substitution σ to an expression φ is the expression φσ obtained from φ by replacing every occurrence of every variable in the substitution by its replacement.

$$\texttt{q(X,Y)}\ \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(b)}, \texttt{Z}\leftarrow\texttt{V}\} = \texttt{q(a,f(b))}$$
$$\texttt{q(X,X)}\ \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(b)}, \texttt{Z}\leftarrow\texttt{V}\} = \texttt{q(a,a)}$$
$$\texttt{q(X,W)}\ \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(b)}, \texttt{Z}\leftarrow\texttt{V}\} = \texttt{q(a,W)}$$
$$\texttt{q(Z,V)}\ \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(b)}, \texttt{Z}\leftarrow\texttt{V}\} = \texttt{q(V,V)}$$

# Cascaded Substitutions

$$\texttt{r(X,Y,Z)}\{\texttt{x}\leftarrow\texttt{a}, \texttt{y}\leftarrow\texttt{f(U)}, \texttt{Z}\leftarrow\texttt{V}\} = \texttt{r(a,f(U),V)}$$

$$\texttt{r(a,f(U),V)}\{\texttt{U}\leftarrow\texttt{d}, \texttt{V}\leftarrow\texttt{e}, \texttt{Z}\leftarrow\texttt{g}\} = \texttt{r(a,f(d),e)}$$

$$\texttt{r(X,Y,Z)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(d)}, \texttt{Z}\leftarrow\texttt{e}, \texttt{U}\leftarrow\texttt{d}, \texttt{V}\leftarrow\texttt{e}\} = \texttt{r(a,f(d),e)}$$

# Composition of Substitutions

The *composition* of substitution $\sigma$ and $\tau$ is the substitution (written *compose*($\sigma$,$\tau$) or, more simply, $\sigma\tau$) obtained by

(1) applying $\tau$ to the replacements in $\sigma$

(2) adding to $\sigma$ pairs from $\tau$ with different variables

(3) deleting any assignments of a variable to itself.

$$\{X \leftarrow a, Y \leftarrow U, Z \leftarrow V\}\{U \leftarrow d, V \leftarrow e, Z \leftarrow g\}$$
$$= \{X \leftarrow a, Y \leftarrow d, Z \leftarrow e\}\{U \leftarrow d, V \leftarrow e, Z \leftarrow g\}$$
$$= \{X \leftarrow a, Y \leftarrow d, Z \leftarrow e, U \leftarrow d, V \leftarrow e\}$$

# Unification

A substitution $\sigma$ is a *unifier* for an expression $\varphi$ and an expression $\psi$ if and only if $\varphi\sigma=\psi\sigma$.

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{b}, \texttt{V}\leftarrow\texttt{b}\} = \texttt{p(a,b)}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{b}, \texttt{V}\leftarrow\texttt{b}\} = \texttt{p(a,b)}$$

If two expressions have a unifier, they are said to be *unifiable*. Otherwise, they are *nonunifiable*.

```
p(X,X)
p(a,b)
```

# Non-Uniqueness of Unification

Unifier 1:

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{b}, \texttt{V}\leftarrow\texttt{b}\} = \texttt{p(a,b)}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{b}, \texttt{V}\leftarrow\texttt{b}\} = \texttt{p(a,b)}$$

Unifier 2:

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(W)}, \texttt{V}\leftarrow\texttt{f(W)}\} = \texttt{p(a,f(W))}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{f(W)}, \texttt{V}\leftarrow\texttt{f(W)}\} = \texttt{p(a,f(W))}$$

Unifier 3:

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{V}\} = \texttt{p(a,V)}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{V}\} = \texttt{p(a,V)}$$

# Most General Unifier

A substitution σ is a *most general unifier* (*mgu*) of two expressions if and only if it is *as general as or more general than* any other unifier.

Theorem: If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{V}\} = \texttt{p(a,V)}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{V}\} = \texttt{p(a,V)}$$

$$\texttt{p(X,Y)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{V}\leftarrow\texttt{Y}\} = \texttt{p(a,Y)}$$
$$\texttt{p(a,V)}\{\texttt{X}\leftarrow\texttt{a}, \texttt{V}\leftarrow\texttt{Y}\} = \texttt{p(a,Y)}$$

# Unification Procedure

One good thing about our language is that there is a simple and inexpensive procedure for computing a most general unifier of any two expressions if it exists.

