Programming in Prolog

CSE 505 – Computing with Logic

Stony Brook University

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

Predicates/Relations

- ***Predicates*** (aka. ***relations***) are building-blocks in predicate calculus: **`p(`**$\mathbf{a_1, a_2, \ldots, a_k}$**`)`**
  - **`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

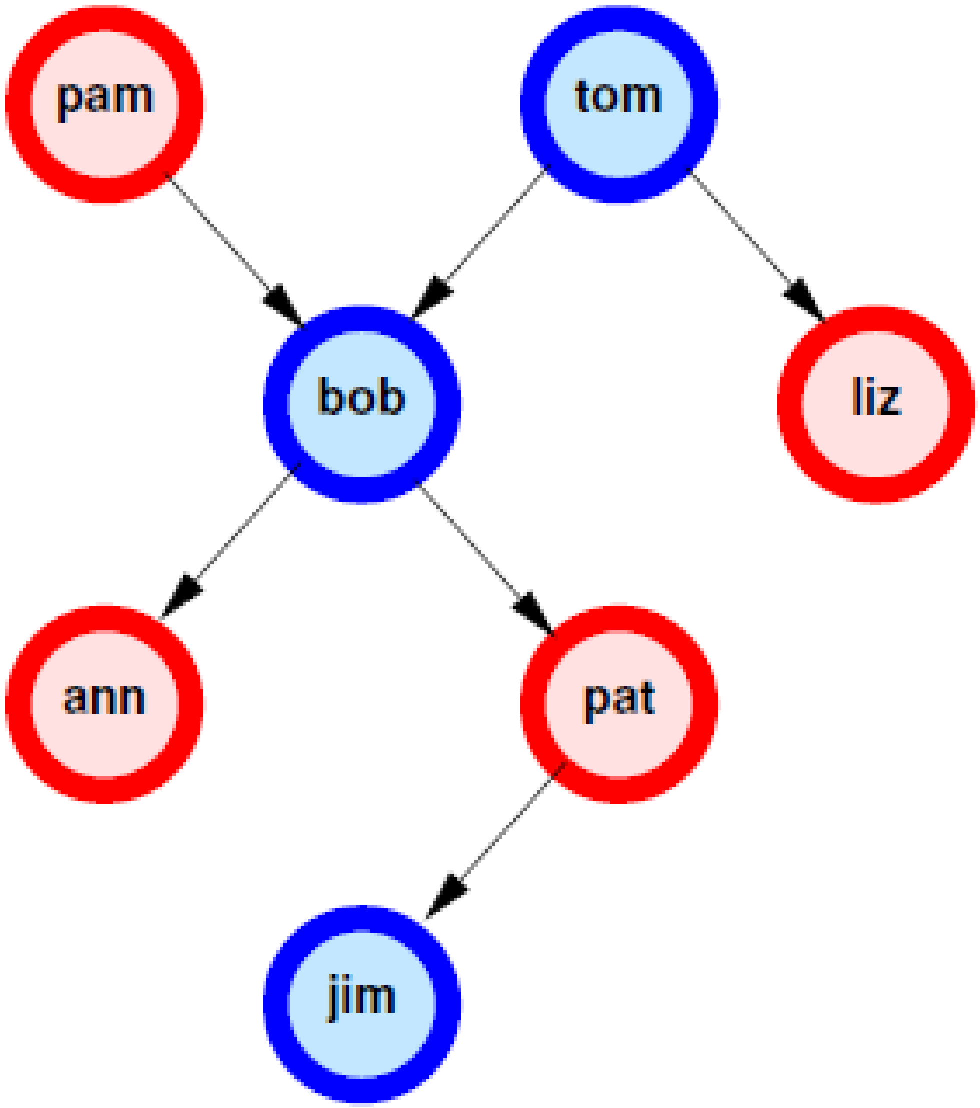
Predicates

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

```
female(pam).
female(pat).
female(ann).
female(liz).
```

Predicates

```
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).
```



Rules

- *Rules*: Consider the predicate **`mother(X, Y)`**: X is the mother of Y.

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

$\forall X,Y\ (parent(X,Y) \wedge female(X) \rightarrow mother(X,Y))$

-In Prolog:

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

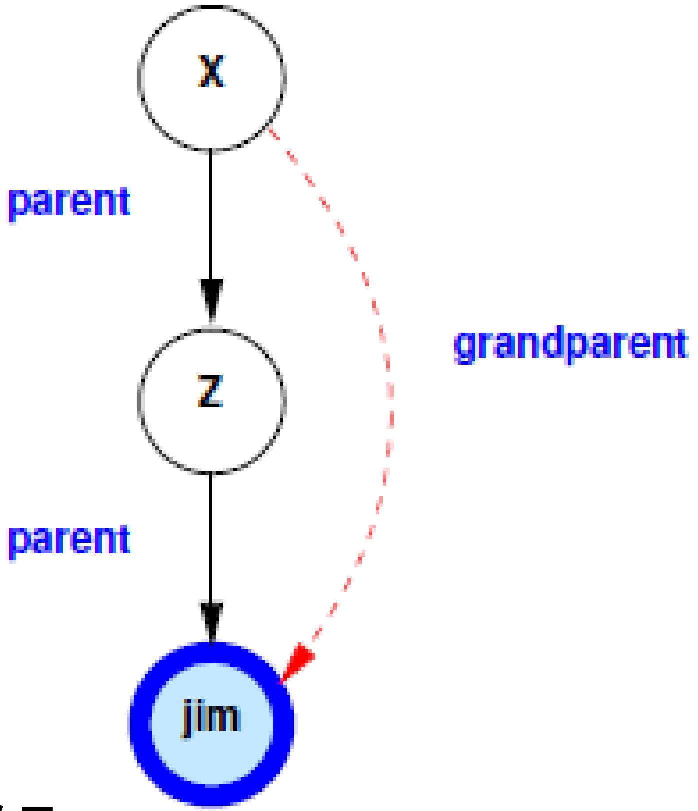
  - all variables are universally quantified outside the rule
  - "," means *and* (conjunction), ":-" means *if* (implication) and ";" means *or* (disjunction).

Rules

- More rules:

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

can be read in two ways:



- For all **X**,**Y** and **Z**, if **X** is a parent of **Z** and **Z** is a parent of **Y**, then **X** is a grandparent of **Y**.

is logical equivalent with:

- For all **X** and **Y**, **X** is a grandparent of **Y** if there is some **Z** such that **X** is a parent of **Z** and **Z** is a parent of **Y**.

Rules

$\forall X,Y,Z\ (parent(X,Z) \wedge parent(Z,Y) \rightarrow grandparent(X,Y))$

$\equiv \forall X,Y,Z\ (\sim(parent(X,Z) \wedge parent(Z,Y)) \vee grandparent(X,Y))$

$\equiv \forall X,Y\ (\forall Z \sim(parent(X,Z) \wedge parent(Z,Y)) \vee grandparent(X,Y))$

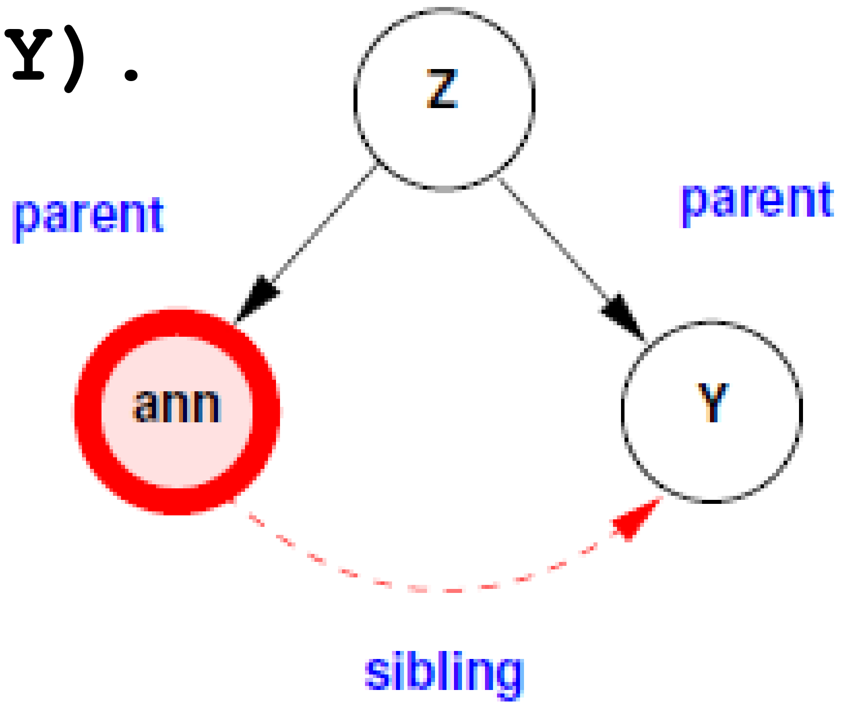
since Z is not in grandparent(X,Y) we can the universal quantifier pass the disjunction

$\equiv \forall X,Y\ (\sim\exists Z(parent(X,Z) \wedge parent(Z,Y)) \vee grandparent(X,Y))$

$\equiv \forall X,Y\ ((\exists Z(parent(X,Z) \wedge parent(Z,Y))) \rightarrow grandparent(X,Y))$

Rules

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

?- sibling(ann,Y).
```



Rules

- More rules:

