

Prolog Tutorial-II

Dr A Sahu
Dept of Computer Science &
Engineering
IIT Guwahati

Outline

- Quick recap:
 - General rule for variables
 - Fact, rule and queries
- Some simple programs
- **Unification**
- **List Map**
- **The Cut**

Prolog is a ‘Declarative’ language

- Clauses are statements about **what is true about a problem**, instead of instructions how to accomplish the solution.
- The Prolog system uses the clauses to work out how to accomplish the **solution by searching through the space of possible solutions**.
- *Not all problems have pure declarative specifications. Sometimes extralogical statements are needed.*

What a program looks like

```
/* At the Zoo */  
elephant(gaj) .  
elephant(aswasthama) .  
  
panda(chi_chi) .  
panda(ming_ming) .
```

Facts

```
dangerous(X) :- big_teeth(X) .  
dangerous(X) :- venomous(X) .  
guess(X, tiger) :- stripey(X), big_teeth(X),  
                 isaCat(X) .  
  
guess(X, zebra) :- stripey(X), isaHorse(X) .
```

Rules

Factorial Program

```
factorial(0, 1).  
factorial(N, F) :- N>0, N1 is N-1,  
    factorial(N1, F1), F is N * F1.
```

The Prolog goal to calculate the factorial of the number 3 responds with a value for W, the goal variable:

```
?- factorial(3, W).  
W=6
```

Complete Syntax of Terms

Term

Constant

Names an individual

Atom	Number
alpha17	0
gross_pay	1
john_smith	57
dyspepsia	1.618
+	2.04e-27
=/=	-13.6
'12Q&A'	

Compound Term

*Names an individual
that has parts*

likes(john, mary)
book(dickens, Z,
cricket)
f(x)
[1, 3, g(a), 7, 9]
-(+(15, 17), t)
15 + 17 - t

Variable

*Stands for an individual
unable to be named when
program is written*

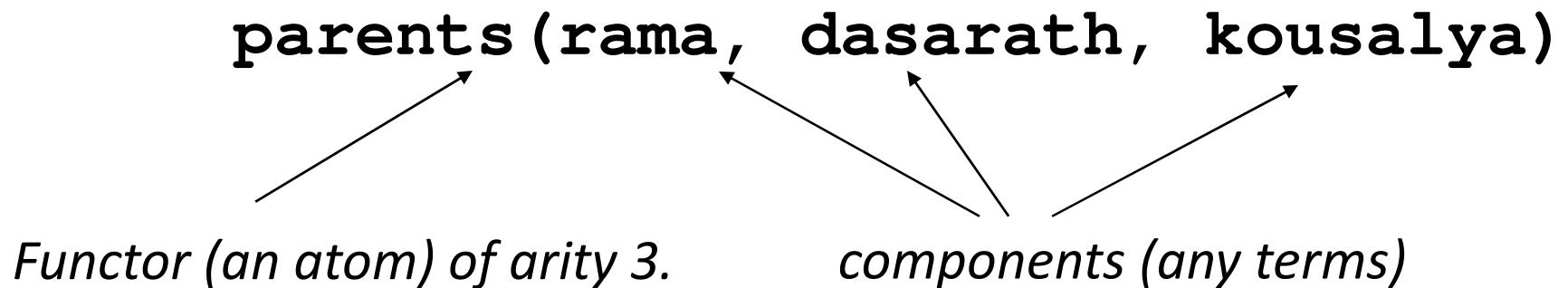
X
Gross_pay
Diagnosis
257
—

General Rules

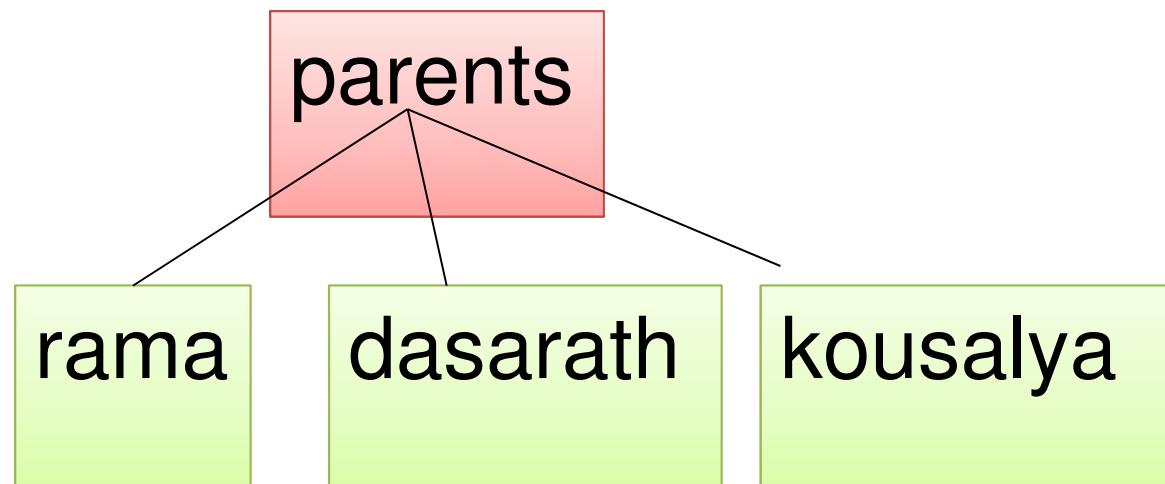
- variable start with
 - Capital letter or underscore
 - Mostly we use Capital **X, Y, Z, L, M** for variable
- atom start with
 - Mostly word written in small letters
 - likes, john, mary in likes (john, mary).
 - elephant gaj in elephant(gaj).

Compound Terms

The parents of Rama are Dasarath and Kousalya.



It is possible to depict the term as a tree:



Examples of operator properties

Position	Operator Syntax	Normal Syntax
Prefix:	-2	-(2)
Infix:	5+17	+(17,5)
Postfix:	N!	!(N)

Associativity: left, right, none.
 $X+Y+Z$ is parsed as $(X+Y)+Z$
because addition is left-associative.

Precedence: an integer.
 $X+Y*Z$ is parsed as $X+(Y*Z)$
because multiplication has higher precedence.

*These are all
the same as
the normal
rules of
arithmetic.*

Logical Operation on Numbers

$X ::= Y$ X and Y stands for the same number

$X =\neq Y$ X and Y stands for different numbers

$X < Y$ X is less then Y

$X > Y$

$X =< Y$ **Not same as in C (\geq , \leq)**

$X \geq Y$

The last point about Compound Terms...

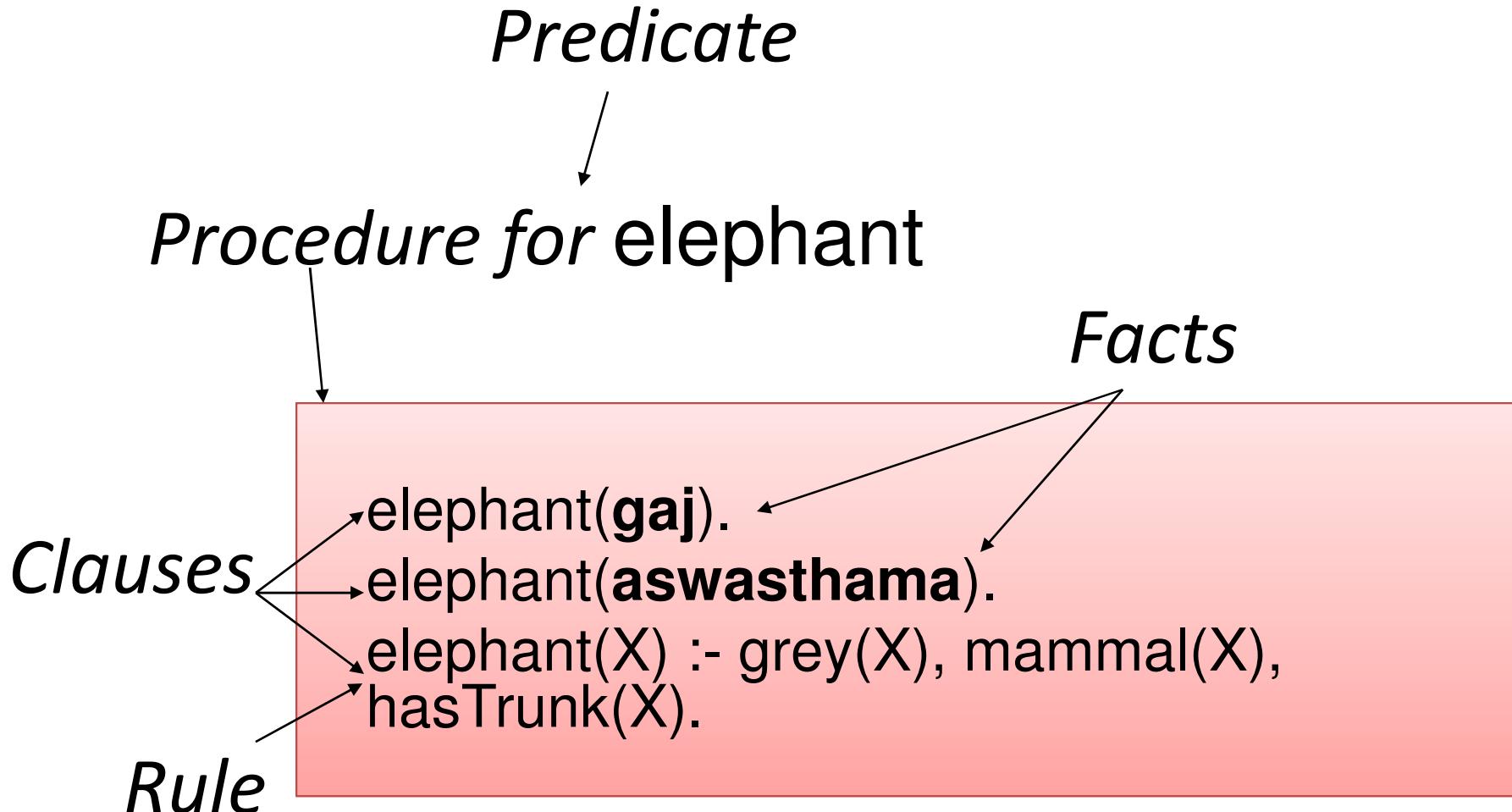
Constants are simply compound terms of arity 0.

