

# Prolog

CSE 595 – Semantic Web

Instructor: Dr. Paul Fodor

Stony Brook University

<http://www3.cs.stonybrook.edu/~pfodor/courses/cse595.html>

# Relations/Predicates

- Predicates are building-blocks in predicate calculus:  $p(a_1, a_2, \dots, a_k)$

- **parent**(X, Y) : X is a parent of Y.

**parent**(pam, bob) . **parent**(bob, ann) .

**parent**(tom, bob) . **parent**(bob, pat) .

**parent**(tom, liz) . **parent**(pat, jim) .

- **male**(X) : X is a male.

**male**(tom) .

**male**(bob) .

**male**(jim) .

We attach meaning to them, but within the logical system they are simply structural building blocks, with no meaning beyond that provided by explicitly-stated interrelationships

# Relations

- **female (X) : X is a female.**

**female (pam) .**

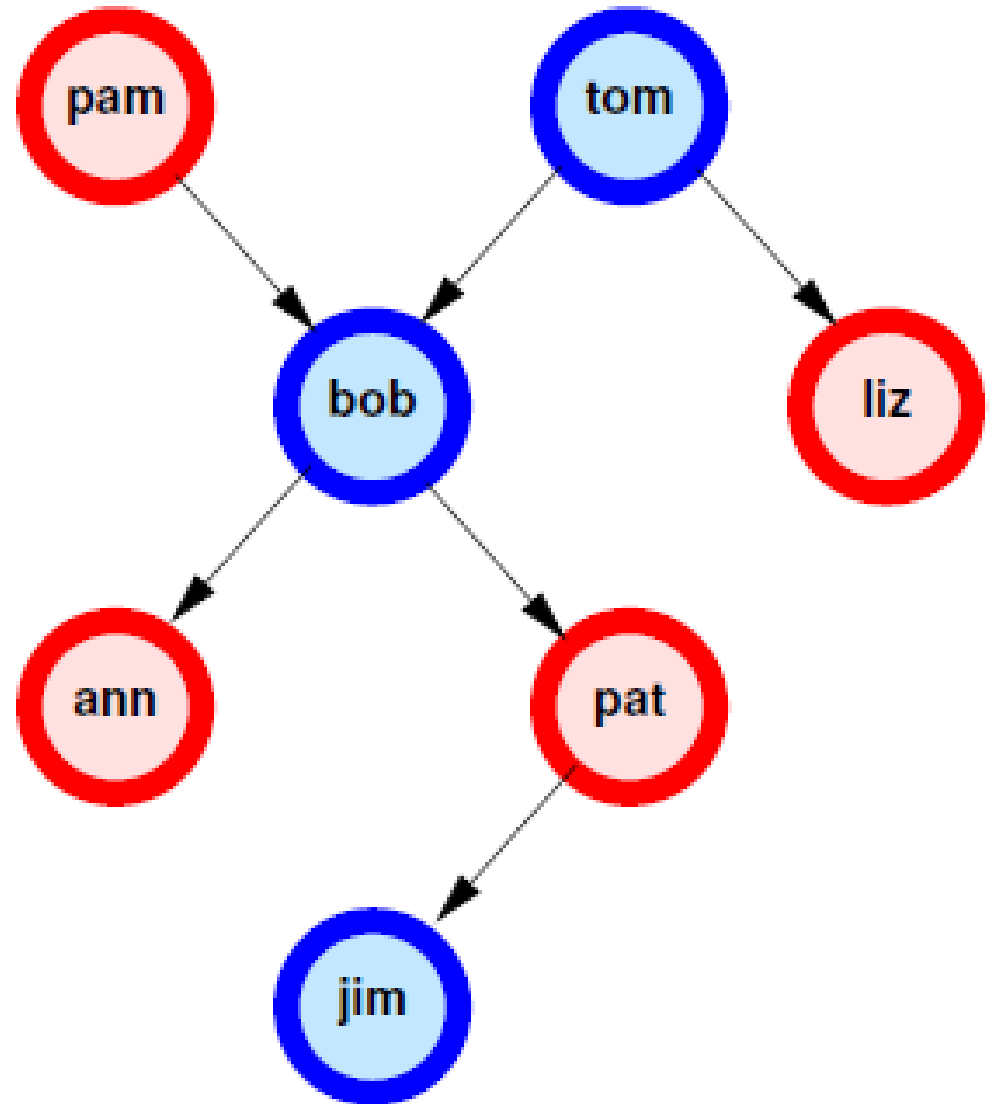
**female (pat) .**

**female (ann) .**

**female (liz) .**

# Relations

parent (pam, bob) .  
parent (tom, bob) .  
parent (tom, liz) .  
parent (bob, ann) .  
parent (bob, pat) .  
parent (pat, jim) .  
female (pam) .  
female (pat) .  
female (ann) .  
female (liz) .  
male (tom) .  
male (bob) .  
male (jim) .



# Relations

- Rules:

- **mother (X, Y)** : X is the mother of Y.

-In First Order Logic (FOL or predicate calculus):

$$\forall X, Y (\text{parent}(X, Y) \wedge \text{female}(X) \Rightarrow \text{mother}(X, Y))$$

-In Prolog:

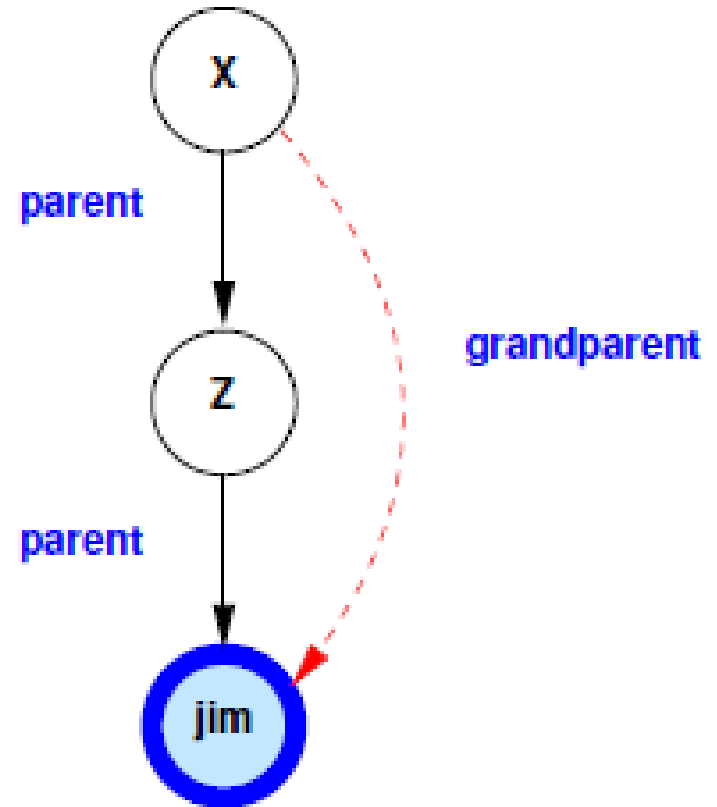
```
mother (X, Y) :-  
    parent (X, Y) ,  
    female (X) .
```

- all variables are universally quantified outside the rule
- “,” means *and* (conjunction), “:-” means *if* (implication) and “;” means *or* (disjunction).

# Relations

- More Relations:

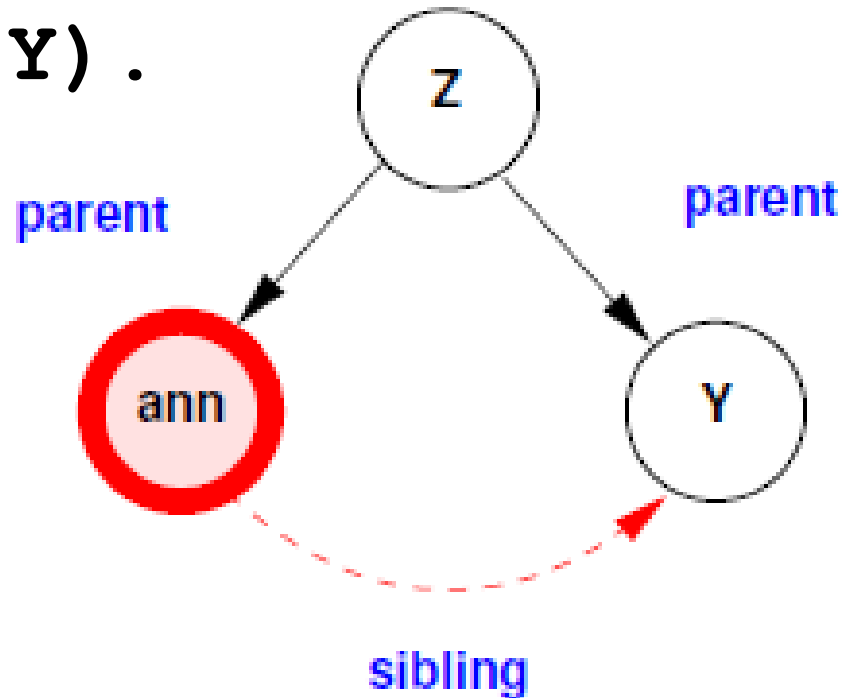
**grandparent (X, Y) :-  
parent (X, Z) ,  
parent (Z, Y) .**



# Relations

```
sibling(X,Y) :- parent(Z,X),  
parent(Z,Y), X \= Y.
```

?- sibling(ann,Y).



