UCS1524 – Logic Programming

Syntax and Semantics of Prolog

Session Meta Data

Author	**Dr. D. Thenmozhi**
Reviewer	
Version Number	1.2
Release Date	30 August 2022

Session Objectives

- Understanding the syntax and semantics of Prolog programming
- Learn about the data objects, matching operations on objects, procedural and declarative semantics of Prolog programs

Session Outcomes

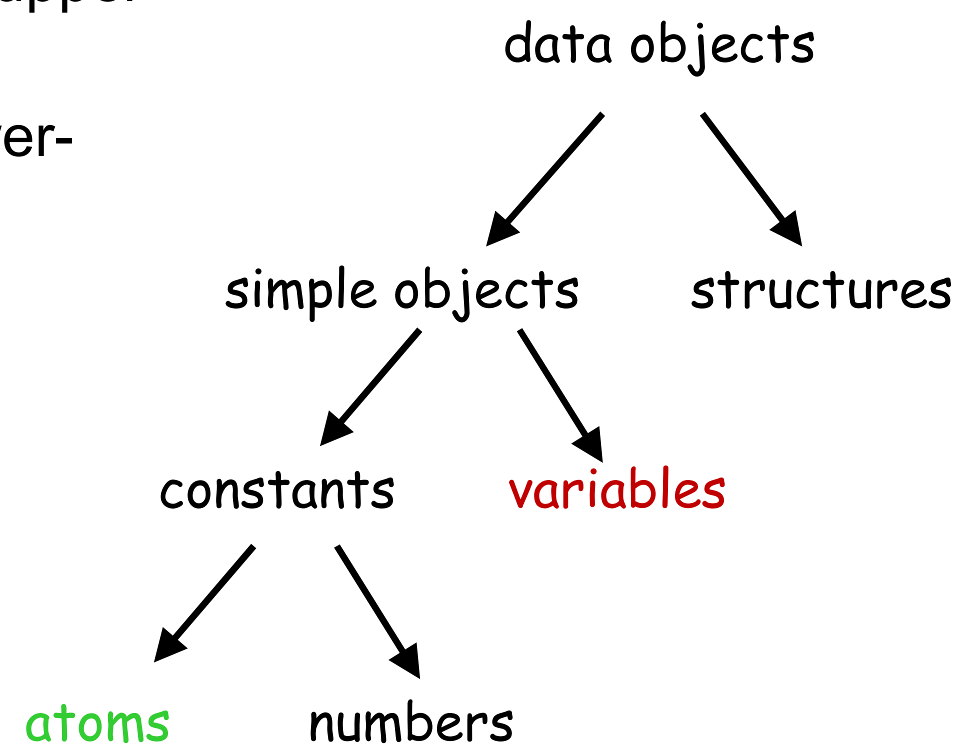
- At the end of this session, participants will be able to
  - explain the syntax and semantics of Prolog programming.

Agenda

- Simple data objects
- Structured objects
- Operation on objects
- Declarative meaning of a program
- Procedural meaning of a Program
- Relation between the declarative and procedural meanings of a program
- Altering the procedural meaning

Data Objects

- Variables start with upper-case letters
- Atoms start with lower-case letters



Atoms and numbers

- Characters:
  - Upper-case letter A, B,…, Z
  - Lower-case letter a, b,…, z
  - Digits 0,1,2,…,9
  - Special characters such as +-*/<>=:.&_~
- **Atoms** can be constructed in three ways:
  - Strings of letters, digits and the underscore character,'_', starting with a lower case letter
    - anna, x25, x_35AB, x___y, miss_Jones
  - Strings of special characters
    - <--->, ===>, …,::=,.:., (except :-)
  - Strings of characters enclosed in single quotes
    - 'Tom', 'South_America', 'Sarah Jones'

Atoms and numbers

- Number:
  - Numbers used in Prolog include integer numbers and real numbers.
    - Integer numbers: 1313, 0, -97
    - Real numbers: 3.14, -0.0035, 100.2
  - In symbolic computation, integers are often used.

Variables

- **Variables** are start with an upper-case letter or an underscore character.
  - Examples: X, Result, _x23, _23
- Anonymous variables:
  - Examples:

```
hasachild( X) :- parent( X, Y).
hasachild( X) :- parent( X, _).
somebody_has_child :- parent( X, Y).
somebody_has_child :- parent( _, _).
somebody_has_child :- parent( X, X).
?- parent( X, _)
```

  - We are interested in people who have children, but not in the names of the children.

Variables

- The lexical scope of variable names is one clause.
  - If the name X occurs in two clauses, then it signifies two different variables.

    hasachild(X) :- parent(X, Y).

    isapoint(X) :- point(X, Y, Z).

  - But each occurrence of X with in the same clause means the same variables.

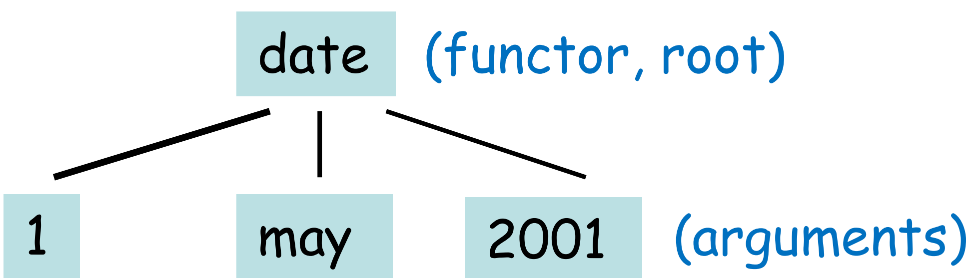
    hasachild(X) :- parent(X, Y).