badger means the same as
badger()

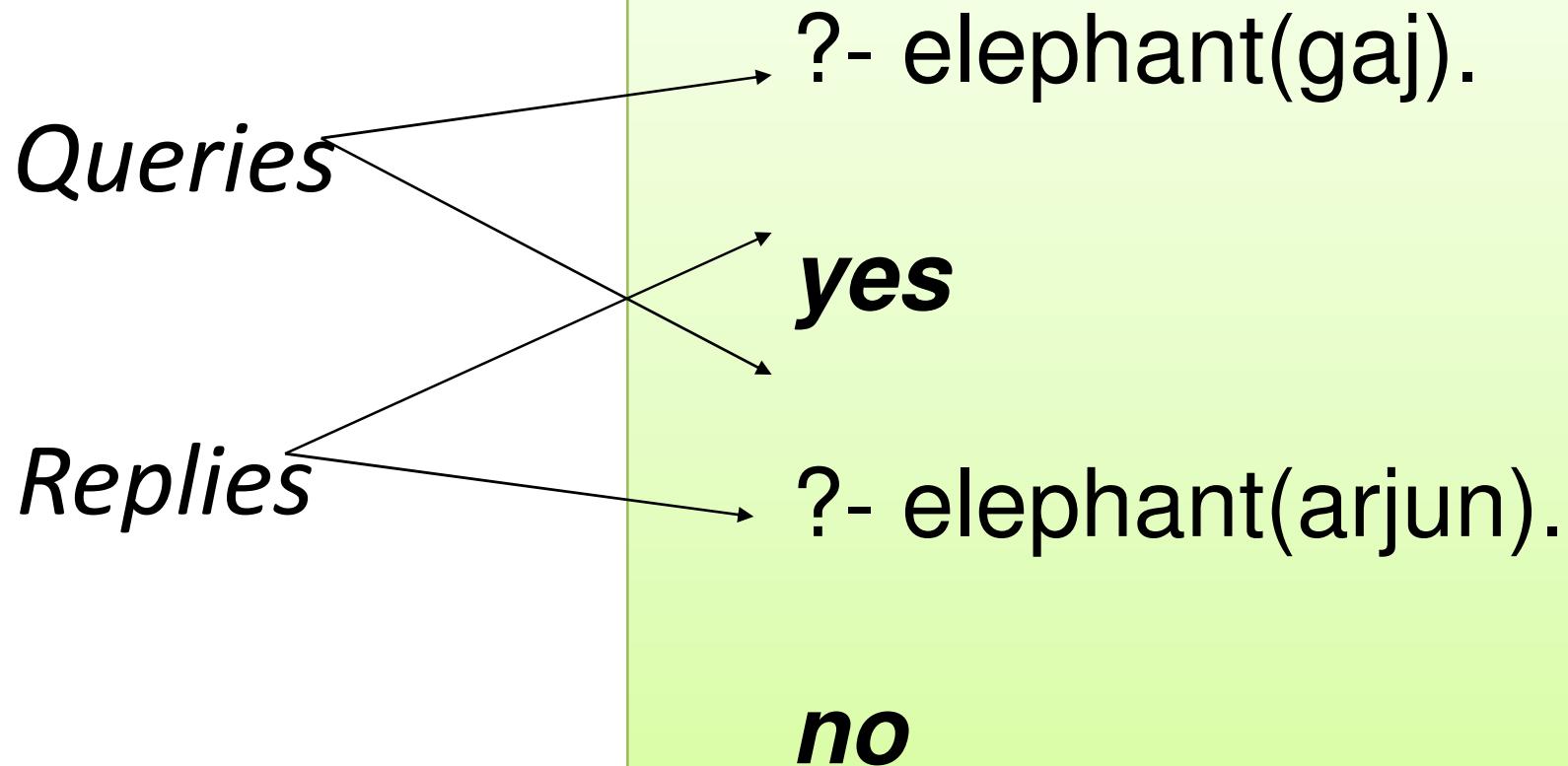
Structure of Prolog Programs

- Programs consist of procedures.
- Procedures consist of clauses.
- Each clause is a fact or a rule.
- Programs are executed by posing queries.

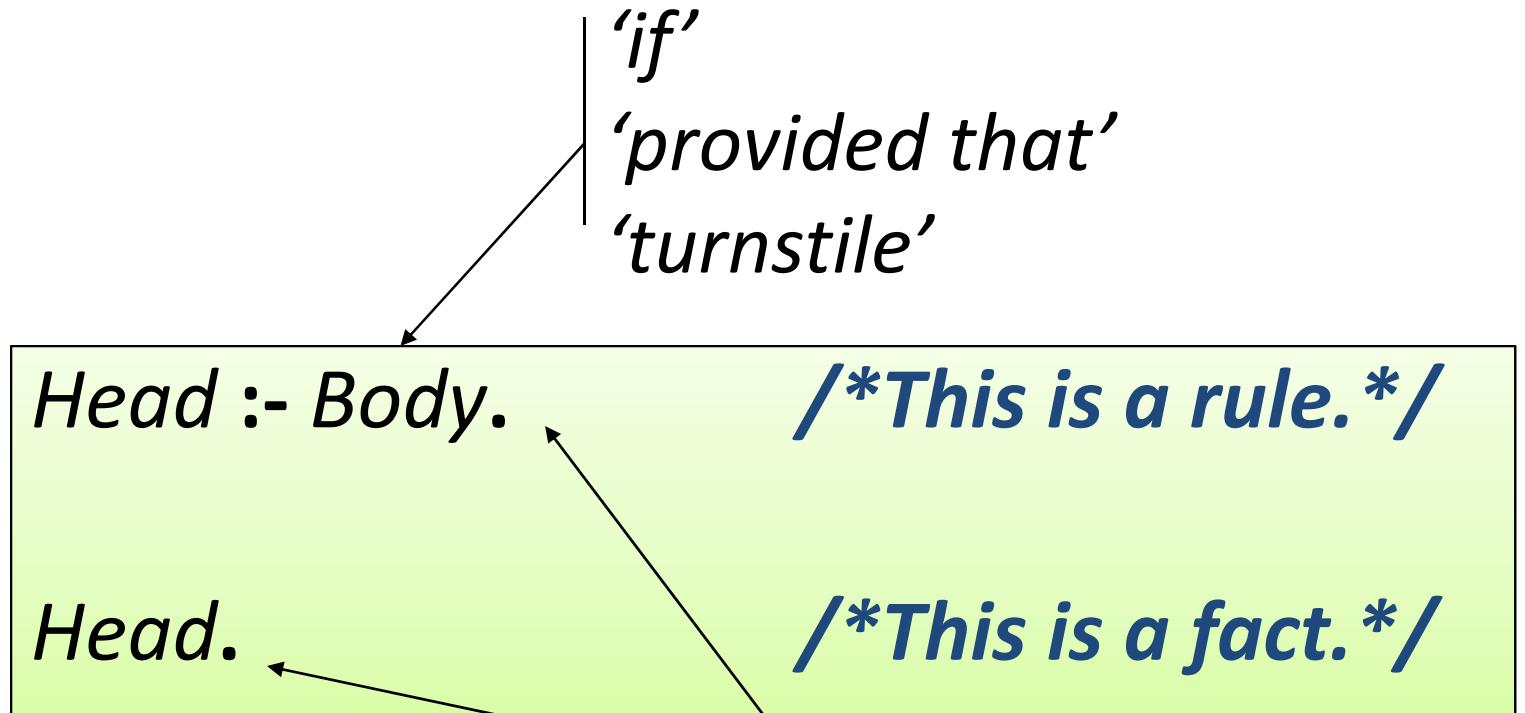
An Example



Example

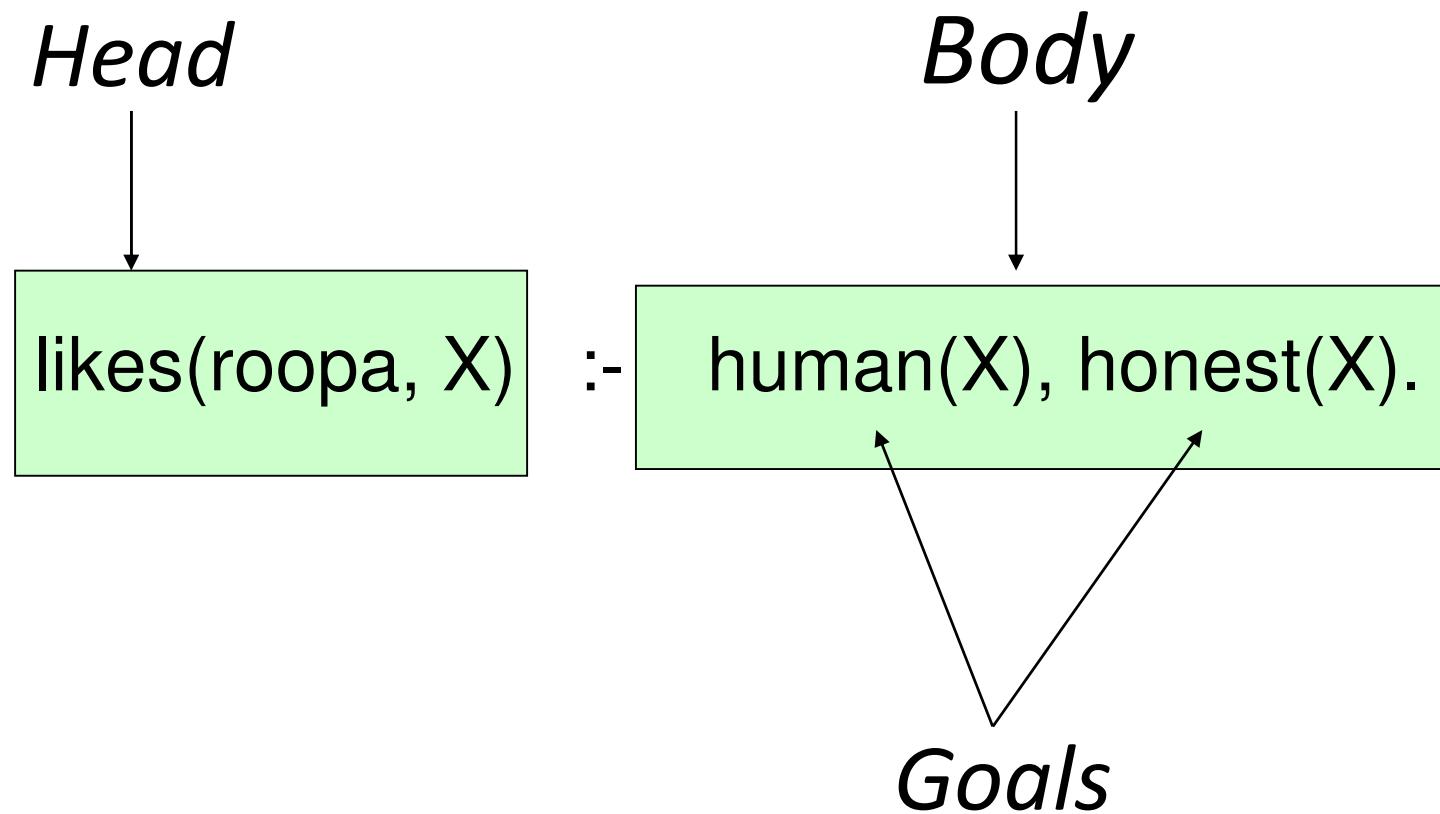


Clauses: Facts and Rules



Full stop at the end.