# Relations

- More Relations:

**cousin (X, Y) :- ...**

**greatgrandparent (X, Y) :- ...**

**greatgreatgrandparent (X, Y) :- ...**



# Recursion

**ancestor (X, Y) :-**

**parent (X, Y) .**

**ancestor (X, Y) :-**

**parent (X, Z) ,**

**ancestor (Z, Y) .**

?- **ancestor (X, jim) .**

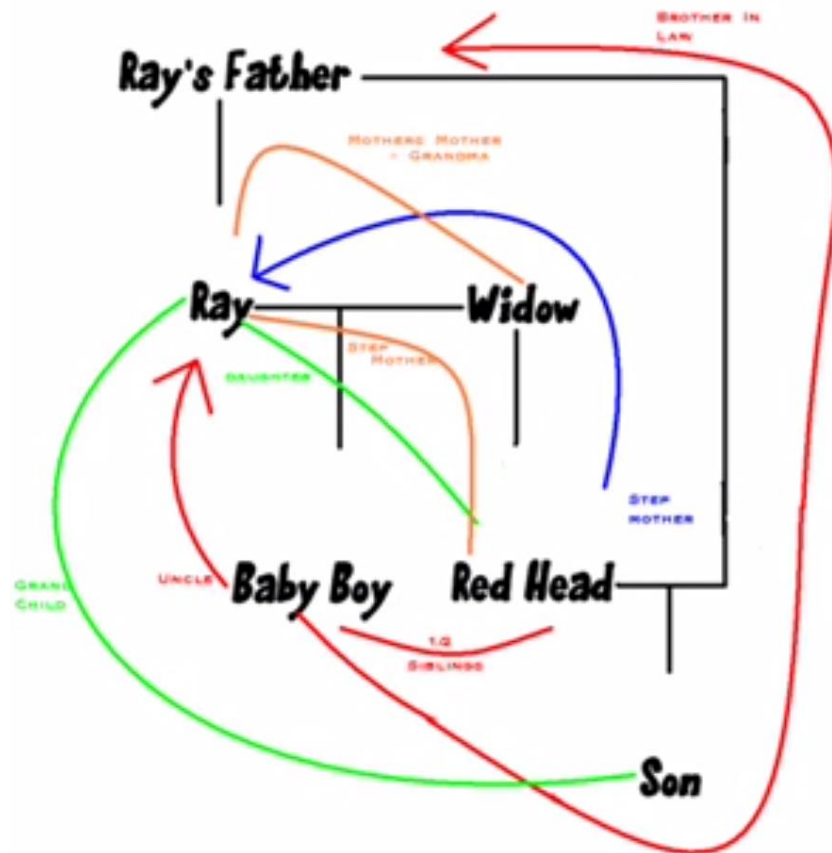
?- **ancestor (pam, X) .**

?- **ancestor (X, Y) .**

# Relations

- How to implement “I’m My Own Grandpa”?

<https://www.youtube.com/watch?v=eYlJH81dSiw>



# Recursion

- What about:

**ancestor** (X, Y) :-

**ancestor** (X, Z),

**parent** (Z, Y) .

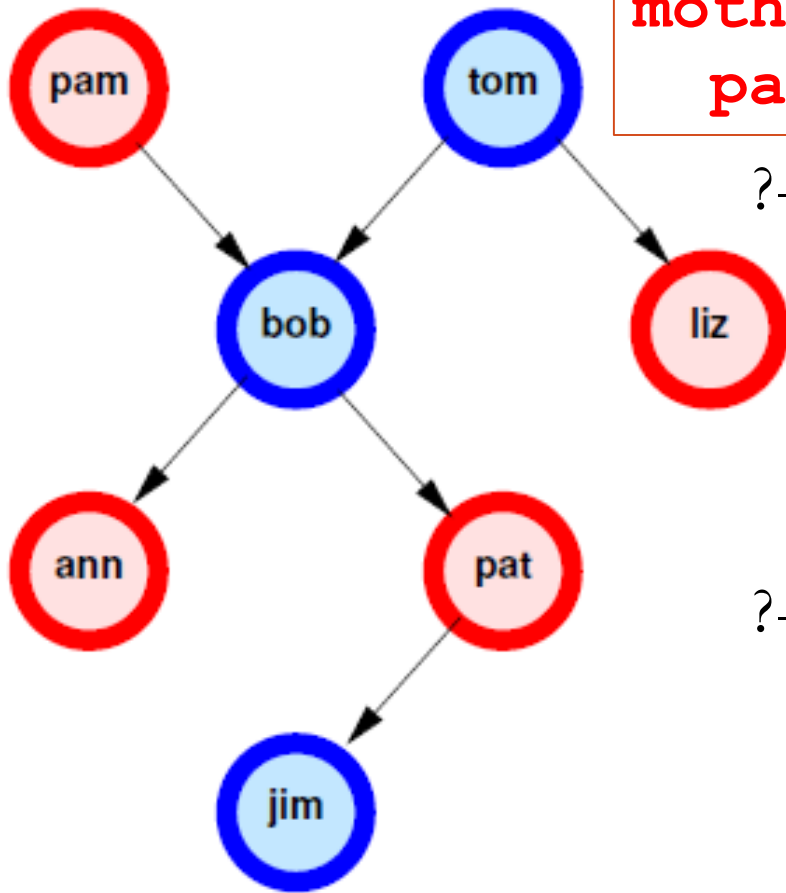
**ancestor** (X, Y) :-

**parent** (X, Y) .

?- **ancestor** (X, Y) .

**INFINITE LOOP**

# Computations in Prolog



```
mother(X, Y) :-  
parent(X, Y), female(X) .
```

?- mother(M, bob).

?- parent(M, bob), female(M).

?- M=pam, female(pam).

**M = pam true**

?- father(M, bob).

?- parent(M, bob), male(M)

(i) ?- M=pam, male(pam).

**fail**

(ii) ?- M=tom, male(tom).

**M = tom true**

# Prolog Execution

- Call: Call a predicate (invocation)
- Exit: Return an answer to the caller
- Fail: Return to caller with no answer
- Redo: Try next path to find an answer

# The XSB Prolog System

- <http://xsb.sourceforge.net>
  - Developed at Stony Brook by David Warren and many contributors
- Overview of Installation:
  - Unzip/untar; this will create a subdirectory XSB
  - Windows: you are done
  - Linux:

```
cd XSB/build
./configure
./makexsb
```

That's it!
  - Cygwin under Windows: same as in Linux

# Use of XSB

- Put your ruleset *and* data in a file with extension .P (or .pl)  

```
p(X) :- q(X,_) .  
q(1,a) .  
q(2,a) .  
q(b,c) .
```
- Don't forget: all rules and facts end with a period (.)
- Comments: /\*...\*/ or %.... (% acts like // in Java/C++)

- Type

.../XSB/bin/xsb

(Linux/Cygwin)

...\XSB\config\x86-pc-windows\bin\xsb (Windows)

where ... is the path to the directory where you downloaded XSB

- You will see a prompt

| ?-

and are now ready to type queries

# Use of XSB

- Loading your program, myprog.P or myprog.pl

?- **[myprog]** .

XSB will compile myprog.P (if necessary) and load it.

Now you can type further queries, e.g.

?- **p(X)** .

?- **p(1)** .

- Some Useful Built-ins:

- **write(X)** – write whatever X is bound to

- **writeln(X)** – write then put newline

- **nl** – output newline

- Equality: **=**

- Inequality: **\=**

<http://xsb.sourceforge.net/manual1/index.html> (Volume 1)

<http://xsb.sourceforge.net/manual2/index.html> (Volume 2)



# Use of XSB

- Some Useful Tricks:

- XSB returns only the first answer to the query

- To get the next, type `; <Return>`. For instance:

```
| ?- q(X).
```

```
X = 2;
```

```
X = 4
```

```
yes
```

- Usually, typing the `;`'s is tedious. To do this programmatically, use this idiom:

```
| ?- (q(_X), write('X='), writeln(_X), fail ; true).
```

`_X` here tells XSB to not print its own answers, since we are printing them by ourselves. (XSB won't print answers for variables that are prefixed with a `_`.)

# Logic Programming Concepts

- In logic, most statements can be written many ways
  - That's great for people but a nuisance for computers.
  - It turns out that if you make certain **restrictions** on the **format of statements** you can prove theorems mechanically
    - Most common restriction is to have a single conclusion implied by a conjunction of premises (i.e., *Horn clauses*)
      - Horn clauses are named for the logician Alfred Horn, who first pointed out their significance in 1951
    - That's what logic programming systems do!

# Syntax of Prolog Programs

- A *Prolog program* is a sequence of clauses
- Each *clause* (sometimes called a *rule* or *Horn rule*) is of the form:

**Head** :- **Body** .

- **Head** is one *term*
- **Body** is a comma-separated list of terms
- A clause with an empty body is called a *fact*

# Logic Programming Concepts

- Operators:
  - conjunction, disjunction, negation, implication
- Universal and existential quantifiers
- Statements
  - sometimes true, sometimes false, sometimes unknown
  - axioms - assumed true
  - theorems - provably true
  - goals - things we'd like to prove true

# Logic Programming Concepts

- A *term* can be a *constant*, *variable*, or *structure* (consisting of a *functor* and a parenthesized list of arguments)
- A *constant* is either an *atom* or a *number*
  - An *atom* is either what looks like an identifier beginning with a lowercase letter, or a single quoted string
  - A *number* looks like an integer or real from some more ordinary language
- A *variable* looks like an identifier beginning with an upper-case letter
- There are NO declarations (vars, terms, or predicates)
  - All types are discovered implicitly

# Logic Programming Concepts

- The Prolog interpreter has a collection of facts and rules in its DATABASE

- Facts (i.e., clauses with empty bodies):

**raining(ny) .                      raining(seattle) .**

- *Facts* are axioms (things the interpreter assumes to be true)

- Prolog provides an automatic way to deduce true results from facts and rules

- A rule (i.e., a clause with both sides):

**wet(X) :- raining(X) .**

- The meaning of a *rule* is that the conjunction of the structures in the body implies the head.

