

# Programming in Prolog

CSE 505 – Computing with Logic

Stony Brook University

<http://www.cs.stonybrook.edu/~cse505>

# Relations

- `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).`

# Relations

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

`female(pam).`

`female(pat).`

`female(ann).`

`female(liz).`

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

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

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

# Relations

parent(pam, bob).

parent(tom, bob).

parent(tom, liz).

parent(bob, ann).

parent(bob, pat).

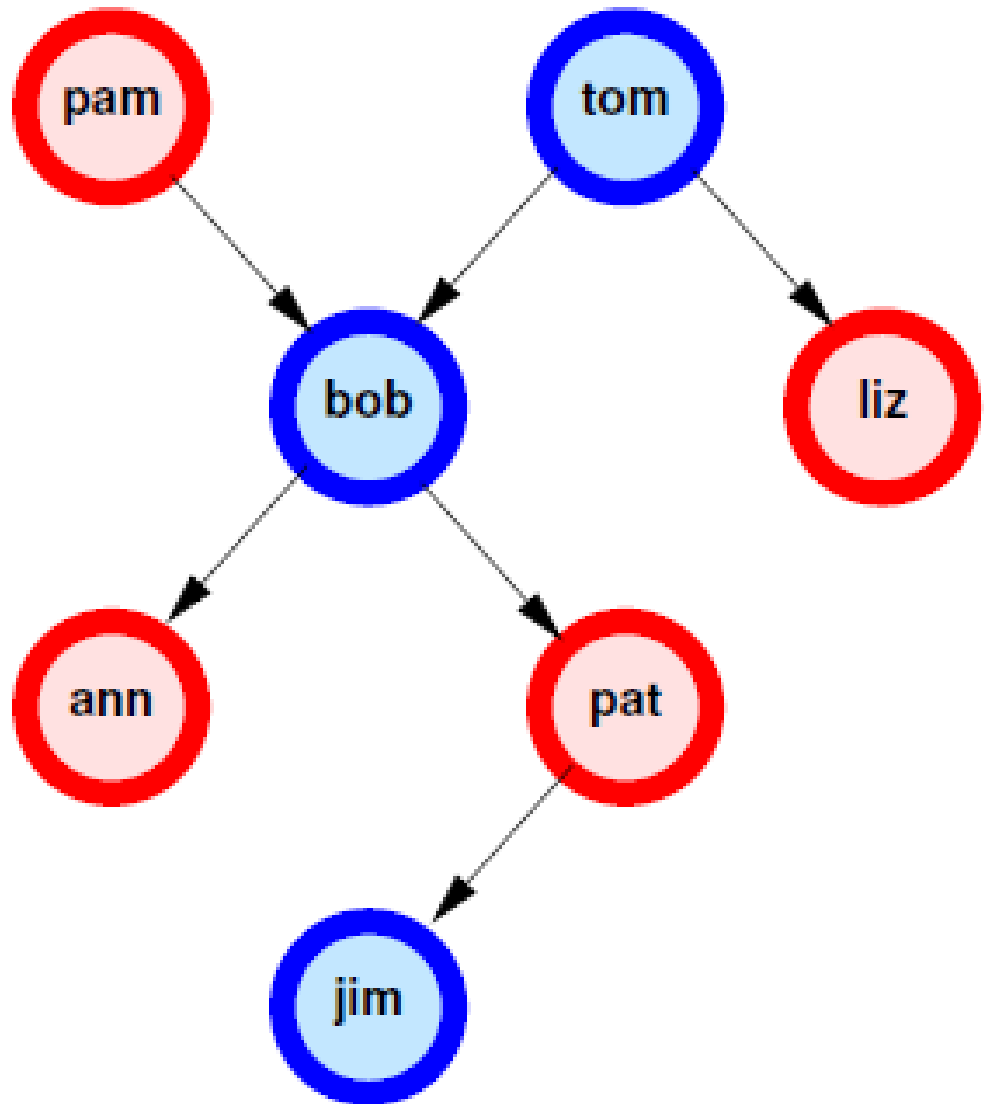
parent(pat, jim).

female(pam).      male(tom).

female(pat).      male(bob).

female(ann).      male(jim).

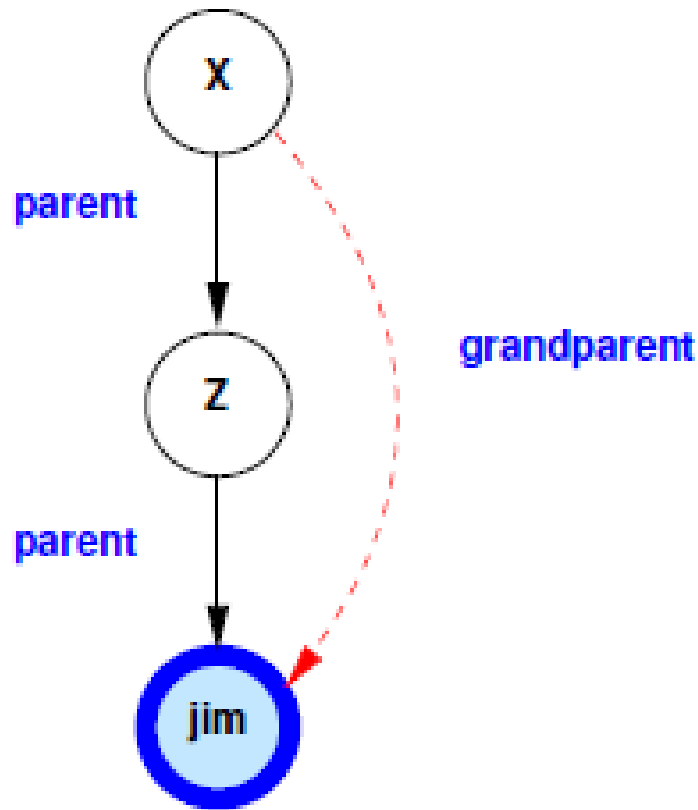
female(liz).



