UNICAM
Università di Camerino
1336

Logic and Constraint Programming

PROLOG

Prof. Fabrizio Fornari

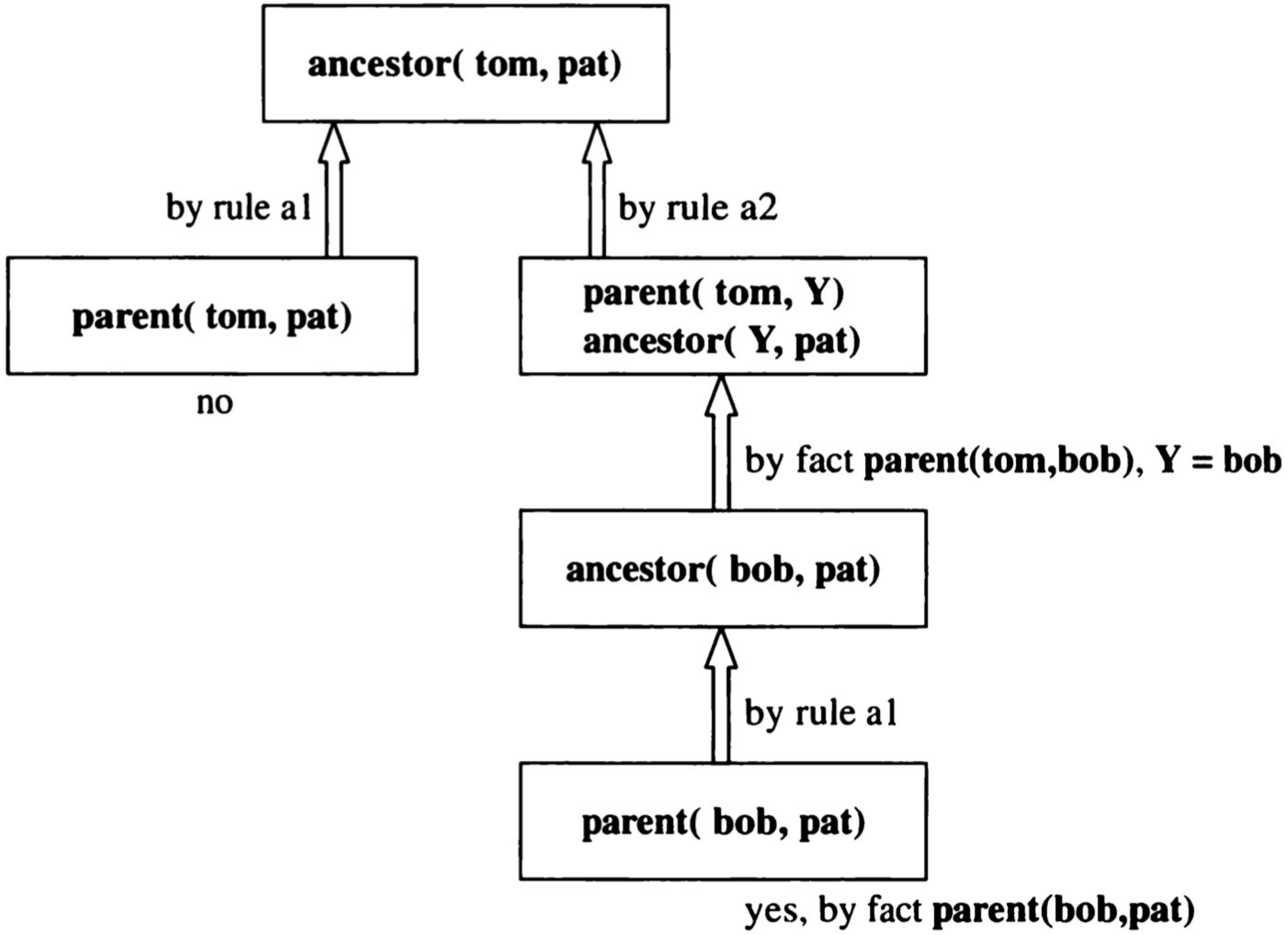
May 20, 2022

How Prolog answers questions

An execution trace
has the form of a tree.