**Note: Single-assignment variables: X must have the same value on both sides.**

- *Query* or *goal* (i.e., a clause with an empty head):

**?- wet(X) .**

# Logic Programming Concepts

- So, rules are theorems that allow the interpreter to infer things
- To be interesting, rules generally contain variables

**employed(X) :- employs(Y,X) .**

can be read as:

*"for all X, X is employed if there exists a Y such that Y employs X"*

- Note the direction of the implication
- Also, the example does NOT say that X is employed

ONLY IF there is a Y that employs X

- there can be other ways for people to be employed
  - like, we know that someone is employed, but we don't know who is the employer or maybe they are self employed:

**employed(bill) .**

# Logic Programming Concepts

- The scope of a variable is the clause in which it appears:
  - Variables whose first appearance is on the **left hand side of the clause (i.e., the head)** have implicit **universal** quantifiers
    - For example, we infer for all possible **X** that they are **employed**  
**employed(X) :- employs(Y,X) .**
  - Variables whose **first appearance is in the body** of the clause have implicit **existential** quantifiers **in that body**
    - For example, there exists some **Y** that **employs X**
    - Note that these variables are also universally quantified outside the rule (by logical equivalences)



# Logic Programming Concepts

```
grandmother (A, C) :-  
    mother (A, B) ,  
    mother (B, C) .
```

can be read as:

*"for all A, C [A is the grandmother of C if there exists a B such that A is the mother of B and B is the mother of C]"*

- We probably want another rule that says:

```
grandmother (A, C) :-  
    mother (A, B) ,  
    father (B, C) .
```

# Recursion

- Transitive closure:
  - Example: a graph declared with facts (true statements)

**edge (1 , 2) .**

**edge (2 , 3) .**

**edge (2 , 4) .**

1) if there's an **edge** from **X** to **Y**, we can **reach Y** from **X**:

**reach (X , Y) :- edge (X , Y) .**

2) if there's an **edge** from **X** to **Z**, and we can **reach Y** from **Z**, then we can **reach Y** from **X**:

**reach (X , Y) :-  
edge (X , Z) ,  
reach (Z , Y) .**

?- reach (X, Y) .

X = 1

Y = 2 ; ← Type a semi-colon repeatedly for

X = 2 more answers

Y = 3 ;

X = 2

Y = 4 ;

X = 1

Y = 3 ;

X = 1

Y = 4 ;

no



reach (X, Y) :- edge (X, Y) .

reach (X, Y) :-

edge (X, Z) ,

reach (Z, Y) .

# Prolog Programs

- We will now explore Prolog programs in more detail:
  - Syntax of Prolog Programs
    - *Terms* can be:
      - Atomic data
      - Variables
      - Structures

# Atomic Data

- *Numeric constants*: integers, floating point numbers (e.g. **1024**, **-42**, **3.1415**, **6.023e23**,...)
- *Atoms*:
  - Identifiers: sequence of letters, digits, underscore, beginning with a lower case letter (e.g. **paul**, **r2d2**, **one\_element**).
  - Strings of characters enclosed in single quotes (e.g. **'Stony Brook'**)

# Variables

- Variables are denoted by identifiers beginning with an Uppercase letter or underscore (e.g. **X**, **Index**, **\_param**).
- *These are Single-Assignment Logical variables:*
  - Variables can be assigned only once
  - Different occurrences of the same variable in a clause denote the same data
- Variables are implicitly declared upon first use
  - Variables are not typed
    - All types are discovered implicitly (no declarations in LP)
  - If the variable does not start with underscore, it is assumed that it appears multiple times in the rule.
    - If it does not appear multiple times, then a warning is produced: "*Singleton variable*"
    - You can use variables preceded with underscore to eliminate this warning

# Variables

- *Anonymous variables* (also called *Don't care variables*): variables **beginning with "\_"**
  - Underscore, by itself (i.e., `_`), represents a variable
    - Each occurrence of `_` corresponds to a different variable; even within a clause, `_` does not stand for one and the same object.
  - A variable with a name beginning with `"_"`, but has more characters. E.g.: **`_radius`**, **`_Size`**
    - we want to give it a descriptive name
    - sometimes it is used to **create relationships within a clause (and must therefore be used more than once)**: a warning is produced: *"Singleton-marked variable appears more than once"*

# Variables

- Warnings are used to identify bugs (most because of copy-paste errors)
  - Instead of declarations and type checking
  - Fix all the warnings in a program, so you know that you don't miss any logical error



# Variables

- Variables can be assigned only once, but that value can be further refined:

$$\begin{aligned} ?- \quad & \mathbf{X=f(Y)} , \\ & \mathbf{Y=g(Z)} , \\ & \mathbf{Z=2} . \end{aligned}$$

Therefore,  $\mathbf{X=f(g(2))}$  ,  $\mathbf{Y=g(2)}$  ,  $\mathbf{Z=2}$

- The order also does not matter:

$$\begin{aligned} ?- \quad & \mathbf{Z=2} , \\ & \mathbf{X=f(Y)} , \\ & \mathbf{Y=g(Z)} . \end{aligned}$$

$\mathbf{X = f(g(2))}$  ,  $\mathbf{Y=g(2)}$  ,  $\mathbf{Z=2}$

- Even infinite structures:

$$?- \quad \mathbf{X=f(X)} .$$

$$\mathbf{X=f(f(f(f(f(f(f(f(f(\dots)))))}))}$$

# Logic Programming Queries

- To run a Prolog program, one asks the interpreter a question
  - This is done by asking a query which the interpreter tries to prove:
    - If it can, it says **yes**
    - If it can't, it says **no**
    - If your query contained variables, the interpreter prints the values it had to give them to make the query true

```
?- wet(ny) .      ?- reach(a, d) .      ?- reach(d, a) .
```

```
Yes              Yes              No
```

```
?- wet(X) .      ?- reach(X, d) .      ?- reach(X, Y) .
```

```
X = ny;          X=a              X=a, Y=d
```

```
X = seattle; ?- reach(a, X) .
```

```
no              X=d
```

# Meaning of Logic Programs

- **Declarative Meaning:** What are the *logical consequences* of a program?
- **Procedural Meaning:** For what values of the variables in the query can I *prove* the query?
  - The user gives the system a goal:
    - The system attempts to find axioms + inference rules to **prove** that goal
    - If goal contains variables, then also gives the values for those variables for which the goal is proven

# Declarative Meaning

```
brown(bear) .          big(bear) .
gray(elephant) .      big(elephant) .
black(cat) .          small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X), big(X) .
```

- The *logical consequences* of a program L is the smallest set such that
  - All facts of the program are in L,
  - If  $\mathbf{H} :- \mathbf{B}_1, \mathbf{B}_2, \dots, \mathbf{B}_n .$  is an instance of a clause in the program such that  $\mathbf{B}_1, \mathbf{B}_2, \dots, \mathbf{B}_n$  are all in L, then  $\mathbf{H}$  is also in L.
  - For the above program we get **dark(cat)** and **dark(bear)** and consequently **dangerous(bear)** in addition to the original facts.

# Procedural Meaning of Prolog

```
brown(bear) .          big(bear) .
gray(elephant) .      big(elephant) .
black(cat) .          small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X), big(X) .
```

- A *query* is, in general, a conjunction of goals:  $G_1, G_2, \dots, G_n$
- To *prove*  $G_1, G_2, \dots, G_n$ :
  - Find a clause  $H :- B_1, B_2, \dots, B_k$  such that  $G_1$  and  $H$  match.
  - Under the substitution for variables, prove  $B_1, B_2, \dots, B_k, G_2, \dots, G_n$

If nothing is left to prove then the proof succeeds!

If there are no more clauses to match, the proof fails!

# Procedural Meaning of Prolog

```
brown (bear) .          big (bear) .
gray (elephant) .      big (elephant) .
black (cat) .           small (cat) .
dark (Z) :- black (Z) .
dark (Z) :- brown (Z) .
dangerous (X) :- dark (X) , big (X) .
```

- To prove: ?- **dangerous (Q)** .

1. Select **dangerous (X) :-dark (X) ,big (X)** and prove **dark (Q) ,big (Q)** .
2. To prove **dark (Q)** select the first clause of dark, i.e. **dark (Z) :-black (Z)** , and prove **black (Q) ,big (Q)** .
3. Now select the fact **black (cat)** and prove **big (cat)** .
4. Go back to step 2, and select the second clause of dark, i.e. **dark (Z) :-brown (Z)** , and prove **brown (Q) ,big (Q)** .

This proof fails!

# Procedural Meaning of Prolog

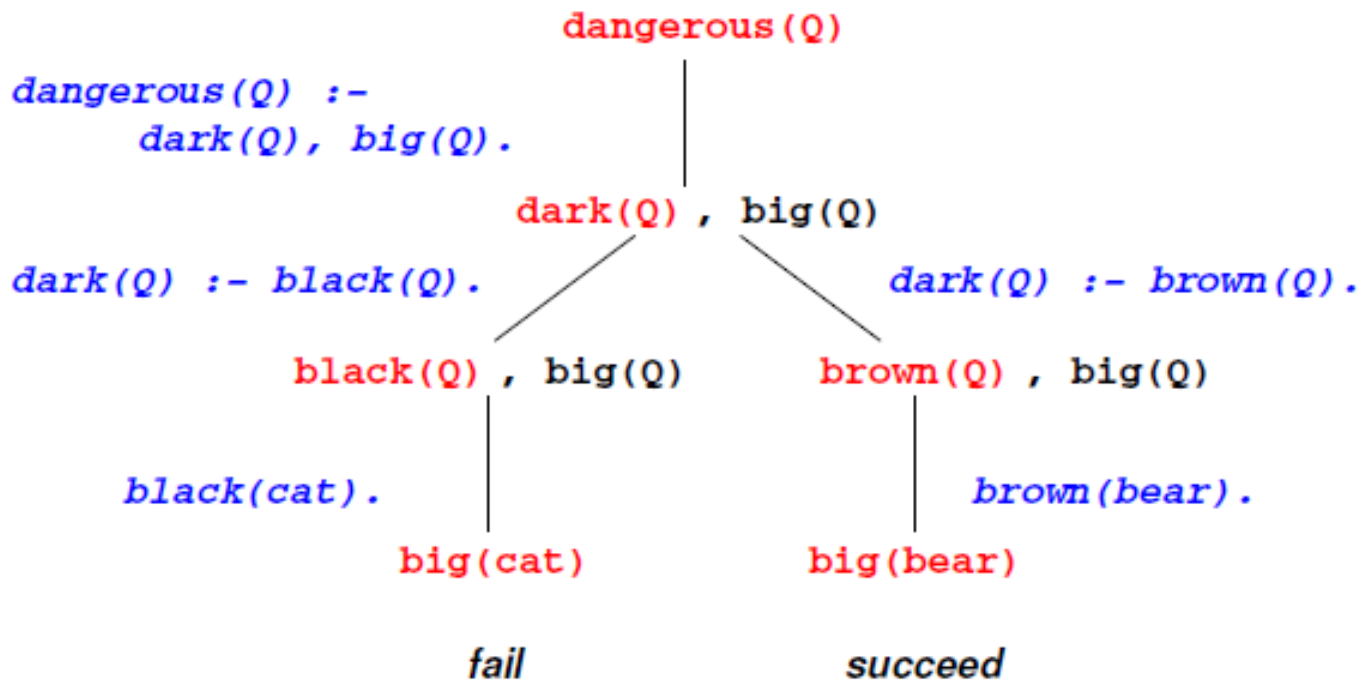
```
brown(bear) .           big(bear) .
gray(elephant) .       big(elephant) .
black(cat) .           small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X), big(X) .
```

