

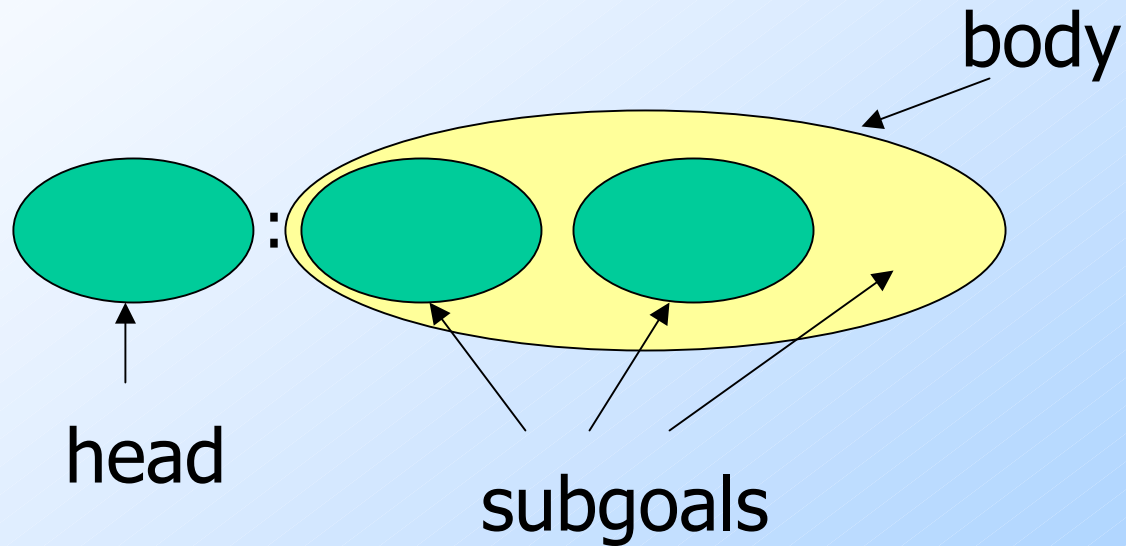
# Datalog

Rules

Programs

Negation

# Review of Logical If-Then Rules



“The head is true if all the subgoals are true.”

# Terminology

- ◆ Head and subgoals are *atoms*.
- ◆ An atom consists of a *predicate* (lower case) applied to zero or more *arguments* (upper case letters or constants).

# Semantics

- ◆ Predicates represent relations.
- ◆ An atom is true for given values of its variables iff the arguments form a tuple of the relation.
- ◆ Whenever an assignment of values to all variables makes all subgoals true, the rule asserts that the resulting head is also true.

# Example

- ◆ We shall develop rules that describe what is necessary to “make” a file.
- ◆ The predicates/relations:
  - ◆  $\text{source}(F) = F$  is a “source” file.
  - ◆  $\text{includes}(F, G) = F$  #includes  $G$ .
  - ◆  $\text{create}(F, P, G) = F$  is created by applying process  $P$  to file  $G$ .

# Example --- Continued

- ◆ Rules to define “view”  $\text{req}(X,Y)$  = file  $Y$  is required to create file  $X$ :

$\text{req}(F,F) :- \text{source}(F)$

$\text{req}(F,G) :- \text{includes}(F,G)$

$G$  is required for  $F$  if there is some process  $P$  that creates  $F$  from  $G$ .

$G$  is required for  $F$  if there is some  $H$  such that  $H$  is required for  $F$  and  $G$  is required for  $H$ .

# Why Not Just Use SQL?

1. Recursion is much easier to express in Datalog.
  - ◆ Viz. last rule for `req`.
2. Rules express things that go on in both FROM and WHERE clauses, and let us state some general principles (e.g., containment of rules) that are almost impossible to state correctly in SQL.

# IDB/EDB

- ◆ A predicate representing a stored relation is called *EDB* (extensional database).
- ◆ A predicate representing a “view,” i.e., a defined relation that does not exist in the database is called *IDB* (intensional database).
- ◆ Head is always IDB; subgoals may be IDB or EDB.



# Datalog Programs

- ◆ A collection of rules is a (Datalog) *program*.
- ◆ Each program has a distinguished IDB predicate that represents the result of the program.
  - ◆ E.g., `req` in our example.

# Extensions

1. Negated subgoals.
2. Constants as arguments.
3. Arithmetic subgoals.

# Negated Subgoals

- ◆ NOT in front of a subgoal means that an assignment of values to variables must make it false in order for the body to be true.
- ◆ Example:  
    `cycle(F) :- req(F,F) & NOT source(F)`

# Constants as Arguments

- ◆ We use numbers, lower-case letters, or quoted strings to indicate a constant.