Body of a (rule) clause contains goals.



Interpretation of Clauses

Clauses can be given a declarative reading or a procedural reading.

Form of clause:

HORN Clause

$$H \ :- \ G_1, \ G_2, \ \dots, \ G_n.$$

Declarative reading:

“That H is provable follows from goals G_1, G_2, \dots, G_n being provable.”

Procedural reading:

“To execute procedure H , the procedures called by goals G_1, G_2, \dots, G_n are executed first.”

Another Example

Program

```
male(rohan).  
male(vivek).  
female(purbi).  
female(neha).  
pair(X, Y) :- male(X),  
             female(Y).
```

Queries

```
?- pair(vivek, X).  
?- pair(ram, sita).  
?- pair(neha, X).  
?- pair(X, purbi).  
?- pair(X, X).  
?- pair(rohan, purbi).  
?- pair(X, dhanush).  
?- pair(X, Y).
```

```
?- consult('test1.pl').
```

```
?-pair(vivek, X).
```

//use ; get next option

Example 2

```
drinks(raj, tea).  
drinks(mann, coffee).  
drinks(sagar, juice).  
drinks(raj, coffee).  
drinks(kush, coffee).
```

```
pair(X, Y, Z) :-  
    drinks(X, Z),  
    drinks(Y, Z).
```

```
?- pair(X, raj, tea).  
?- pair(mann, sagar, coffee).  
?- pair(raj, mann, coffee).  
?- pair(raj, raj, coffee).  
?- pair(X, Y, coffee).  
?- pair(sita, gouri, juice).  
?- pair(X, Y, Z).
```

This definition forces X and Y to be

```
pair(X, Y, Z) :- drinks(X, Z), drinks(Y, Z), X \== Y.
```

Another Examples: Density Calculation

```
%popultaion.pl  Population in Million  
pop(usa,280).      pop(india,1000).  
pop(china,1200).   pop(brazil,130).  
area(usa,3). /* millions of sq miles */  
area(india,1). area(china,4).  
area(brazil,3).
```

```
density(X,Y) :- pop(X,P),  
               area(X,A),  
               Y is P/A.
```

The population density of country X is Y, if:

The population of X is P, **and**

The area of X is A, **and**

Y is calculated by dividing P by A.

Examples: Density Calculation

```
?- consult(population.pl).  
% population compiled 0.00 sec,  
1,548 bytes
```

Yes

```
?- density(usa,D).
```

```
D = 93.3333
```

Yes

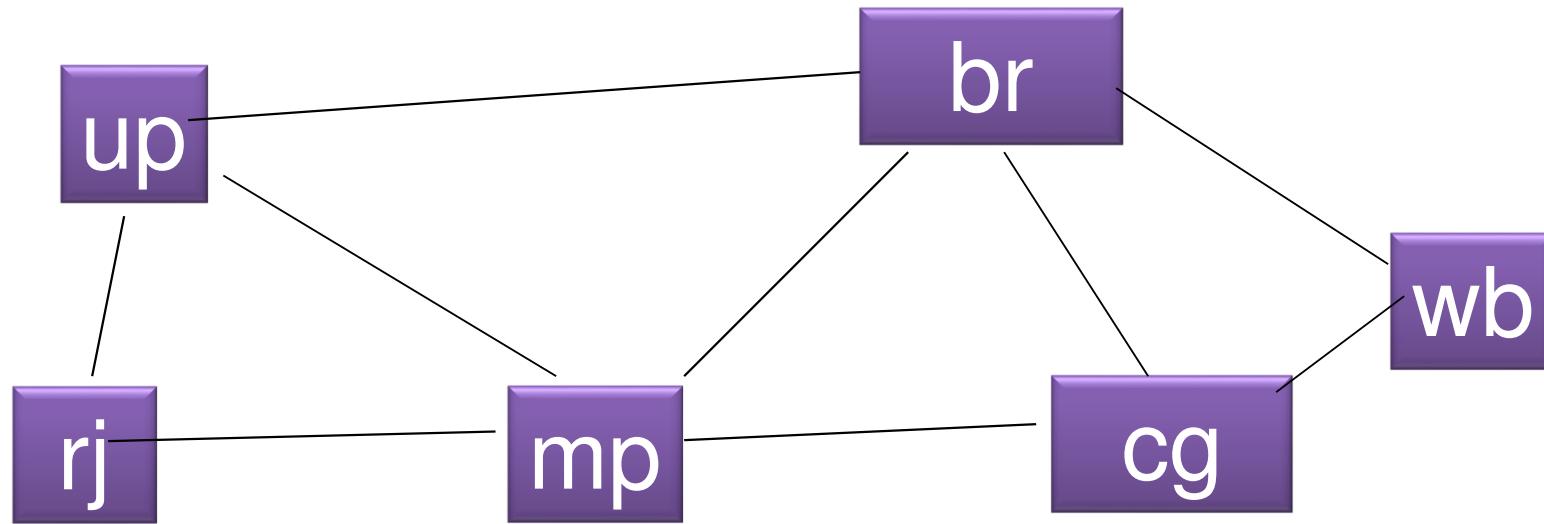
```
?- density(china,D).
```

```
D = 300
```

Yes

Example 3: Boarder of Indian States

- (a) Representing a symmetric relation.
- (b) Implementing a strange ticket condition.



How to represent this relation?
Note that borders are symmetric.

Example 3: Boarder of India States

This relation represents one ‘direction’ of border:

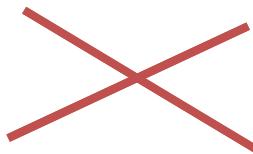
```
border(cg, wb).  
border(cg, br).  
border(br, wb).  
border(mp, cg).  
border(mp, br).  
border(mp, up).  
border(up, br).  
border(rj, mp).  
border(rj, up).
```

What about the other?

(a) Say `border(wb, cg)` .
`border(cg, wb)` .
⋮

(b) Say
`adjacent(X, Y) :- border(X, Y)` .
`adjacent(X, Y) :- border(Y, X)` .

(c) Say
`border(X, Y) :- border(Y, X)` .



Example 3: Boarder of India States

Now a somewhat strange type of discount ticket. For the ticket to be valid, one must pass through an intermediate state.

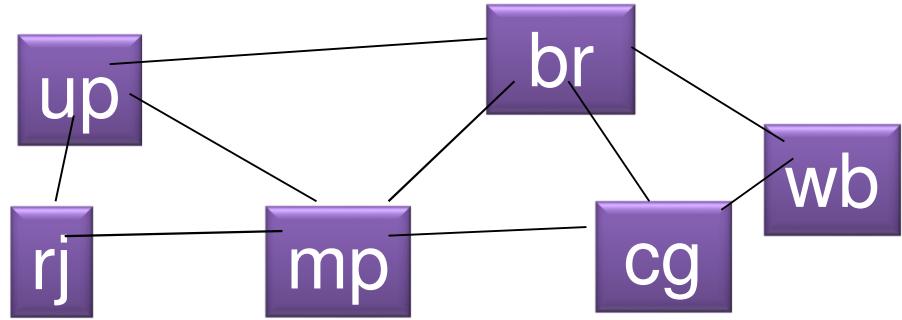
A valid ticket between a start and end state obeys the following rule:

```
valid(X, Y) :- adjacent(X, Z), adjacent(Z, Y)
```

Example 3: Boarder of India States

```
border(CG, WB).  
border(CG, BR).  
border(BR, WB).  
border(MP, CG).  
border(MP, BR).  
border(MP, UP).  
border(UP, BR).  
border(RJ, MP).  
border(RJ, UP).
```

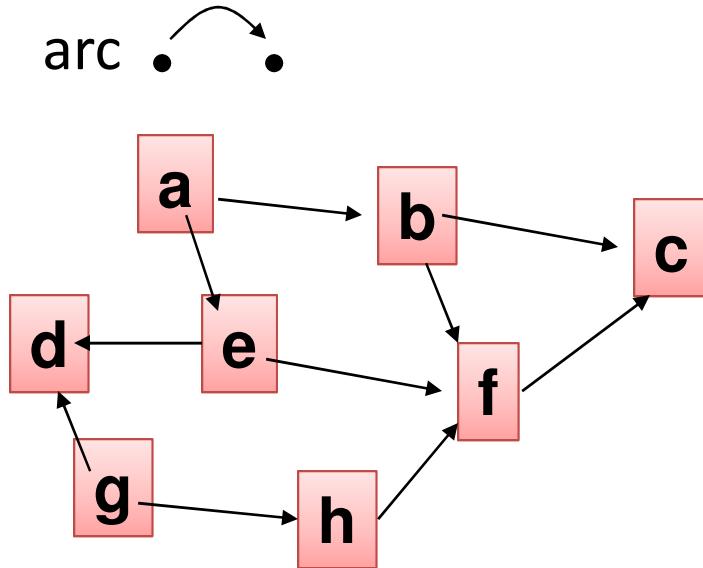
```
adjacent(X, Y) :- border(X, Y).  
adjacent(X, Y) :- border(Y, X).
```



```
valid(X, Y) :-  
    adjacent(X, Z),  
    adjacent(Z, Y)
```

```
?- valid(rj, CG).  
?- valid(rj, WB).  
?- valid(MP, MP).  
?- valid(X, WB).  
?- valid(CG, X).  
?- valid(X, Y).
```