The top goal is satisfied
when a path is found from
the root node (top goal) to
a leaf node labelled ‘yes’
(a goal that is satisfied).

The execution of Prolog
programs is the
searching for such paths.

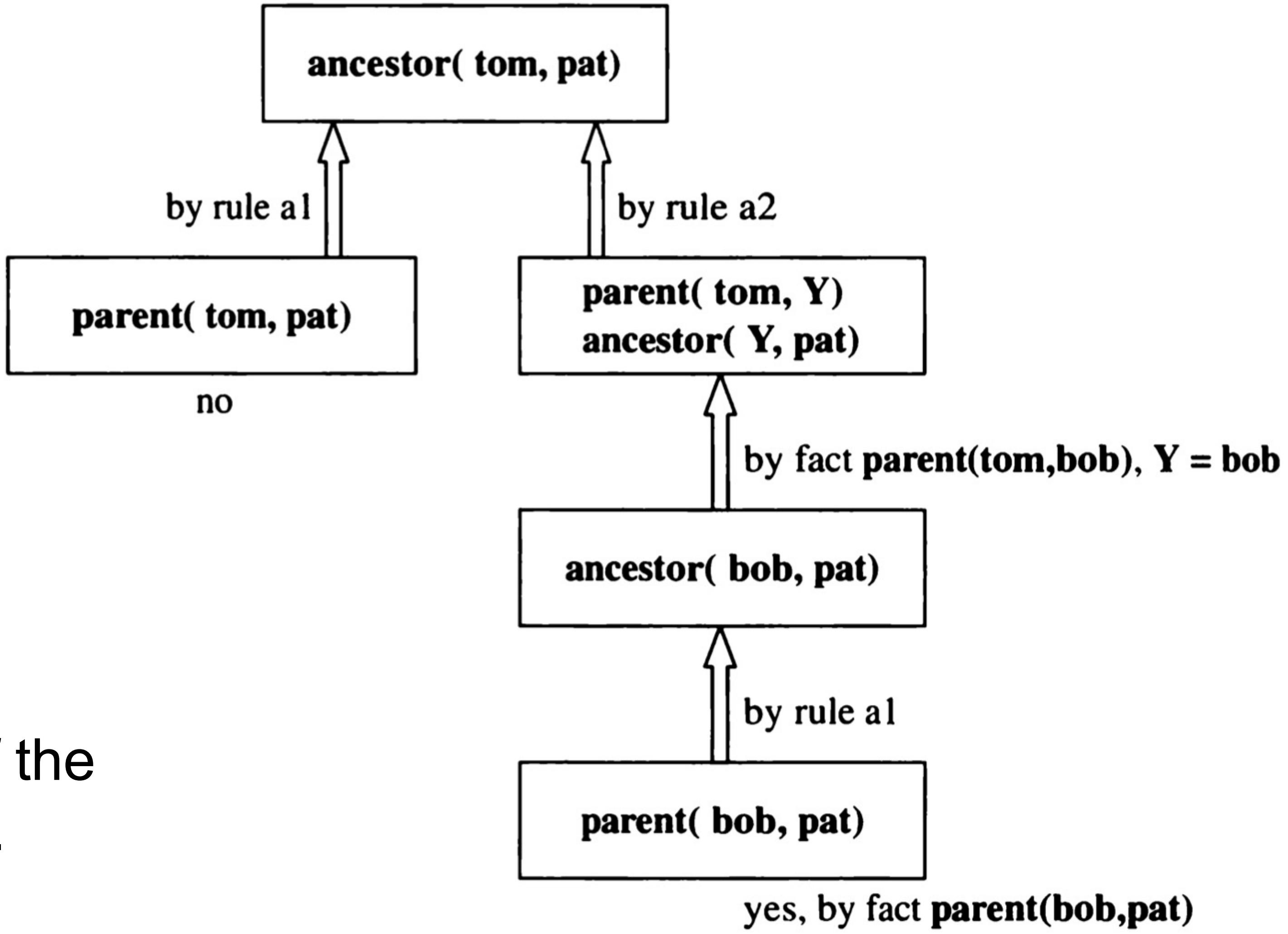


How Prolog answers questions

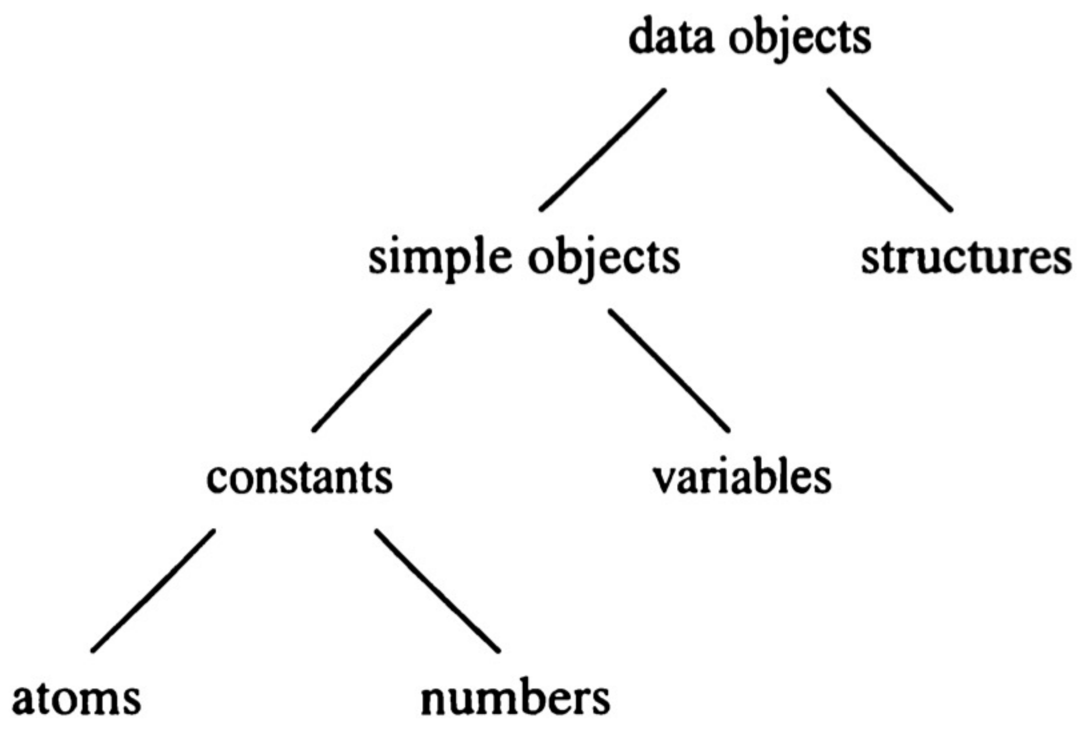
During the search Prolog may enter an unsuccessful branch.

When Prolog discovers that a branch fails it automatically *backtracks* to previous node and tries to apply an alternative clause at that node.

Automatic backtracking is one of the distinguishing features of Prolog.