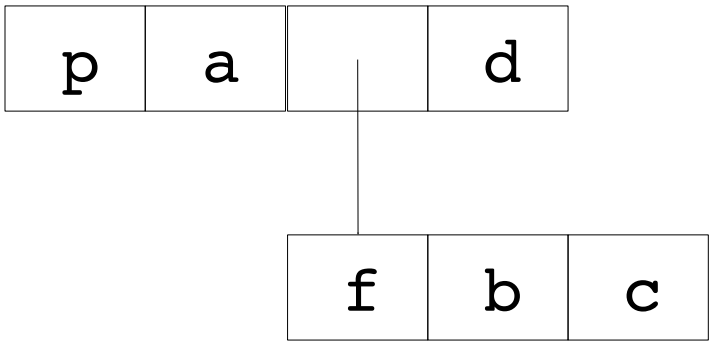
# Expression Structure

Each expression is treated as a sequence of its immediate subexpressions.

Linear Version:

```
p(a, f(b, c), d)
```

Structured Version:



# Unification Procedure

(1) If two expressions being compared are identical, succeed.

(2) If neither is a variable and at least one is a constant, fail.

(3) If one of the expressions is a variable, proceed as described shortly.

(4) If both expressions are sequences, iterate across the expressions, comparing each subexpression as described above.

# Dealing With Variables

If one of the expressions is a variable, check whether the variable has a binding in the current substitution.

(a) If so, try to unify the *binding* with the other expression.

(b) If no binding, check whether the other expression contains the variable. If the variable occurs within the expression, fail; otherwise, set the substitution to the composition of the old substitution and a new substitution in which variable is bound to the other expression.

# Example

**Call:** `p(X,b)`, `p(a,Y)`, {}

**Call:** `p`, `p`, {}
**Exit:** {}

**Call:** `X`, `a`, {}
**Exit:** {}{`X`←`a`} = {`X`←`a`}

**Call:** `b`, `Y`, {`X`←`a`}
**Exit:** {`X`←`a`} {`Y`←`b`} = {`X`←`a`, `Y`←`b`}

**Exit:** {`X`←`a`, `Y`←`b`}

# Example

**Call:** `p(X,X)`, `p(a,Y)`, {}

**Call:** `p`, `p`, {}
**Exit:** {}

**Call:** `X`, `a`, {}
**Exit:** {}{`X`←`a`} = {`X`←`a`}

**Call:** `X`, `Y`, {`X`←`a`}
**Call:** `a`, `Y`, {`X`←`a`}
**Exit:** {`X`←`a`} {`Y`←`a`} = {`X`←`a`, `Y`←`a`}
**Exit:** {`X`←`a`, `Y`←`a`}

**Exit:** {`X`←`a`, `Y`←`a`}

# Example

**Call:** `p(X,X)`, `p(a,b)`, {}

**Call:** `p`, `p`, {}
**Exit:** {}

**Call:** `X`, `a`, {}
**Exit:** {}{`X`←`a`} = {`X`←`a`}

**Call:** `X`, `b`, {`X`←`a`}
**Call:** `a`, `b`, {`X`←`a`}
**Exit:** *false*
**Exit:** *false*

**Exit:** *false*

# Example

**Call:** `p(X,X)`, `p(Y,f(Y))`, {}

**Call:** `p`, `p`, {}
**Exit:** {}

**Call:** `X`, `Y`, {}
**Exit:** {}{`X`←`Y`} = {`X`←`Y`}

**Call:** `X`, `f(Y)`, {`X`←`Y`}
**Call:** `Y`, `f(Y)`, {`X`←`Y`}
**Exit:** *false*
**Exit:** *false*

**Exit:** *false*

# Reason

Circularity Problem:

$$\{X \leftarrow f(Y), Y \leftarrow f(Y)\}$$

Unification Problem:

$$p(X,X)\{X \leftarrow f(Y), Y \leftarrow f(Y)\} = p(f(Y),f(Y))$$

$$p(Y,f(Y))\{X \leftarrow f(Y), Y \leftarrow f(Y)\} = p(f(Y),f(f(Y)))$$

Before assigning a variable to an expression, first check that the variable does not occur within that expression.