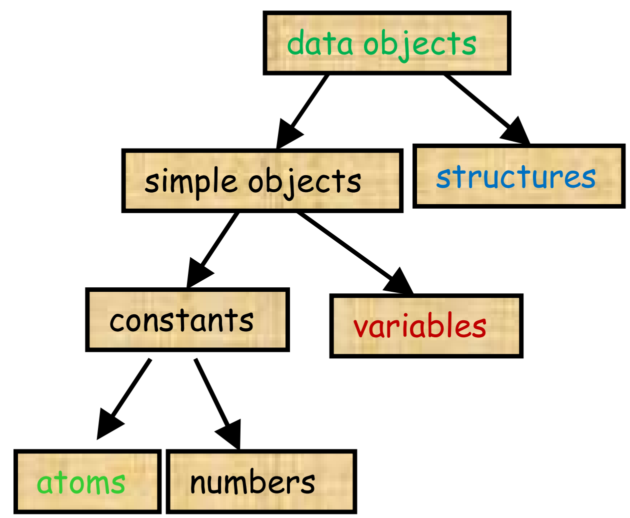
- The same atom always means the same object in any clause throughout the whole program.

Structures

- **Structured objects** are objects that have several components.
- All structured objects can be pictured as trees.
  - The root of the tree is the functor.
  - The offsprings of the root are the components.
  - Components can also be variables or other structures.

    date(Day, may, 2001)
  - Example: **date(1, may, 2001)**





- All **data objects** in Prolog are terms.

Structures

- Let

  a point be defined by its two coordinates,

  a line segment be defined by two points, and

  a triangle be defined by three points.

- Choose the following functors:

  point for points,

  seg for line segments, and

  triangle for triangles.

- Representation:

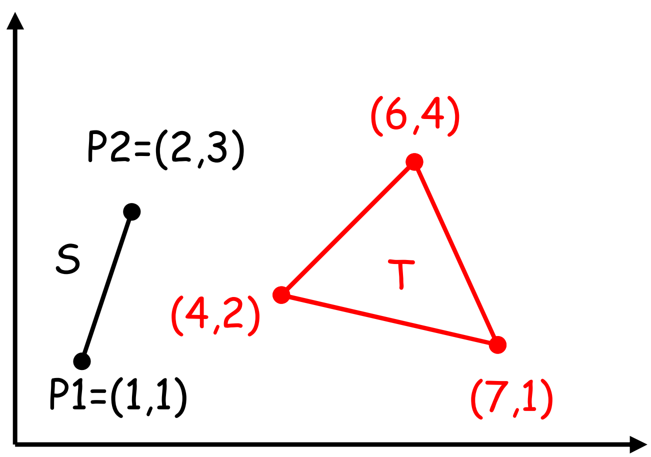
  P1 = point(1, 1)

  P2 = point(2, 3)

  S = seg(P1, P2)

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

  T = triangle(point(4,2), point(6,4), point(7,1))



Structures

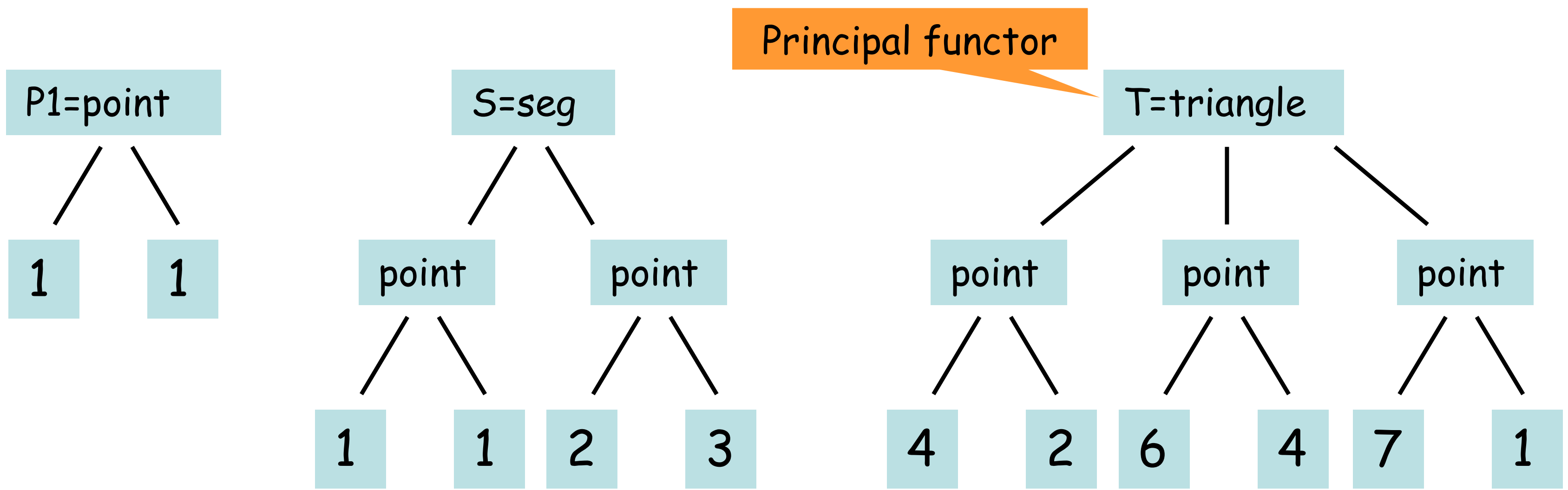
- Tree representation of the objects:

  P1 = point(1, 1)

  S = seg(P1, P2)

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

  T = triangle(point(4,2), point(6,4), point(7,1))



Structures

- Each functor is defined by two things:
  - The name, whose syntax is that of atoms;
  - The arity—that is, the number of arguments.

  - For example:

    point(X1, Y1) and **point(X, Y, Z)** are different.

    - The Prolog system will recognize the difference by the number of arguments, and will interpret this name as two functors.

Structures

- The tree structure corresponding to the arithmetic expression (a + b)*(c - 5).