# Relations

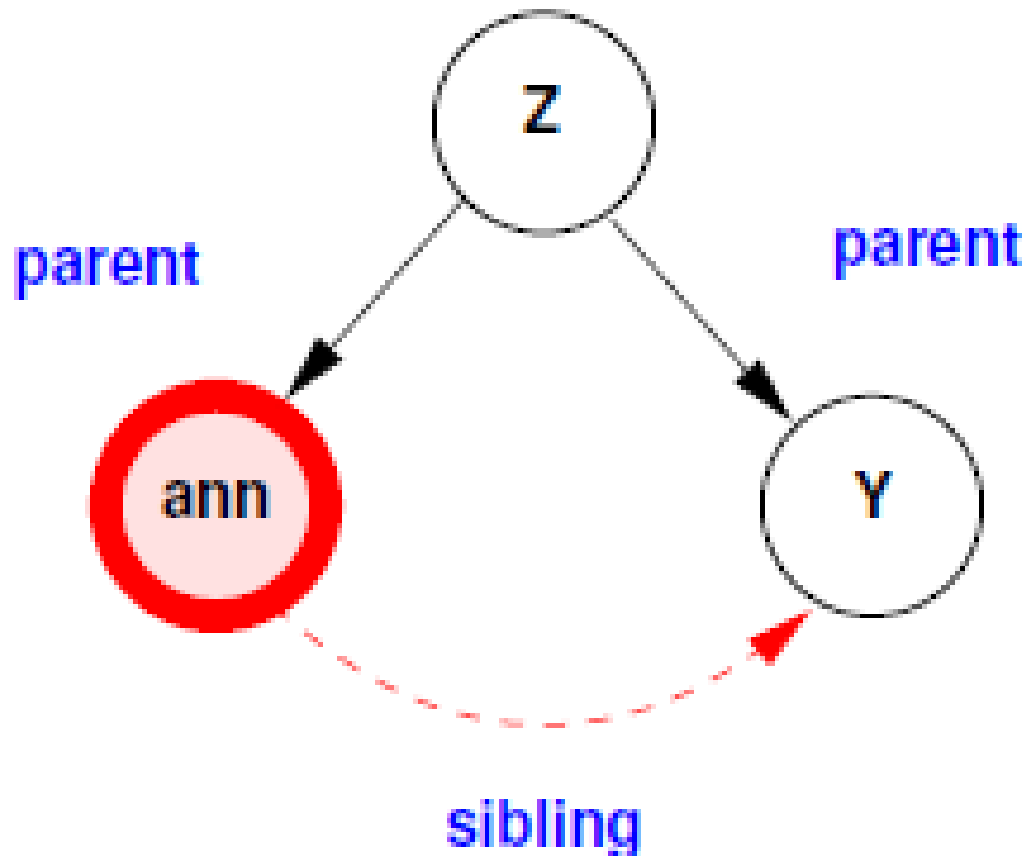
- More Relations:

$\text{grandparent}(X, Y) \text{ :- } \text{parent}(X, Z), \text{parent}(Z, Y).$



# Relations

$\text{sibling}(X, Y) \text{ :- parent}(Z, X), \text{parent}(Z, Y), X \neq Y.$



# Relations

- More Relations:

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

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

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

ancestor(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).

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

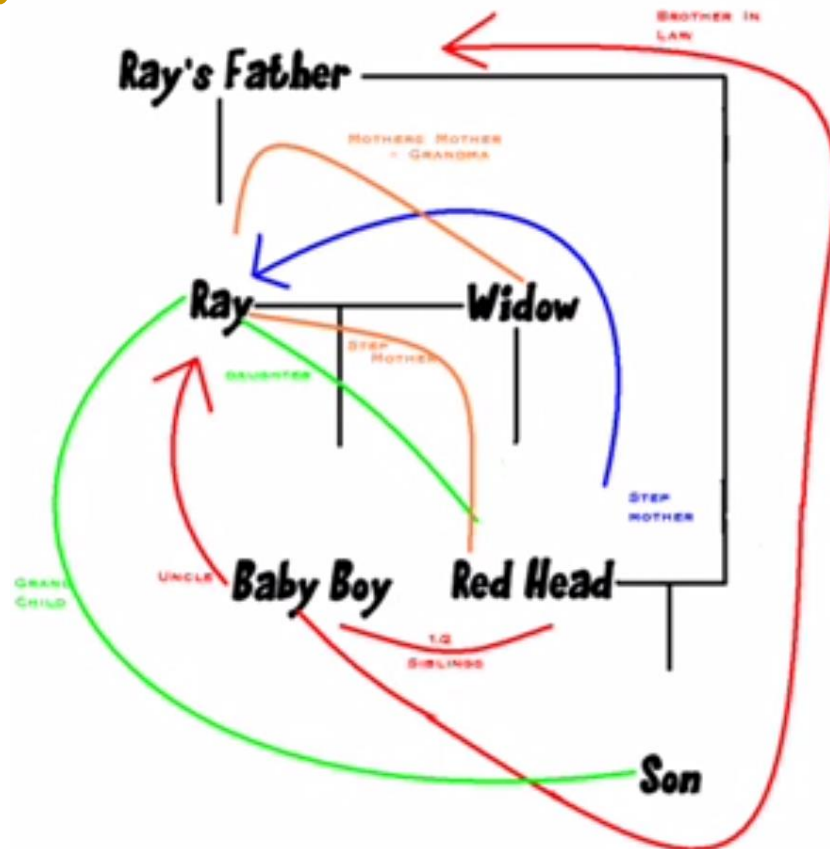
<https://www.youtube.com/watch?v=eYlJH81dSiw>



# 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

# Recursion

- Transitive closure:

`edge (1, 2) .`

`edge (2, 3) .`

`edge (2, 4) .`

`reachable (X, Y) :- edge (X, Y) .`

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

?- reachable(X,Y) .