This is called the *occur check* test.

Prolog does not do the occur check (and is proud of it).

# General Goals and Rules

# Procedure With Variables

Procedure without variables uses *equality* tests.

```
p(a,b)
p(b,c)
s(a,c) :- p(a,b) & p(b,c)

s(a,c)?
```

Procedure with variables uses *unification*.

```
p(a,b)
p(b,c)
s(X,Z) :- p(X,Y) & p(Y,Z)

s(a,c)?
```

# Step 1 - Atoms with Base Relations

Given an atom with a base relation and a substitution:

(a) Compare the goal to each factoid in our dataset.

(b) If there is an extension of the given substitution that unifies the goal and the factoid, add to our list of answers.

(c) Once all relevant factoids examined, return answers.

# Example 1 - Atoms with Base Relations

Goal: `p(X,Y)`

Substitution: {X←a}

Dataset: {p(a,b), p(a,c), p(b,c)}

Result: [{X←a, Y←b}, {X←a, Y←c}]

# Step 2 - Negations

Given a negation and a substitution:

(a) Execute the procedure on the target of the negation and the given substitution.

(b) If the result is empty, return a singleton list containing the given substitution, indicating success.

(c) Otherwise, return the empty list of answers, indicating failure.

# Example 2 - Negations

Goal: `~p(X,Y)`
Substitution: {X←a, Y←d}
Dataset: {p(a,b), p(a,c), p(b,c)}
Result: [{X←a, Y←d}]

Goal: `~p(X,Y)`
Substitution: {X←a, Y←c}
Dataset: {p(a,b), p(a,c), p(b,c)}
Result: []

# Step 3 - Conjunctions

Given a conjunction and a substitution:

(a) Execute our procedure on the first conjunct and the given substitution to get a list of answers.

(b) Iterate through the list of substitutions, calling the procedure recursively on the remaining conjuncts with each substitution in turn.

(c) Collect the answers from recursive calls and return.

# Example 3 - Conjunctions

Goal: `p(X,Y) & p(Y,Z)`
Substitution: {X←a}
Dataset: {p(a,b), p(a,c), p(b,c)}

Call: p(X,Y), {X←a}
Result: [{X←a, Y←b},{X←a, Y←c}]

Call: p(Y,Z), {X←a, Y←b}
Result: [{X←a, Y←b, Z←c}]

Call: p(Y,Z), {X←a, Y←c}
Result: p(Y,Z): []

Overall Result: [{X←a, Y←b, Z←c}]

# Step 4 - Atoms with View Relations

Given atom with view relation and a substitution:

(a) Iterate through the rules in our program.

(b) *Copy each rule, replacing variables with new variables*.

(c) Try to unify the given goal and the new rule head.

(d) Call the procedure recursively on the body of the rule.

(e) Return substitutions from all successful cases.

# Example 4 - Atoms with View Relations

Goal: `q(X,Y)`
Substitution: {X←a}
Rule: `q(X,Z) :- p(X,Y) & p(Y,Z)`
Dataset: {p(a,b),p(a,c),p(b,c)}

Copy of rule: `q(U,W) :- p(U,V) & p(V,W)`

Unification: `q(U,W)` `q(X,Y)` {X←a}
Result: {U←a,W←Y,X←a}

New Goal: {p(U,V) & p(V,W)}
New Substitution: {U←a,W←Y,X←a}

Result: [{U←a,W←c,X←a,V←b,Y←c}]

# Compound Terms

Compound terms compound the difficulty

Rule

```
s(X,f(Y,Z)) :- p(X,g(Y)) & p(Y,X)
```

Query

```
s(h(X),X)
```

Subgoal

```
p(h(f(Y,Z)),g(Y)) & p(Y,h(f(Y,Z)))
```

# Efficiency Enhancements

Multiple substitutions

Different substitutions used for goals and rules

*Good: Rules are not copied*

Evaluation of conjuncts is pipelined

Once each answer to a conjunct is computed,
the other conjuncts are checked immediately;
then other answers generated and checked.