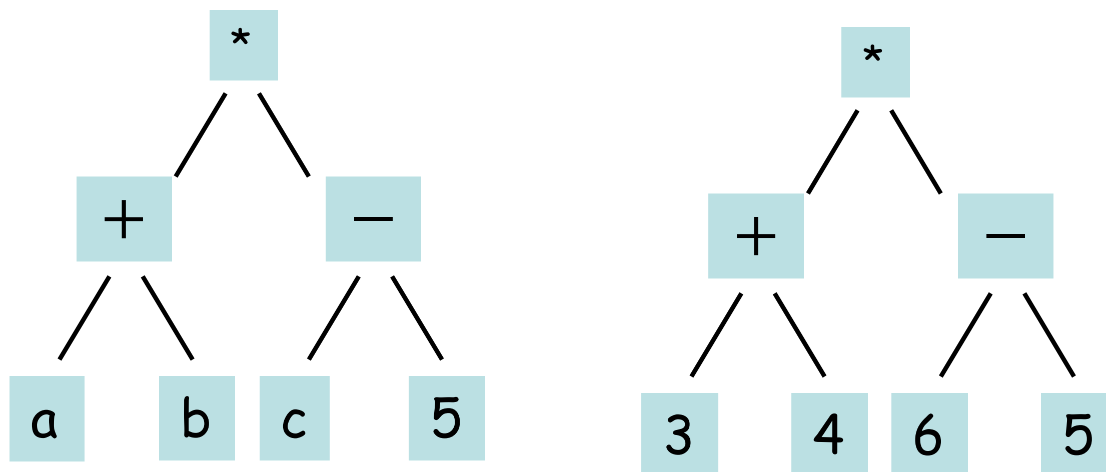
infix notation

  – Using the simples ‘*’,’+’ and ‘-’ as functors

    ***(+(a, b), -(c, 5))**

    ?- X = *(+(a, b), -(c, 5)).

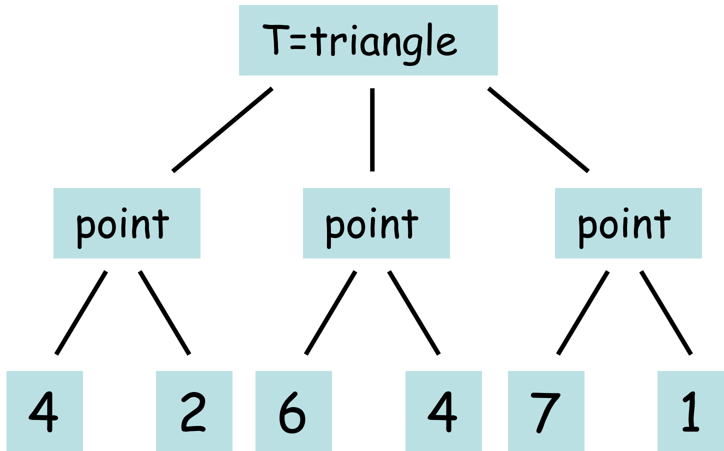
    ?- X is *(+(3, 4), -(6, 5)).



  – In fact, Prolog also allows us to use the infix notation. (Details will be discussed in Arithmetics topic)

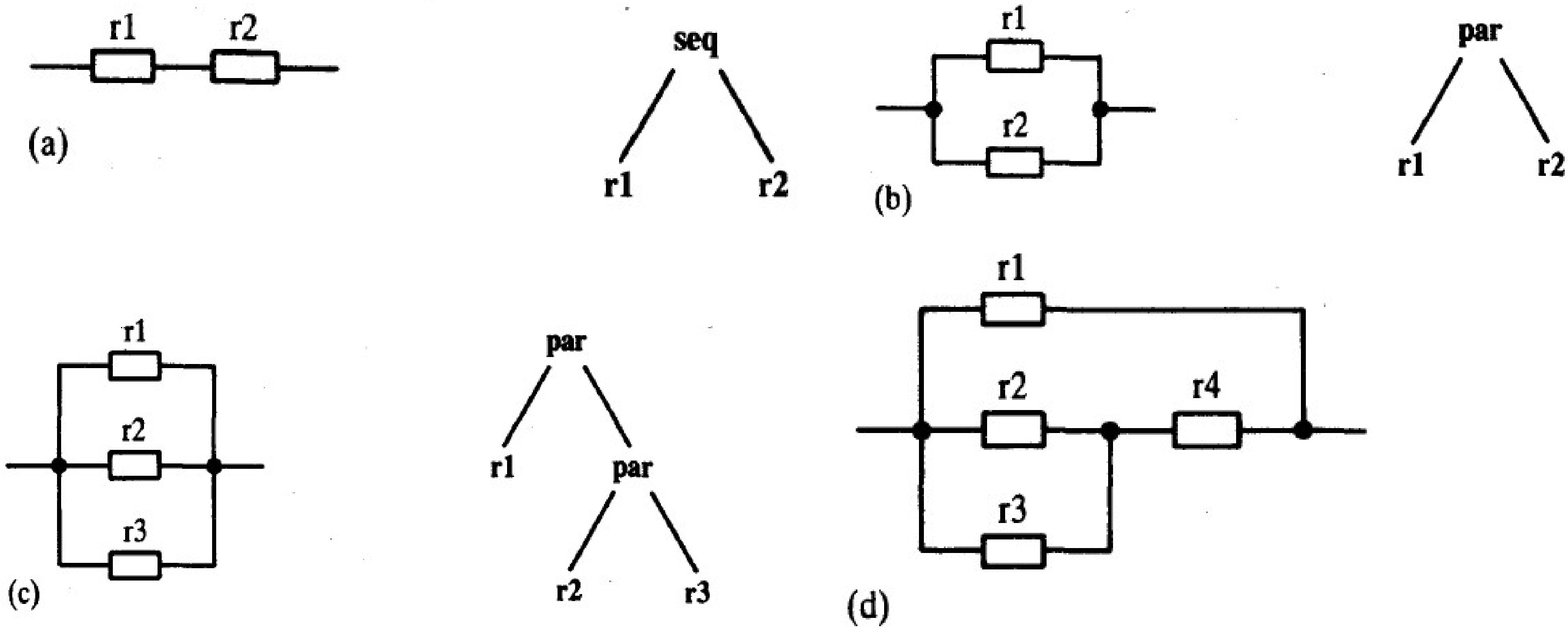
Exercise

- Suggest a representation for rectangles, squares **or** circles as structured Prolog objects.
  - For example, a rectangle can be represented by four points (or maybe three points only).
  - Write some example terms that represent some concrete objects of there types using the suggested representation.