- To prove: **?- dangerous(Q) .**
  5. Now select **brown(bear)** and prove **big(bear)** .
  6. Select the fact **big(bear)** .

There is nothing left to prove, so the proof succeeds

# Procedural Meaning of Prolog

```
brown(bear) .          big(bear) .
gray(elephant) .      big(elephant) .
black(cat) .          small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X) , big(X) .
```





# Procedural Meaning of Prolog

- The Prolog interpreter works by what is called **BACKWARD CHAINING** (also called *top-down, goal directed*)
  - It begins with the thing it is trying to prove and works backwards looking for things that would imply it, until it gets to facts.
- It is also possible to work forward from the facts trying to see if any of the things you can prove from them are what you were looking for
  - This methodology is called *bottom-up resolution*
  - It can be very time-consuming
  - Example: Answer set programming, DLV, Potassco (the Potsdam Answer Set Solving Collection), OntoBroker
- Fancier logic languages use both kinds of chaining, with special smarts or hints from the user to bound the searches

# Procedural Meaning of Prolog

- When it attempts resolution, the Prolog interpreter pushes the current goal onto a stack, makes the first term in the body the current goal, and goes back to the beginning of the database and starts looking again.
- If it gets through the first goal of a body successfully, the interpreter continues with the next one.
- If it gets all the way through the body, the goal is satisfied and it backs up a level and proceeds.

# Procedural Meaning of Prolog

- The Prolog interpreter starts at the beginning of your database (**this ordering is part of Prolog**, NOT of logic programming in general) and looks for something with which to unify the current goal
  - If it finds a fact, great; it succeeds,
  - If it finds a rule, it attempts to satisfy the terms in the body of the rule depth first.
  - This process is motivated by the *RESOLUTION PRINCIPLE*, due to Robinson, 1965:
    - It says that if  $C1$  and  $C2$  are Horn clauses, where  $C2$  represents a true statement and the head of  $C2$  unifies with one of the terms in the body of  $C1$ , then we can replace the term in  $C1$  with the body of  $C2$  to obtain another statement that is true if and only if  $C1$  is true



# Procedural Meaning of Prolog

- If it fails to satisfy the terms in the body of a rule, the interpreter **undoes** the unification of the left hand side (BACKTRACKING) (this includes un-instantiating any variables that were given values as a result of the unification) **and keeps looking through the database for something else with which to unify**
- If the interpreter gets to the end of database without succeeding, it **backs** out a level (that's how it might **fail** to satisfy something in a body) and continues from there.

# Procedural Meaning of Prolog

- PROLOG IS NOT PURELY DECLARATIVE:
  - The ordering of the database and the left-to-right pursuit of sub-goals gives a deterministic imperative semantics to searching and backtracking
  - Changing the order of statements in the database can give you different results:
    - It can lead to infinite loops
    - It can result in inefficiency

# Procedural Meaning of Prolog

- Transitive closure with left recursion in Prolog will run into an infinite loop:

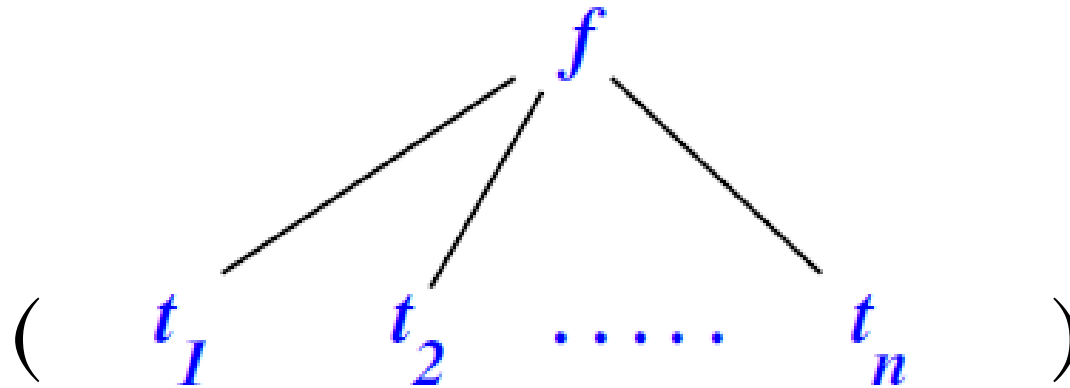
```
reach (X, Y) :-  
    reach (X, Z) ,  
    edge (Z, Y) .  
reach (X, Y) :-  
    edge (X, Y) .
```

```
?- reach (A, B) .
```

**Infinite loop**

# Structures

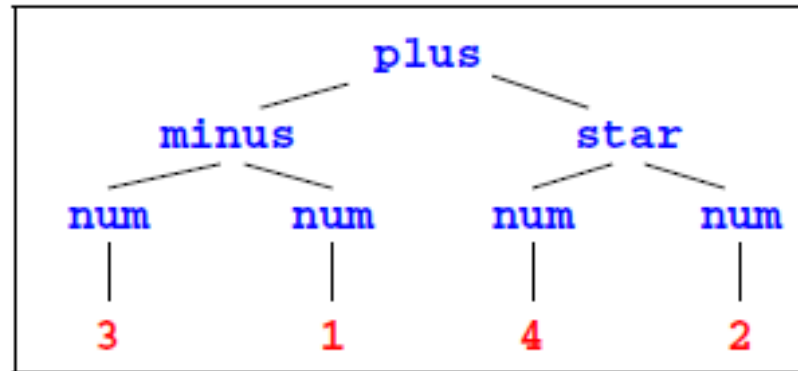
- If  $\mathbf{f}$  is an identifier and  $\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_n$  are terms, then  $\mathbf{f}(\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_n)$  is a term



- In the above,  $\mathbf{f}$  is called a *functor* and  $\mathbf{t}_i$ s are called *arguments*
- Structures are used to group related data items together (in some ways similar to struct in C and objects in Java)
  - Structures are used to construct trees (and, as a special case of trees, **lists**)

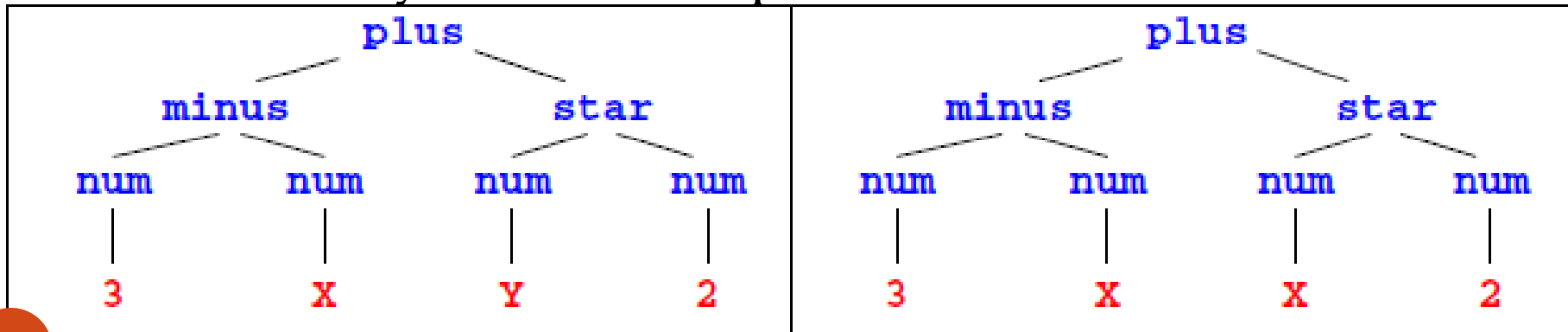
# Trees

- Example: expression trees:



`plus (minus (num (3) , num (1) ) , star (num (4) , num (2) ) ) )`

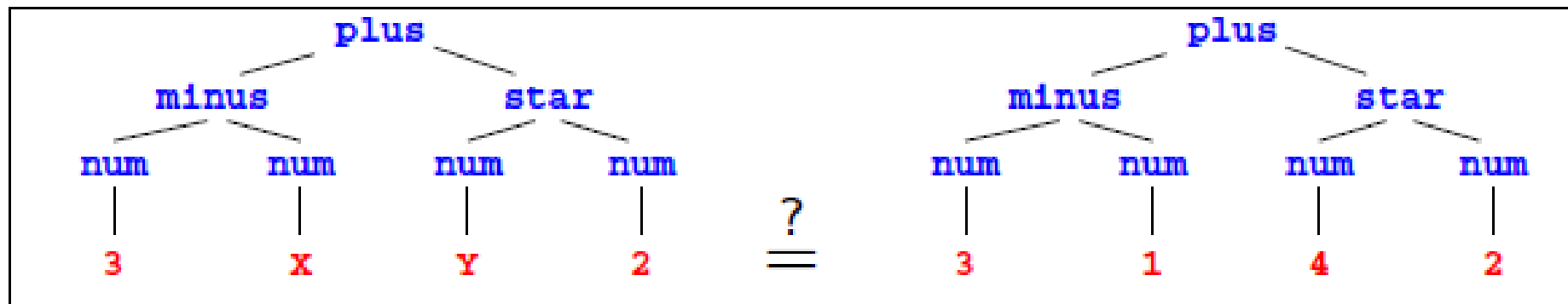
- Data structures may have variables AND the same variable may occur multiple times in a data structure





# Matching

- $t_1 = t_2$ : finds substitutions for variables in  $t_1$  and  $t_2$  that make the two terms identical
- (We'll later introduce *unification*, a related operation that has logical semantics)



Yes, with  $X = 1$ ,  $Y = 4$ .

