Prolog Tutorial-2

Dr A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline

- *Terms as data structures, unification*
- Is
- List
- The Cut
- Assignment 3 uploaded....

Prolog Relational Database: Example

[a, b, c]= [a|[b,c]] = [Head is symbol | Tail is list]

Relation **append** is a set of tuples of the form (X,Y,Z) where Z consist if X followed by the element of Y.

X	Y	Z
[]	[]	[]
[a]	[]	[a]
...	...	....
[a,b]	[c,d]	[a,b,c,d]
.....	......	....

Relation are also called ***predicates.***

Query : Is a given tuple in relation append?

```
?-append([a],[b],[a,b]).
yes
```

```
?-append([a],[b],[]).
no
```

Writing append relation in prolog

Rules

```
append([],Y,Y).
append([H|X],Y,[H|Z]):-
                      append(X,Y,Z).
```

Queries

```
?-append ([a,b],[c,d],[a,b,c,d]).
 yes
?-append([a,b],[c,d],Z).
Z=[a,b,c,d]
?-append([a,b],Y,[a,b,c,d]).
Y=[c,d]
```

Prolog Program Structure

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

panda(chi_chi).
panda(ming_ming).
```

Facts

Rules

```
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).
```

Factorial Program

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

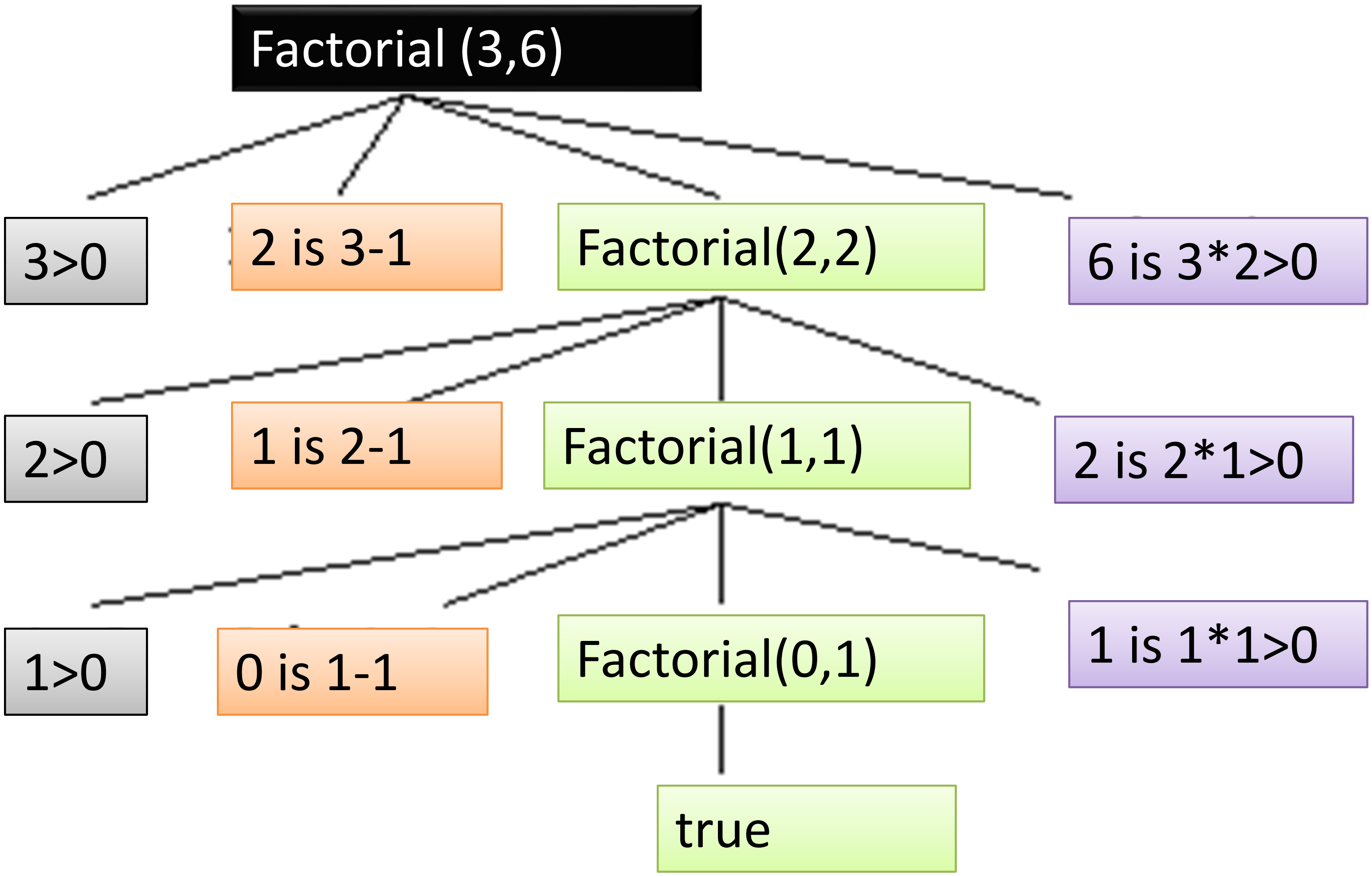
The Prolog goal to calculate the factorial of the number 3 responds with a value for W, the goal variable:

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

W=6

Factorial Program Evaluation

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



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.
- 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