Example

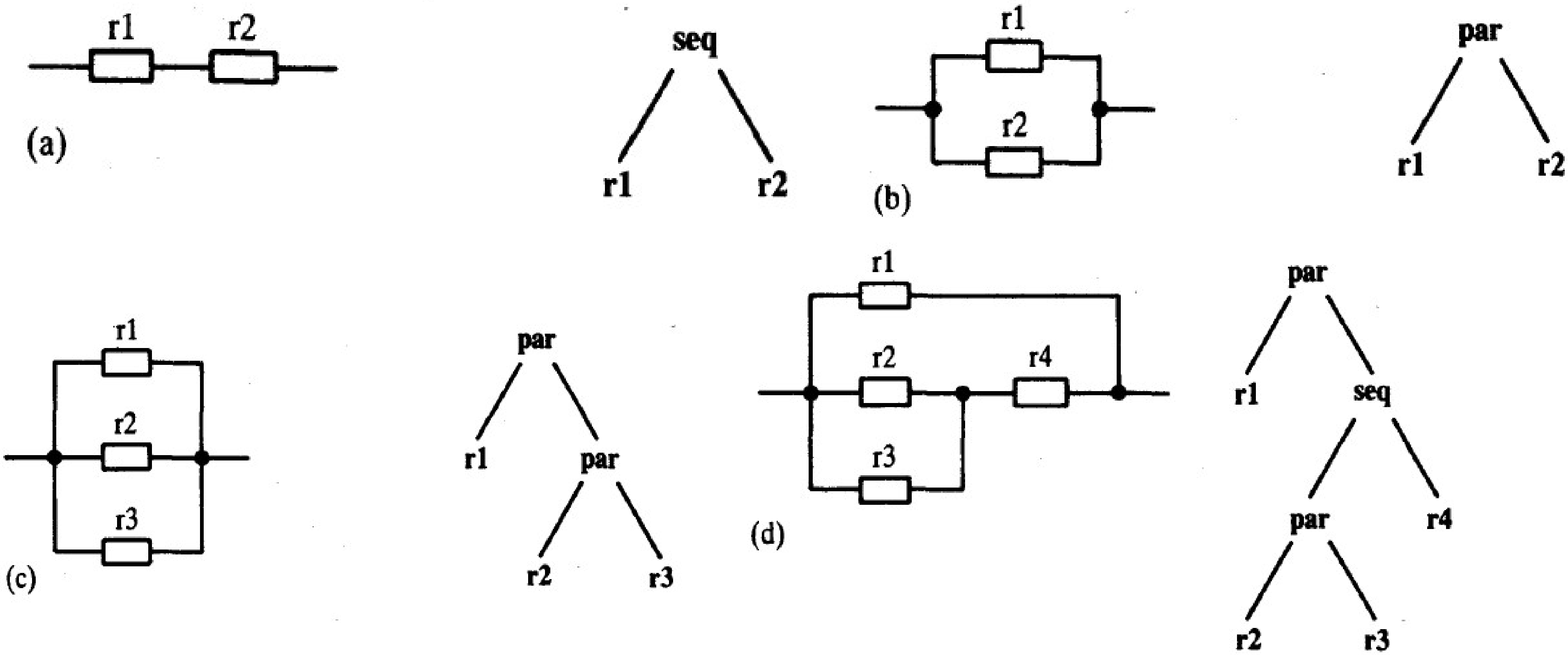
- Let atoms r1,r2,r3 are r4 are resistors. Then the electric circuits can be represented as



What is the structure of (d)

Example

- Let atoms r1,r2,r3 are r4 are resistors. Then the electric circuits can be represented as



What is the structure of (d)

Matching

- The most important operation on terms is matching.
- Matching is a process that takes as input two terms and checks whether they match.
  - Fails: if the terms do not match
  - Succeeds: if the terms do match
- Given two terms, we say that they match if:
  - they are identical , or
  - the variable in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.
  - For example:
    - the terms **date(D, M, 2001)** and **date(D1, may, Y1)** match
    - the terms **date(D, M, 2001)** and **date(D1, M1, 1444)** do not match

Matching

- The request for matching, using the operator ‘=‘:

| ?- date(D, M, 2001) = date(D1, may, Y1).

D1 = D
M = may
Y1 = 2001
Yes

| ?- date(D, M, 2001) = date(D1, may, Y1),
date(D, M, 2001) = date(15, M, Y).

D = 15
D1 = 15
M = may
Y = 2001
Y1 = 2001
yes

Matching

- Matching in Prolog always
  - results in the most general instantiation
  - leaves the greatest possible freedom for further instantiations if further matching is required

Matching

- The general rules to decide whether two terms, S and T, match are as follows:
  - If S and T are constants then S and T match only if they are the same object.

    ```
    | ?- date( D, M, 2001) = date(D1, may, 2001).
    D1 = D
    M = may
    yes
    ```

  - If S is a variable and T is anything, then they match, and S is instantiated to T.

  - If S and T are structures then they match only if S and T have the same principal functor, and all their corresponding components match.

    ```
    | ?- date( date(D, M1, 2003), may, 2001) = date( date(D, D, F), may, 2001).
    F = 2003
    M1 = D
    yes
    ```

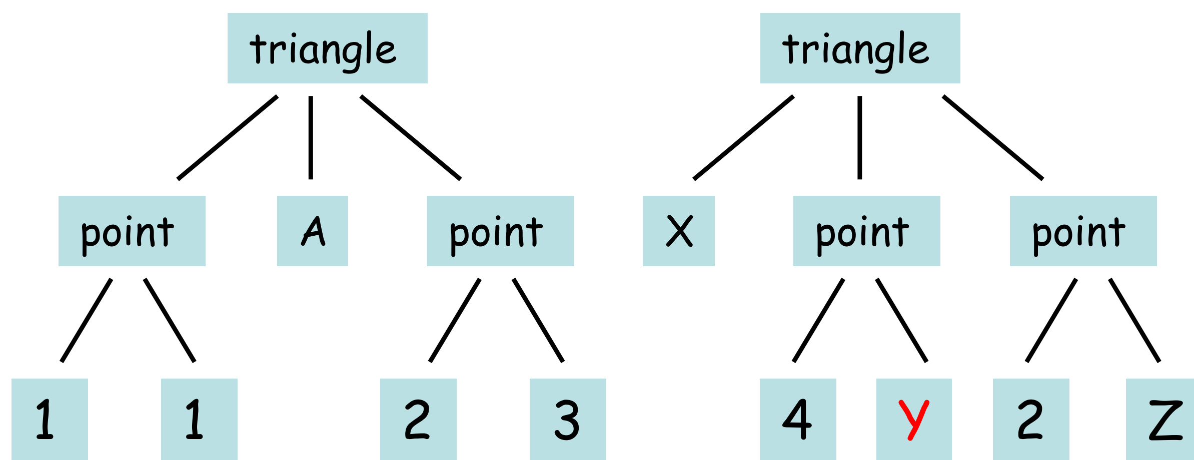
    The resulting instantiation is determined by the matching of the components.