```
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).
```

Recursion

- What about:

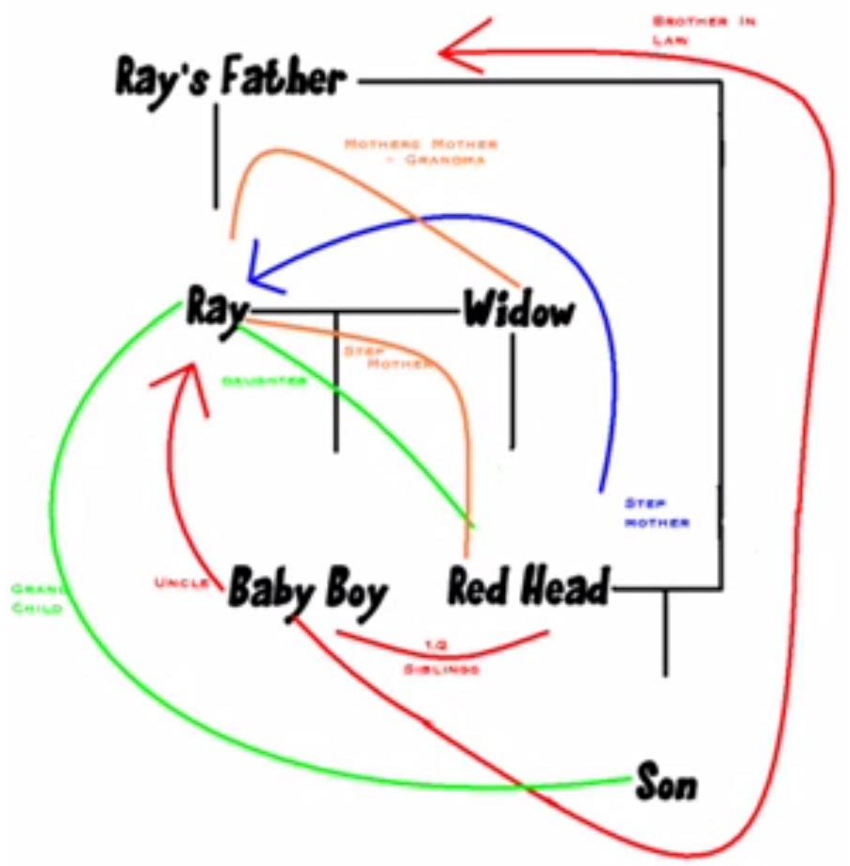
```
ancestor(X,Y) :-
    ancestor(X,Z),
    parent(Z,Y).
ancestor(X,Y) :-
    parent(X,Y).

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

INFINITE LOOP

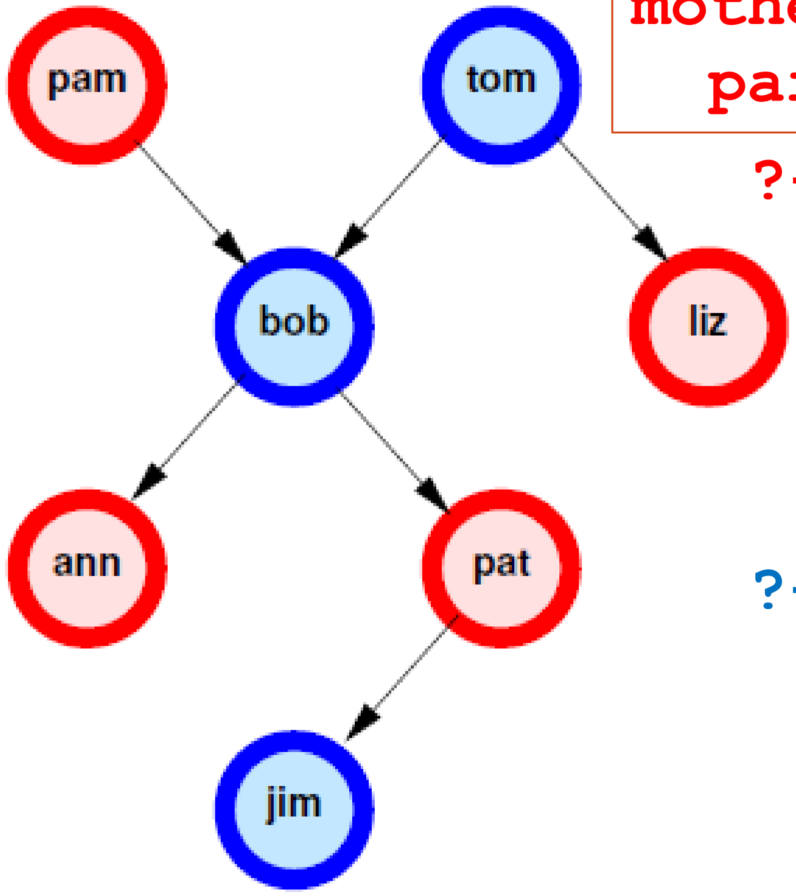
Rules

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

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



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

```
father(X,Y):-
   parent(X,Y),male(X).
```

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).` returns `X=1; X=2; X=b`

  `?- p(1).` returns `true`

- 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:

```
| ?- p(X).
X = 1;
X = 2
yes
```

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

```
| ?- (p(_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!

Logic Programming Concepts

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

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

- 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)
  - Rules (i.e., clauses 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**.
    - **Single-assignment variables: X must have the same value on both sides.**

- Prolog provides an automatic way to deduce true results from facts and rules:
  - ***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).
employed(X) :- self_employed(X).
```

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., in 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 in LP

- 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`**:

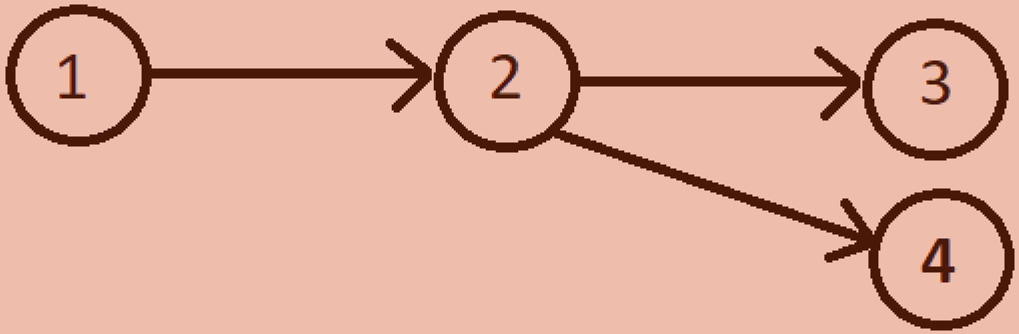
```
reach(X,Y)  :-  edge(X,Y).
```

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;
  X = 2
  Y = 3;
  X = 2
  Y = 4;
  X = 1
  Y = 3;
  X = 1
  Y = 4;
  no
```

← Type a semi-colon repeatedly for more answers



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

Prolog Syntax

- 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, <u>beginning with a lower case letter</u> (e.g. **`paul`**, **`r2d2`**, **`one_element`**).
  - Strings of characters enclosed in <u>single quotes</u> (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 is does not appear multiple times, then a warning is produced: "*Singleton variable*"
    - You can use variables preceded with underscore to eliminate this warning

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

- ***Anonymous variables*** (also called *Don't care variables*): variables **beginning with "_"**
  - Underscore, by itself (i.e., _), represents a variable

```
has_a_child(X) :- parent(X,_).
```

    - Each occurrence of _ corresponds to a different variable; even within a clause, _ does not stand for one and the same object

```
somebody_has_a_child :- parent(_,_).
```

  - 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), but are not returned in the head of the clause (we don't need to use _ in this case since it is not singleton variable): `r :- p(_X), q(_X).`

Variables

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

```
?- X=f(Y),
      Y=g(Z),
      Z=2.
```

Therefore, `X=f(g(2)), Y=g(2), Z=2`

- The order also does not matter (nether in the rule or in the matching/unification order):

```
?- 2=Z,
      f(Y)=X,
      g(Z)=Y.
```

Therefore, `X = f(g(2)), Y=g(2), Z=2`

- Even infinite structures:

```
?- X=f(X).
X=f(f(f(f(f(f(f(f(f(f(...))
```

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

- **Semantics:**
  - **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?
    - That is, 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 **H :- B**$_1$**,B**$_2$**, …, B**$_n$**.** is an instance of a clause in the program such that **B**$_1$**,B**$_2$**, …, 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)** in addition to the original facts.

L = {**brown(bear), big(bear), gray(elephant), big(elephant), black(cat), small(cat), dark(cat), dark(bear), dangereous(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,…,G_n.**
- To *prove* **G_1,G_2,…,G_n**:
  - Find a clause **H :- B_1,B_2, …, B_k** such that **G_1** and **H** match
  - Under the ***substitution*** for variables, prove **B_1,B_2,…,B_k,G_2,…,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).
```