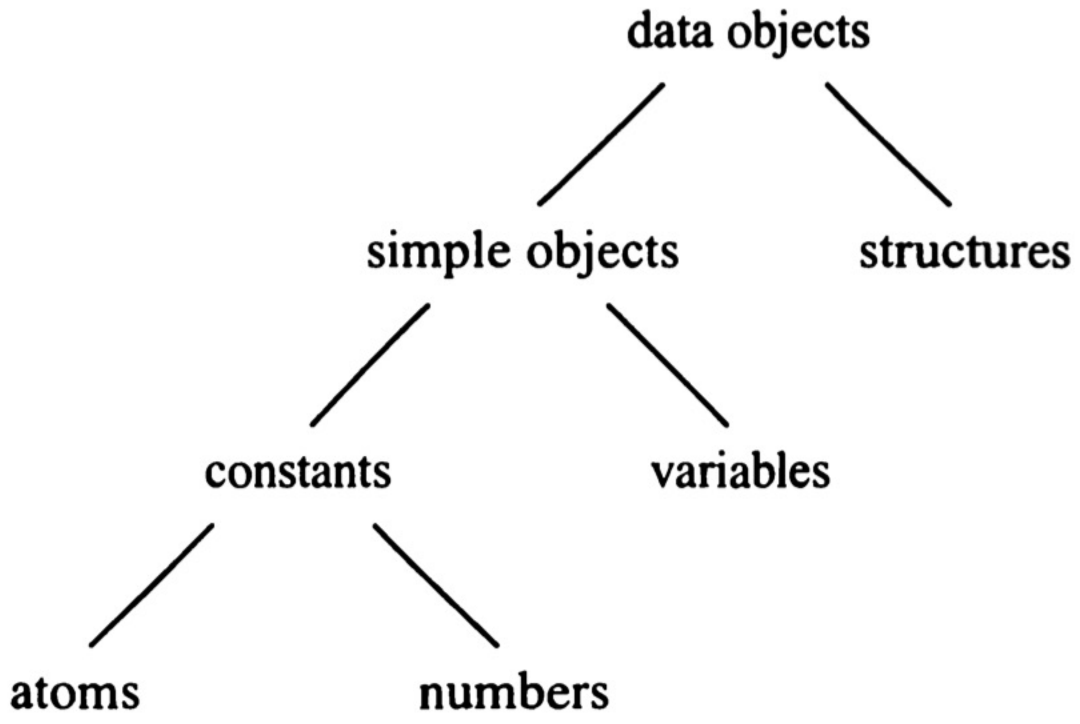
Syntax and Semantics of Prolog's concepts