Matching

- Matching

  | ?- triangle(point(1,1), A, point(2,3))= triangle(X, point(4,Y), point(2,Z)).

  A = point(4,Y)
  X = point(1,1)
  Z = 3
  yes

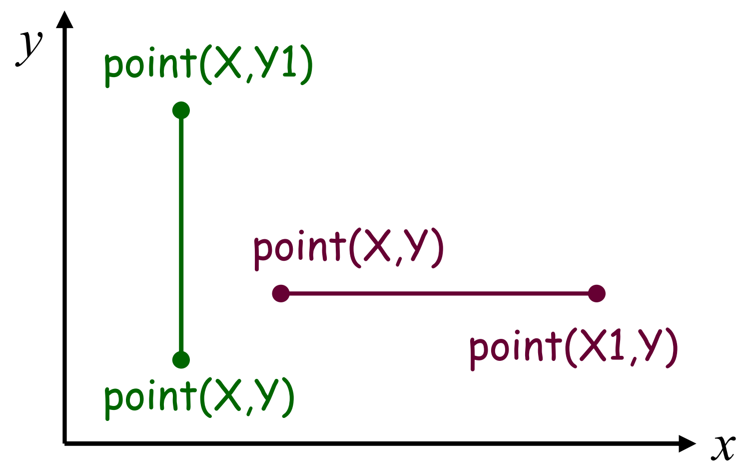


Matching

- Vertical and horizontal line segments
  - 'Vertical' is a unary relation.
  - A segment is vertical if the x-coordinates of its end-points are equal.
  - The property 'horizontal' is similarly formulated, with x and y interchanged.

vertical(seg(point(X,Y), point(X, Y1))).

horizontal(seg(point(X,Y), point(X1, Y))).



Matching

- An example:

```
point(1,1).
point(1,2).
point(2,4).
seg(point(1,1), point(1,2)).
seg(point(1,1), point(2,4)).
seg(point(1,2), point(2,4)).
vertical( seg( point( X, Y), point( X, Y1))).
horizontal( seg( point( X, Y), point( X1, Y))).
```

```
| ?- vertical( seg( point(1,1), point( 1,2))).
   yes

| ?- vertical( seg( point(1,Y), point(2,Y))).
   no

| ?- horizontal( seg( point(1,1), point(2,Y))).
   Y = 1
   yes
```

Matching

| ?- horizontal(seg(point(1,1), P)).

P = point(_,1)

Yes

| ?- vertical(seg(point(1,1), P)).

P = point(1,_)

Yes

- The answer means: Yes, any segment that ends at any point (1,_), which means anywhere on the vertical line x =1.

| ?- vertical(S), horizontal(S).

S = seg(point(A,B), point(A,B))

yes

- The answer means: Yes, any segment that is degenerated to a point has the property of being vertical and horizontal at the same time.

Declarative meaning of Prolog programs

- Consider a clause:

  P :- Q, R.

  - Some declarative reading of this clause are:
    - P is true if Q and R are true.
    - From Q and R follows P.
  - Two procedural reading of this clause are:
    - To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.
    - To satisfy P, *first* satisfy Q and *then* R.
  - Difference:
    - The procedural readings do not only define the logical relations between the head of the clause and the goals in the body, but also the order in which the goals are processed.

Declarative meaning of Prolog programs

- The declarative meaning:
  - The declarative meaning of programs determines whether a given goal is true, and if so, for what values of variables it is true.
- Instance v.s. variant:
  - An instance of a clause C is the clause C with each of its variables substituted by some term.
  - An variant of a clause C is such an instance of the clause C where each variable is substituted by another variable.

Declarative meaning of Prolog programs

- For example, consider the clause:

  hasachild(X) :- parent(X, Y).

  - Two variants of this clause are:

    hasachild(A) :- parent(A, B).

    hasachild(X1) :- parent(X1, X2).

  - Instances of this clause are:

    hasachild(peter) :- parent(peter, X).

    hasachild(barry) :- parent(barry, small(caroline)).

Declarative meaning of Prolog programs

- A goal G is true if and only if
  - (1) there is a clause C in the program such that
  - (2) there is a clause instance I of C such that
    - (a) the head of I is identical to G, and
    - (b) all the goals in the body of I are true.

  - In general, a question to the Prolog system is a list of goals separated by commas (,) (the conjunction of goals).
  - A list of goals is true if all the goals in the list are true for the same instantiation of variables.

Declarative meaning of Prolog programs

- Prolog also accepts the disjunction of goals:
  - Any one of the goals in a disjunction has to be true.

    P :- Q; R.

    P is true if Q is true or R is true.
- Example:

  P :- Q, R; S, T, U.