X = 1

Y = 2;    **Type a semi-colon repeatedly**

X = 2

Y = 3;

X = 2

Y = 4;

X = 1

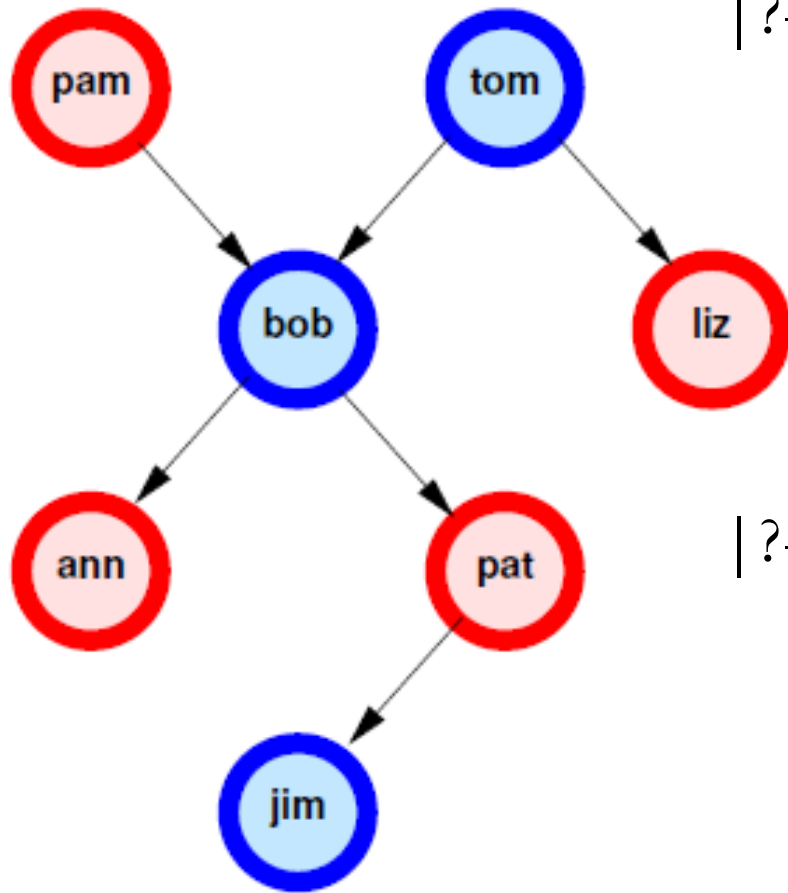
Y = 3;

X = 1

Y = 4;

no

# Computations in Prolog



| ?- 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)  
     $p(X) :- q(X, \_).$   
     $q(1, a).$   
     $q(2, a).$   
     $q(b, c).$   
     $? - p(X).$
- 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

| ?- [myprog].

XSB will compile myprog.P (if necessary) and load it. Now you can type further queries, e.g.

| ?- p(X).

| ?- p(1).

Etc.

- 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)

(c) Paul Fodor (CS Stony Brook) and Elsevier

# 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 `_`.)

# Syntax of Prolog Programs

- A *program* is a sequence of clauses.
- Each *clause* 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*.
- A clause is also sometimes called a *rule*.

# Terms

- Atomic data
- Variables
- Structures

# Atomic Data

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

# Variables

- Variables are denoted by identifiers beginning with an *Uppercase letter* or *underscore* (e.g. X, Index, \_param).
  - Underscore, by itself (i.e., \_), represents an anonymous variable.
- Different occurrences of the same variable in a clause denote the same data.
  - Each occurrence of an anonymous variable is treated as a different data.
- Variables are implicitly declared upon first use.
- Variables are not typed.

# Variables

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

$$X=f(Y),$$

$$Y=g(Z),$$

$$Z=2.$$

- Even infinite structures:  $S=f(S)$ .
- We'll come to this topic later when we discuss about structures.

# 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?



# 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) .
```

- *Logical consequence of a program* L is the smallest set such that
  - All facts of the program are in L,
  - If  $H :- B_1, B_2, \dots, B_n$  is an instance of a clause in the program such that  $B_1, B_2, \dots, B_n$  are all in L, then H is also in L.
- For the above program we get `dark(cat)` and `dark(bear)` and consequently `dangerous(bear)`.

# 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 that 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)**. **This proof fails!**
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)**.

# Procedural Meaning of Prolog

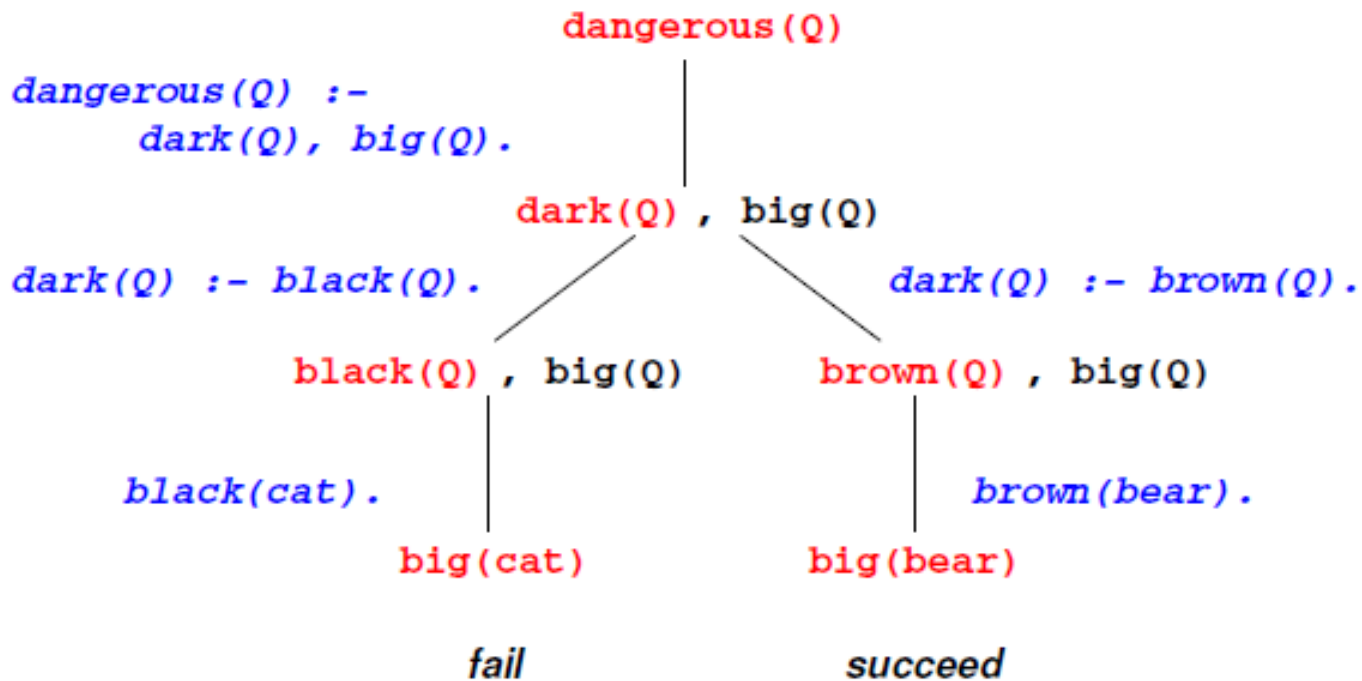
```
brown(bear) .           big(bear) .  
gray(elephant) .       big(elephant) .  
black(cat) .           small(cat) .  
dark(Z) :- black(Z) .  
dark(Z) :- brown(Z) .  
dangerous(X) :- dark(X) , big(X) .
```

- 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

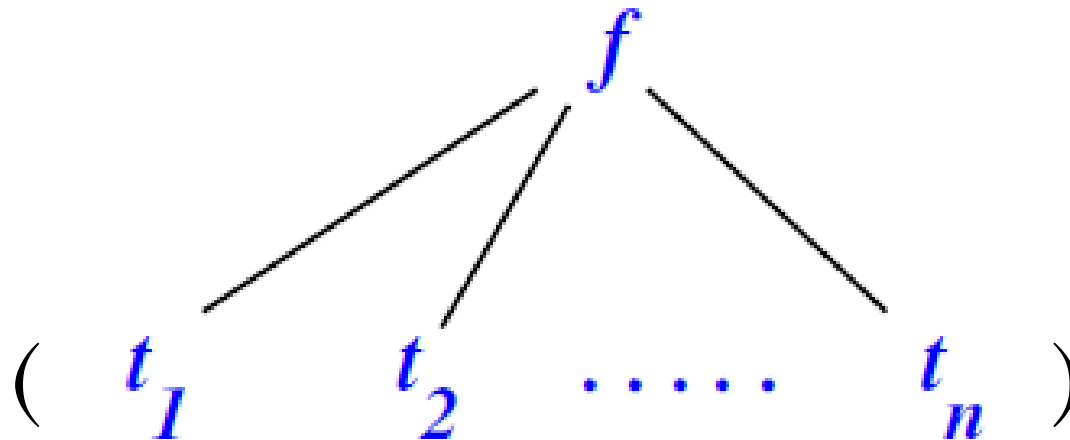
# Procedural Meaning of Prolog