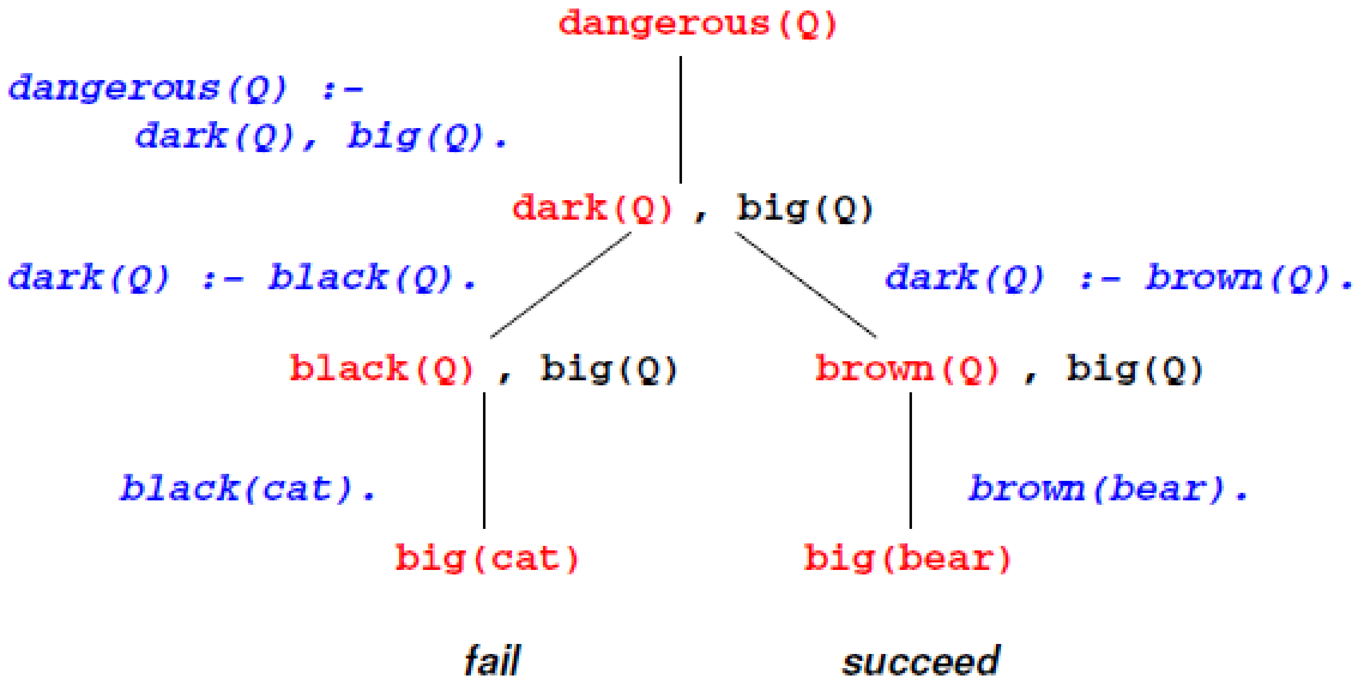
- To prove: `?- dangerous(Q).`

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).
```



Procedural Meaning of Prolog

- This procedural meaning/semantics of Prolog interpreters 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 systems: Answer set programming, DLV, Potassco (the Potsdam Answer Set Solving Collection: clasp, clingo), OntoBroker
    - They have very good applications in scheduling and planning (similar to constraint solving)

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 <u>beginning of your database</u> (**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

Order of clauses

- Danger of indefinite looping – consider:

```
p :- p.
```

says that **`p`** is true if **`p`** is true.

- This is declaratively correct, but procedurally is quite useless and causes problems to Prolog: the question **`?- p.`** will loop infinitely: the goal **`p`** is replaced by the same goal **`p`**; this will be in turn replaced by **`p`**, etc.

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).
```

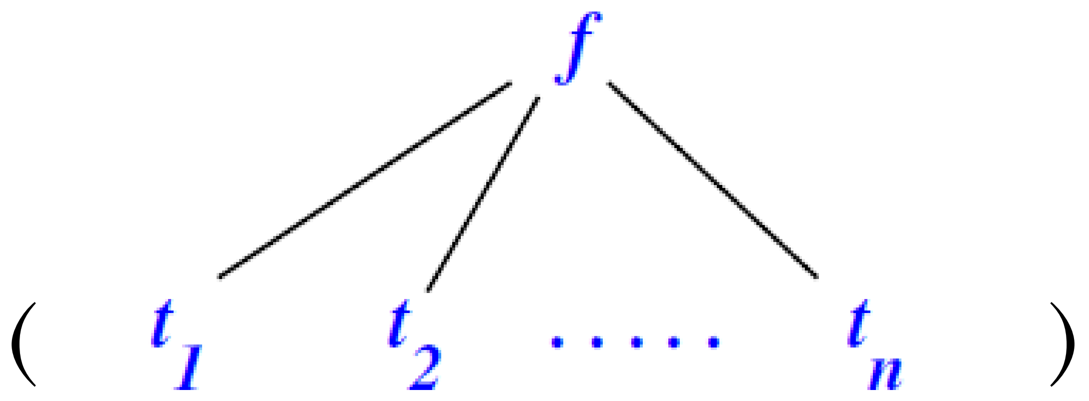
- Query:

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

Infinite loop (it does not matter what edges you have in the DB, they will never be used)

Structures

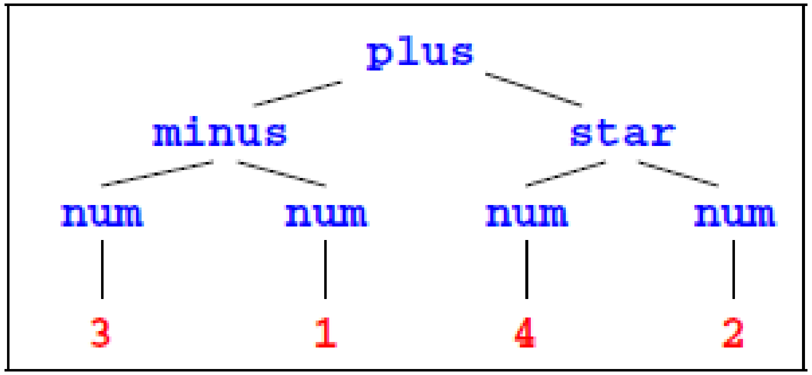
- If **`f`** is an identifier and **`t`**$_1$**`,`** **`t`**$_2$**`,`** …**`,`** **`t`**$_n$ are terms, then **`f(t`**$_1$**`,`** **`t`**$_2$**`,`** …**`,`** **`t`**$_n$**`)`** is a term



- In the above, **`f`** is called a ***functor*** and **`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**)

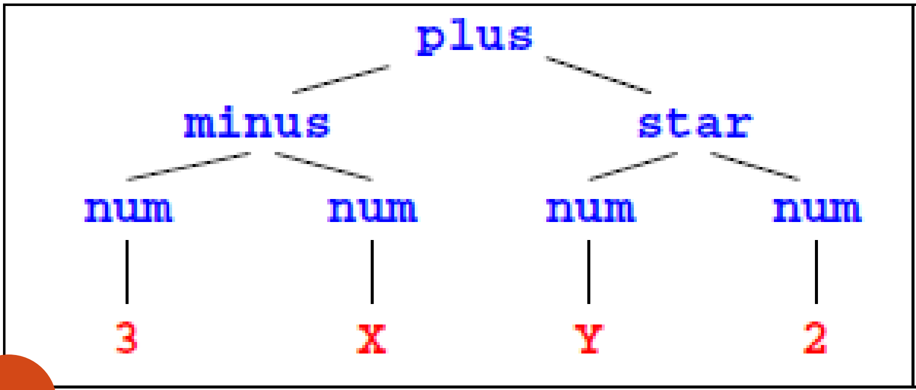
Trees

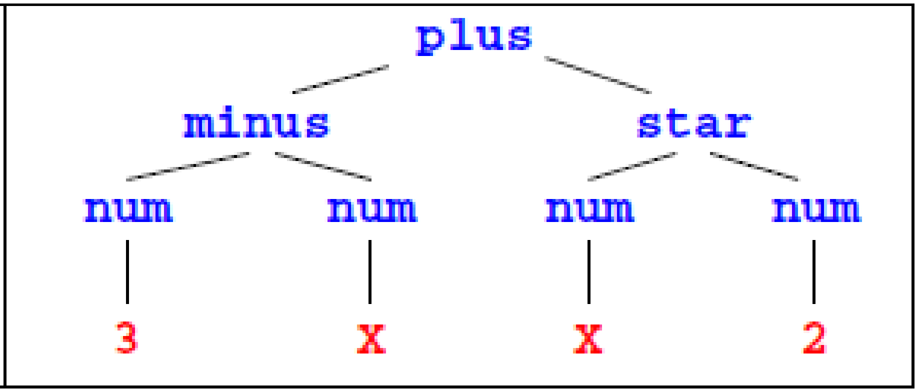
- Example: expression trees:



```
?- E = 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

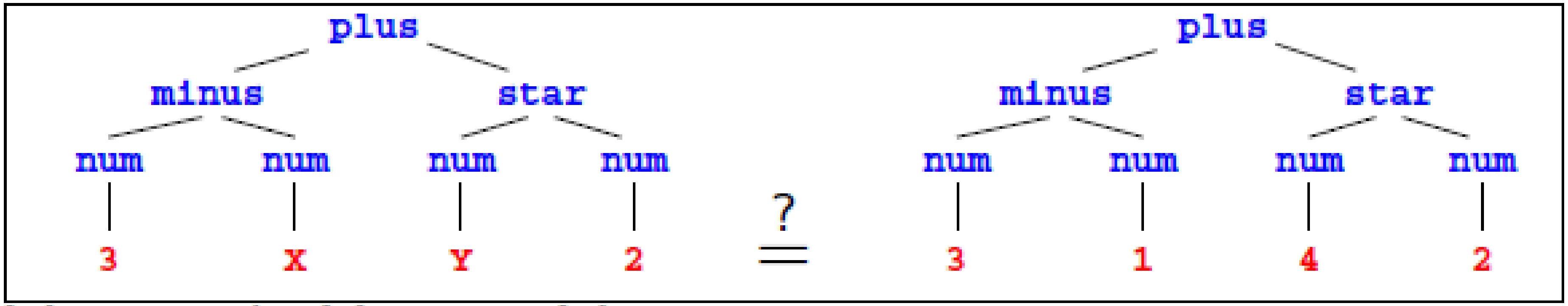




Matching

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