Graph Example

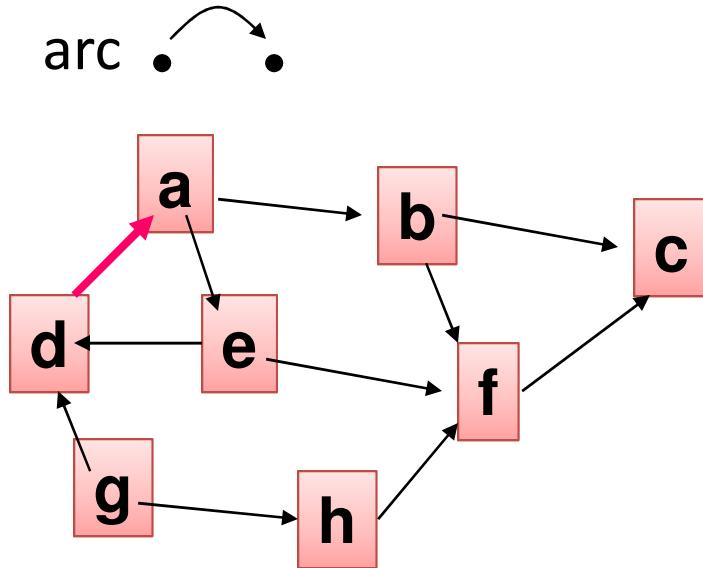


$a(g, h)$. $a(g, d)$. $a(e, d)$.
 $a(h, f)$. $a(e, f)$. $a(a, e)$.
 $a(a, b)$. $a(b, f)$. $a(b, c)$. $a(f, c)$.
 $\text{path}(X, X)$.
 $\text{path}(X, Y) :- a(X, Z), \text{path}(Z, Y)$.

Prolog can distinguish between the 0-ary constant a (the name of a node) and the 2-ary functor a (the name of a relation).

```
?- path(f, f).  
?- path(a, c).  
?- path(g, e).  
?- path(g, X).  
?- path(X, h).
```

But what happens if...



arc(g, h). arc(g, d). arc(e, d).
arc(h, f). arc(e, f). arc(a, e).
arc(a, b). arc(b, f). arc(b, c).
arc(f, c).

arc(d, a).

path(X, X).

path(X, Y) :- arc(X, Z), path(Z, Y).

This program works only for acyclic graphs.
The program may infinitely loop given a cyclic graph.

We need to leave a ‘trail’ of visited nodes
== > (to be seen later).

Unification in Prolog

Unification

- Two terms unify
 - if substitutions can be made for any variables in the terms so that the terms are made identical.
 - If no such substitution exists, the terms do not unify.

Unification Examples

| ?- X=1+2 .

X = 1+2

yes

| ?- f(g(Y))=f(X) .

X = g(Y)

yes

| ?- X=f(Y) .

X = f(Y)

yes

Unification

- The Unification Algorithm proceeds by recursive descent of the two terms.
 - Constants unify if they are identical
 - Variables unify with any term, including other variables
 - Compound terms unify if their functors and components unify.

Unification Examples: 1

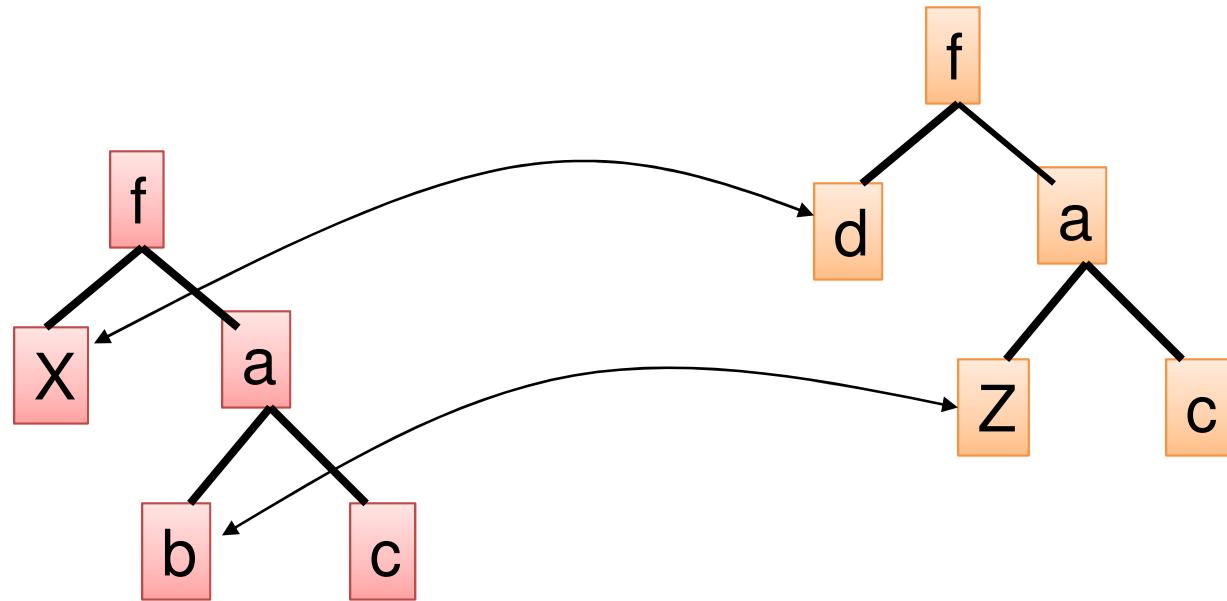
The terms $f(X, a(b, c))$ and $f(d, a(Z, c))$ unify.

| ?- $f(X, a(b, c)) = f(d, a(Z, c)).$

$X = d$

$Z = b$

yes



The terms are made equal if d is substituted for X , and b is substituted for Z .

We also say X is instantiated to d and Z is instantiated to b , or $X/d, Z/b$.

Unification : Examples 2

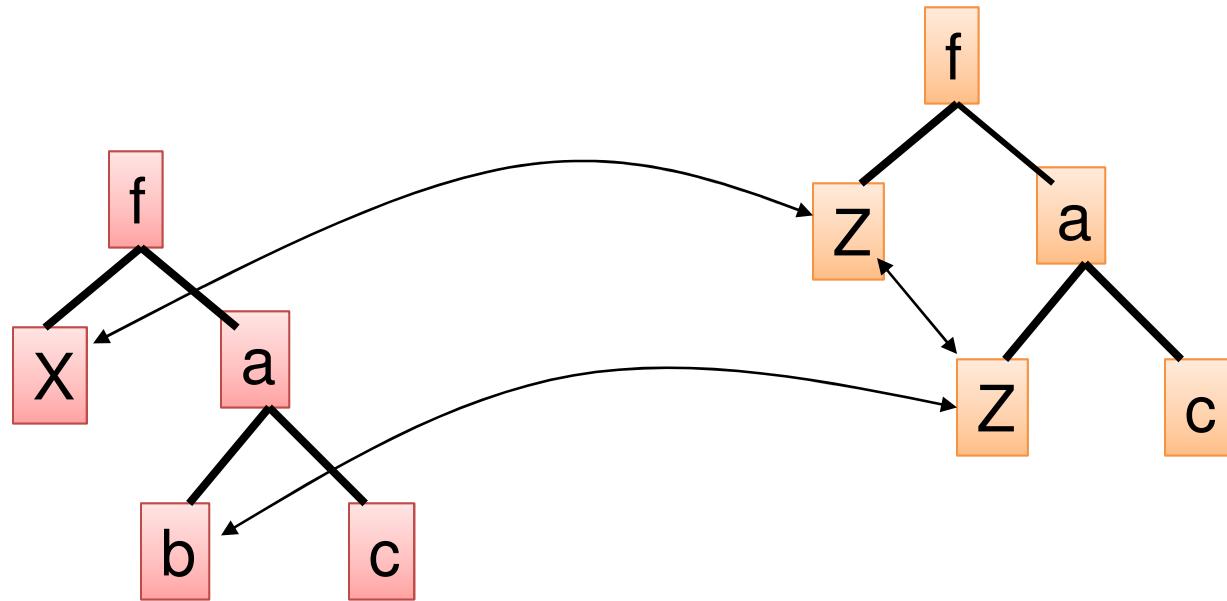
The terms $f(X, a(b,c))$ and $f(Z, a(Z, c))$ unify.

| ?- $f(X, a(b,c)) = f(Z, a(Z, c))$.

$X = b$

$Z = b$

yes



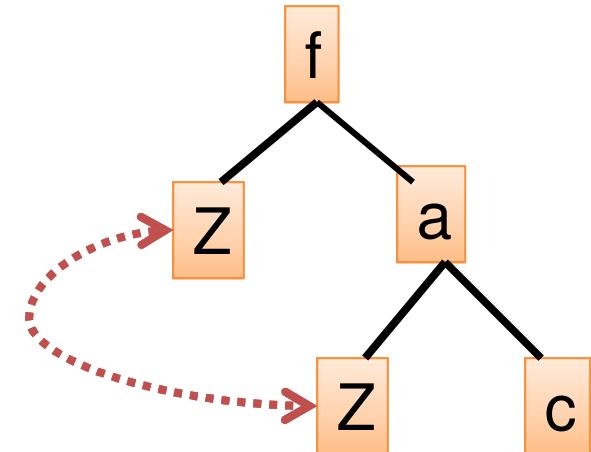
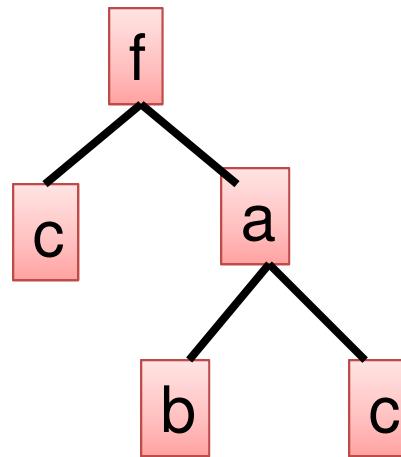
Note that Z co-references within the term.
Here, X/b, Z/b.