```
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) .
```



# Structures

- If  $f$  is an identifier and  $t_1, t_2, \dots, t_n$  are terms, then  $f(t_1, t_2, \dots, t_n)$  is a term.

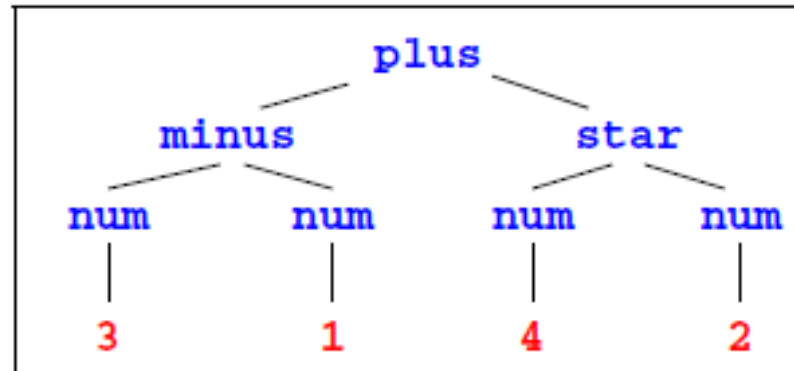


- In the above,  $f$  is called a *functor* and  $t_i$  is an *argument*.
- Structures are used to group related data items together (in some ways similar to struct in C and objects in Java).
- Structures are used to construct trees (and, as a special case, lists).

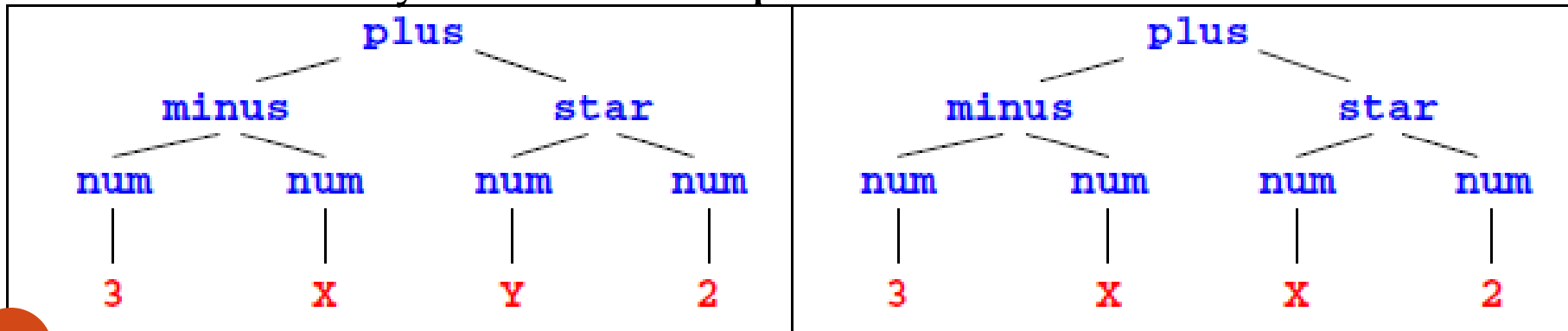
# Trees

- Example: expression trees:

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

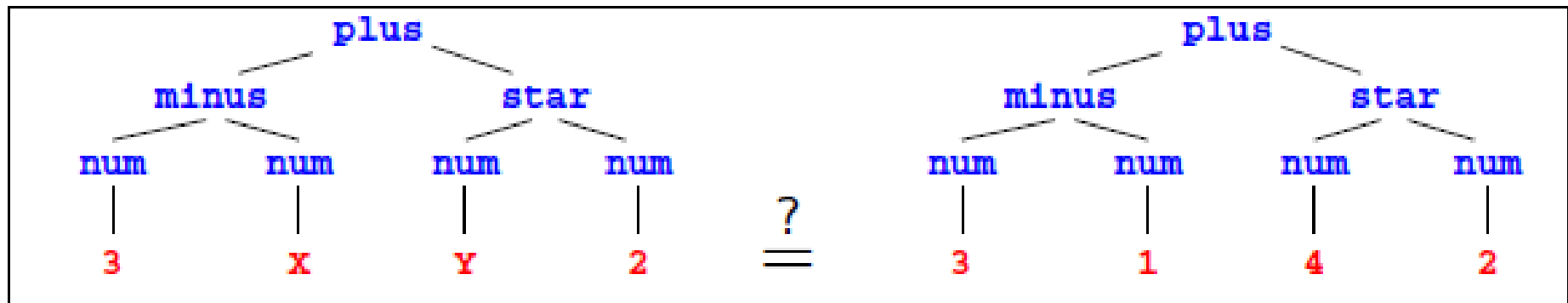


- Data structures may have variables. And the same variable may occur multiple times in a data structure.



# Matching

- (We'll later introduce *unification*, a related operation that has logical semantics).
- $t1 = t2$ : find substitutions for variables in  $t1$  and  $t2$  that make the two terms identical.



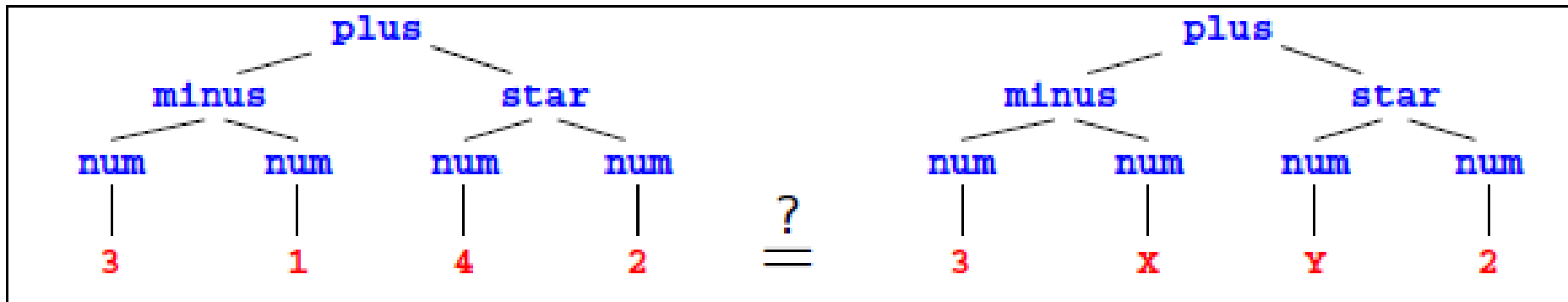
Yes, with  $X = 1$ ,  $Y = 4$ .



# Matching

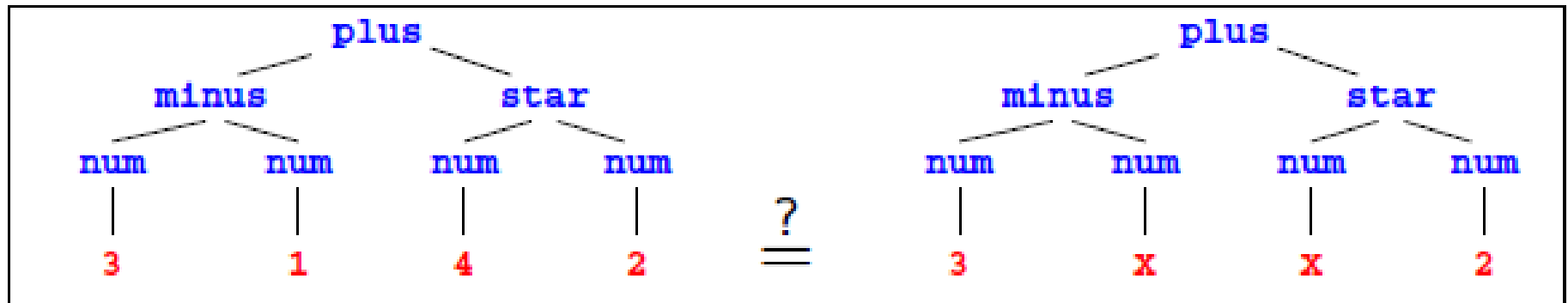
- General Rule to decide whether two terms,  $S$  and  $T$  match are as follows:
  - If  $S$  and  $T$  are constants,  $S=T$  if both are same object
  - If  $S$  is a variable and  $T$  is anything,  $T=S$
  - If  $T$  is variable and  $S$  is anything,  $S=T$
  - If  $S$  and  $T$  are structures,  $S=T$  if
    - $S$  and  $T$  have same functor