- ◆ Example:

```
req( "foo.c" , "stdio.h" ) :-
```

- ◆ Note that empty body is OK.
- ◆ Mixed constants and variables also OK.

# Arithmetic Subgoals

- ◆ Comparisons like  $<$  may be thought of as infinite, binary relations.
  - ◆ Here, the set of all tuples  $(x,y)$  such that  $x < y$ .
- ◆ Use infix notation for these predicates.
- ◆ Example:

```
composite(A) :- divides(B,A) &  
                B > 1 & B != A
```


# Evaluating Datalog Programs

1. Nonrecursive programs.
2. Naïve evaluation of recursive programs without IDB negation.
3. Seminaïve evaluation of recursive programs without IDB negation.
  - ◆ Eliminates some redundant computation.

# Safety

- ◆ When we apply a rule to finite relations, we need to get a finite result.
- ◆ Simple guarantee: *safety* = all variables appear in some nonnegated, relational (not arithmetic) subgoal of the body.
  - ◆ Start with the join of the nonnegated, relational subgoals and select/delete from there.

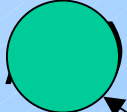
# Examples: Nonsafety

$p$    $:- q(Y)$   $X$  is the problem

Both X and Y  
are problems

$bachelor(X) :- NOT\ married$  

$bachelor(X) :- person(X) \ \&$

$NOT\ married(X)$  

Y is still a problem



# Nonrecursive Evaluation

- ◆ If (and only if!) a Datalog program is not recursive, then we can order the IDB predicates so that in any rule for  $p$  (i.e.,  $p$  is the head predicate), the only IDB predicates in the body precede  $p$ .

# Why?

- ◆ Consider the *dependency graph* with:
  - ◆ Nodes = IDB predicates.
  - ◆ Arc  $p \rightarrow q$  iff there is a rule for  $p$  with  $q$  in the body.
- ◆ Cycle involving node  $p$  means  $p$  is recursive.
- ◆ No cycles: use topological order to evaluate predicates.

# Applying Rules

- ◆ To evaluate an IDB predicate  $p$  :
  1. *Apply* each rule for  $p$  to the current relations corresponding to its subgoals.
    - ◆ “Apply” = If an assignment of values to variables makes the body true, insert the tuple that the head becomes into the relation for  $p$  (no duplicates).
  2. Take the union of the result for each  $p$ -rule.

# Example

$p(X, Y) \text{ :- } q(X, Z) \ \& \ r(Z, Y) \ \& \ Y < 10$

$Q = \{(1,2), (3,4)\};$

$R = \{(2,5), (4,9), (4,10), (6,7)\}$

◆ Assignments making the body true:

$(X, Y, Z) = (1, 5, 2), (3, 9, 4)$

◆ So  $P = \{(1,5), (3,9)\}$ .

# Algorithm for Nonrecursive

```
FOR (each predicate p in  
    topological order) DO  
    apply the rules for p to  
    previously computed relations  
    to compute relation P for p;
```

# Naïve Evaluation for Recursive

```
make all IDB relations empty;  
WHILE (changes to IDB) DO  
  FOR (each IDB predicate p) DO  
    evaluate p using current  
      values of all relations;
```

# Important Points

- ◆ As long as there is no negation of IDB subgoals, then each IDB relation “grows,” i.e., on each round it contains at least what it used to contain.
- ◆ Since relations are finite, the loop must eventually terminate.
- ◆ Result is the *least fixedpoint* (*minimal model*) of rules.

# Seminaïve Evaluation

- ◆ Key idea: to get a new tuple for relation  $P$  on one round, the evaluation must use some tuple for some relation of the body that was obtained on the previous round.
- ◆ Maintain  $\Delta P$  = new tuples added to  $P$  on previous round.
- ◆ “Differentiate” rule bodies to be union of bodies with one IDB subgoal made “ $\Delta$ .”



# Example ("make files")

$r(F, F) :- s(F)$

$r(F, G) :- i(F, G)$

$r(F, G) :- c(F, P, G)$

$r(F, G) :- r(F, H) \ \& \ r(H, F)$

- ◆ Assume EDB predicates  $s, i, c$  have relations  $S, I, C$ .