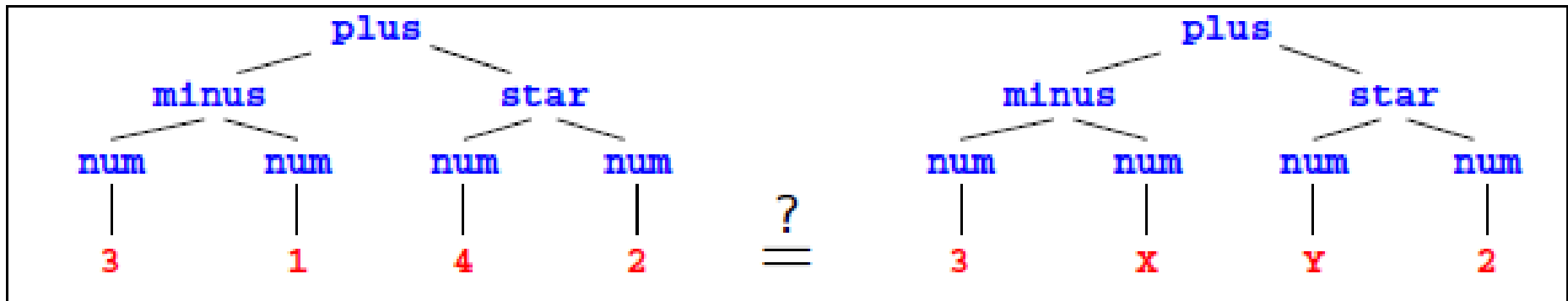
# Matching

- Matching: given two terms, we can ask if they "*match*" each other
  - A constant matches with itself: **42** unifies with **42**
  - A variable matches with anything:
    - if it matches with something other than a variable, then it instantiates,
    - if it matches with a variable, then the two variables become associated.
      - **A=35, A=B** → **B** becomes **35**
      - **A=B, A=35** → **B** becomes **35**
  - Two structures match if they:
    - Have the same functor,
    - Have the same arity, and
    - Match recursively
      - **foo (g (42) , 37)** matches with **foo (A, 37)** ,  
**foo (g (A) , B)** , etc.

# Matching

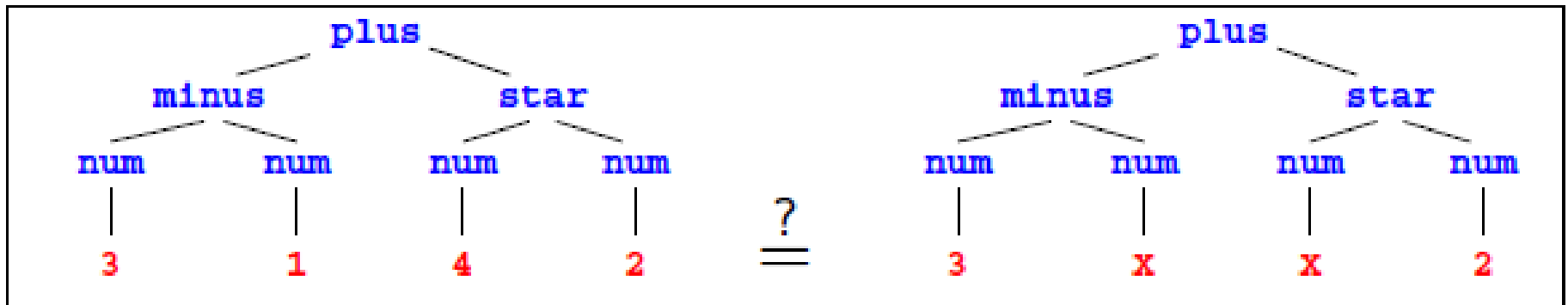
- The general Rules to decide whether two terms **S** and **T** *match* are as follows:
  - If **S** and **T** are constants, **S=T** if both are same object
  - If **S** is a variable and **T** is anything, **T=S**
  - If **T** is variable and **S** is anything, **S=T**
  - If **S** and **T** are structures, **S=T** if
    - **S** and **T** have same functor, same arity, and
    - All their corresponding arguments components have to match

# Matching



Yes, with  $X = 1$ ,  $Y = 4$ .

# Matching



No!  $X$  cannot be 1 and 4 at the same time.

# Matching

- Which of these match?
  - **A**
  - **100**
  - **func (B)**
  - **func (100)**
  - **func (C, D)**
  - **func (+ (99, 1) )**

# Matching

- Which of these match?
  - **A**
  - **100**
  - **func (B)**
  - **func (100)**
  - **func (C, D)**
  - **func (+ (99, 1) )**
- **A** matches with **100**, **func (B)**, **func (100)**, **func (C, D)**, **func (+ (99, 1) )**.
- **100** matches only with **A**.
- **func (B)** matches with **A**, **func (100)**, **func (+ (99, 1) )**
- **func (C, D)** matches with **A**.
- **func (+ (99, 1) )** matches with **A** and **func (B)**.

# Accessing arguments of a structure

- Matching is the predominant means for accessing structures arguments
- Let `date('Sep', 1, 2020)` be a structure used to represent dates, with the month, day and year as the three arguments (**in that order!**)

then `date(M,D,Y) = date('Sep',1,2020)`.  
makes

`M = 'Sep', D = 1, Y = 2020`

- If we want to get only the day, we can write

`date(_, D, _) = date('Sep', 1, 2020)`.

Then we only get: `D = 1`

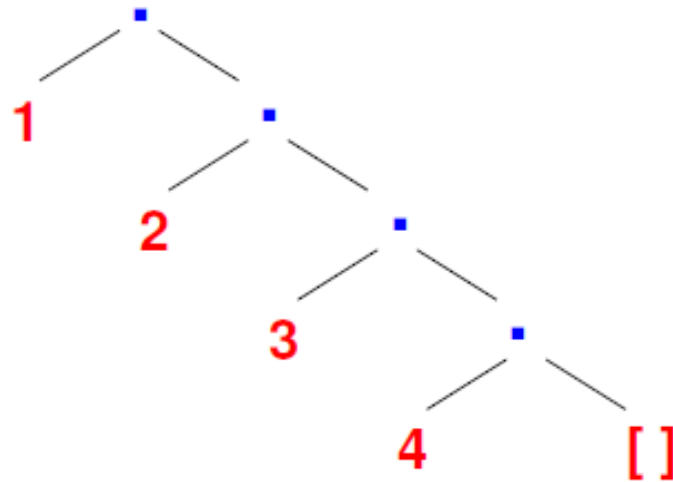


# Lists

- Prolog uses a special syntax to represent and manipulate lists
  - $[1, 2, 3, 4]$ : represents a list with **1**, **2**, **3** and **4**, respectively.
  - This can also be written as  $[1 | [2, 3, 4]]$ : a list with **1** as the *head* (first element) and  $[2, 3, 4]$  as its *tail* (the list of remaining elements).
    - If  $\mathbf{x} = 1$  and  $\mathbf{y} = [2, 3, 4]$  then  $[\mathbf{x} | \mathbf{y}]$  is same as  $[1, 2, 3, 4]$ .
  - The empty list is represented by  $[\ ]$  or **nil**
  - The symbol " $|$ " (*pipe*) and is used to separate the beginning elements of a list from its tail
    - For example:  $[1, 2, 3, 4] = [1 | [2, 3, 4]] = [1 | [2 | [3, 4]]] = [1, 2 | [3, 4]] = [1, 2, 3 | [4]] = [1 | [2 | [3 | [4 | []]]]]$

# Lists

- Lists are special cases of trees (syntactic sugar, i.e., internally, they use structures)
- For instance, the list `[1, 2, 3, 4]` is represented by the following structure:



- where the function symbol `./2` is the list constructor:  
`[1, 2, 3, 4]` is same as `.(1, .(2, .(3, .(4, [])))`

# Lists

- *Strings*: are sequences of characters surrounded by double quotes **"abc"**, **"John Smith"**, **"to be, or not to be"**.
- A string is equivalent to a list of the (numeric) character codes:

**? - X = "abc" .**

**X = [97, 98, 99]**

# Programming with Lists

- **member**/2 finds if a given element occurs in a list:
  - The program:

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

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

- Example queries:

```
?- member (2, [1, 2, 3]) .
```

```
?- member (X, [1, i, s, t]) .
```

```
?- member (f(X), [f(1), g(2), f(3), h(4)]) .
```

```
?- member (1, L) .
```

# Programming with Lists

- **append/3** concatenate two lists to form the third list:
  - The program:
    - Empty list **append L** is **L**:  
**append ([], L, L) .**
    - Otherwise, break the first list up into the head **X**, and the tail **L**: if **L** **append M** is **N**, then **[X|L]** **append M** is **[X|N]**:  
**append ([X|L], M, [X|N]) :-  
append (L, M, N) .**
  - Example queries:
    - ?- **append ([1,2], [3,4], X) .**
    - ?- **append (X, Y, [1,2,3,4]) .**
    - ?- **append (X, [3,4], [1,2,3,4]) .**
    - ?- **append ([1,2], Y, [1,2,3,4]) .**

# Programming with Lists

- Is the predicate a function?
  - **No.** We are not applying arguments to get a result. Instead, we are proving that a theorem holds. Therefore, we can leave any variables unbound.