    - 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.

# Accessing arguments of a structure

- Matching is the predominant means for accessing a structures arguments.
- Let `date('Sep', 1, 2015)` 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, 2015)` makes

`M = 'Sep', D = 1, Y = 2015.`

- If we want to get only the day, we can write `date( _, D, _) = date('Sep', 1, 2015).`

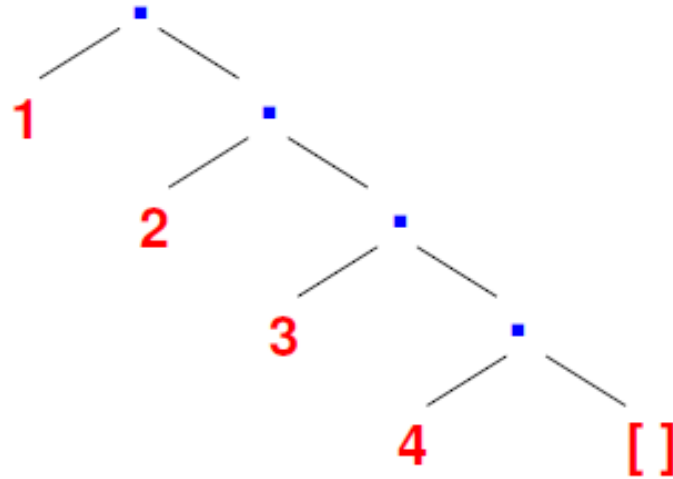
Then we only get: `D = 1.`

# Lists

- Prolog uses a special syntax to represent and manipulate lists (syntactic sugar = internally, it uses structures):
  - $[1,2,3,4]$ : represents a list with 1, 2, 3 and 4, respectively.
  - This can also be written as  $[1 \mid [2,3,4]]$ : a list with 1 as the *head* (first element) and  $[2,3,4]$  as its *tail* (the list of remaining elements).
  - If  $X = 1$  and  $Y = [2,3,4]$  then  $[X \mid Y]$  is same as  $[1,2,3,4]$ .
  - The empty list is represented by  $[]$  or *nil*.
  - The symbol " $\mid$ " (called *cons*) and is used to separate the beginning elements of a list from its tail.
  - For example:  $[1,2,3,4] = [1 \mid [2,3,4]] = [1 \mid [2 \mid [3,4]]] = [1,2 \mid [3,4]]$

# Lists

- Lists are special cases of trees (i.e., (syntactic sugar = internally, it uses 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, []))))`

# Programming with Lists

- First example: `member/2`, to find 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)]) .
```