```
| ?- X=1+2.
    X = 1+2
    yes
| ?- f(g(Y))=f(X).
  X = g(Y)
  yes
| ?- X=f(Y).
  X = f(Y)
     yes
```

Unification Examples: 1

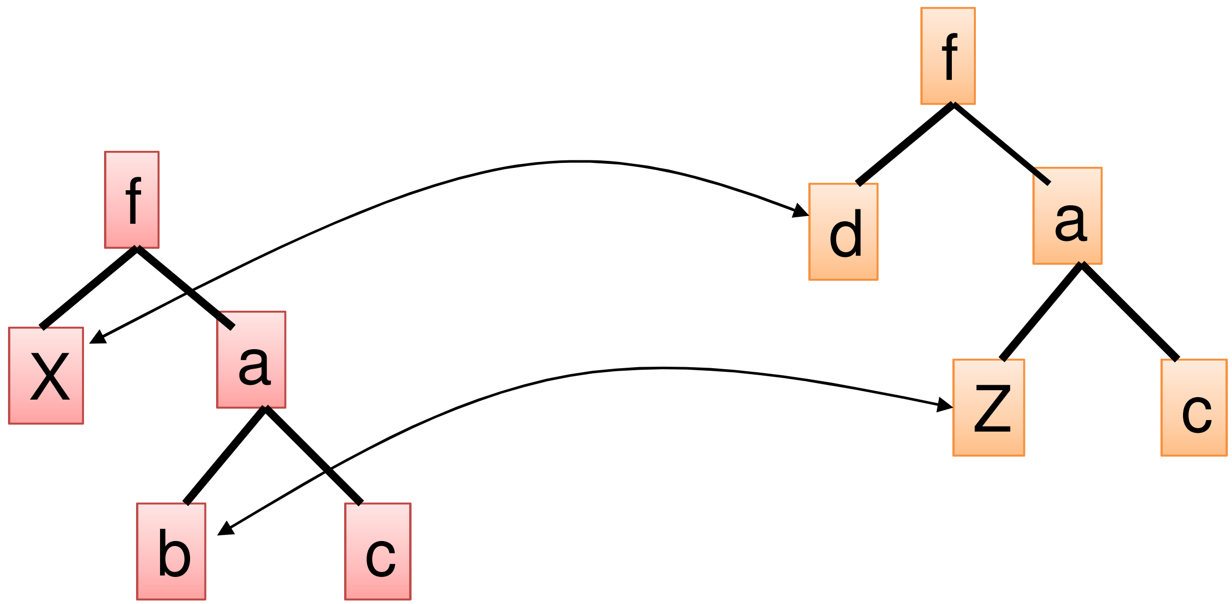
The terms f(X, a(b,c)) and f(d, a(Z, c)) unify.

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

X = d

Z = b

yes

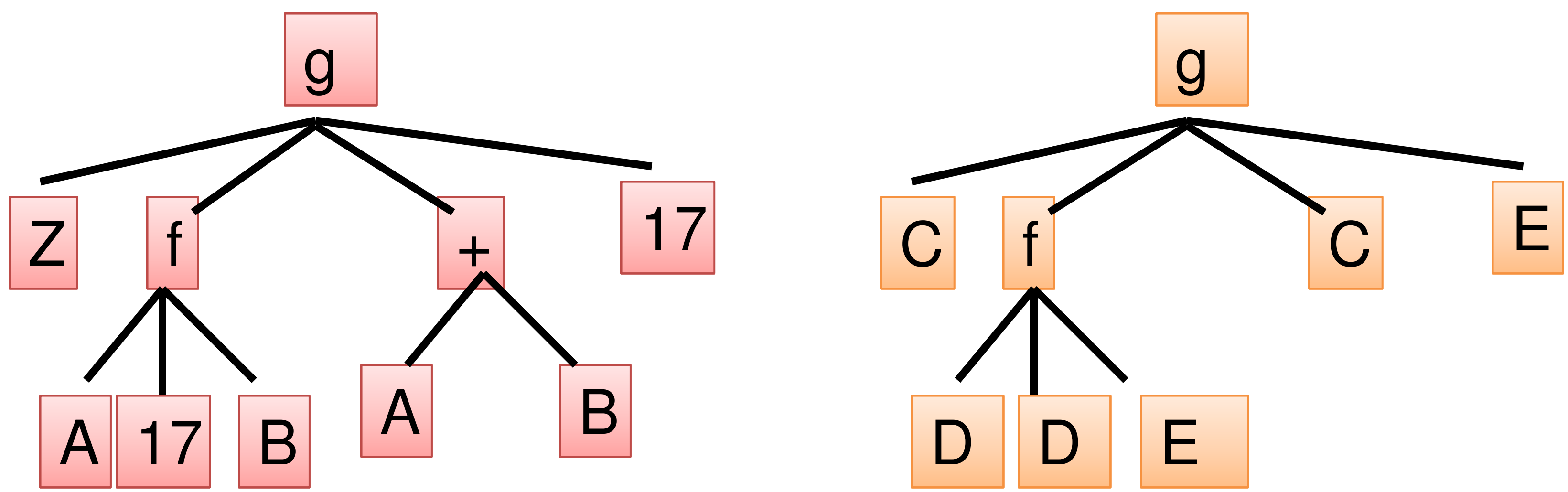


The terms are made equal if d is substituted for X, and b is substituted for Z.
We also say X is instantiated to d and Z is instantiated to b, or X/d, Z/b.

Unification: 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
```



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

Notice T isn't 'used'

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 **a** and **b**, the inner product (dot product) of **a** and **b** is defined

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

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|C]):-  odd(C).
```

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

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

Just use a separate clause for each choice

Input: [1, 3, w, 5, goat]
Output: [1, 9, w, 25, goat]

```
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

```
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 two clauses :

$$H_1 \text{ :- } B_1, B_2, \ldots, B_i, !, B_k, \ldots, B_m.$$

$$H_2 \text{ :- } B_n, \ldots B_p.$$

– If H_1 matches, goals $B_1 \ldots B_i$ 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 $B_k \ldots B_m$ may backtrack amongst themselves, but if goal B_k fails, then the predicate fails (the subsequent clauses are not matched).

$$H_1 \text{ :- } B_1,\ B_2,\ \ldots,\ B_i,\ \ !,\ \ B_k,\ \ldots,\ B_m.$$

$$H_2 \text{ :- } B_n,\ \ldots\ B_p.$$

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.

commit

$$H_1 \text{ :- } B_1, B_2, \ldots, B_i, \,!, B_k, \ldots, B_m.$$

fail calling goal

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 >= 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 using 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