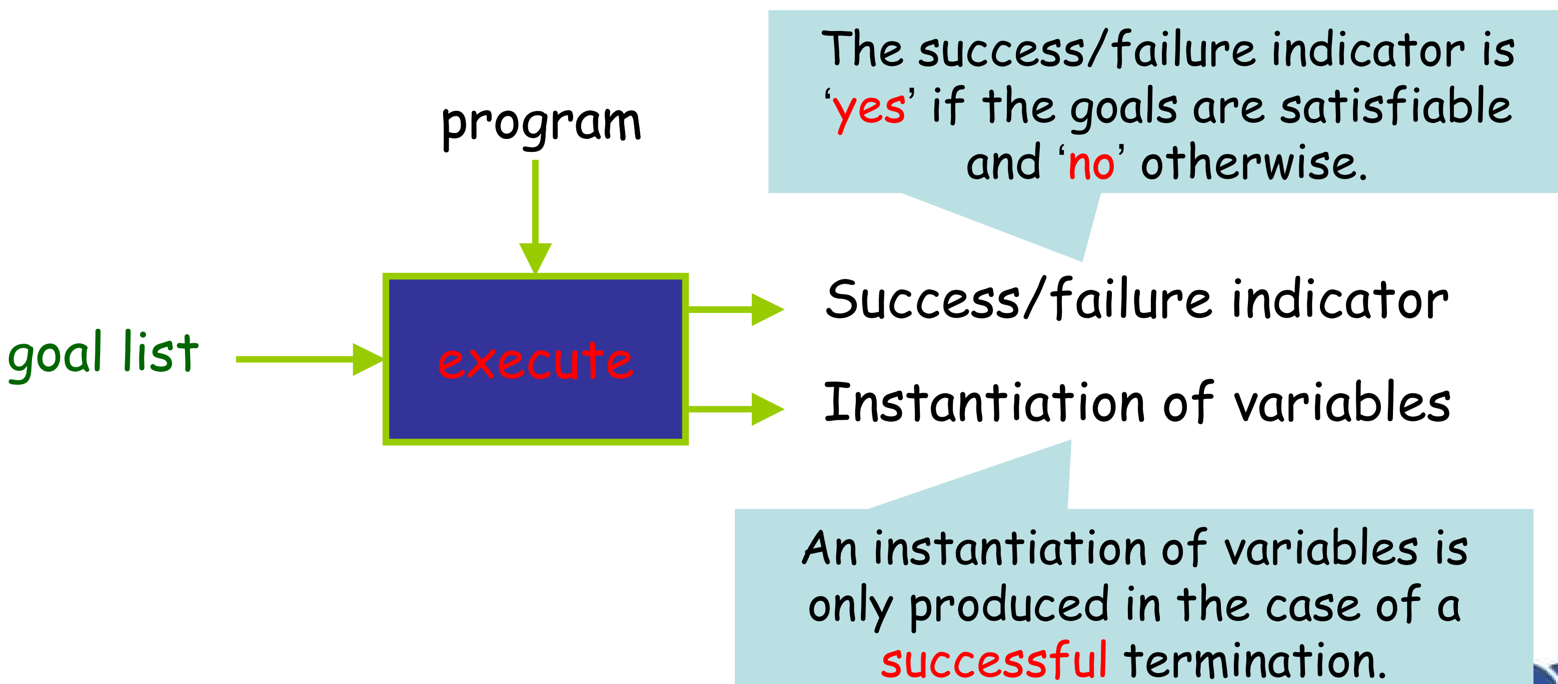
  P :- (Q, R); (S, T, U).

  P :- Q, R.

  P :- S, T, U.

Procedural meaning

- The procedural meaning:
  - The procedural meaning specifies how Prolog answers questions.
  - The procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program.



Procedural meaning

- Procedure execute:
  - If the goal list G1,…,Gm is empty then terminate with success.
  - If the goal list is not empty then called SCANNING.
  - SCANNING:
    - Scan through the clauses in the program from top to bottom until the first clause, C, is found such that the head of C matches the first goal G1. If there is no such clause then terminate with failure.
    - If there is such a clauses C, then rename the variables in C to obtain a variant C’ of C, such that C’ and the list G1, …,Gm have no common variables.
    - Match G1 and the head of C’. Replace G1 with the body of C’ (except the facts) to obtain a new goal list.
  - Execute this new goal list.

Procedural meaning

- Program:

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

```
{trace}
| ?- dark(X), big(X). (goal list: dark(X), big(X))
    1   1  Call: dark(_16) ? (dark(Z):- black(Z);
                                  goal list: black(X), big(X))
    2   2  Call: black(_16) ?
    2   2  Exit: black(cat) ? (yes)
    1   1  Exit: dark(cat) ? (X = cat; goal list: big(X))
    3   1  Call: big(cat) ? (no)
    3   1  Fail: big(cat) ?
    1   1  Redo: dark(cat) ? (X != cat, backtrack;
                                  goal list: dark(X), big(X))
    2   2  Call: brown(_16) ? (dark(Z):- brown(Z);
                                  goal list: brown(X), big(X))
    2   2  Exit: brown(bear) ?
    1   1  Exit: dark(bear) ? (yes, X= bear; goal list: big(X))
    3   1  Call: big(bear) ? (yes)
    3   1  Exit: big(bear) ?
X = bear
yes
{trace}
```

Procedural meaning

- Whenever a recursive call to execute fails, the execution returns to SCANNING, continuing at the program clause C that had been last used before.

- Prolog abandons the whole part of the unsuccessful execution and backtracks to the point where this failed branch of the execution was start.

- When the procedure backtracks to a certain point, all the variable instantiations that were done after that point are undone.

- Even after a successful termination the user can force the system to backtrack to search for more solutions.

Exercise

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

- In which of the two cases does Prolog have to do more work before the answer is found?

  |?- big(X), dark(X).

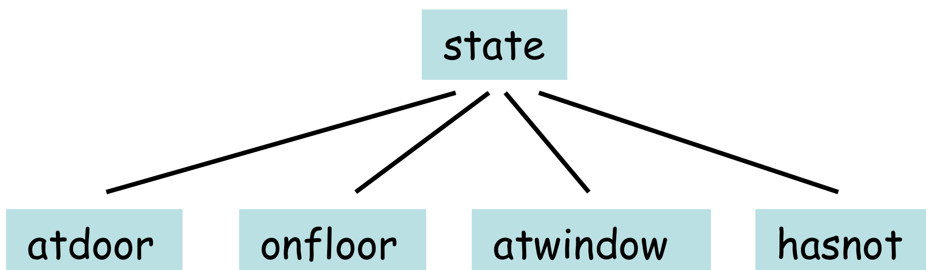
  |?- dark(X), big(X).