# Programming with Lists

- append/3: concatenate two lists to form the third list:
  - The program:

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

```
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 example

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

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

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

# Append example

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

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

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

# Append example

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

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

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

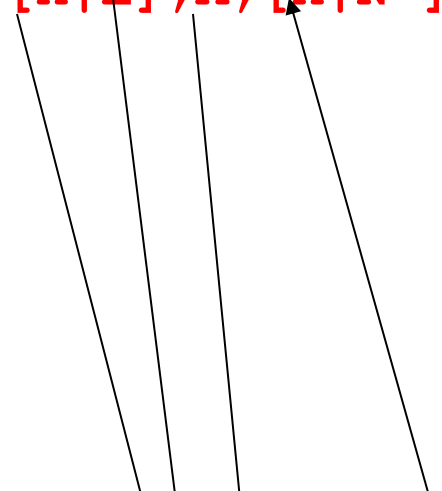
`append([1, 2], [3, 4], X) ?`

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

# Append example

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

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



The diagram shows four arrows originating from the recursive call `append(L, M, N')` in the second line of code. The first arrow points to the first row of the table, the second to the second row, the third to the first row, and the fourth to the second row.

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

# Append example

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

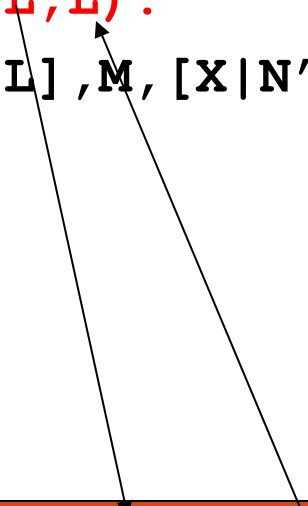
**append**( [X|L] , M, [X|N' ] ) :- **append**( L, M, N' ) .

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

# Append example

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

# Append example

`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]`

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

# Programming with Lists

- `len/2` finds the length of a list (first argument).:

- The program:

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

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

- Example queries:

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

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

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



# Arithmetic

?-  $1 + 2 = 3$ .

*no*

- 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 to 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) .
```

```
?- length(List, 4) .
```

# Conditional Evaluation

- Consider the computation of  $n!$  (i.e. the factorial of  $n$ )

**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**

**).**

# Therefore, Prolog Syntax:

- Assignments with arithmetic expressions is done using the keyword "is".
- If-then-else is written as  
( cond -> then-part ; else-part )
- If more than one action needs to be performed in a rule, they are written one after another, separated by a comma.
- Arithmetic expressions are not directly used as arguments when calling a predicate; they are first evaluated, and then passed to the called predicate.

# Arithmetic Operators

- Integer/Floating Point operators:  $+$ ,  $-$ ,  $*$ ,  $/$ 
  - Automatic detection of Integer/Floating Point
- Integer operators:  $\text{mod}$ ,  $//$  (div)
- Int  $\leftrightarrow$  Float operators: floor, ceiling
- Comparison operators:  $<$ ,  $>$ ,  $=<$ ,  $>=$ ,  
 $Expr1 ::= Expr2$  (succeeds if expression  
 $Expr1$  evaluates to a number equal to  $Expr2$ ),  
 $Expr1 =\backslash = Expr2$  (succeeds if expression  
 $Expr1$  evaluates to a number non-equal to  $Expr2$ )

# Programming with Lists

- Define `delete/3`, to remove a given element from a list (called `select/3` in XSB's basics library):
  - E.g. `delete([1,2,3], 2, X)` should succeed with `X = [1,3]`.

# Programming with Lists

- E.g. `delete([1,2,3], 2, X)` should succeed with `X = [1,3]`.

- The program:

```
delete([X|Ys], X, Ys).
```

```
delete([Y|Ys], X, [Y|Zs]) :-  
    delete(Ys, X, Zs).
```

- Example queries:

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

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

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

```
?- delete(X, Y, [1,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]`?

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

# Programming with Lists

- The program:

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

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

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

```
    delete(Ys, X, 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]`?

**`rev([], []).`**

**`rev([X|Xs], Ys) :- rev(Xs, Zs),  
append(Zs, [X], Ys).`**

- How long does it take to evaluate `rev([1, 2, ..., n], X)`?
  - $T(n) = T(n - 1) + \text{time to add 1 element to the end of an } n - 1 \text{ element list}$   
 $= T(n - 1) + n - 1 = T(n - 2) + n - 2 + n - 1 = \dots$
  - $\rightarrow T(n) = O(n^2)$

# Making rev/2 faster

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

- The program:

```
rev(L1, L2) :- rev(L1, [], L2).
```

```
rev([X|Xs], AccBefore, AccAfter) :-  
    rev(Xs, [X|AccBefore], AccAfter).
```

```
rev([], Acc, Acc). % Base case
```

- Example query:

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

```
    which calls rev([2,3], [1], X)
```

```
    which calls rev([3], [2,1], X)
```

```
    which calls rev([], [3,2,1], X)
```

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

# Tree Traversal

```
preorder(Root, [Root]) :- leaf(Root).  
preorder(Root, [Root|L]) :-  
    node(Root, Child1, Child2),  
    preorder(Child1, L1),  
    preorder(Child2, L2),  
    append(L1, L2, L).
```

- 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”

# Difference Lists

- When  $X = [1, 2, 3 \mid Y]$ ,  $X$  is a list with 1, 2, 3 as its first three elements, followed by  $Y$ .
  - Now if  $Y = [4 \mid Z]$  then  $X = [1, 2, 3, 4 \mid Z]$ .
  - We can think of  $Z$  as “pointing to” the end of  $X$ .
  - **We can now add an element to the end of  $X$  in constant time!!**
    - (e.g.  $Z = [5 \mid W]$ )
- Open-tailed lists are also called *difference lists* in Prolog.