```
?- plus(minus(num(3),num(X)),star(num(Y),num(2))) =
plus(minus(num(3),num(1)),star(num(4),num(2))).
```



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

Matching through Examples

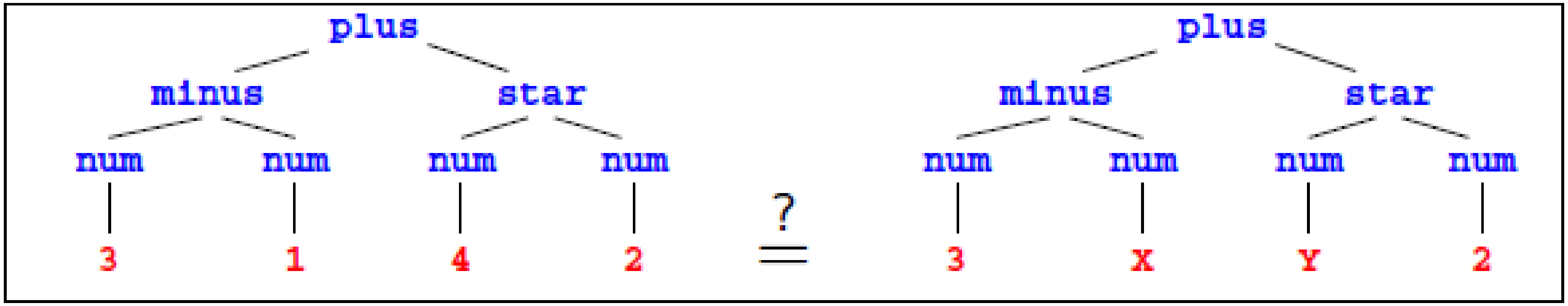
- Matching examples: given two terms, we can ask if they "*match*" each other
  - A constant matches with itself: **`42`** unifies with **`42`**
  - A variable matches with anything:
    - if it matches with something other than a variable, then it instantiates,
    - if it matches with a variable, then 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 Algorithm

- 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 Examples

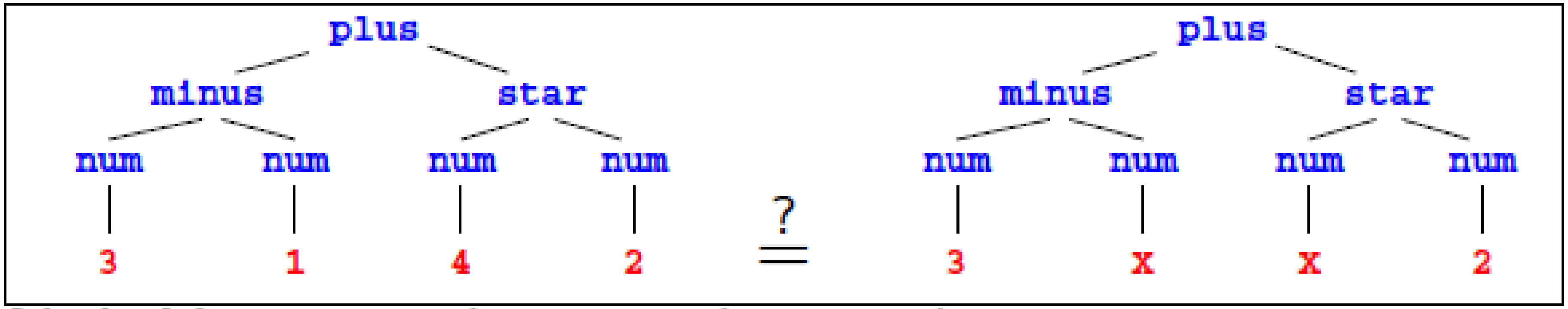
```
?- plus(minus(num(3),num(1)),star(num(4),num(2)))
= plus(minus(num(3),num(X)),star(num(Y),num(2))).
```



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

Matching Examples

```
?- plus(minus(num(3),num(1)),star(num(4),num(2)))
= plus(minus(num(3),num(X)),star(num(X),num(2))).
```



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

Matching Examples

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

```
?- A = 100.
   A = 100
   Yes
?- A = func(B).
   A = func(_123)
   Yes
```

```
?- func(100) = func(B,C).
       No
```

Matching Examples

- 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('Jan', 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('Jan',1,2020).`

makes

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

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

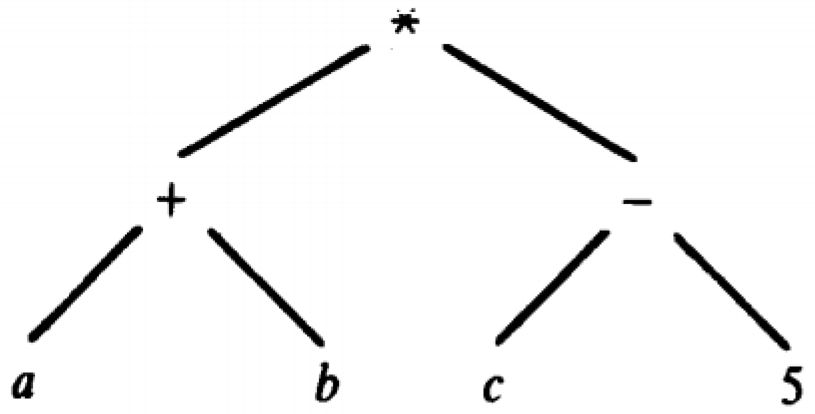
```
?-date(_, D, _) = date('Jan', 1, 2020).
```

Then we only get: `D = 1`

Operators are also functors

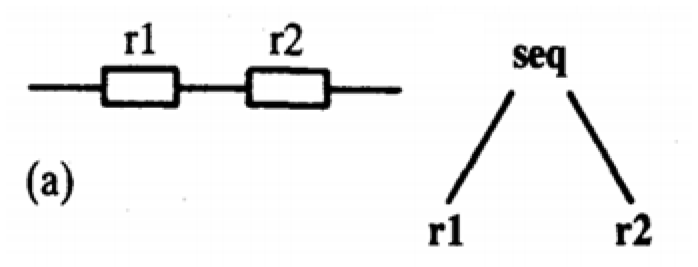
- All structured objects in Prolog are trees – including **unevaluated** arithmetical expressions:

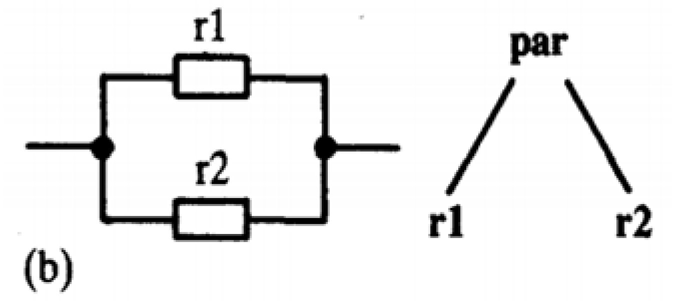
```
?- E = (a+b)*(c-5).
    E = *( +( a, b), -( c, 5) )
    yes
```

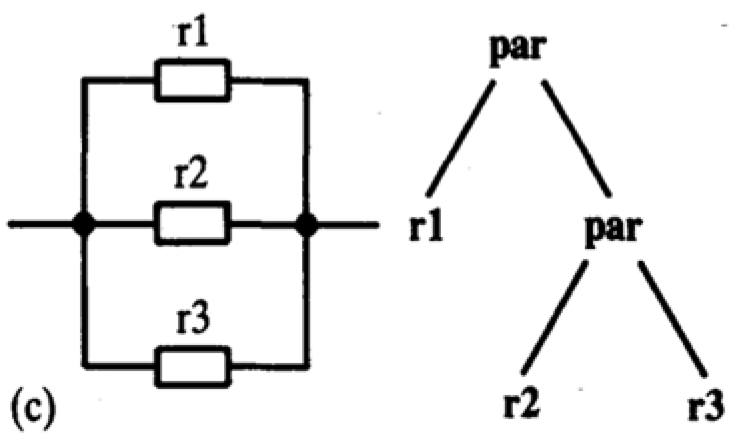


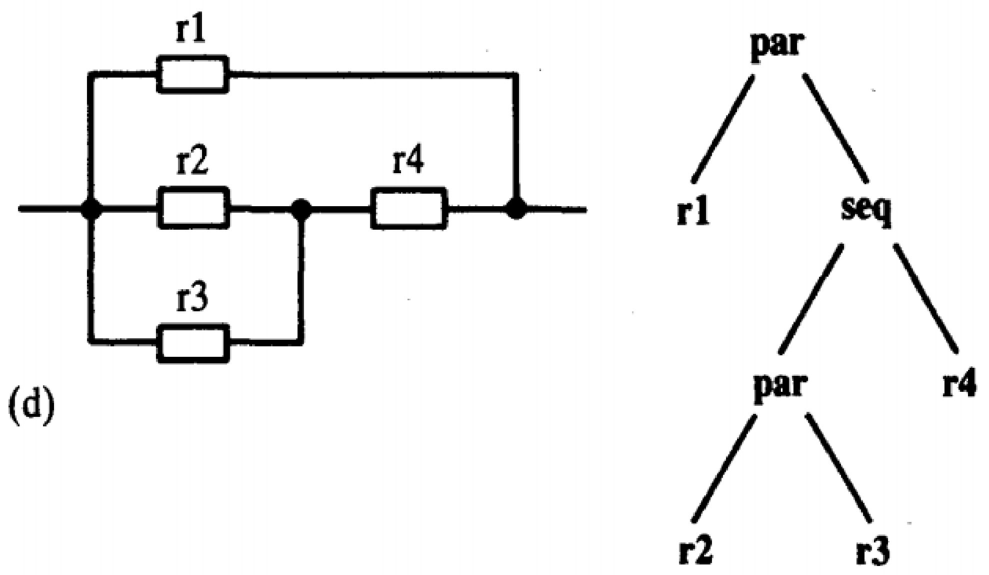
Note: arithmetical operators are not evaluated by default – we will see this later.

Other Structure Examples: Simple electronic circuits





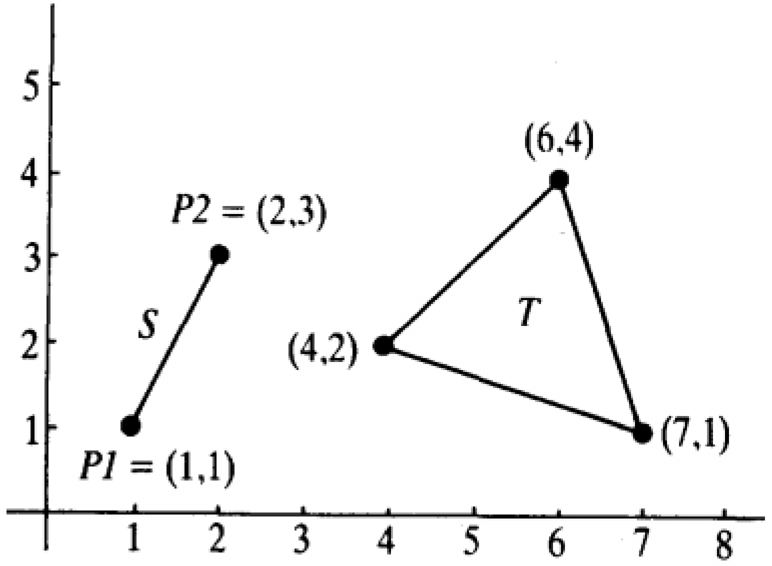




Simple geometric objects

- Let us choose the following functors: **`point`** for points, **`seg`** for line segments, and **`triangle`** for triangle