# Example --- Continued

- ◆ Initialize:  $R = \Delta R = \sigma_{\#1=\#2}(S \times S) \cup I \cup \pi_{1,3}(C)$
- ◆ Repeat until  $\Delta R = \phi$ :
  1.  $\Delta R = \pi_{1,3}(R \bowtie \Delta R \cup \Delta R \bowtie R)$
  2.  $\Delta R = \Delta R - R$
  3.  $R = R \cup \Delta R$

# Function Symbols in Rules

- ◆ Extends Datalog by allowing arguments built from constants, variables, and function names, recursively applied.
- ◆ Function names look like predicate names, but are allowed only within the arguments of atoms.
  - ◆ Predicates return true/false; functions return arbitrary values.

# Example

- ◆ Instead of a string argument like "101 Maple" we could use a term like `addr(street("Maple"), number(101))`
- ◆ Compare with the XML term  
`<ADDR><STREET>Maple</STREET>  
<NUMBER>101</NUMBER>  
</ADDR>`

# Another Example

## ◆ Predicates:

1.  $\text{isTree}(X) = X$  is a binary tree.
2.  $\text{label}(L) = L$  is a node label.

## ◆ Functions:

1.  $\text{node}(A, L, R) =$  a tree with root labeled  $A$ , left subtree  $L$ , and right subtree  $R$ .
2.  $\text{null} =$  0-ary function (constant) representing the empty tree.

# Example --- Continued

## ◆ The rules:

```
isTree(null) :-
```

```
isTree(node(L, T1, T2)) :-
```

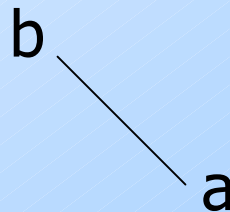
```
    label(L) &
```

```
    isTree(T1) &
```

```
    isTree(T2)
```

# Example --- Concluded

- ◆ Assume  $\text{label}(a)$  and  $\text{label}(b)$  are true.
  - ◆ I.e.,  $a$  and  $b$  are in the relation for  $\text{label}$ .
- ◆ Infer  $\text{isTree}(\text{node}(a, \text{null}, \text{null}))$ .
- ◆ Infer  $\text{isTree}(\text{node}(b, \text{null}, \text{node}(a, \text{null}, \text{null})))$ .



# Evaluation of Rules With Function Symbols

- ◆ Naïve, seminaïve still work when there are no negated IDB subgoals.
- ◆ They both lead to the unique least fixedpoint (minimal model).
- ◆ But... this fixedpoint may not be reached in any finite number of rounds.
  - ◆ The `isTree` rules are an example.



# Problems With IDB Negation

- ◆ When rules have negated IDB subgoals, there can be several minimal models.
- ◆ Recall: *model* = set of IDB facts, plus the given EDB facts, that make the rules true for every assignment of values to variables.
  - ◆ Rule is true unless body is true and head is false.

# Example: EDB

$\text{red}(X, Y)$  = the Red bus line runs from  $X$  to  $Y$ .

$\text{green}(X, Y)$  = the Green bus line runs from  $X$  to  $Y$ .

# Example: IDB

$\text{greenPath}(X, Y) =$  you can get from  $X$  to  $Y$  using only Green buses.

$\text{monopoly}(X, Y) =$  Red has a bus from  $X$  to  $Y$ , but you can't get there on Green, even changing buses.

# Example: Rules

```
greenPath(X,Y) :- green(X,Y)
```

```
greenPath(X,Y) :-  
    greenPath(X,Z) &  
    greenPath(Z,Y)
```

```
monopoly(X,Y) :- red(X,Y) &  
    NOT greenPath(X,Y)
```

# EDB Data

red(1,2), red(2,3), green(1,2)



# Two Minimal Models

1. EDB + greenPath(1,2) + monopoly(2,3)
2. EDB + greenPath(1,2) + greenPath(2,3) + greenPath(1,3)

greenPath(X,Y) :- green(X,Y)

greenPath(X,Y) :- greenPath(X,Z) &  
greenPath(Z,Y)

monopoly(X,Y) :- red(X,Y) &  
NOT greenPath(X,Y)



# Stratified Models

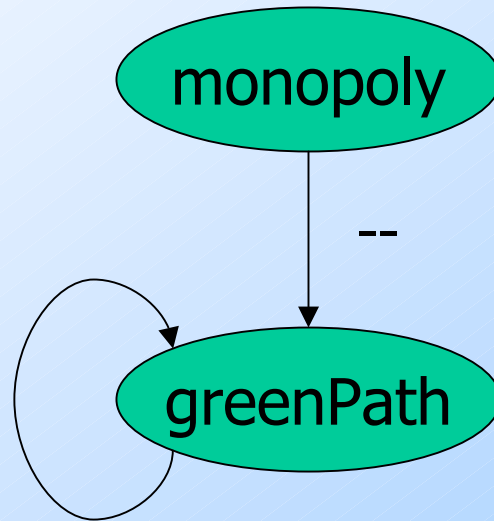
- 1. Dependency graph* describes how IDB predicates depend negatively on each other.
- 2. Stratified Datalog* = no recursion involving negation.
- 3. Stratified model* is a particular model that “makes sense” for stratified Datalog programs.