# 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

(Chap. 8.5.3 of Bratko)

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

# Difference Lists: Conventions

- The preorder traversal program may be written as:

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

# 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

- Path from one node to another one:

```
path(G,A,B,P) :- path1(G,A,[B],P).
```

```
path1(_,A,[A | P1],[A | P1]).
```

```
path1(G,A,[Y | P1],P) :-  
    adjacent(X,Y,G),  
    \+ member(X,[Y | P1]),  
    path1(G,A,[X,Y | P1],P).
```

# Graphs in Prolog

- Acyclic graph path:

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

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

# Graphs in Prolog

- Cycle from a given node:
  - 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(B,A,G),  
    path(G,A,B,P1),  
    length(P1,L),  
    L > 2,  
    append(P1,[A],P).
```



# Logical Puzzles

- Eight queens problem:
  - place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.
  - We represent the positions of the queens as a list of numbers  $1..N$  (e.g.,  $[4,2,7,3,6,8,5,1]$  means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc.)

# Logical Puzzles

- Eight queens problem:
  - place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.
  - We represent the positions of the queens as a list of numbers  $1..N$  (e.g.,  $[4, 2, 7, 3, 6, 8, 5, 1]$  means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc.)

# Logical Puzzles

- Eight queens problem:
  - using the permutations of the numbers  $1..N$  we guarantee that no two queens are in the same row
  - The only test that remains to be made is the diagonal test

# Logical Puzzles

- Eight queens problem:

queens(N,Qs) :- range(1,N,Rs), perm(Rs,Qs), test(Qs).

% range(A,B,L) :- L is the list of numbers A..B

range(A,A,[A]).

range(A,B,[A | L]) :- A < B, A1 is A+1, range(A1,B,L).

% perm(Xs,Zs):-the list Zs is a permutation of the list Xs

perm([],[]).

perm(Qs,[Y | Ys]) :- del(Y,Qs,Rs), perm(Rs,Ys).

del(X,[X | Xs],Xs).

del(X,[Y | Ys],[Y | Zs]) :- del(X,Ys,Zs).

# Logical Puzzles

- Eight queens problem:

`% test(Qs) :- Qs is a non-attacking queens solution`

`test(Qs) :- test(Qs,1,[],[]).`

`% test(Qs,X,Cs,Ds) :- the queens in Qs, representing  
columns X to N, are not in conflict with the diagonals Cs  
and Ds`

`test([],_,_,_).`

`test([Y | Ys],X,Cs,Ds) :-`

`C is X-Y, \+ memberchk(C,Cs),`

`D is X+Y, \+ memberchk(D,Ds),`

`X1 is X + 1, test(Ys,X1,[C | Cs],[D | Ds]).`

# Aggregates in XSB

- `setof(?Template, +Goal, ?Set)` : ?Set is the set of all instances of Template such that Goal is provable.
- `bagof(?Template, +Goal, ?Bag)` has the same semantics as `setof/3` except that the third argument returns an unsorted list that may contain duplicates.
- `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.
- `tfindall(?Template, +Goal, ?List)` is similar to predicate `findall/3`, but the Goal must be a call to a single tabled predicate.

# XSB Prolog

- Negation: *not* ( $\backslash +$ ): 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:
    - ?- see('a.txt'), read(X), seen.
    - ?- tell('a.txt'),  
write('Hello, World!'), told.



# 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.**

- **Green cut**

`gamble(X) :- gotmoney(X), !.`

`gamble(X) :- gotcredit(X), \+ gotmoney(X).`

- **cut says “stop looking for alternatives”**
- by explicitly writing `\+ gotmoney(X)`, it guarantees that the second rule will always work even if the first one is removed by accident or changed

- **Red cut**

`gamble(X) :- gotmoney(X), !.`

`gamble(X) :- gotcredit(X).`

# Cut (logic programming)

- Consider:

$p(a). p(b).$

$q(a). q(b). q(c).$

$?- p(X), !.$

$X=a ;$

no

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

$X=a Y=a ;$

$X=a Y=b ;$

$X=a Y=c ;$

# 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

- **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.

# Assert and retract

- **asserta(C)**

Assert clause C into database above other clauses with the same key predicate. The key predicate of a clause is the first predicate encountered when the clause is read from left to right.

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

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

- **retract(C)**

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

# Prolog terms and clauses

- **clause(H,B)**

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

- **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 Nth argument of E



# Prolog terms and clauses

- `=..`

converts between term and list. For example,

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

`L = [parent, a, _X001]`

# Prolog Programming

- **Recursion is king to Computational Problem Solving**
  - learn to express algorithmic ideas in an abstract manner
- Prolog Programming Contest:
  - <http://people.cs.kuleuven.be/~bart.demoen/PrologProgrammingContests/>
  - First 10 contests book:  
<https://dtai.cs.kuleuven.be/ppcbook/ppcbook.pdf>
  - Fun: the winning Stony Brook team in 1998:
- <http://people.cs.kuleuven.be/~bart.demoen/PrologProgrammingContests/contest98.html>



# Prolog Programming

- Patterns: Triangle (1995 Portland, USA)
  - Write a predicate `triangle/1`, which is called with its argument `N` instantiated to a non-negative integer, and which draws a triangle of size `N` on the screen:

The diagram illustrates the recursive construction of a triangle pattern of size 7. It shows the pattern for size 7 as the sum of the pattern for size 6 and the pattern for size 1.

Pattern for size 7 (left):

```

_ _ _ _ _ *
_ _ _ _ * *
_ _ _ * * *
_ _ * * * *
_ * * * * *
_ * * * * *
_ * * * * *

```

Pattern for size 6 (middle):

```

_ _ _ _ _ _
_ _ _ _ _
_ _ _ _
_ _ _
_ _
_

```

Pattern for size 1 (right):