Atoms

1. Strings of letters, digits, and the underscore character, '_', starting with lower-case letter:

anna	x_
nil	x___y
X25	alpha_beta_procedure
X_25	miss_Jones
X_25AB	

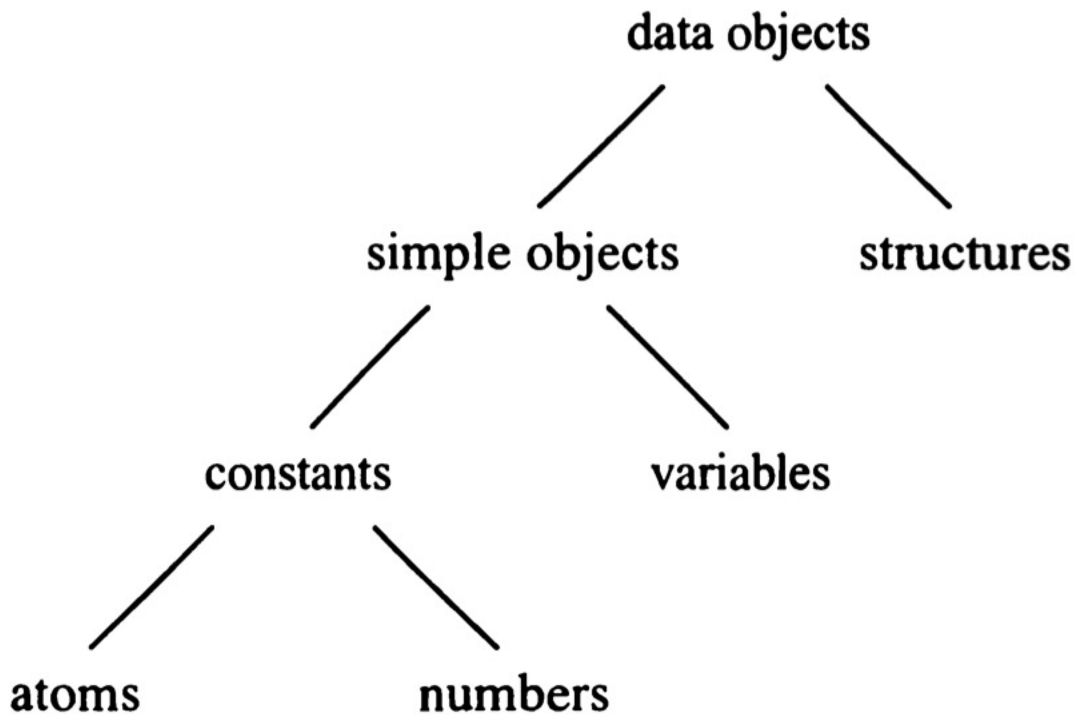


Atoms

2. Strings of special characters:

```
<--->
======>
+
...
.:.
::=
```

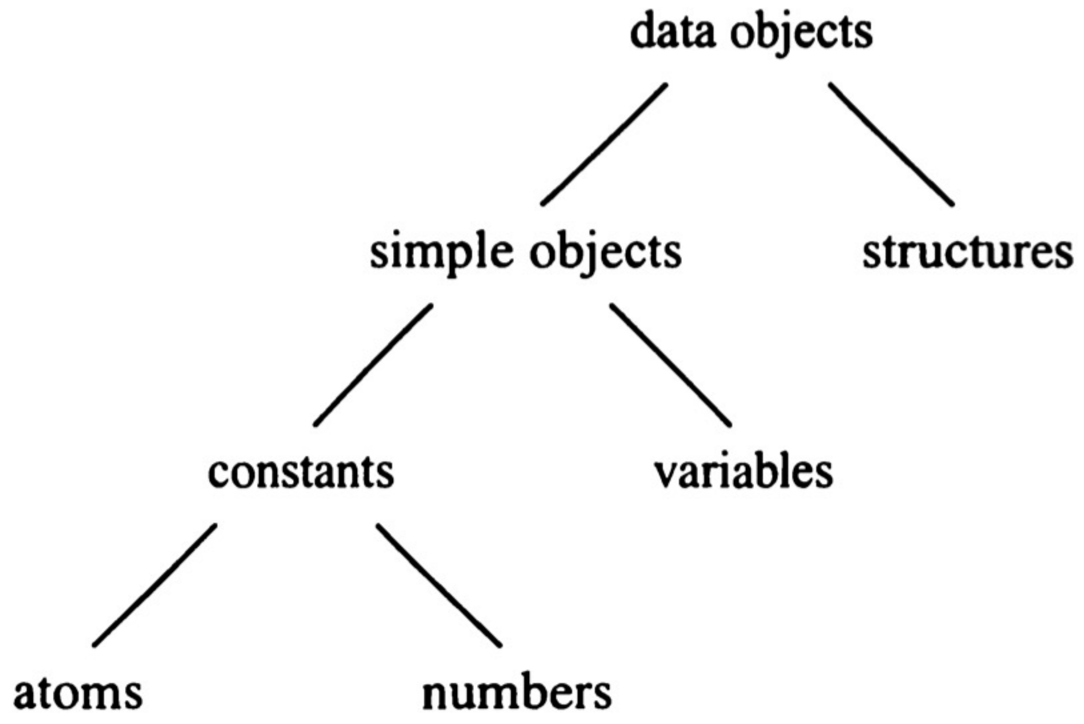
But some strings of special characters already have a predefined meaning. E.b., ‘:-’



Atoms

3. Strings of characters enclosed in single quotes:

‘Tom’
‘South_America’
‘Sarah Jones’