```
?- P1 = point(1,1),
 P2 = point(2,3),
 S = seg(P1,P2),
 T = triangle(point(4,2),
  point(6,4), point(7,1)).
```



```
S = seg(point(1,1), point(2,3))
```

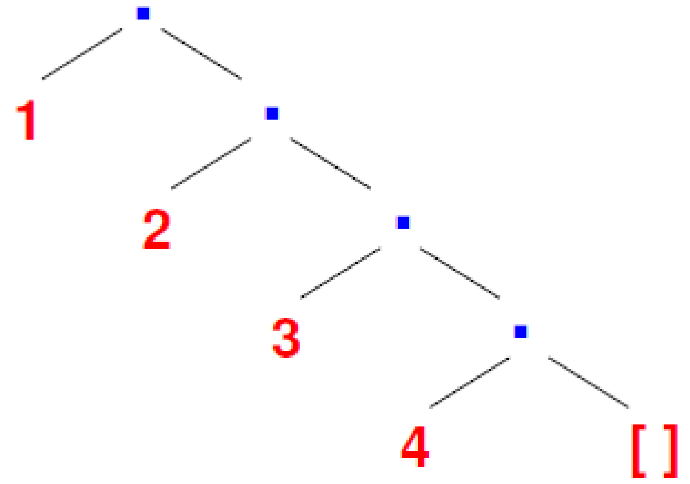
  - In three-dimensional space then we could use another functor: **`point3(X,Y,Z)`** different from **`point`** because each functor is defined by two things: name and arity

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 `X = 1` and `Y = [2,3,4]` then `[X|Y]` is same as `[1,2,3,4]`.
  - The empty list is represented by `[]` or `nil`
  - The symbol "`|`" (called ***pipe***) and is used to separate the beginning elements of a list from its tail
    - As opposed to functional programming languages like SML, Prolog's pipe operator can have multiple elements in the prefix of the list, e.g., `[1,2|[3,4]]`
    - 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 (a 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 <u>equivalent to a list of the (numeric) character codes</u>:

```
?- 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,[l,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.

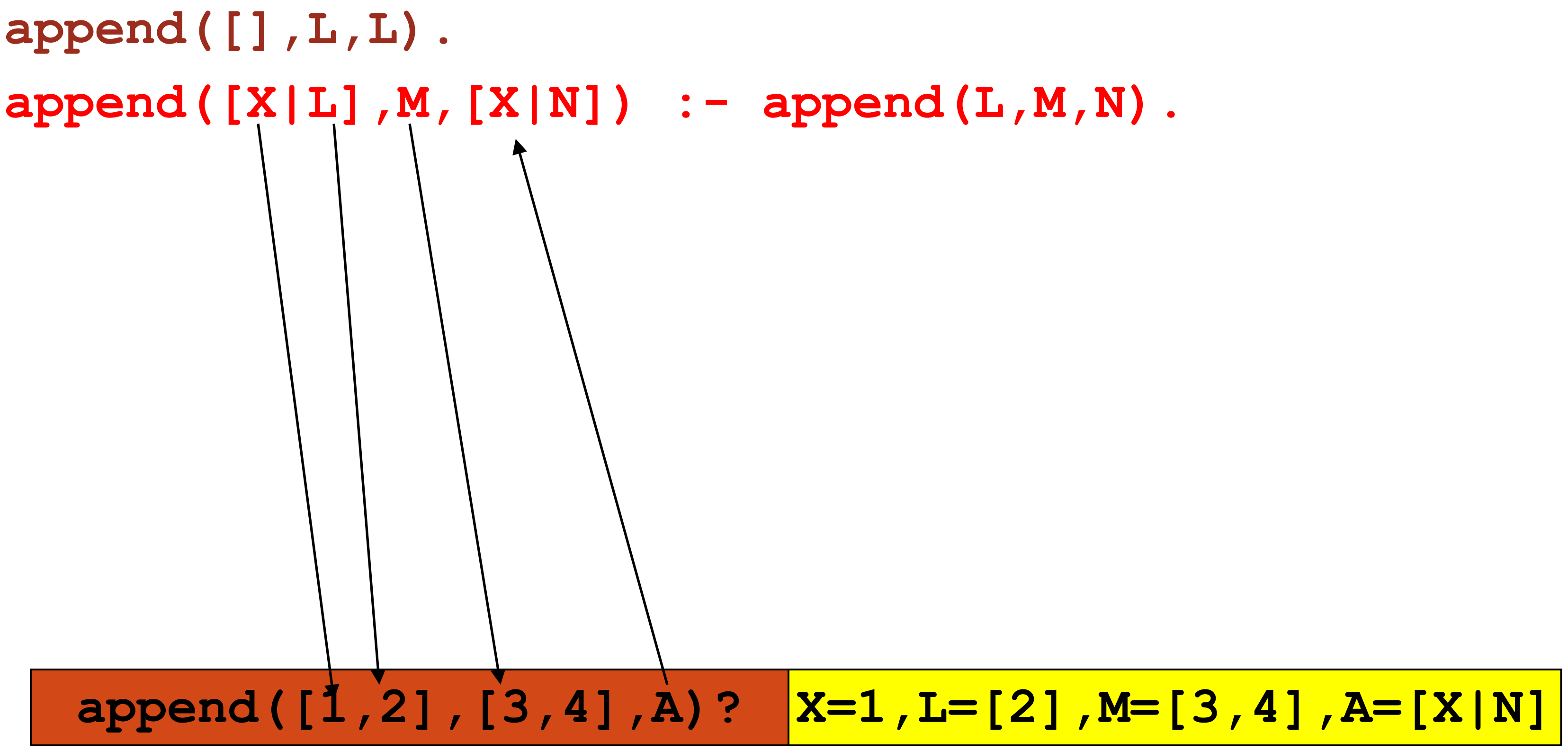
```
?- append(L, [2, 3], [1, 2, 3]).
     L = [ 1 ]
?- append([ 1 ], L, [1, 2, 3]).
     L = [2, 3]
?- append(L1, L2, [1, 2, 3]).
  L1 = [],              L2 = [1, 2, 3];
  L1 = [1],             L2 = [2, 3];
  L1 = [1, 2],          L2 = [3] ;
  L1 = [1, 2, 3],       L2 = [];
  no
```

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).
```

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

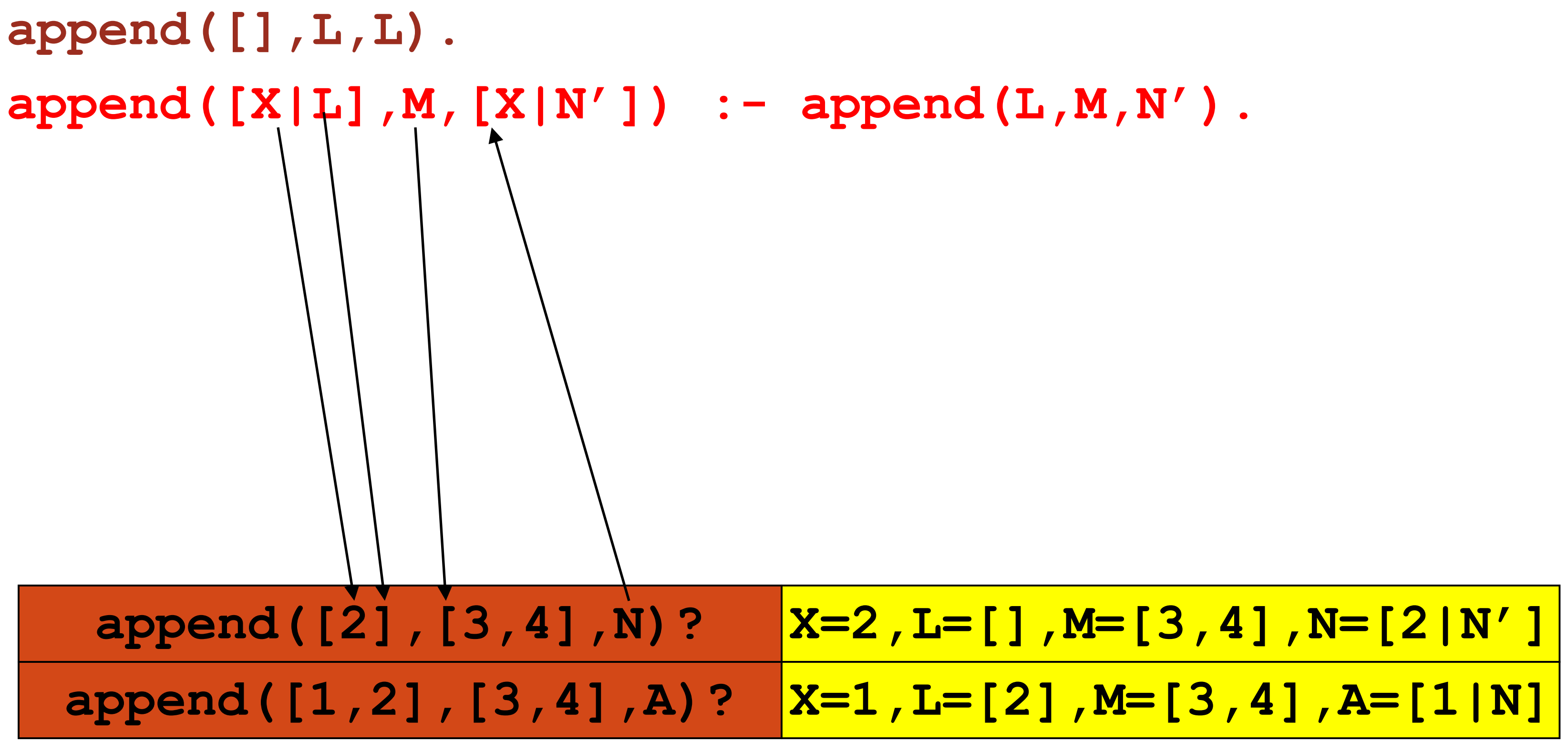
```
X=1,L=[2],M=[3,4],A=[X|N]
```

Append example trace

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

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’).
```