```

*
*
*
*
*
*
*

```

The equation is represented as:  $\text{Pattern}_7 = \text{Pattern}_6 + \text{Pattern}_1$

# Prolog Programming

- Patterns: Triangle (1995 Portland, USA)

```
triangle(N) :- Stars = 1,
```

```
    triangle(N, Stars).
```

```
triangle(_).
```

```
triangle(Spaces, Stars) :-
```

```
    Spaces > 0, writeN(Spaces, ' '),
```

```
    writeN(Stars, '* '), nl,
```

```
    Spaces1 is Spaces - 1,
```

```
    Stars1 is Stars + 1,
```

```
    triangle(Spaces1, Stars1).
```

# Prolog Programming

- Actions: Star (1999 Santa Cruz, NM)
  - Write a predicate `star/1`, which is called with a list of commands as input, and which draws on the screen the picture corresponding to the list of commands. There are 5 different commands: up, down, right, left, star. Imagine there is a rectangular grid on the screen, and that you start off at some position. The command up obviously means that you move up one step in the grid on the screen. Similarly reasonable interpretations can be made for down, left and right. The star command means that you must write a `*` on the screen at the current position.

# Prolog Programming

- ?- star([down,star,left,down,right,up,star,right]).

a \* in the left column on the screen.

```
star(Commands) :-
    collect_stars(Commands,Stars),
    Stars = [star(StartCol,StartLine0)|RestStars],
    max_line(RestStars,StartLine0,StartLine),
    write_stars(Stars,StartLine,StartCol).

write_stars([],_,_).
write_stars(Lines,StartLine,StartCol) :-
    Lines = [_|_],
    write1line(Lines,StartLine,StartCol),
    findall(star(A,B),(member(star(A,B),Lines),
        B \== StartLine), RestLines),
    StartLine1 is StartLine - 1,
    write_stars(RestLines,StartLine1,StartCol).

write1line(Lines,StartLine,StartCol) :-
    findall(Col,member(star(Col,StartLine),Lines),Cols),
    sort(Cols,ColsS),
    writecols(ColsS,StartCol),
    nl.

writecols([],_).
writecols([X|R],Col) :-
    Col1 is Col + 1,
    ( Col == X ->
        write('*'),
        writecols(R,Col1)
    ;
        write(' ')
    )
```

# Prolog Programming

```
collect_stars(Commands,Stars) :-
    once(collect_stars(Commands,0,0,UnorderedStars)),
    sort(UnorderedStars,Stars).

collect_stars([],_,_,[]).
collect_stars([up | Commands],I,J,Stars) :-
    J1 is J + 1,
    collect_stars(Commands,I,J1,Stars).
collect_stars([down | Commands],I,J,Stars) :-
    J1 is J - 1,
    collect_stars(Commands,I,J1,Stars).
collect_stars([left | Commands],I,J,Stars) :-
    I1 is I - 1,
    collect_stars(Commands,I1,J,Stars).
collect_stars([right | Commands],I,J,Stars) :-
    I1 is I + 1,
    collect_stars(Commands,I1,J,Stars).
collect_stars([star | Commands],I,J,[star(I,J) | Stars]) :-
    collect_stars(Commands,I,J,Stars).

max_line([],Max,Max).
max_line([star(_,L) | RestStars],MaxSoFar,Max) :-
    ( L > MaxSoFar ->
        max_line(RestStars,L,Max)
    ;
        max_line(RestStars,MaxSoFar,Max)
    ).
```

# Prolog Programming

- Puzzles: M-Queens (2001 Paphos, Cyprus):
  - Find all maximally safe configurations of queens on an  $M \times M$  chess board: a configuration is maximally safe if no queen can be added without the configuration becoming unsafe (a queen attacking another queen).
  - Write a predicate `mqueens/2`, which, given as input a number, unifies the second argument with all maximally safe configurations through backtracking.



# Prolog Programming

?- mqueens(1,L).

L = [1]

there is queen in the first column in row 1

?- mqueens(2,L).

L = [1,none]

there is queen in the first column in row 1

L = [2,none]

there is queen in the first column in row 2

L = [none,1]

there is no queen in the first column,  
there is queen in the second column in row 1

L = [none,2]

?- mqueens(3,L).

L = [1,3,none]   L = [1,none,2]   L = [2,none,1]   L = [2,none,3]

L = [3,1,none]   L = [3,none,2]   L = [none,1,3]   L = [none,2,none]

L = [none,3,1]

# Prolog Programming

```
mqueens(M,Solution) :-  
    findall(sq(A,B),(numlist(1,M,Rows),  
                    member(A,Rows),member(B,Rows)),Squares),  
    mqueens(M,Squares,[],Solution).
```

```
mqueens(M,NotAttacked,PartialSolution,Solution) :-  
    ( M == 0 ->  
        NotAttacked = [],  
        Solution = PartialSolution  
    ;  
        M1 is M - 1,  
        (  
            member(sq(M,X),NotAttacked),  
            safe(PartialSolution,1,X),  
            delete_attacked(NotAttacked,M,X,NotAttacked1)  
        ;  
            X = none,  
            NotAttacked1 = NotAttacked  
        ),  
        mqueens(M1,NotAttacked1,[X|PartialSolution],Solution)  
    ).
```

# Prolog Programming

```
safe([],_,_).
safe([X|R],Dist,Y) :-
    ( X == none ->
        true
    ;
        X \= Y,
        abs(X - Y) \= Dist
    ),
    Dist1 is Dist + 1,
    safe(R,Dist1,Y).
```

```
delete_attacked([],_,_,[]).
delete_attacked([sq(A,B)|R],I,J,Squares) :-
    ( (A == I ; B == J ; A-B == I-J ; A+B == I+J) ->
        delete_attacked(R,I,J,Squares)
    ;
        Squares = [sq(A,B)|S],
        delete_attacked(R,I,J,S)
    ).
```

# Prolog Programming

- Puzzles: M-Queens (2001 Paphos, Cyprus):
  - Find all maximally safe configurations of queens on an  $M \times M$  chess board: a configuration is maximally safe if no queen can be added without the configuration becoming unsafe (a queen attacking another queen).
  - Write a predicate `mqueens/2`, which, given as input a number, unifies the second argument with all maximally safe configurations through backtracking.