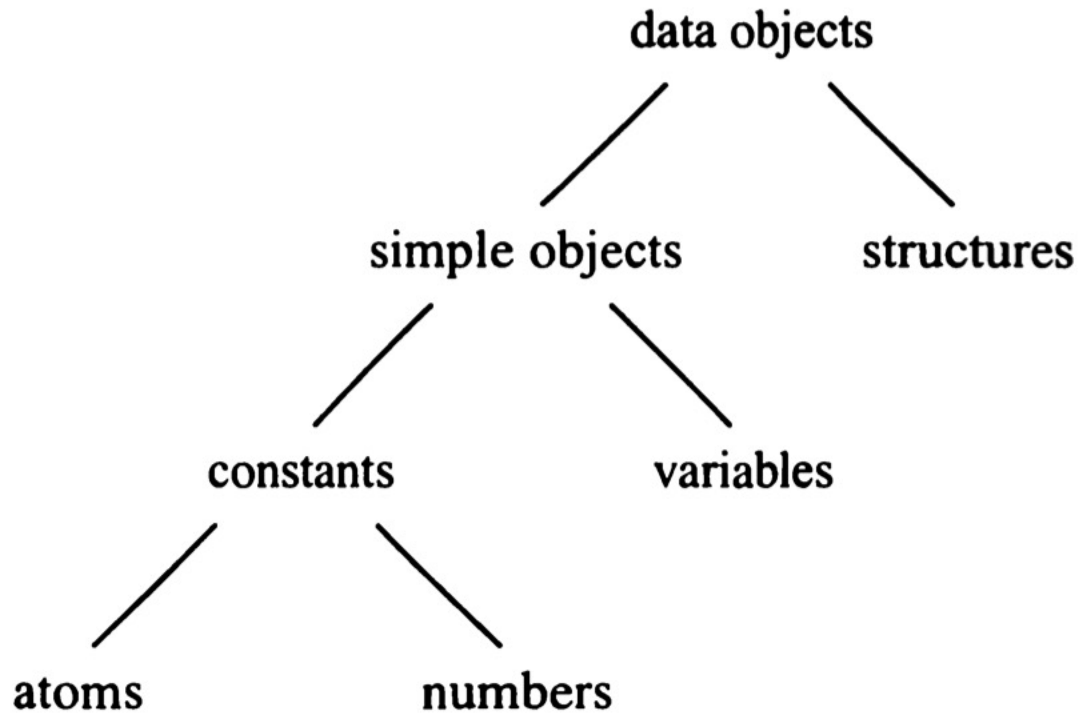
Numbers

Integer numbers:

1 1313 0 -97

Real numbers:

3.14 -0.0035 100.2. 7.15E-9

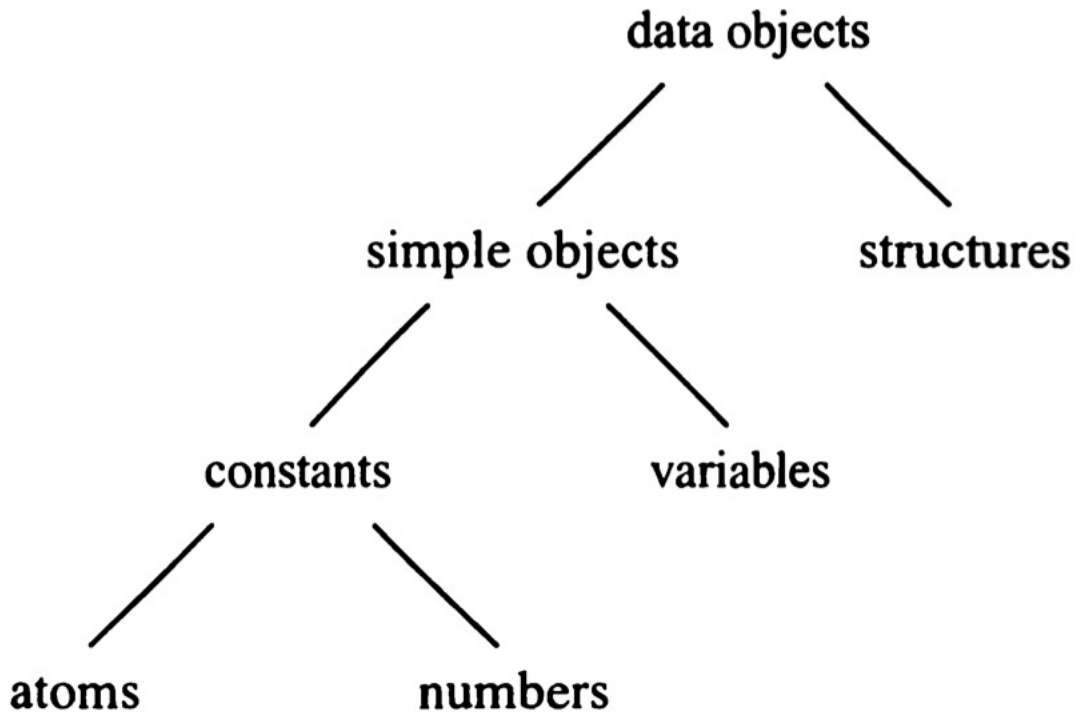


Variables

Variables are strings of letters, digits, and underscore characters.