Example: monkey and banana

- Problem:
  - 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, climb the box, push the box around and grasp the banana if standing on the box directly under the banana.
  - Can the monkey get the banana?

Example: monkey and banana

- The representation of the problem:
  - The initial state:
    (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)



○ The goal of the game:

state(_, _, _, has)

Example: monkey and banana

- Four types of moves:
  (1) grasp banana,
  (2) climb box,
  (3) push box,
  (4) walk around.
- A three-place relation:

  move(State1, Move, State2)

$$\text{State1} \xrightarrow{\text{Move}} \text{State2}$$

'grasp':

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

Example: monkey and banana

'walk':

```
move( state( P1, onfloor, Box, Has),
           walk( P1, P2),
           state( P2, onfloor, Box, Has)).
```

'climb':

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

'push':

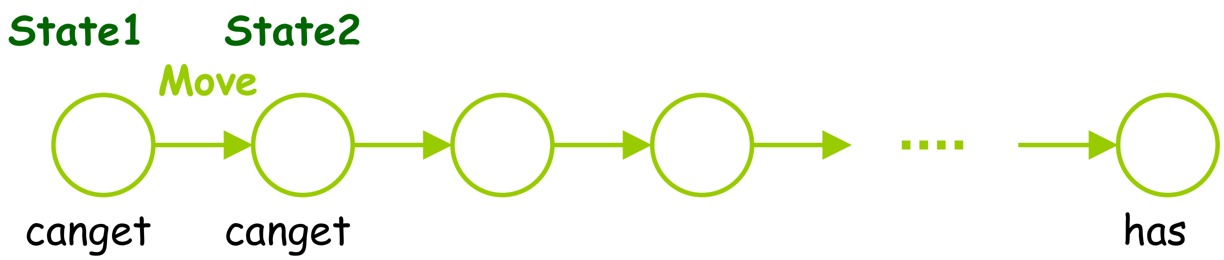
```
move( state( P1, onfloor, P1, Has),
           push( P1, P2),
           state( P2, onfloor, P2, Has)).
```

Example: monkey and banana

- Question: can the monkey in some initial state **State** get the banana?

```
    canget( State)

canget( state( _, _, _, has)).
canget( State1) :-
      move( State1, Move, State2),
    canget( State2).
```



Example: monkey and banana

```
% Figure 2.14   A program for the monkey and banana problem.

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

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

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

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

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

canget( State1)  :- move( State1, Move, State2), canget( State2).
```

Example: monkey and banana

```
| ?- canget( state( atdoor, onfloor, atwindow, hasnot)).
true ?
Yes (The monkey can grasp the banana from this state.)

{trace} (Figure 2.15)
| ?- canget( state( atdoor, onfloor, atwindow, hasnot)).
    1    1  Call: canget(state(atdoor,onfloor,atwindow,hasnot)) ?
    2    2  Call: move(state(atdoor,onfloor,atwindow,hasnot),_45,_85) ?
    2    2  Exit: move(state(atdoor,onfloor,atwindow,hasnot),
                  walk(atdoor,_73),state(_73,onfloor,atwindow,hasnot)) ?
    3    2  Call: canget(state(_73,onfloor,atwindow,hasnot)) ?
    4    3  Call: move(state(_73,onfloor,atwindow,hasnot),_103,_143) ?
    4    3  Exit: move(state(atwindow,onfloor,atwindow,hasnot),
                  climb,state(atwindow,onbox,atwindow,hasnot)) ?
    5    3  Call: canget(state(atwindow,onbox,atwindow,hasnot)) ?
    6    4  Call: move(state(atwindow,onbox,atwindow,hasnot),_158,_198) ?
    6    4  Fail: move(state(atwindow,onbox,atwindow,hasnot),_158,_186) ?
    5    3  Fail: canget(state(atwindow,onbox,atwindow,hasnot)) ?
    4    3  Redo: move(state(atwindow,onfloor,atwindow,hasnot),
                  climb,state(atwindow,onbox,atwindow,hasnot)) ?
```

Example: monkey and banana

```
    4  3 Exit: move(state(atwindow,onfloor,atwindow,hasnot),
              push(atwindow,_131),state(_131,onfloor,_131,hasnot)) ?
    5  3 Call: canget(state(_131,onfloor,_131,hasnot)) ?
    6  4 Call: move(state(_131,onfloor,_131,hasnot),_161,_201) ?
    6  4 Exit: move(state(_131,onfloor,_131,hasnot),
              climb,state(_131,onbox,_131,hasnot)) ?
    7  4 Call: canget(state(_131,onbox,_131,hasnot)) ?
    8  5 Call: move(state(_131,onbox,_131,hasnot),_216,_256) ?
    8  5 Exit: move(state(middle,onbox,middle,hasnot),
              grasp,state(middle,onbox,middle,has)) ?
    9  5 Call: canget(state(middle,onbox,middle,has)) ?
    9  5 Exit: canget(state(middle,onbox,middle,has)) ?
    7  4 Exit: canget(state(middle,onbox,middle,hasnot)) ?
    5  3 Exit: canget(state(middle,onfloor,middle,hasnot)) ?
    3  2 Exit: canget(state(atwindow,onfloor,atwindow,hasnot)) ?
    1  1 Exit: canget(state(atdoor,onfloor,atwindow,hasnot)) ?
true ?
(62 ms) yes
{trace}
```

Order of clauses and goals - Danger of indefinite looping

- Such a clause can cause problems to Prolog:

  p :- p.

  Consider the question:

  ?- p.

  – In such case Prolog will enter an infinite loop.

- In the monkey and banana program, what could happen if the order of the clauses are different?

  – Let us assume that the 'walk' clause appears first.