*Good: Saves work when only few answers needed.*

*Good: Avoids problems due to infinite answer sets.*

Upshot: This is complicated. Don't try this at home. Leave it to the professionals.

# Tracing

Facts and Rules

```
p(a,b)
p(b,c)
s(X,Z) :- p(X,Y) & p(Y,Z)
```

Trace

```
Call: s(X,Z)
| Call: p(X,Y)
| Exit: p(a,b)
| Call: p(b,Z)
| Exit: p(b,c)
Exit: s(a,c)
```

# Backup Tracing

Facts and Rules

```
p(a,b)
p(b,c)
s(X,Z) :- p(X,Y) & p(Y,Z)
```

Trace

```
Call: s(X,Z)
| Call: p(X,Y)
| Exit: p(a,b)
| Call: p(b,Z)
| Exit: p(b,c)
Exit: s(a,c)
```

```
Redo: s(X,Z)
| Redo: p(b,Z)
| Fail: p(b,Z)
| Redo: p(X,Y)
| Exit: p(b,c)
| Call: p(c,Z)
| Fail: p(c,Z)
Fail: s(X,Z)
```

# Summary

# Comparison of Evaluation Strategies

**Bottom-Up Evaluation**

- Easy to understand
- Computes all results
- Computes subresults just once
- Wasteful when want just one or a few answers, not all
- Problematic on logic programs with infinite models

**Top-Down Evaluation**

- Less waste when want one or a few answers
- More complicated
- Subqueries evaluated multiple times
- Possibility of infinite loops on programs w/ finite models

# But …

**Bottom-Up Evaluation**

Can be focussed using Magic Sets

**Top-Down Evaluation**

Top-Down can avoid duplication through caching

Infinite Loops can be avoided using iterative deepening

*The arms race continues.*

Sierra

# Sierra

Sierra is browser-based IDE (interactive development environment) for Epilog.

Saving and loading files

Viewing and Editing datasets
Querying datasets
Transformation tools for datasets

Interpreter (for view definitions, action definitions)
Trace capability (useful for debugging rules)
Analysis tools (error checking and optimizing rules)

http://epilog.stanford.edu/homepage/sierra.php

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

**Lambda**

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

**Query**

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

**Transform**

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

## Library

Save Revert

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

## Library

Save Revert

```
anc(X,Y) :- p(X,Y)
anc(X,Z) :- p(X,Y) & anc(Y,Z)
```

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

## Library

Save Revert

```
anc(X,Y) :- p(X,Y)
anc(X,Z) :- p(X,Y) & anc(Y,Z)
```

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

## Library

Save Revert

```
anc(X,Y) :- p(X,Y)
anc(X,Z) :- p(X,Y) & anc(Y,Z)
```

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

## Compute

Query

Show Next 100 result(s) ☐ Autorefresh

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
```

## Library

Save Revert

```
anc(X,Y) :- p(X,Y)
anc(X,Z) :- p(X,Y) & anc(Y,Z)
```

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
```

## Compute

Query `anc(b,Z)`

Show Next 100 result(s) ☐ Autorefresh

```
anc(b,c)
anc(b,d)
anc(b,e)
```

epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

## Lambda

Save Revert Sort

```
p(a,b)
p(b,c)
p(c,d)
p(d,e)
p(e,f)
```

## Library

Save Revert

```
anc(X,Y) :- p(X,Y)
anc(X,Z) :- p(X,Y) & anc(Y,Z)
```

## Query

Pattern `goal(X,Z)`

Query `p(X,Y) & p(Y,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
goal(a,c)
goal(b,d)
goal(c,e)
goal(d,f)
```

## Transform

Condition `p(X,Y)`

Conclusion `~p(X,Y) & p(Y,X)`

Expand ☑ Expand on update

Execute ☐ Run on clock tick

```
~p(a,b)
p(b,a)
~p(b,c)
p(c,b)
~p(c,d)
p(d,c)
~p(d,e)
p(e,d)
~p(e,f)
p(f,e)
```

## Compute

Query `anc(b,Z)`

Show Next 100 result(s) ☑ Autorefresh

```
anc(b,c)
anc(b,d)
anc(b,e)
anc(b,f)
```