They start with an upper-case letter or an underscore character:

X
Result
Object2
Participant_list
ShoppingList
_A
_x23
_23



Structures

Structures can be used to represent geometric objects

What defines a point?

Two coordinates

P1 = point(1,1)
P2 = point(2,3)

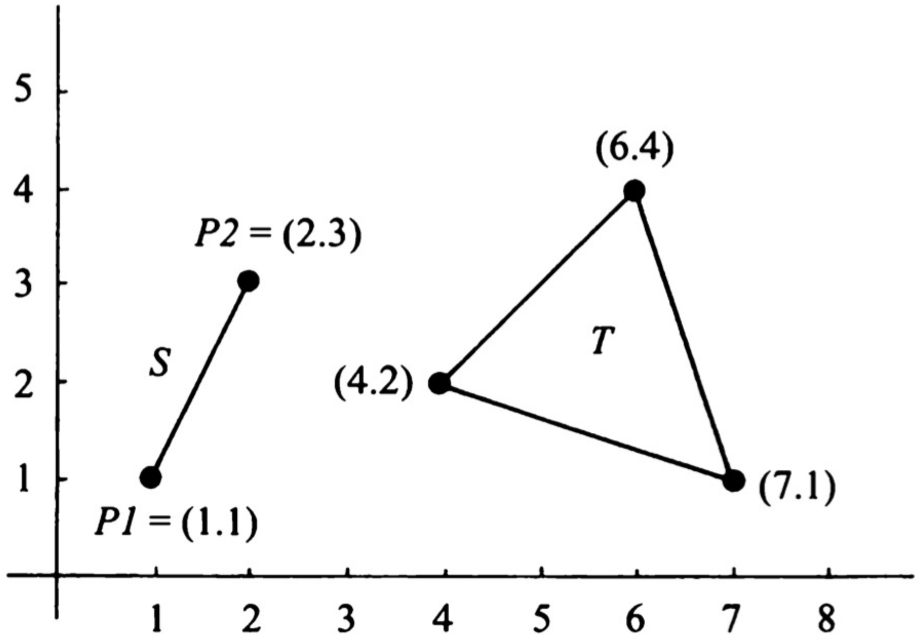
What defines a segment?

Two points

S = seg(P1, P2) = seg(point(1,1), point(2,3))

A triangle can be defined by three points

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



Structures

Structures can be used to represent geometric objects

What defines a point?

Two coordinates

P1 = point(1,1)
P2 = point(2,3)

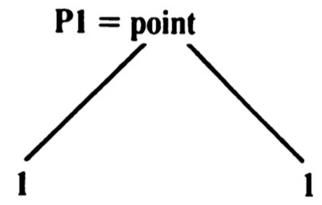
What defines a segment?