Unification : Examples 3

The terms $f(c, a(b, c))$ and $f(Z, a(Z, c))$ do not unify.

| ?- $f(c, a(b, c)) = f(Z, a(Z, c))$.

no

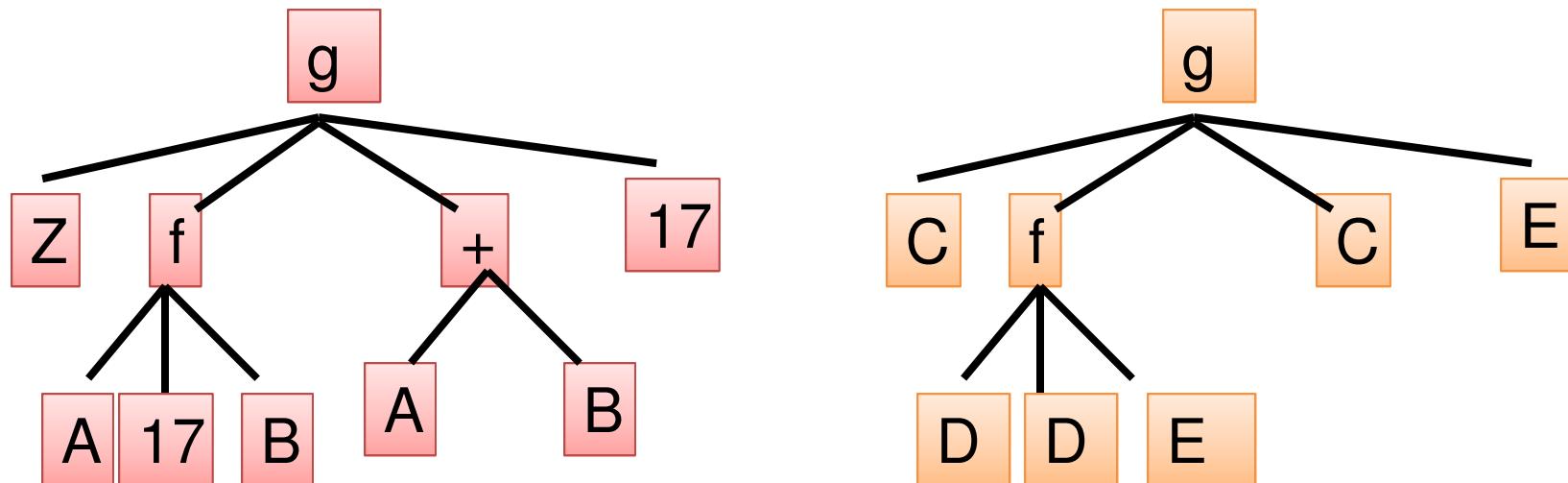


No matter how hard you try, these two terms cannot be made identical by substituting terms for variables.

Unification: Big Example

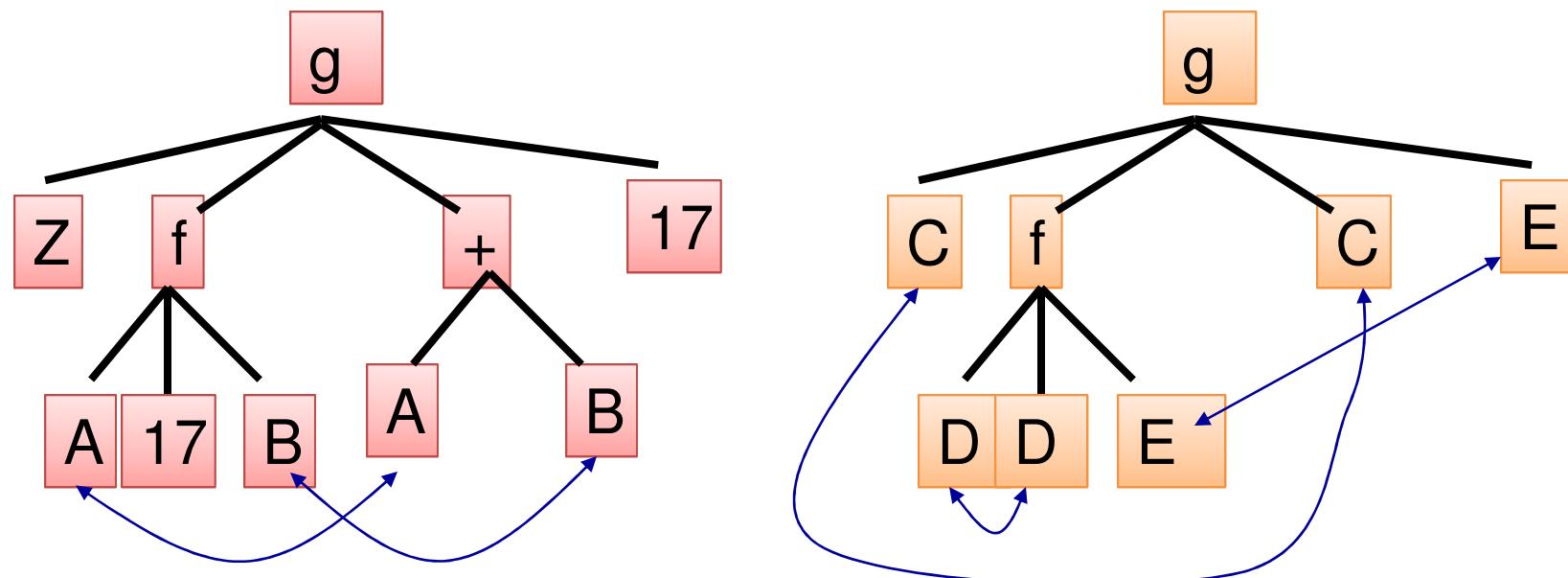
Do terms $g(Z, f(A, 17, B), A+B, 17)$ and $g(C, f(D, D, E), C, E)$ unify?

```
| ?- g(Z, f(A, 17, B), A+B, 17) = g(C, f(D, D, E), C, E).  
A = 17    B = 17    C = 17+17  D = 17  E = 17  Z = 17+17  
yes
```



Unification: Big Example

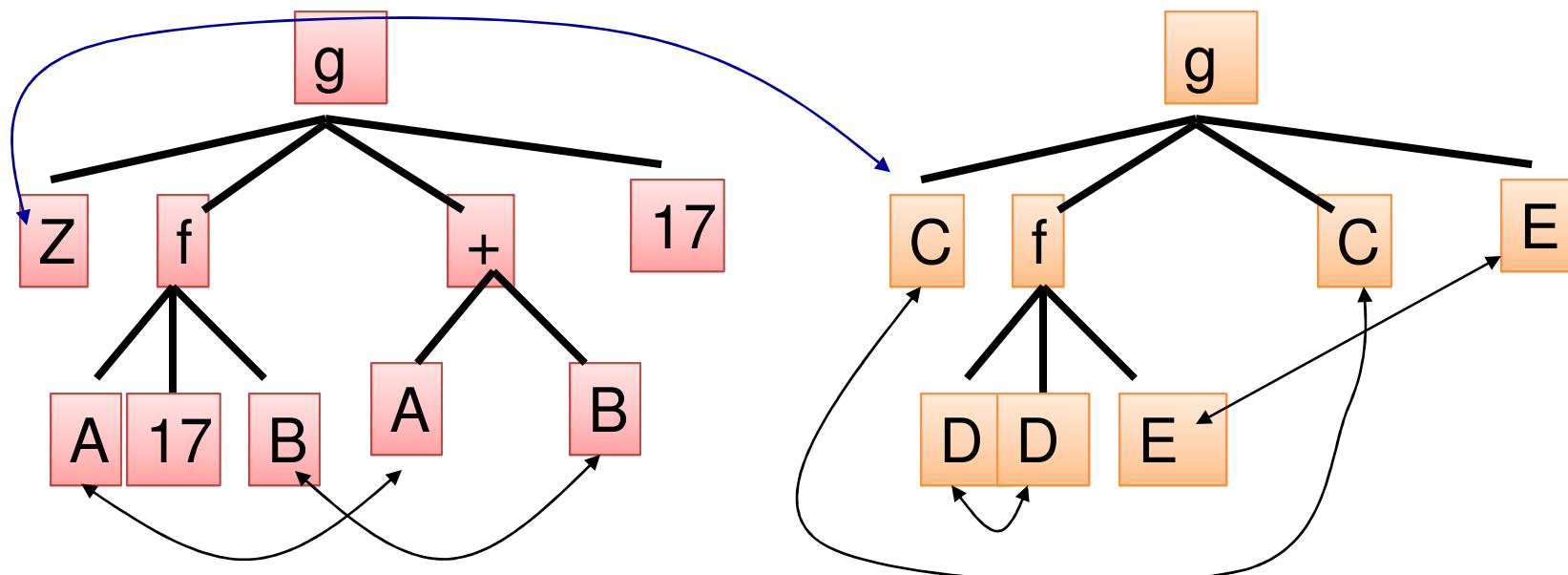
First write in the co-referring variables.