?- `append(L, [2, 3], [1, 2, 3]).`

`L = [ 1 ]`

?- `append([ 1 ], L, [1, 2, 3]).`

`L = [2, 3]`

?- `append(L1, L2, [1, 2, 3]).`

`L1 = [], L2 = [1, 2, 3];`

`L1 = [1], L2 = [2, 3];`

`L1 = [1, 2], L2 = [3];`

`L1 = [1, 2, 3], L2 = [];`

`no`

# Append example trace

```
append([], L, L) .
```

```
append([X|L], M, [X|N]) :- append(L, M, N) .
```

```
append([1,2], [3,4], X) ?
```

# Append example trace

`append([], L, L) .`

`append([X|L], M, [X|N]) :- append(L, M, N) .`

<code>append([1, 2], [3, 4], A) ?</code>	<code>X=1, L=[2], M=[3, 4], A=[X N]</code>
--	--



# Append example trace

`append([], L, L) .`

`append([X|L], M, [X|N]) :- append(L, M, N) .`

`append([2], [3, 4], N) ?`

`append([1, 2], [3, 4], A) ?`

`X=1, L=[2], M=[3, 4], A=[X|N]`

# Append example trace

`append([], L, L) .`

`append([X|L], M, [X|N']) :- append(L, M, N') .`

<code>append([2], [3, 4], N) ?</code>	<code>X=2, L=[], M=[3, 4], N=[2 N']</code>
<code>append([1, 2], [3, 4], A) ?</code>	<code>X=1, L=[2], M=[3, 4], A=[1 N]</code>

# Append example trace

**append ([ ] , L , L) .**

**append ([X|L] , M , [X|N'] ) :- append (L , M , N' ) .**

**append ([ ] , [3 , 4] , N' ) ?**

**append ([2] , [3 , 4] , N) ?**

**append ([1 , 2] , [3 , 4] , A) ?**

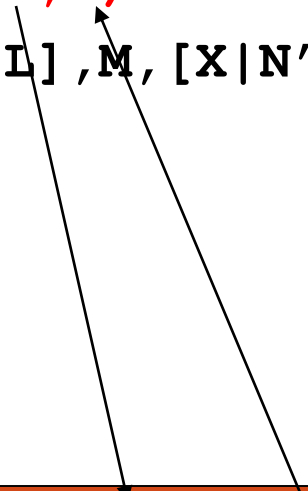
**X=2 , L= [ ] , M= [3 , 4] , N= [2 | N' ]**

**X=1 , L= [2] , M= [3 , 4] , A= [1 | N]**

# Append example trace

**append([], L, L) .**

**append([X|L], M, [X|N']) :- append(L, M, N') .**



<b>append([], [3, 4], N') ?</b>	<b>L = [3, 4], N' = L</b>
<b>append([2], [3, 4], N) ?</b>	<b>X=2, L=[], M=[3, 4], N=[2 N']</b>
<b>append([1, 2], [3, 4], A) ?</b>	<b>X=1, L=[2], M=[3, 4], A=[1 N]</b>

# Append example trace

`append([], L, L) .`

`append([X|L], M, [X|N']) :- append(L, M, N') .`

`A = [1|N]`

`N = [2|N']`

`N' = L`

`L = [3,4]`

Answer: `A = [1,2,3,4]`

`append([], [3,4], N') ?`

`L = [3,4], N' = L`

`append([2], [3,4], N) ?`

`X=2, L=[], M=[3,4], N=[2|N']`

`append([1,2], [3,4], A) ?`

`X=1, L=[2], M=[3,4], A=[1|N]`

# Programming with Lists

- **len**/2 to find the length of a list (the first argument):
  - The program:

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

```
len([_ | Xs], N+1) :-  
    len(Xs, N).
```

- Example queries:

```
?- len([], X).  
    X = 0
```

```
?- len([l,i,s,t], 4).  
    false
```

```
?- len([l,i,s,t], X).  
    X = 0+1+1+1+1
```

# Arithmetic

?-  $1+2 = 3$ .

*false*

- In Predicate logic, the basis for Prolog, the only symbols that have a meaning are the predicates themselves
- In particular, function symbols are **uninterpreted**: have no special meaning and can only be used to construct data structures

# Arithmetic

- Meaning for arithmetic expressions is given by the built-in predicate "**is**":

**?- X is 1 + 2.**

*succeeds, binding X = 3.*

**?- 3 is 1 + 2.**

*succeeds.*

- General form: **R is E** where **E** is an expression to be evaluated and **R** is matched with the expression's value
- **Y is X + 1**, where **X** is a free variable, will give an error because **X** does not (yet) have a value, so, **X + 1** cannot be evaluated



# The list length example revisited

- **length/2 finds** the length of a list (first argument):
  - The program:

```
length([], 0).  
length([_|Xs], M) :-  
    length(Xs, N),  
    M is N+1.
```

- Example queries:

```
?- length([], X).  
?- length([l,i,s,t], 4).  
?- length([l,i,s,t], X).  
    X = 4  
?- length(List, 4).  
    List = [_1, _2, _3, _4]
```

# Conditional Evaluation

- Conditional operator: the if-then-else construct in Prolog:
  - *if A then B else C* is written as **( A -> B ; C )**
    - To Prolog this means: try **A**. If you can prove it, go on to prove **B** and ignore **C**. If **A** fails, however, go on to prove **C** ignoring **B**.

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

```
?- max(1, 2, X).  
X = 2.
```

# Conditional Evaluation

- Consider the computation of  $n!$  (i.e. the factorial of  $n$ )  
% `factorial(+N, -F)`  
`factorial(N, F) :- ...`
- $N$  is the input parameter and  $F$  is the output parameter!
- The body of the rule specifies how the output is related to the input
  - For factorial, there are two cases:  $N \leq 0$  and  $N > 0$ 
    - if  $N \leq 0$ , then  $F = 1$
    - if  $N > 0$ , then  $F = N * \text{factorial}(N - 1)$

```
factorial(N, F) :-
```

```
(N > 0
```

```
-> N1 is N-1,
```

```
    factorial(N1, F1),
```

```
    F is N*F1
```

```
; F = 1
```

```
).
```

```
?- factorial(12,X) .
```

```
X = 479001600
```

# Imperative features

- Other imperative features: we can think of prolog rules as imperative programs **w/ backtracking**

```
program :-
```

```
    member (X, [1, 2, 3, 4]),  
    write (X),  
    nl,  
    fail;  
    true.
```

```
?- program. % prints all solutions
```

- **fail**: always fails, causes backtracking
- **!** is the cut operator: prevents other rules from matching (we will see it later)

# Arithmetic Operators

- Integer/Floating Point operators:  $+$ ,  $-$ ,  $*$ ,  $/$ 
  - Automatic detection of Integer/Floating Point
- Integer operators:  $\text{mod}$ ,  $//$  (integer division)
- Comparison operators:  $<$ ,  $>$ ,  $=<$ ,  $>=$ ,

$Expr1 ::= Expr2$  (succeeds if expression

$Expr1$  evaluates to a number equal to  $Expr2$ ),

$Expr1 \neq Expr2$  (succeeds if expression

$Expr1$  evaluates to a number non-equal to  $Expr2$ )

# Programming with Lists

- Quicksort:

```
quicksort([], []).
```

```
quicksort([X0|Xs], Ys) :-  
    partition(X0, Xs, Ls, Gs),  
    quicksort(Ls, Ys1),  
    quicksort(Gs, Ys2),  
    append(Ys1, [X0|Ys2], Ys).
```

```
partition(Pivot, [], [], []).
```

```
partition(Pivot, [X|Xs], [X|Ys], Zs) :-  
    X =< Pivot,  
    partition(Pivot, Xs, Ys, Zs).
```

```
partition(Pivot, [X|Xs], Ys, [X|Zs]) :-  
    X > Pivot,  
    partition(Pivot, Xs, Ys, Zs).
```

# Programming with Lists

- Quicksort:

```
quicksort([], []).
```

```
quicksort([X0|Xs], Ys) :-  
    partition(X0, Xs, Ls, Gs),  
    quicksort(Ls, Ys1),  
    quicksort(Gs, Ys2),  
    append(Ys1, [X0|Ys2], Ys).
```

```
partition(Pivot, [], [], []).
```

```
partition(Pivot, [X|Xs], [X|Ys], Zs) :-  
    X =< Pivot,  
    !, % cut
```

```
    partition(Pivot, Xs, Ys, Zs).
```

```
partition(Pivot, [X|Xs], Ys, [X|Zs]) :-  
    partition(Pivot, Xs, Ys, Zs).
```

# Programming with Lists

- We want to define **delete/3**, to remove a given element from a list (called **select/3** in XSB's **basics** library):
  - **delete(2, [1,2,3], X)** should succeed with **X=[1,3]**
  - **delete(X, [1,2,3], [1,3])** should succeed with **X=2**
  - **delete(2, X, [1,3])** should succeed with **X=[2,1,3]**; **X=[1,2,3]**; **X=[1,3,2]**; **fail**
  - **delete(2, [2,1,2], X)** should succeed with **X=[1,2]**; **X=[2,1]**; **fail**



# Programming with Lists

- **Algorithm:**

- When **X** is selected from **[X | Ys]**, **Ys** results as the rest of the list
- When **X** is selected from the tail of **[H | Ys]**, **[H | Zs]** results, where **Zs** is the result of taking **X** out of **Ys**

# Programming with Lists

- The program:

```
delete(X, [], _) :- fail. % not needed  
delete(X, [X|Ys], Ys).  
delete(X, [Y|Ys], [Y|Zs]) :-  
    delete(X, Ys, Zs).
```

- Example queries:

```
?- delete(s, [l,i,s,t], Z).  
X = [l, i, t]  
?- delete(X, [l,i,s,t], Z).  
?- delete(s, Y, [l,i,t]).  
?- delete(X, Y, [l,i,s,t]).
```

# Permutations

- Define **permute**/2, to find a permutation of a given list.
  - E.g. **permute** ([1, 2, 3], **x**) should return **x**=[1, 2, 3] and upon backtracking, **x**=[1, 3, 2], **x**=[2, 1, 3], **x**=[2, 3, 1], **x**=[3, 1, 2], and **x**=[3, 2, 1].
  - Hint: What is the relationship between the permutations of [1, 2, 3] and the permutations of [2, 3]?