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]

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)?	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]

Append example trace

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

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]

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
      ).
```

Query:

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

Conditional Evaluation

- Standard logic priorities of operators apply: "," (conjunction) has higher priority than ";" (disjunction)

```
P:- Q;R.
```

is read: **P** is true if **Q** is true or **R** is true.

  - The meaning of the clause is the same as the meaning of the following two clauses together:

```
P:- Q.
P:- R.
```

- The comma binds stronger than the semicolon, so the clause

  `P:- Q,R; S,T,U.` `%`Parenthesis can be used for grouping

is understood as **`P :- (Q,R);(S,T,U).`** and means the same as the clauses: **`P:- Q,R.`**

```
P:- S,T,U.
```

Conditional Evaluation

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

```
% factorial(+N, -F)
factorial(N, F) :- ...
```

Comments in Prolog: **`N`** is the input parameter ("+") and **`F`** is the output parameter ("-"). "?" is used when the parameter can be either input or output.

- The body of the rule species how the output is related to the input:
  - For factorial, there are two cases: **`N <= 0`** and **`N > 0`**
    - if **`N <= 0`**, then **`F = 1`**
    - if **`N > 0`**, then **`F = N * 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 with 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

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

Expr1 =:= *Expr2* (succeeds if expression *Expr1* evaluates to a number equal to *Expr2*'s evaluated value),

Expr1 =\= *Expr2* (succeeds if expression *Expr1* evaluates to a number non-equal to *Expr2*'s evaluated value)

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, [2,1,2], X)`** should succeed with **`X=[1,2]; X =[2,1]; fail`**
  - **`delete(2, X, [1,3])`** should succeed with **`X=[2,1,3]; X =[1,2,3]; X=[1,3,2]; 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]).
```

Programming with Lists

- The program:

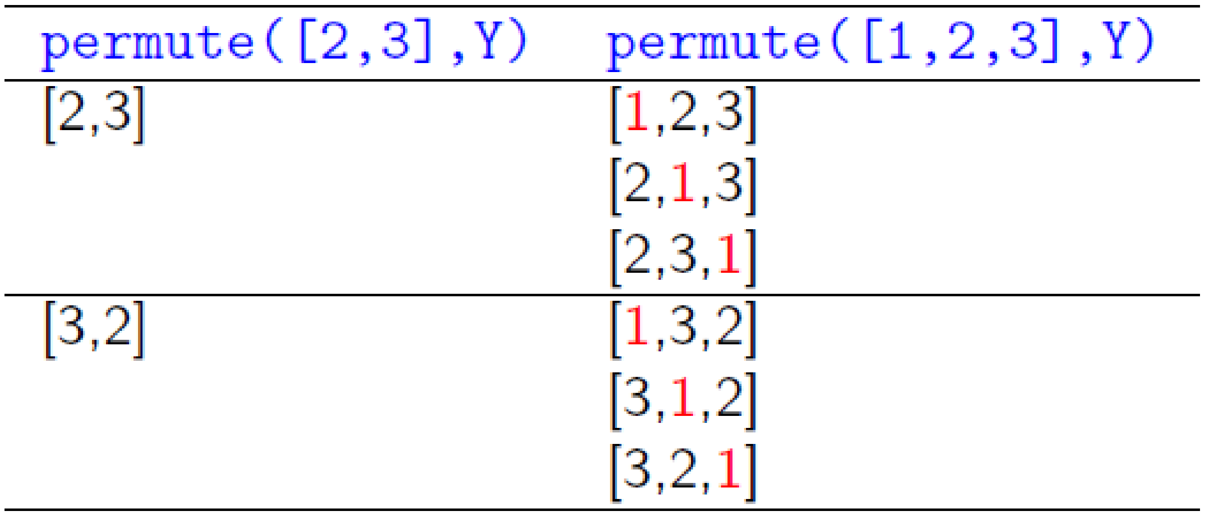
```
delete(X, [X|Ys], Ys).
delete(X, [Y|Ys], [Y|Zs]) :-
     delete(X, Ys, Zs).
```

- 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]`**?



permute([2,3],Y)	permute([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 **permute** program:

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

- Example query:

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

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 = ...`**

$\rightarrow$ **`T(n) = O(n`**^{2}**`)`** (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]
```

`T(n)=O(n)` (linear)

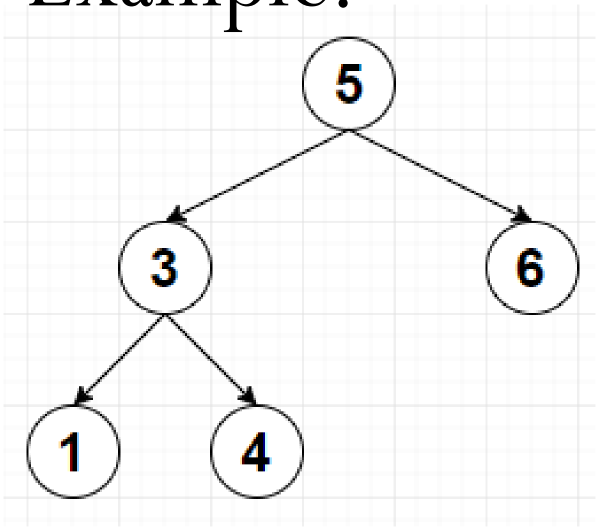
Palindrome

- Check if a list is a palindrome:

```
palindrome(X)  :-
      rev(X,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:

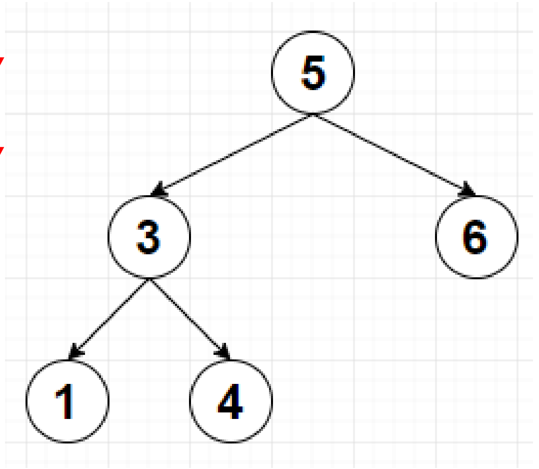


```
node(5, 3, 6).
node(3, 1, 4).
leaf(1).
leaf(4).
leaf(6).
```

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. **How to append 2 lists in shorter time?**

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 **`X=[1,2,3|Y]`**, **`X`** is a list with **`1`**, **`2`**, **`3`** as its first three elements, followed by **`Y`**
  - Now if **`Y=[4|Z]`** then **`X=[1,2,3,4|Z]`**
    - We can now think of **`Z`** as "pointing to" the end of **`X`**
  - **We can now add an element to the end of `X` in constant time!!**
    - And continue adding more elements, e.g. **`Z=[5|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. **`X = [1,2,3|Z], Z`**
- Most Prolog programmers use the notation **`List-Tail`** to denote a list **`List`** with tail **`Tail`**.
  - e.g. **`X-Z`**
  - Note that "**`-`**" is used as a <u>data structure infix symbol</u> (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).
?- 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([], X, [X]). % linear time to add w/out diff lists
add([H|T], X, [H|T2]) :-
      add(T, X, T2).
?- add([1,2,3],4,L).
```

  - Much better with diff. lists:

```
add(L-T, X, L2-T2) :- % constant time
      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 `add` 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-[]).% helper
palindromeHelp(A-A). % an empty list
palindromeHelp([_|A]-A).%1-element list
palindromeHelp([C|A]-D) :-
    B=[C|D],
    palindromeHelp(A-B).
?- palindrome([1,2,2,1]).
yes
?- palindrome([1,2,3,2,1]).
yes
?- palindrome([1,2,3,4,5]).
no
```

Tree Traversal, Revisited

```
preorder(Node, List) :-
  preorder1(Node, List-[]).
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).
```

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

Tree Traversal, Revisited

```
node(5, 3, 6).
node(3, 1, 4).
leaf(1).
leaf(4).
leaf(6).
?- preorder(5, List).
     List = [5,3,1,4,6];
     no
```

Difference Lists

- The preorder traversal program may be rewritten simpler 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

- The inorder traversal program:

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

Difference Lists

- The postorder traversal program:

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

Negation

- The built-in Prolog predicate **\+**/1 provides negation as failure: we derive **`\+ p`** as **`true`** if we fail to derive **`p`**

```
p :- q, \+ r.
q.
```

derives: **`\+ r`** and **`p`** as **`true`**

- Consider the rule:

```
legal(X) :- \+ illegal(X).
?- legal(smoking).
yes
```

- Prolog attempts to prove **`illegal(X)`**. If a proof for that goal can be found, the original goal (i.e., **`\+illegal(X)`**) fails. If no proof can be found, the original goal succeeds.

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 structure term: a pair of two args. nodes and edges: **`graph([a,b,c,d,f,g], [e(a,b),e(b,c),e(b,f)])`**
    - or just the list of arcs: **`[a-b, b-c, b-f]`** (same problem with isolated nodes as above)
  - adjacency-list term: **`[n(a,[b]), n(b,[c,f]), n(d,[])]`**

Graphs in Prolog

- Path predicate to find paths 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 also return **all paths via backtracking**
  - We will solve it using the graph as a data object, like in the term:

```
G=graph([a,b,c,d,f,g],[e(a,b),e(b,c),e(b,f)]
```

directed vs. undirected graphs

- **`adjacent`** for <u>directed edges</u>:

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

- **`adjacent`** for <u>undirected edges</u> (ie. 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,Cycle)`** to find a closed path (cycle) **`Cycle`** starting at a given node **`A`** in the graph **`G`**
  - The predicate should return all cycles via backtracking