Unification: Big Example

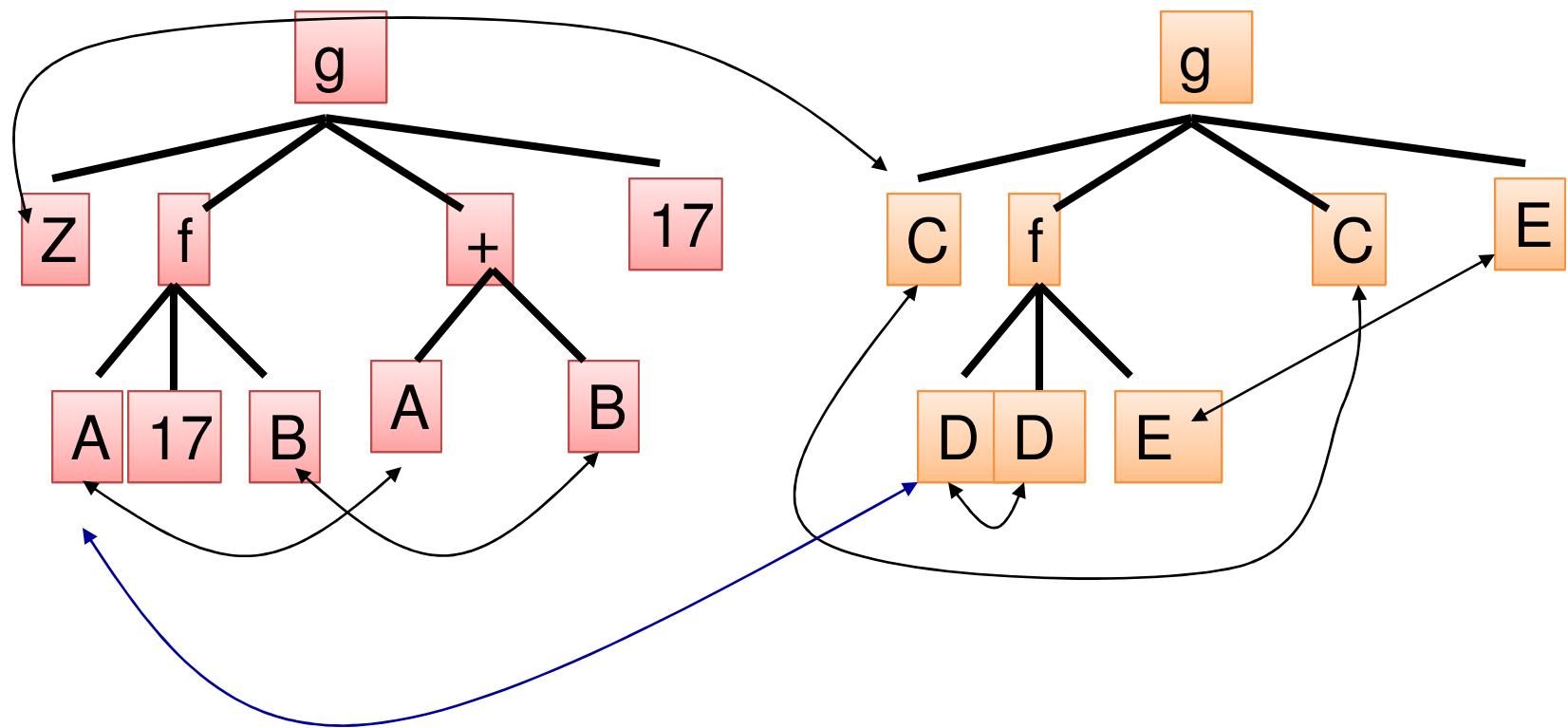
Z/C, C/Z

Now proceed by recursive descent We go top-down, left-to-right, but the order does not matter as long as it is systematic and complete.