# Dependency Graph

- ◆ Nodes = IDB predicates.
- ◆ Arc  $p \rightarrow q$  iff there is a rule for  $p$  that has a subgoal with predicate  $q$ .
- ◆ Arc  $p \rightarrow q$  labeled – iff there is a subgoal with predicate  $q$  that is negated.



# Monopoly Example

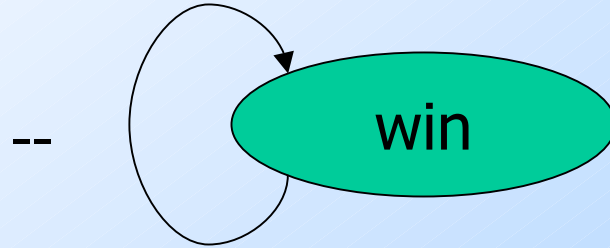


# Another Example: “Win”

```
win(X) :- move(X,Y) & NOT win(Y)
```

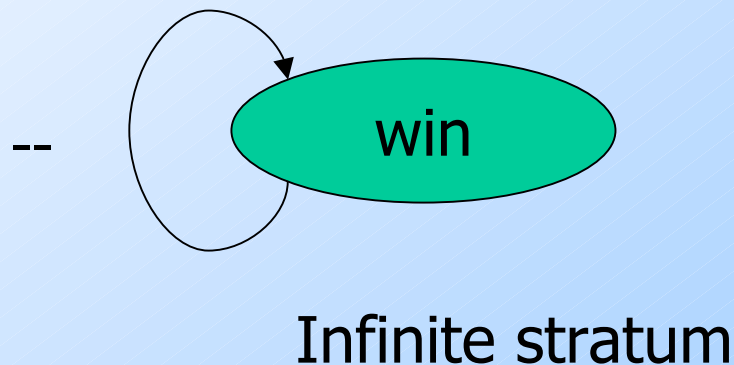
- ◆ Represents games like Nim where you win by forcing your opponent to a position where they have no move.

# Dependency Graph for "Win"

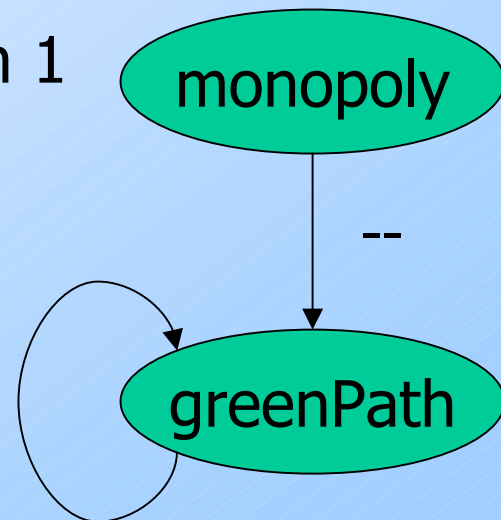


# Strata

- ◆ The stratum of an IDB predicate is the largest number of  $-$ 's on a path from that predicate, in the dependency graph.
- ◆ Examples:



Stratum 1



Stratum 0

# Stratified Programs

- ◆ If all IDB predicates have finite strata, then the Datalog program is *stratified*.
- ◆ If any IDB predicate has the infinite stratum, then the program is *unstratified*, and no stratified model exists.

# Stratified Model

- ◆ Evaluate strata  $0, 1, \dots$  in order.
- ◆ If the program is stratified, then any negated IDB subgoal has already had its relation evaluated.
  - ◆ Safety assures that we can “subtract it from something.”
  - ◆ Treat it as EDB.
- ◆ Result is the stratified model.

# Examples

- ◆ For “Monopoly,” `greenPath` is in stratum 0: compute it (the transitive closure of `green`).
- ◆ Then, `monopoly` is in stratum 1: compute it by taking the difference of `red` and `greenPath`.
- ◆ Result is first model proposed.
- ◆ “Win” is not stratified, thus no stratified model.