```
cycle(G,A,Cycle) :-
      adjacent(A,B,G),
      path(G,B,A,P1),
      Cycle = [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,Cycle) :-
      adjacent(A,B,G),
      path(G,B,A,P),
      Cycle = [A|P].
?- 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,Cycle),
      writeln(Cycle),
      fail; true.
```

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.
  - even if we collect only some variables, it groups the results on the other vars
  - if **`Goal`** is unsatisfiable, both **`setof`** and **`bagof`** fail
- **`findall(Template,Goal,List)`** is similar to predicate **`bagof`**/3, except that, if **`Goal`** is unsatisfiable, it succeeds binding **`List`** to the empty list, and all variables are existential(not collected)
- **`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
```

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 - above**
  - **Green cut: if someone deletes !, then the rule is correct**

```
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
- **`number(X)`**
- **`atomic(X)`**

Testing for variables

- **`ground(G)`**

  Tests whether **`G`** has unbound logical variables
- **`var(X)`**

  Tests whether **`X`** is bound to a Prolog variable
- **`is_list(L)`**

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

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).
```

Control / Meta-predicates

- **`call(P)`**

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

- **`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

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,[]]
```

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(inc1(X,Y),[5,6],R).` returns **`R=[6,7]`**

where **`inc1(X,Y)`** was defined as:

```
      inc1(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).
inc1(X,Y) :-
     Y is X+1.
?- map(inc1(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
```

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.

Assert and retract

```
?- assert(fast(ann)).
?- assert(slow(tom)).
?- assert((faster(X,Y) :-
    fast(X),slow(Y))).
?- faster(A, B).
    A = ann
    B = tom
?- retract(slow(tom)).
?- faster(A, B).
    no
```

Programming style

- To reduce the danger of programming errors and to produce programs that are readable and easy to understand, easy to debug and to modify
  - Clauses should be short: their body should typically contain no more than a few goals
  - Mnemonic names for predicates and variables should be used: names should indicate the meaning of relations and the role of data objects
- The layout of programs is important:
  - Spacing, blank lines and indentation should be consistently used for the sake of readability
  - Clauses about the same procedure should be clustered together

Programming style

- Each goal can be placed on a separate line
- The cut operator should be used with care and **should be placed on a different line**
- if Condition then Goal else Goal2 with **`->`** **`;`** translates into Prolog using cut:

```
        Condition, !, Goal1 ; Goal2
```

- There should be blank lines between clauses for different relations

Debugging

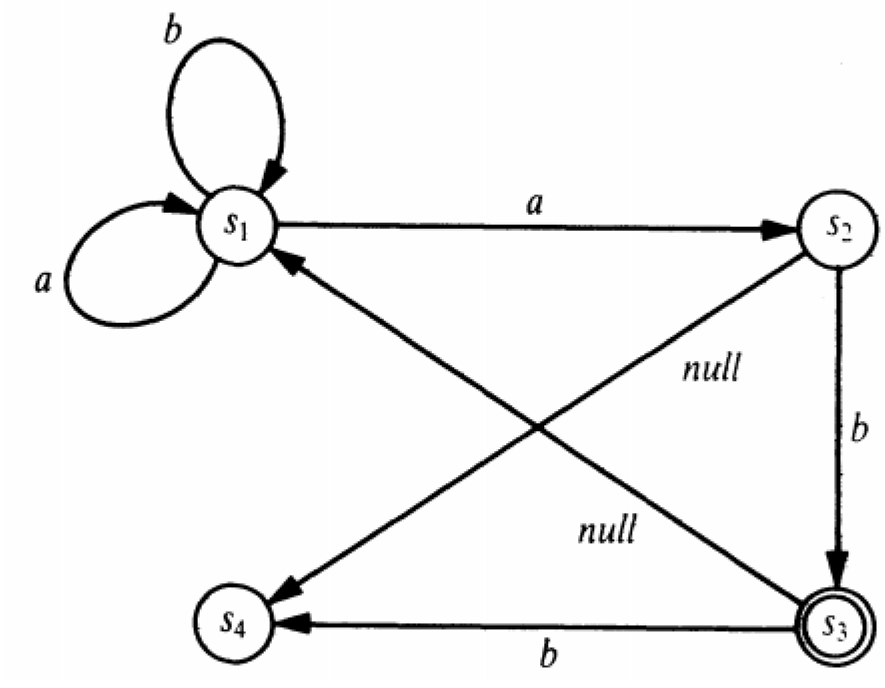
- When a program does not do what it is expected to do the main problem is to locate the error
- The basis of Prolog debugging is tracing activated by the built-in predicate **`trace`**

```
?- trace.
?- SomeGoal.
```

- Steps:
  - 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
- You can skip calls with **`s`**, abort with **`a`** and leap with **`l`**
- Exit tracing with **`?- notrace.`**

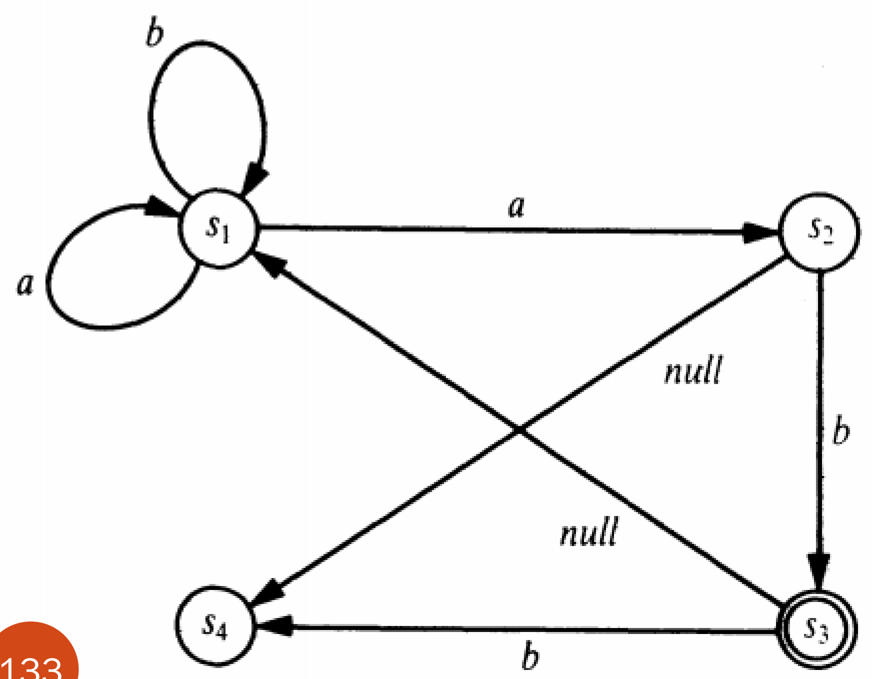
Prolog Example: Non-deterministic finite automata

- A unary relation **`final`** which defines the final states of the automaton
- A three-argument relation trans which defines the state transitions so that **`trans(S1, X, S2)`** means that a transition from a state **`S1`** to **`S2`** is possible when the current input symbol **`X`** is read.
- A binary relation **`silent(S1, S2)`** meaning that a silent move is possible from **`S1`** to **`S2`**



Non-deterministic finite automata

- A unary relation **`final`** which defines the final states of the automaton
- A three-argument relation trans which defines the state transitions so that **`trans(S1, X, S2)`** means that a transition from a state **`S1`** to **`S2`** is possible when the current input symbol **`X`** is read.
- A binary relation **`silent(S1, S2)`** meaning that a silent move is possible from **`S1`** to **`S2`**



```
final(s3).
trans(s1, a, s1).
trans(s1, a, s2).
trans(s1, b, s1).
trans(s2, b, s3).
trans(s3, b, s4).
silent(s2, s4).
silent(s3, s1).
```

Non-deterministic finite automata

- A binary relation **`accepts`** defines the acceptance of a string from a given state:

```
accepts(S, []) :-
     final( S).
accepts(S, [X|Rest]) :-
     trans(S, X, S1),
     accepts(S1, Rest).
accepts(S, String) :-
     silent(S, S1),
     accepts(S1, String).
?- accepts(s1, [a,a,a,b]).
     yes
```

In which state our automaton can be in initially so that it will accept the string **`ab`**: `?- accepts(S, [a,b]).`

```
     S=s1;
     S=s3
```

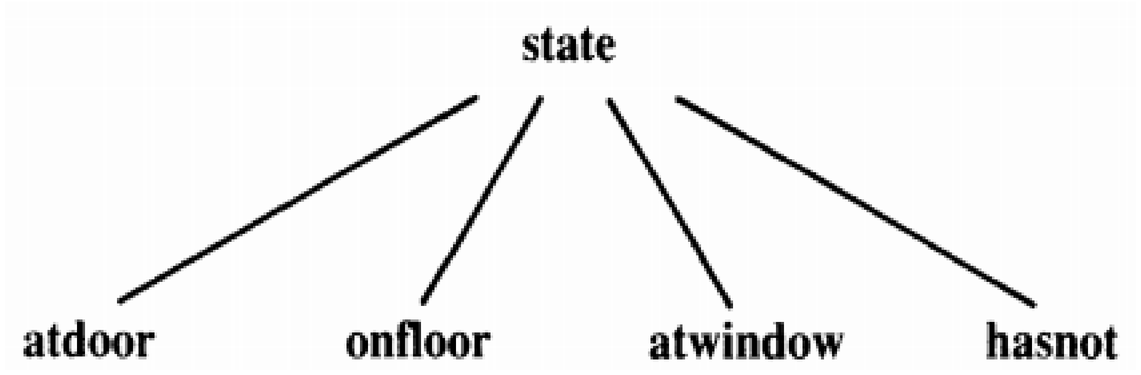
Simple Planning Example: monkey and banana

- There is a monkey at the door into a room.
- In the middle of the room a banana is hanging from the ceiling.
- The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor.
- At the window of the room there is a box the monkey may use.
- The monkey can perform the following actions:
  - walk on the floor
  - push the box around (if it is already at the box)
  - climb the box
  - grasp the banana if standing on the box directly under the banana
- Can the monkey get the banana?

Example: monkey and banana

- The initial state of the world is determined by:
  - (1) Monkey is at door
  - (2) Monkey is on floor
  - (3) Box is at window
  - (4) Monkey does not have banana