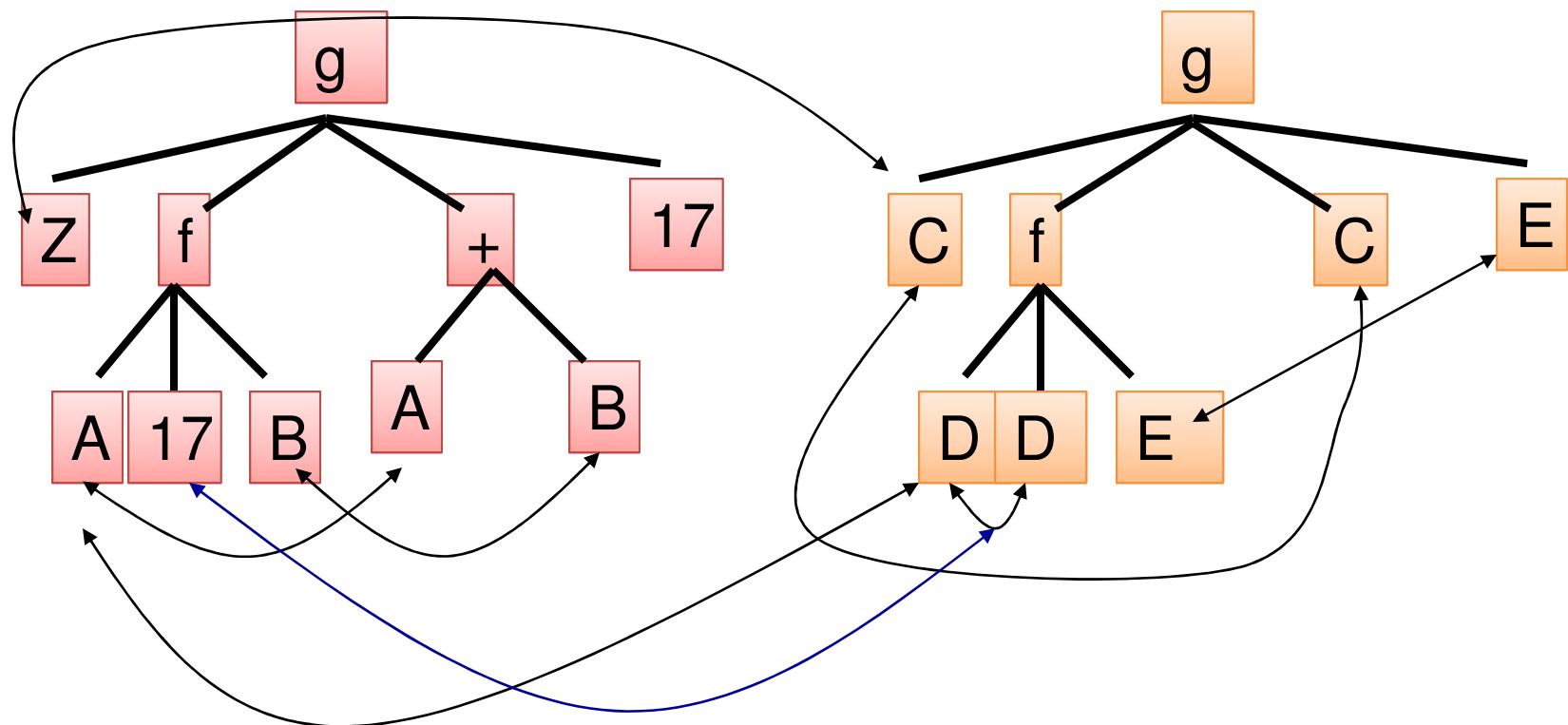
Unification: Big Example

Z/C, C/Z, A/D, D/A



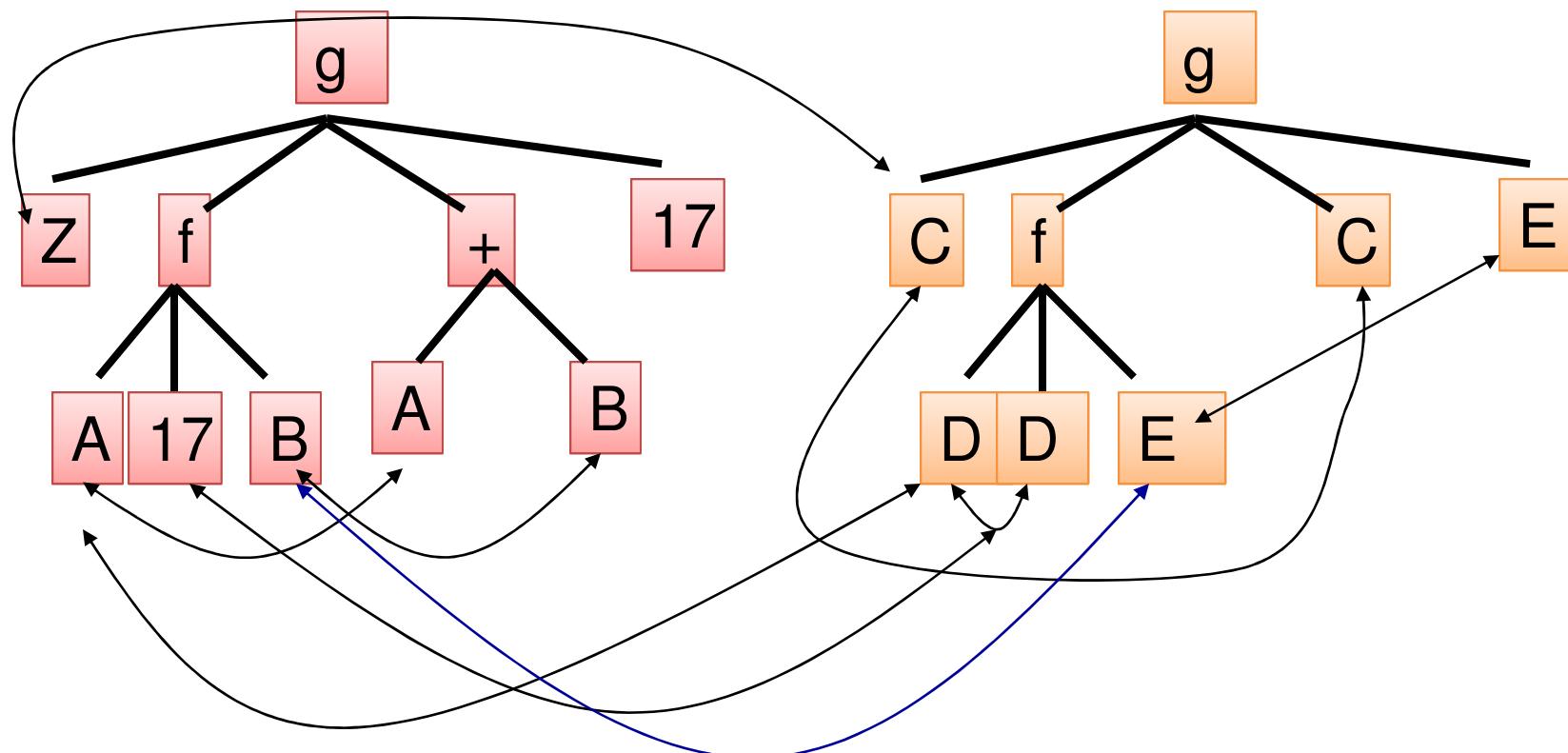
Unification: Big Example

Z/C, C/Z, A/17, D/17



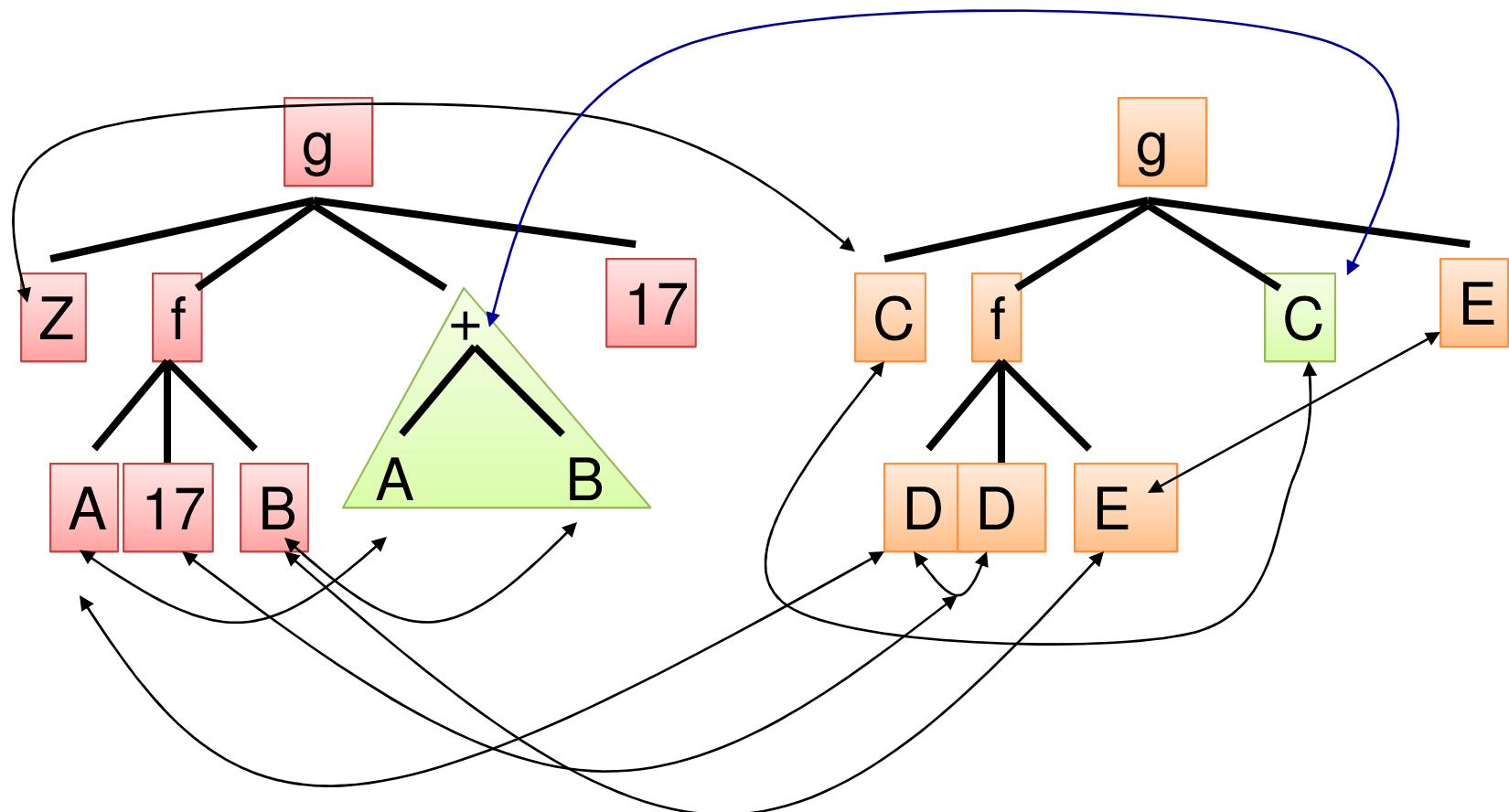
Unification: Big Example

Z/C, C/Z, A/17, D/17, B/E, E/B



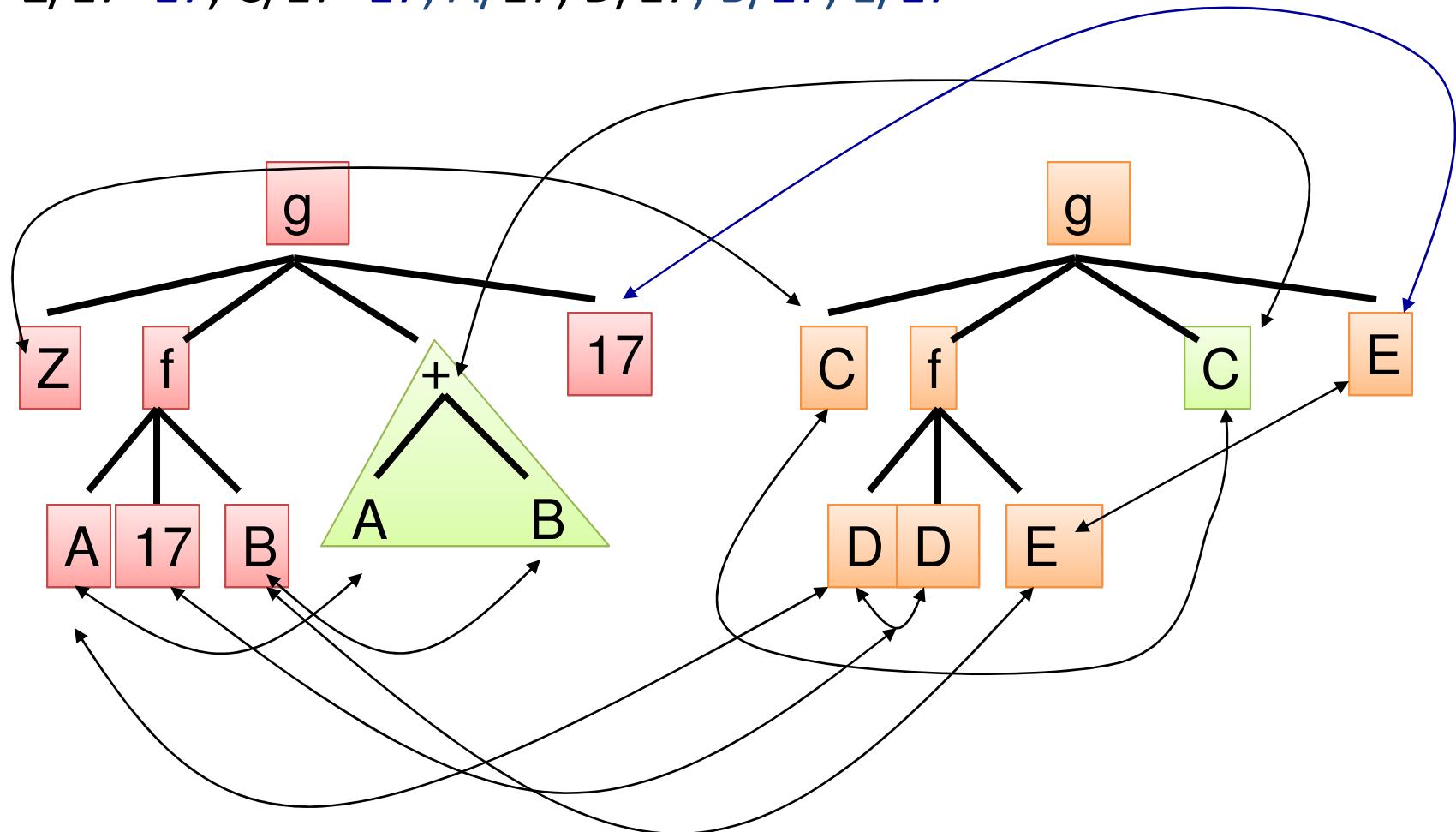
Unification: Big Example

Z/17+B, C/17+B, A/17, D/17, B/E, E/B



Unification: Big Example

Z/17+17, C/17+17, A/17, D/17, B/17, E/17



A Brief Diversion into Arithmetic

A Brief Diversion into Arithmetic

- The built-in predicate ‘**is**’ takes two arguments.
- It interprets its second as an arithmetic expression, and unifies it with the first.
- Also, ‘**is**’ is an infix operator.

```
?- x is 2 + 2 * 2.
```

```
X = 6
```

```
?- 10 is (2 * 0) + 2 << 4.
```

```
no
```

```
?- 32 is (2 * 0) + 2 << 4.
```

```
yes
```

is

is

- But ‘is’ cannot solve equations
- so the expression must be ‘ground’
(contain no free variables).

```
?- Y is 2 * X.
```

no

```
?- X is 15, Y is 2 * X.
```

```
X = 15, Y= 30.
```

A Brief Diversion into Anonymous Variables

```
/* member(Term, List) */
```

```
member(X, [X|T]).           Notice T isn't 'used'
```

```
member(X, [H|T]) :- member(X, T).
```

Notice H isn't 'used'

```
member(X, [X|_]).
```



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



Length of a List

```
/* length(List, Length) */
```

Naïve method:

```
length([], 0).
```

```
length([H|T], N) :- length(T, NT),  
N is NT+1.
```

Length of a List

```
/* length(List, Length) */
```

Tail-recursive method:

```
length(L, N) :- acc(L, 0, N).
```

```
/* acc(List, Before, After) */
```

```
acc([], A, A).
```

```
acc([H|T], A, N) :- A1 is A+1,  
                  acc(T, A1, N).
```

Sum of List and Sum of Square list

```
Lsum( [ ] , 0 ) .
```

```
Lsum( [H | T] , TSum) :-
```

```
    Lsum( T , Sum1 ) , Tsum is H+ Sum1 .
```

```
Lsqsum ( [ ] , 0 ) .
```

```
Lsqsum( [H | T] , TSqSum) :-
```

```
    Lsqsum( T , Sum1 ) , TSqSum is H*H +  
    Sum1 .
```

Length of a List

?– **length**([apple, pear], N) .

N=2

?– **length**([alpha], 2) .

no

?– **length**(L, 3) .