<b>permute</b> ([2,3], Y)	<b>permute</b> ([1,2,3], Y)
[2,3]	[1,2,3]
	[2,1,3]
	[2,3,1]
[3,2]	[1,3,2]
	[3,1,2]
	[3,2,1]

# Programming with Lists

- The program:

```
permute([], []).
```

```
permute([X|Xs], Ys) :-
```

```
    permute(Xs, Zs),
```

```
    delete(X, Ys, Zs).
```

- Example query:

```
?- permute([1,2,3], X).
```

# The Issue of Efficiency

- Define a predicate, **rev**/2 that finds the reverse of a given list
  - E.g. **rev**([1, 2, 3], **X**) should succeed with **X**=[3, 2, 1]
  - Hint: what is the relationship between the reverse of [1, 2, 3] and the reverse of [2, 3]? Answer: **append**([3, 2], [1], [3, 2, 1])  
**rev**([], []).  
**rev**([X|Xs], Ys) :- **rev**(Xs, Zs),  
**append**(Zs, [X], Ys).
- How long does it take to evaluate **rev**([1, 2, ..., n], **X**)?
  - **T**(n) = **T**(n - 1) + time to add 1 element to the end of an n - 1 element list = **T**(n - 1) + n - 1 =  
**T**(n - 2) + n - 2 + n - 1 = ...
  - **T**(n) = **O**(n<sup>2</sup>) (quadratic)

# Making rev/2 faster

- Keep an accumulator: stack all elements seen so far
  - i.e. a list, with elements seen so far in reverse order
- The program:

```
rev(L1, L2) :- rev_h(L1, [], L2).  
rev_h([X|Xs], AccBefore, Out) :-  
    rev_h(Xs, [X|AccBefore], Out).  
rev_h([], Acc, Acc). % Base case
```

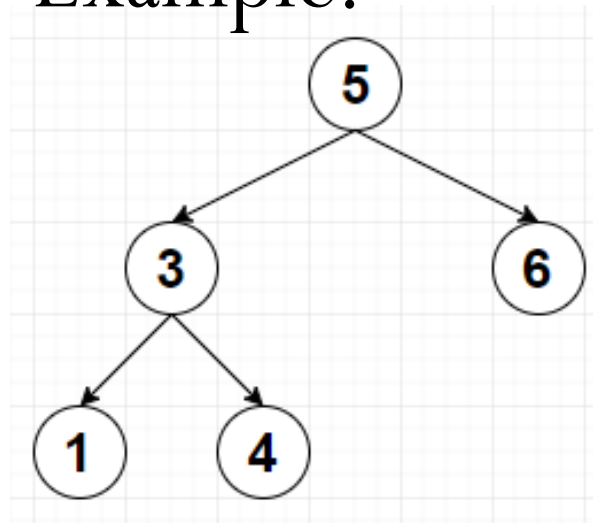
- Example query:

```
?- rev([1,2,3], X).
```

```
will call rev_h([1,2,3], [], X)  
which calls rev_h([2,3], [1], X)  
which calls rev_h([3], [2,1], X)  
which calls rev_h([], [3,2,1], X)  
which returns X = [3,2,1]
```

# Tree Traversal

- Assume you have a binary tree, represented by
  - **node/3** facts for internal nodes: **node (a, b, c)** means that **a** has **b** and **c** as children
  - **leaf/1** facts: for leaves: **leaf (a)** means that **a** is a leaf
  - Example:



**node (5, 3, 6) .**

**node (3, 1, 4) .**

**leaf (1) .**

**leaf (4) .**

**leaf (6) .**

# Tree Traversal

- Write a predicate `preorder/2` that traverses the tree (starting from a given node) and returns the list of nodes in pre-order

```
preorder (Root, [Root]) :-
```

```
    leaf (Root) .
```

```
preorder (Root, [Root|L]) :-
```

```
    node (Root, Child1, Child2) ,
```

```
    preorder (Child1, L1) ,
```

```
    preorder (Child2, L2) ,
```

```
    append (L1, L2, L) .
```

```
?- preorder (5, L) .
```

```
L = [5, 3, 1, 4, 6]
```

- The program takes  $O(n^2)$  time to traverse a tree with  $n$  nodes



# Difference Lists

- The lists in Prolog are singly-linked; hence we can access the first element in constant time, **but** need to scan the entire list to get the **last** element
- However, unlike functional languages like Lisp or SML, we can use **variables** in data structures:
  - We can exploit this to make lists “*open tailed*” (also called *difference lists* in Prolog): end the list with a variable tail and pass that variable, so we can add elements at the end of the list

# Difference Lists

- When  $\mathbf{x} = [1, 2, 3 | \mathbf{Y}]$ ,  $\mathbf{x}$  is a list with 1, 2, 3 as its first three elements, followed by  $\mathbf{Y}$ 
  - Now if  $\mathbf{Y} = [4 | \mathbf{Z}]$  then  $\mathbf{x} = [1, 2, 3, 4 | \mathbf{Z}]$ 
    - We can think of  $\mathbf{Z}$  as “pointing to” the end of  $\mathbf{x}$
- **We can now add an element to the end of  $\mathbf{x}$  in constant time!**
  - and continue adding more elements, e.g.  $\mathbf{Z} = [5 | \mathbf{W}]$

# Difference Lists: Conventions

- A *difference list* is represented by two variables: one referring to the entire list, and another to its (uninstantiated) tail
  - e.g.  $\mathbf{X} = [1, 2, 3 | \mathbf{Z}], \mathbf{Z}$
- Most Prolog programmers use the notation **List-Tail** to denote a list **List** with tail **Tail**.
  - e.g.  $\mathbf{X-Z}$
  - Note that “-” is used as a data structure symbol (not used for arithmetic here)

# Difference Lists

- Append 2 open ended lists:

```
dappend(X,T, Y,T2, L,T3) :-
```

```
    T = Y,
```

```
    T2 = T3,
```

```
    L = X.
```

```
?- dappend([1,2,3|T],T, [4,5,6|T2],T2, L,T3) .
```

```
L = [1,2,3,4,5,6|T3]
```

- Simplified version:

```
dappend(X,T, T,T2, X,T2) .
```

- More simplified **notation** (with "-"):

```
dappend(X-T, T-T2, X-T2) .
```

```
?- dappend([1,2,3|T]-T, [4,5,6|T2]-T2, L-T3) .
```

```
L = [1,2,3,4,5,6|T2]
```

# Difference Lists

- Add an element at the end of a list:

```
add(L-T, X, L2-T2) :-
```

```
    T = [X|T2],
```

```
    L = L2.
```

```
?- add([1,2,3|T]-T, 4, L-T2).
```

```
L = [1,2,3,4|T2]
```

- We can simplify it as:

```
add(L-T, X, L-T2) :-
```

```
    T = [X|T2].
```

- This can be simplified more like:

```
add(L-[X|T2], X, L-T2).
```

- Alternative using **dappend**:

```
add(L-T, X, L-T2) :-
```

```
    dappend(L-T, [X|T2]-T2, L-T2).
```

# Difference Lists

- Check if a list is a palindrome:

```
palindrome(X) :-  
    palindromeHelp(X-[]).  
palindromeHelp(A-A). % an empty list  
palindromeHelp([_|A]-A). % 1-element list  
palindromeHelp([C|A]-D) :-  
    palindromeHelp(A-B),  
    B=[C|D].  
?- palindrome([1,2,2,1]).  
yes  
?- palindrome([1,2,3,2,1]).  
yes  
?- palindrome([1,2,3,4,5]).  
no
```

# Tree Traversal, Revisited

```
preorder1(Node, List, Tail) :-  
    node(Node, Child1, Child2),  
    List = [Node|List1],  
    preorder1(Child1, List1, Tail1),  
    preorder1(Child2, Tail1, Tail).  
preorder1(Node, [Node|Tail], Tail) :-  
    leaf(Node).  
preorder(Node, List) :-  
    preorder1(Node, List, []).
```

- The program takes  $O(n)$  time to traverse a tree with  $n$  nodes

# Difference Lists: Conventions

- The preorder traversal program may be rewritten as:

```
preorder1 (Node, [Node|L]-T) :-  
    node (Node, Child1, Child2),  
    preorder1 (Child1, L-T1),  
    preorder1 (Child2, T1-T).  
preorder1 (Node, [Node|T]-T).
```



# Difference Lists: Conventions

- The inorder traversal program:

```
inorder1 (Node, L-T) :-  
    node (Node, Child1, Child2),  
    inorder1 (Child1, L-T1),  
    T1 = [Node|T2],  
    inorder1 (Child2, T2-T).  
inorder1 (Node, [Node|T]-T).  
inorder (Node, L) :-  
    inorder1 (Node, L-[]).
```

# Difference Lists: Conventions

- The postorder traversal program:

```
postorder1 (Node, L-T) :-  
    node (Node, Child1, Child2),  
    postorder1 (Child1, L-T1),  
    postorder1 (Child2, T1-T2),  
    T2 = [Node|T].  
postorder1 (Node, [Node|T]-T).  
postorder (Node, L) :-  
    postorder1 (Node, L-[]).
```