```
state(atdoor, onfloor, atwindow, hasnot)
```



Example: monkey and banana

- The goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is **`has`**:
  `state(_, _, _, has)`
- There are four types of moves:
  - (1) grasp banana
  - (2) climb box
  - (3) push box
  - (4) walk around
- Not all moves are possible in every possible state of the world
  - the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet

Example: monkey and banana

- The goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is **`has`**:

```
state(_, _, _, has)
```

- There are four types of moves:
  - (1) grasp banana
  - (2) climb box
  - (3) push box
  - (4) walk around
- Moves are formalized as a three-place relation

```
move(State1, M, State2)
```

`State1` is the state before the move, **`M`** is the move executed and **`State2`** is the state after the move.

  - Not all moves are possible in every possible state of the world
    - the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet

Example: monkey and banana

- Grasp banana:

```
move( state(middle, onbox, middle, hasnot),
      grasp,
      state(middle, onbox, middle, has) ).
```

- Climb box:

```
move( state(P, onfloor, P, H),
      climb, state(P, onbox, P, H) ).
```

- Push box:

```
move( state(P1, onfloor, P1, H),
      push(P1, P2), state(P2, onfloor, P2, H) ).
```

- Walk:

```
move( state( P1, onfloor, B, H),
      walk( P1, P2),
      state( P2, onfloor, B, H) ).
```

Example: monkey and banana

- The question that our program will have to answer is:

Can the monkey in some initial state **`S`** get the banana?

This can be formulated as a predicate **`canget(S)`**

- For any state S in which the monkey already has the banana, the predicate canget must certainly be true; no move is needed in this case

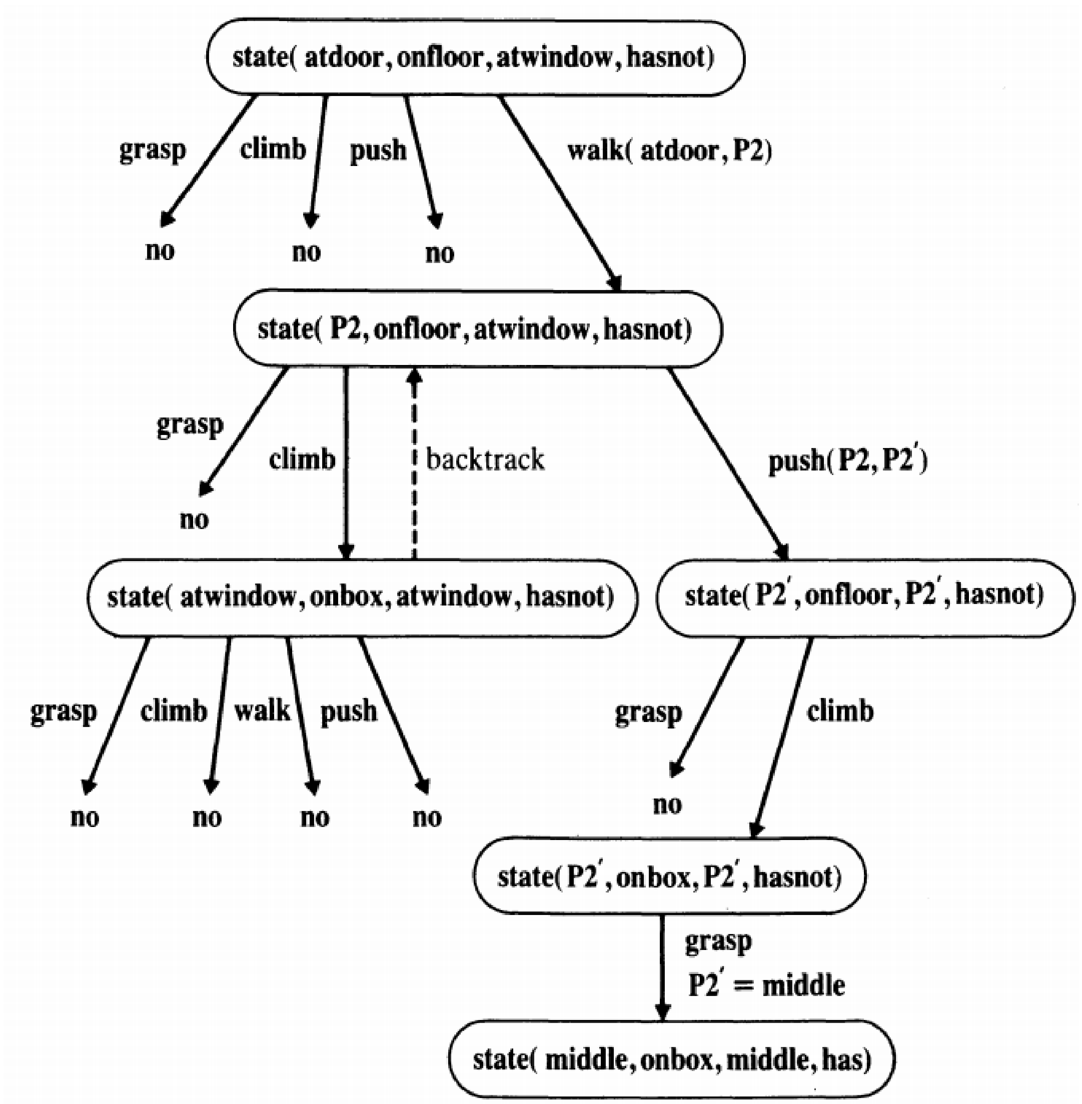
```
canget(state(_, _, _, has)).
```

- In other cases one or more moves are necessary: the monkey can get the banana in any state **`S1`** if there is some move **`M`** from state **`S1`** to some state **`S2`**, such that the monkey can then get the banana in state **`S2`** (in zero or more moves):

```
canget(S1) :-
      move(S1, M, S2),
      canget(S2).
```

Example: monkey and banana

```
?- canget(state(atdoor, onfloor, atwindow, hasnot)).
```


state(atdoor, onfloor, atwindow, hasnot)
grasp
climb
push
walk(atdoor, P2)
no
no
no
state(P2, onfloor, atwindow, hasnot)
grasp
climb
backtrack
push(P2, P2′)
no
state(atwindow, onbox, atwindow, hasnot)
state(P2′, onfloor, P2′, hasnot)
grasp
climb
walk
push
no
no
no
no
grasp
climb
no
state(P2′, onbox, P2′, hasnot)
grasp
P2′ = middle
state(middle, onbox, middle, has)

Order of clauses and goals

- Danger of indefinite looping

  `p:-p.`

says that **`p`** is true if **`p`** is true.

- This is declaratively correct, but procedurally is quite useless and causes problems to Prolog: the question **`?- p.`** will loop infinitely: the goal **`p`** is replaced by the same goal **`p`**; this will be in turn replaced by **`p`**, etc.
- In the monkey and banana program, the clauses about the move relation must be ordered: grasp, climb, push, walk
  - According to the procedural semantics of prolog, the order of clauses indicates that the monkey prefers grasping to climbing, climbing to pushing, etc. This order of preferences in fact helps the monkey to solve the problem. If walk is first, then infinite loop.

Example: Travel planning

- What days of the week is there a direct flight from NY to London?
- How can I get from NY to Edinburgh on Thursday?
- I have to visit Milan, Paris and Zurich, starting from London on Tuesday and returning to London on Friday. In what sequence should I visit these cities so that I have no more than one flight each day of the tour?
- `timetable(Place1, Place2, ListOfFlights):`

```
timetable(london, edinburgh,
      [9:40 / 10:50 / ua4733 / alldays,
      19:40 / 20:50 / ua4833 / [mo,tu,we,th,fr,su] ]).

flight(Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
      timetable(Place1, Place2, FlightList),
      member( Deptime / Arrtime / Fnum / Daylist ,
            Flightlist),
      flyday(Day, Daylist).
```

Example: Travel planning

- `route(Place1, Place2, Day, Route):`

```
route(Place1, Place2, Day, [Place1-Place2:Fnum:Dep]):-
        flight(Place1, Place2, Day, Fnum, Dep, Arr).
route(P1, P2, Day, [P1-P3:Fnum1:Dep1 | Route]):-
        flight(P1, P3, Day, Fnum1, Dep1, Arr1),
        route(P3, P2, Day, Route),
        deptime(Route, Dep2),
        transfer(Arr1, Dep2).
flyday(Day, Daylist) :-
        member( Day, Daylist).
flyday(_Day, alldays).
deptime([_P1-_P2 : _Fnum : Dep | _], Dep).

?- route(ny, bucharest, th, R).
        R = [ny-zurich:ua322:1:30,
              zurich-bucharest:sr806:16:10]
```

XSB Tabling

- XSB also supports Datalog with ***TABLING (memoization)*:**

  ```
  :- auto_table.
  ```

  at the top of the program file, OR

  ```
  :- table p/1.
  ```

  for the predicates that you want to table.

  - Tabling has many advantages – the most important one is that it solves the problem of infinite recursion
- Another negation in XSB called **`tnot`** is used for ***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 <u>positive</u> relations:
  - Ok: …**`:-p(X,Y),tnot(foobar(X,Y)),`**…
  - Not ok: …**`:-p(X,Z),tnot(foobar(X,Y)),`** …

Example: Nqueens

```
queens([]).  % when place queen in empty list, solution found
queens([Row/Col | Rest]) :-
      queens(Rest),
      member(Col, [1,2,3,4,5,6,7,8]),
      safe( Row/Col, Rest).
safe(Anything, []). % the empty board is always safe
safe(Row/Col, [Row1/Col1 | Rest]) :-
      Col =\= Col1, % same column?
      Col1 - Col =\= Row1 - Row, % check diagonal
      Col1 - Col =\= Row - Row1,
      safe(Row/Col, Rest).
member(X, [X | Tail]).
member(X, [Head | Tail]) :-
      member(X, Tail).

?- queens([1/C1,2/C2,3/C3,4/C4,5/C5,6/C6,7/C7,8/C8]).
```