L = [_, _, _]

Yes

List: Inner Product

A list of n integers can be used to represent an n -vector (a point in n -dimensional space). Given two vectors \mathbf{a} and \mathbf{b} , the inner product (dot product) of \mathbf{a} and \mathbf{b} is defined

$$\overrightarrow{a} \cdot \overrightarrow{b} = \sum_{i=1}^n a_i b_i$$

As you might expect, there are naïve and tail-recursive ways to compute this.

List Inner Product : Naive

```
inner([], [], 0) .
```

```
inner([A|As], [B|Bs], N) :-  
    inner(As, Bs, Ns),  
    N is Ns + (A * B) .
```

List Inner Product: Tail recursive

```
inner(A, B, N) :- dotaux(A, B, 0, N) .
```

```
dotaux([], [], V, V) .
```

```
dotaux([A|As], [B|Bs], N, Z) :-
```

```
    N1 is N + (A * B) ,
```

```
    dotaux(As, Bs, N1, Z) .
```

Mutual recursion

```
even([]).  
even([B|C]) :- odd(C).  
odd([A|C]) :- even(C).
```

Peter Norvig: Any mutual recursion can be converted to direct recursion using procedural inlining... *some time it find difficult....*

```
even([]).  
even([A,B|C]) :- even(C).  
odd([A]).  
odd([A,B|C]) :- odd(C).
```

Mutual Recursion: conversion to direct recursion

Mapping: The Full Map

List of numbers

sqlist (, **)** *List of squares
of numbers*

sqlist ([] , []) .

sqlist ([X|T] , [Y|L]) :-

Y is X * X,

sqlist (T , L) .

Mapping: The Full Map

Map each list element (a number) to a term $s(A,B)$ where A is the number and B is its square.

Input: [1, 9, 15]

Output: [s(1,1), s(9, 81), s(15, 225)]

sqterm([], []).

sqterm([X|T], [s(X,Y)|L]) :-
 Y is X * X,
 sqterm(T, L).

General Scheme for Full Map

```
/* fullmap(In, Out) */

fullmap( [ ] , [ ] ) .

fullmap( [ X | T ] , [ Y | L ] ) :-  
    transform( X, Y ) ,  
    fullmap( T, L ) .
```

Simple Transformation

/*Here is a typical transformation table...*/

```
transform(cat, gatto).  
transform(dog, cane).  
transform(hamster, criceto).  
transform(X, X) .
```

A '*catchall*' rule

```
?- fullmap([cat, dog, goat], Z).  
Z = [gatto, cane, goat]
```

Multiple Choices in Prolog

Multiple Choices in Prolog

Sometimes the map needs to be sensitive to the input data:

Input: [1, 3, w, 5, goat]

Output: [1, 9, w, 25, goat]

*Just use a separate clause
for each choice*

```
squint( [], [] ).
```

```
squint( [X|T], [Y|L] ) :-  
    integer(X),  
    Y is X * X, squint(T, L).
```

```
squint( [X|T], [X|L] ) :-  
    squint(T, L).
```

Clause for
Integer data

Clause for
other data

Multiple Choices

Using the infix binary compound term `*`, it is easy enough to give some mathematical reality to the map:

Input: [1, 3, w, 5, goat]

Output: [1, 9, w*w, 25, goat*goat]

Just use a separate clause for each choice

```
squint ( [ ] , [ ] ) .
```

```
squint ( [X|T] , [Y|L] ) :-
```

```
    integer(X) ,
```

```
    Y is X * X, squint(T, L) .
```

```
squint ( [X|T] , [X*X|L] ) :-
```

```
    squint(T, L) .
```

Clause for
Integer data

Clause for
other data

Partial Maps

Given an input list, partially map it to an output list.

```
evens([], []).
```

```
evens([X|T], [X|L]) :-
```

0 is X mod 2,

```
    evens(T, L)
```

```
evens([X|T], L) :-
```

1 is X mod 2,

```
    evens(T, L).
```

```
?- evens([1, 2, 3, 4, 5, 6], Q).
```

```
Q = [2, 4, 6].
```

General Scheme for Partial Maps

```
partial([], []).  
partial([X|T], [X|L]) :-  
    include(X), partial(T, L)  
partial([X|T], L) :- partial(T, L).
```

For example,

```
include(X) :- X >= 0.
```

```
?- partial([-1, 0, 1, -2, 2], X),  
X = [0, 1, 2].
```

Backtracking and Non-determinism

Backtracking and Non-determinism

```
member(X, [X|_]).
```

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

```
?- member(fred, [john, fred, paul, fred]).
```

Yes

Deterministic query

```
?- member(X, [john, fred, paul, fred]).
```

X = john;

X = fred;

X = paul;

X = fred

no

Nondeterministic query

Answer no says:

All the solution displayed, no more solution

The problem of controlling backtracking

```
colour(cherry, red) .  
colour(banana, yellow) .  
colour(apple, red) .  
colour(apple, green) .  
colour(orange, orange) .  
colour(X, unknown) .
```

Generic one
True for every one
If match earlier, we
should skip this

```
?- colour(banana, X) .
```

X = *yellow*

```
?- colour(physalis,  
X) .
```

X = *unknown*

```
?- colour(cherry, X) .
```

X = *red*;

X = *unknown*;

no

(Pressed semicolon)
Is there any other
solution ?

The cut: in Prolog

- A built-in predicate, used not for its logical properties, but for its effect. Gives control over backtracking.
- The cut is spelled: !
- The cut always succeeds.
- When backtracking over a cut, the goal that caused the current procedure to be used fails.

How it works

- Suppose that goal H has two clauses :

$H_1 :- B_1, B_2, \dots, B_i, !, B_k, \dots, B_m.$

$H_2 :- B_n, \dots B_p.$

- If H_1 matches, goals $B_1 \dots B_i$ may backtrack among themselves.
- If B_1 fails, H_2 will be attempted. **But as soon as the ‘cut’ is crossed, Prolog commits to the current choice. All other choices are discarded.**

Commitment to the clause

Goals $B_k \dots B_m$ may backtrack amongst themselves, but if goal B_k fails, then the predicate fails (the subsequent clauses are not matched).

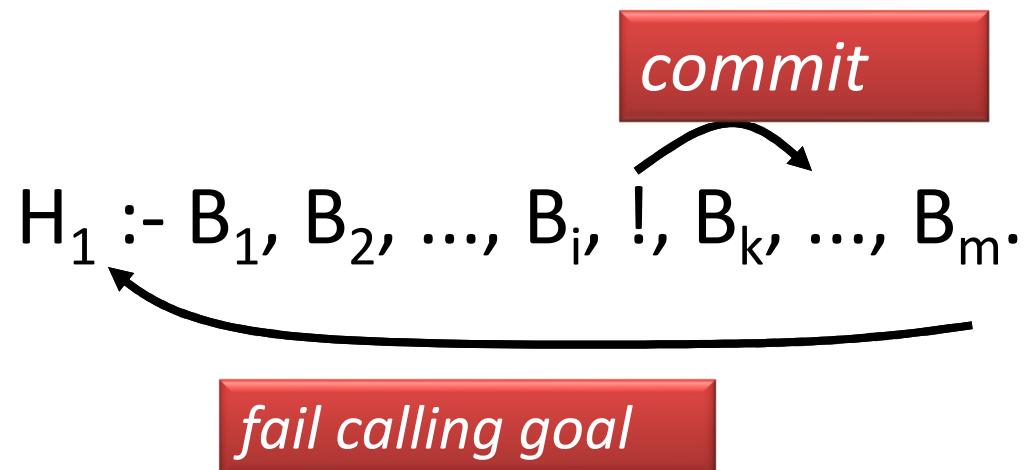
$H_1 :- B_1, B_2, \dots, B_i, !, B_k, \dots, B_m.$

$H_2 :- B_n, \dots B_p.$

How to remember it

Think of the ‘cut’ as a ‘fence’ which, when crossed as a success, asserts a commitment to the current solution.

However, when a failure tries to cross the fence, the entire goal is failed.



Cut :Use 1

To define a deterministic (functional) predicate.

A deterministic version of member, which is more efficient for doing ‘member checking’ because it needn’t give multiple solutions:

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

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

```
?- membercheck (fred, [joe, john, fred, paul]) .
```

yes.

```
?- membercheck (X, [a, b, c]) .
```

X = a;

no.

Answer no says:

a is solution, but there are other solution too

Cut : Use 2

To specify exclusion of cases by ‘committing’ to the current choice.

The goal **max (X, Y, Z)** instantiates Z to the greater of X and Y:

```
max(X, Y, X) :- X >= Y.  
max(X, Y, Y) :- X < Y.
```

Note that each clause is a logically correct statement about maxima. A version using cut can get rid of the second test, and might look like this:

```
max(X, Y, X) :- X >= Y, !.  
max(X, Y, Y) .
```

Max with cut

```
max(X, Y, X) :- X >= Y, !.  
max(X, Y, Y).
```

- If max is called with $X \geq Y$, the first clause will succeed, and the cut assures that the second clause is never made.
- The advantage is that the test does not have to be made twice if $X < Y$.

The disadvantage is that each rule does not stand on its own as a logically correct statement about the predicate. To see why this is unwise, try

```
?- max(10, 0, 0).
```

Max with cut

So, it is better to use the cut and both tests will give a program that backtracks correctly as well as trims unnecessary choices.

```
max(X, Y, X) :- X >= Y, !.  
max(X, Y, Y) :- X < Y.
```

Or, if your clause order might suddenly change (because of automatic program rewriting), you might need:

```
max(X, Y, X) :- X >= Y, !.  
max(X, Y, Y) :- X < Y, !.
```

Cut: One more example

rem_dups, without cut:

```
rem_dups( [ ] , [ ] ) .  
rem_dups( [ F | Rest ] , NRest ) :-  
    member( F , Rest ) ,  
    rem_dups( Rest , NRest ) .  
rem_dups( [ F | Rest ] , [ F | NRest ] ) :-  
    not( member( F , Rest ) ) ,  
    rem_dups( Rest , NRest ) .
```

Cut: One more example

rem_dups, with **cut**:

```
rem_dups( [ ] , [ ] ) .  
rem_dups( [ F | Rest ] , NRest ) :-  
    member( F , Rest ) , ! ,  
    rem_dups( Rest , NRest ) .  
rem_dups( [ F | Rest ] , [ F | NRest ] ) :-  
    rem_dups( Rest , NewRest ) .
```

Thanks