# Graphs in Prolog

- There are several ways to represent graphs in Prolog:
  - represent each edge separately as one clause (fact):  
**edge (a , b) .**  
**edge (b , c) .**
    - isolated nodes cannot be represented, unless we have also **node/1** facts
  - the whole graph as one data object: as a pair of two sets (nodes and edges): **graph ([a , b , c , d , f , g] , [e (a , b) , e (b , c) , e (b , f) ])**
    - list of arcs: **[a-b , b-c , b-f]**
  - adjacency-list: **[n (a , [b] ) , n (b , [c , f] ) , n (d , []) ]**

# Graphs in Prolog

- Path from one node to another one:
  - a predicate **path (G, A, B, P)** to find an acyclic path **P** from node **A** to node **B** in the graph **G**.
  - The predicate should return all paths via backtracking.
  - We will solve it using the graph as a data object, like in **graph ([a, b, c, d, f, g], [e(a, b), e(b, c), e(b, f)])**

# Graphs in Prolog

- **adjacent** for directed edges:

```
adjacent (X, Y, graph (_, Es) ) :-  
    member (e (X, Y) , Es) .
```

- **adjacent** for undirected edges (no distinction between the two vertices associated with each edge):

```
adjacent (X, Y, graph (_, Es) ) :-  
    member (e (X, Y) , Es) .
```

```
adjacent (X, Y, graph (_, Es) ) :-  
    member (e (Y, X) , Es) .
```

# Graphs in Prolog

- Path from one node to another one:

```
path (G, A, B, P) :-  
    pathHelper (G, A, [B], P) .  
  
% Base case  
pathHelper (_, A, [A|P1], [A|P1]) .  
pathHelper (G, A, [Y|P1], P) :-  
    adjacent (X, Y, G) ,  
    \+ member (X, [Y|P1]) ,  
    pathHelper (G, A, [X, Y|P1], P) .
```

# Graphs in Prolog

- Cycle from a given node in a directed graph:
  - a predicate **cycle (G, A, P)** to find a closed path (cycle) **P** starting at a given node **A** in the graph **G**.
  - The predicate should return all cycles via backtracking

```
cycle (G, A, P) :-  
    adjacent (A, B, G) ,  
    path (G, B, A, P1) ,  
    P = [A | P1] .
```

# Complete program in XSB

```
:- import member/2 from basics.
```

```
adjacent(X,Y,graph(_,Es)) :-  
    member(e(X,Y),Es).
```

```
path(G,A,B,P) :-  
    pathHelper(G,A,[B],P).
```

```
pathHelper(_,A,[A|P1],[A|P1]).  
pathHelper(G,A,[Y|P1],P) :-  
    adjacent(X,Y,G),  
    \+ member(X,[Y|P1]),  
    pathHelper(G,A,[X,Y|P1],P).
```

```
cycle(G,A,P) :-  
    adjacent(A,B,G),  
    path(G,B,A,P1),  
    P = [A|P1].
```

```
?- Graph = graph([a,b,c,d,f,g],[e(a,b), e(b,c), e(c,a), e(a,e), e(e,a)]),  
    cycle(Graph,a,P),  
    writeln(P),  
    fail;  
    true.
```



# Aggregates in XSB

- **setof (Template, Goal, Set)** : **Set** is the set of all instances of **Template** such that **Goal** is provable.
- **findall (Template, Goal, List)** is similar to predicate **bagof**/3, except that variables in **Goal** that do not occur in **Template** are treated as existential, and alternative lists are not returned for different bindings of such variables.
- **bagof (Template, Goal, Bag)** has the same semantics as **setof**/3 except that the third argument returns an unsorted list that may contain duplicates. **X^Goal** will not bind **X**
- **tfindall (Template, Goal, List)** is similar to predicate **findall**/3, but the **Goal** must be a call to a single tabled predicate.

# Aggregates in XSB

`p(1,1).`

`p(1,2).`

`p(2,1).`

`?- setof(Y,p(X,Y),L).`

`L=[1,2]`

`?- findall(Y,p(X,Y),L).`

`L=[1,2,1]`

`?- bagof(Y,p(X,Y),L).`

`X=1, L=[1,2] ;`

`X=2, L=[1] ;`

`fail`

# XSB Prolog

- Negation:  $\backslash +$  is negation-as-failure
- Another negation called **tnot** (*TABLING = memoization*)
  - Use: ... :- ..., **tnot (foobar (X))** .
  - All variables under the scope of **tnot** must also occur to the left of that scope in the body of the rule in other positive relations:
    - Ok: ... :- **p (X, Y)** , **tnot (foobar (X, Y))** , ...
    - Not ok: ... :- **p (X, Z)** , **tnot (foobar (X, Y))** , ...
- XSB also supports Datalog:
  - :- **auto\_table** .at the top of the program file

# XSB Prolog

- Read/write from and to files:
  - Edinburgh style:

```
?- tell('a.txt'),  
   write('Hello, World!'), told.
```

```
?- see('a.txt'), read(X), seen.
```

# XSB Prolog

- Read/write from and to files:
  - ISO style:

```
?- open('a.txt', write, X),  
   write(X, 'Hello, World!'),  
   close(X).
```

# Cut (logic programming)

- Cut (**!** in Prolog) is a goal which always succeeds, **but cannot be backtracked past.**

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

**max (X, \_, X) .**

- cut says “stop looking for alternatives”
  - no check is needed in the second rule anymore because if we got there, then **X =< Y** must have failed, so **X > Y** must be true.
  - red cut: if someone deletes **!**, then the rule is incorrect
- Green cut

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

**max (X, Y, X) :- X > Y.**

- by explicitly writing **X > Y**, it guarantees that the second rule will always work even if the first one is removed by accident or changed (cut is deleted)

# Cut (logic programming)

- No backtracking pass the guard, but ok after:

`p(a) . p(b) .`

`q(a) . q(b) . q(c) .`

`?- p(X) , ! .`

`X=a ;`

`no`

`?- p(X) , ! , q(Y) .`

`X=a , Y=a ;`

`X=a , Y=b ;`

`X=a , Y=c ;`

`no`

# Testing types

- **atom (X)**

Tests whether **X** is bound to a symbolic atom.

```
?- atom(a) .
```

```
yes
```

```
?- atom(3) .
```

```
no
```

- **integer (X)**

Tests whether **X** is bound to an integer.

- **real (X)**

Tests whether **X** is bound to a real number.



# Testing for variables

- **is\_list(L)**

Tests whether **L** is bound to a list.

- **ground(G)**

Tests whether **G** has unbound logical variables.

- **var(X)**

Tests whether **X** is bound to a Prolog variable.

# Control / Meta-predicates

- **call (P)**

Force **P** to be a goal; succeed if **P** does, else fail.

- **clause (H, B)**

Retrieves clauses from memory whose head matches **H** and body matches **B**. **H** must be sufficiently instantiated to determine the main predicate of the head.

- **copy\_term (P, NewP)**

Creates a new copy of the first parameter (with new variables).

- It is used in iteration through non-ground clauses, so that the original calls are not bound to values

# Control / Meta-predicates

- Write a Prolog relation

**map (BinaryRelation, InputList, OutputList)**

which applies a binary relation on each of the elements of the list **InputList** as the first argument and collects the second argument in the result list.

Example:

?- **map (incl (X, Y) , [5, 6] , R) .** returns **R=[6, 7]**

where **incl (X, Y)** was defined as:

```
incl (X, Y) :-  
    Y is X+1.
```

# Control / Meta-predicates

```
map(_BinaryCall, [], []) .
```

```
map(BinaryCall, [X|T], [Y|T2]) :-
```

```
    copy_term(BinaryCall, BinaryCall2),
```

```
    BinaryCall2 =.. [_F,X,Y],
```

```
    call(BinaryCall2),
```

```
    map(BinaryCall, T, T2) .
```

```
incl(X,Y) :-
```

```
    Y is X+1.
```

```
?- map(incl(X,Y), [5,6], R) .
```

```
R = [6,7]
```

# Control / Meta-predicates

```
square(X,Y) :-
```

```
    Y is X*X.
```

```
?- map(square(E, E2), [2,3,1], R).
```

```
R = [4,9,1];
```

```
no
```

# Control / Meta-predicates

- Use the relation **map** to implement a relation **pairAll (E, L, L2)** which pairs the element **E** with each element of the list **L** to obtain **L2**.

Examples:

```
?- pairAll(1, [2, 3, 1], L2) .
```

```
returns L2=[[1, 2], [1, 3], [1, 1]]
```

```
?- pairAll(1, [], L2) .
```

```
returns L2=[] .
```

# Control / Meta-predicates

```
pair(E2, (_E1,E2)).
```

```
pairAll(E,L,L2):-
```

```
    map(pair(E2, (E,E2)), L, L2).
```

```
?- pairAll(1, [2,3,1], R).
```

```
R = [(1,2), (1,3), (1,1)]
```

# Assert and retract

- **asserta (C)**

Assert clause **C** into database above other clauses with the same predicate.

- **assertz (C) , assert (C)**

Assert clause **C** into database below other clauses with the same predicate.

- **retract (C)**

Retract **C** from the database. **C** must be sufficiently instantiated to determine the predicate.



# Prolog terms

- **functor (E, F, N)**

**E** must be bound to a functor expression of the form '**f** ( . . . )'. **F** will be bound to '**f**', and **N** will be bound to the number of arguments that **f** has.

- **arg (N, E, A)**

**E** must be bound to a functor expression, **N** is a whole number, and **A** will be bound to the **N**th argument of **E**

# Prolog terms and clauses

- `=..`

converts between term and list. For example,

```
?- parent(a,X) =.. L.
```

```
L = [parent, a, _X001]
```

```
?- [1] =.. X.
```

```
X = [., 1, []]
```