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

```
{trace}
| ?- canget( state( atdoor, onfloor, atwindow, hasnot)).
   1    1  Call: canget(state(atdoor,onfloor,atwindow,hasnot)) ?
   2    2  Call: move(state(atdoor,onfloor,atwindow,hasnot),_45,_85) ?
   2    2  Exit: move(state(atdoor,onfloor,atwindow,hasnot),
                 walk(atdoor,_73),state(_73,onfloor,atwindow,hasnot)) ?
   3    2  Call: canget(state(_73,onfloor,atwindow,hasnot)) ?
   4    3  Call: move(state(_73,onfloor,atwindow,hasnot),_1 _143) ?
   4    3  Exit: move(state(_73,onfloor,atwindow,hasnot),
                 walk(_73,_131),state(_131,onfloor,atwindow,
   5    3  Call: canget(state(_131,onfloor,atwindow,hasnot)) ?
   6    4  Call: move(state(_131,onfloor,atwindow,hasnot),_161,_2
   6    4  Exit: move(state(_131,onfloor,atwindow,hasnot),
                 walk(_131,_189),state(_189,onfloor,atwindow,hasnot)) ?
```

Infinite loop

Program variations through reordering of clauses and goals

- According to the declarative semantics of Prolog we can, without affecting the declarative meaning, change

  (1) the order of clauses in the program, and

  (2) the order of goals in the bodies of clauses.

- For example: the predecessor program

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

Program variations through reordering of clauses and goals

```
% Four versions of the predecessor
    program

% The original version
pred1( X, Z)  :-
  parent( X, Z).
pred1( X, Z)  :-
  parent( X, Y),
  pred1( Y, Z).

% Variation a: swap clauses of the
    original version
pred2( X, Z)  :-
  parent( X, Y),
  pred2( Y, Z).
pred2( X, Z)  :-
  parent( X, Z).

% Variation b: swap goals in second
    clause of the original version
pred3( X, Z)  :-
  parent( X, Z).
pred3( X, Z)  :-
  pred3( X, Y),
  parent( Y, Z).

% Variation c: swap goals and
    clauses of the original version
pred4( X, Z)  :-
  pred4( X, Y),
  parent( Y, Z).
pred4( X, Z)  :-
  parent( X, Z).
```

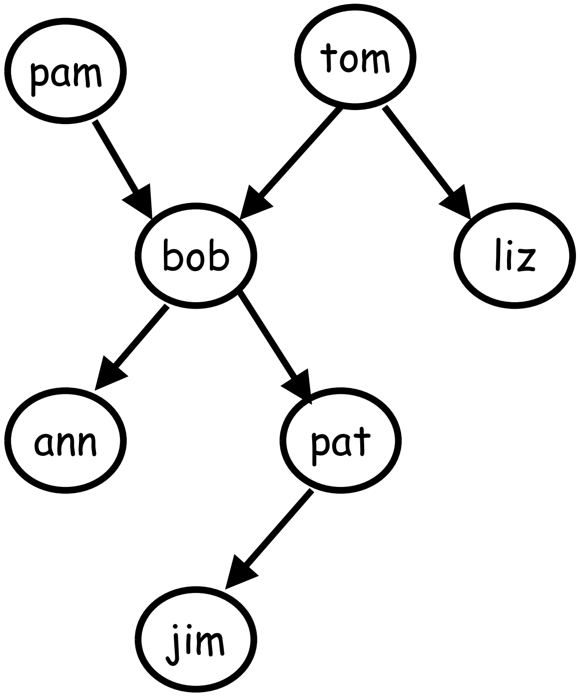
Program variations through reordering of clauses and goals

?- pred1(tom, pat).
Yes

?- pred2(tom, pat).
yes

?- pred3(tom, pat).
yes

?- pred4(tom, pat).
'More core needed' or 'Stack overflow'



- In the last case Prolog cannot find the answer.
- Programming rule: it is usually best to try the simplest idea first.

Program variations through reordering of clauses and goals

- What types of questions can particular variations answer, and what types can they not answer?
  - Pred1, and pred2 are both able to reach an answer for any type of question about predecessors.
  - Pred4 can never reach an answer.
  - Pred3 sometimes can and sometimes cannot.
    - For example:

    ?- pred3(liz, jim).

Combining declarative and procedural views

- There are programs that are declaratively correct, but do not work in practice.
  - For example:

    predecessor(X, Z) :- predecessor(X, Z).
- However, we should not forget about the declarative meaning because
  - the declarative aspects are normally easier to formulated and understand
  - it is often rather easy to get a working program once we have a program that is declaratively correct
- A useful practical approach:
  - Concentrate on the declarative aspects of the program
  - Test the resulting program
  - If it fails procedurally try to rearrange the clauses and goals into a suitable order

Summary

- simple data objects (atoms, numbers, variables)
- structured objects
- matching as the fundamental operation on objects
- declarative (or non-procedural) meaning of a program
- procedural meaning of a Program
- relation between the declarative and procedural meanings of a program
- Altering the procedural meaning by reordering clauses and goals

Check your understanding

Given some 'point' facts as follows.

point(1,1). point(1,2). point(1,3).
point(2,1). point(2,2). point(2,3).
point(3,1). point(3,2). point(3,3).

Please define a **rectangle** relation to check if the following instances construct a rectangle.

a. (1,1), (1, 2), (2,1), (2,2)

b. (1,1), (1, 2), (1,3), (2,2)

Check your understanding

Given some ‘point’ facts as follows.

```
point(1,1).     point(1,2).  point(1,3).
point(2,1).     point(2,2).  point(2,3).
point(3,1).     point(3,2).  point(3,3).
```

Please define a **rectangle** relation to check if the following instances construct a rectangle.

a. |?- rectangle(point(1,1), point(2, 1), point(2,2), point(1,2)).

Yes

b. |?- rectangle(point(1,1), point(1, 2), point(1,3), point(2,2)).

no

Rule:

```
rectangle(point(X1,Y1), point(X2, Y2), point(X3,Y3), point(X4,Y4)) :- Y1=:=Y2,
    X2=:=X3, X4=:=X1, Y3=:=Y4.
```

Check your understanding

- Consider the following program:

```
f( 1, one).
f( s(1), two).
f( s(s(1)), three).
f( s(s(s(X))), N) :- f( X, N).
```

How the Prolog answer the following questions?

  - ?- f(s(1), A).
  - ?- f(s(s(1)), two).
  - ?- f(s(s(s(s(s(s(1)))))), C).
  - ?- f(D, three).

Check your understanding

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

- In which of the two cases does Prolog have to do more work before the answer is found?

  |?- big(X), dark(X).

  |?- dark(X), big(X).