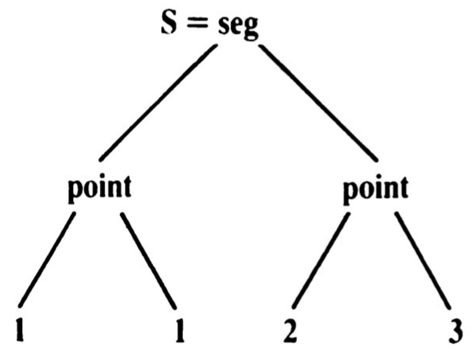
Two points

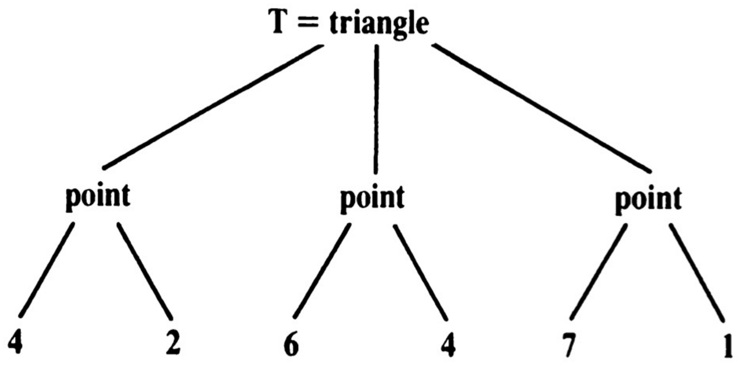
S = seg(P1, P2) = seg(point(1,1), point(2,3))

A triangle can be defined by three points

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







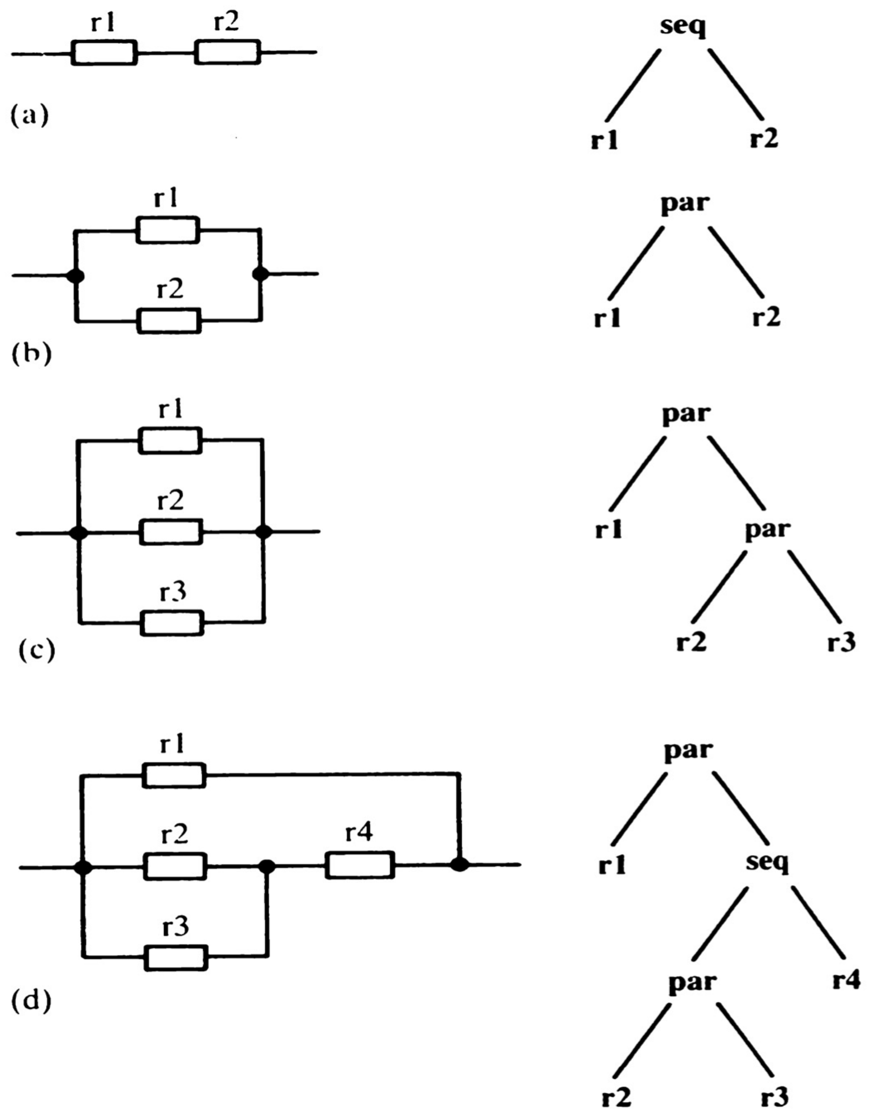
Prolog and Electric Circuits

Simple electric circuits

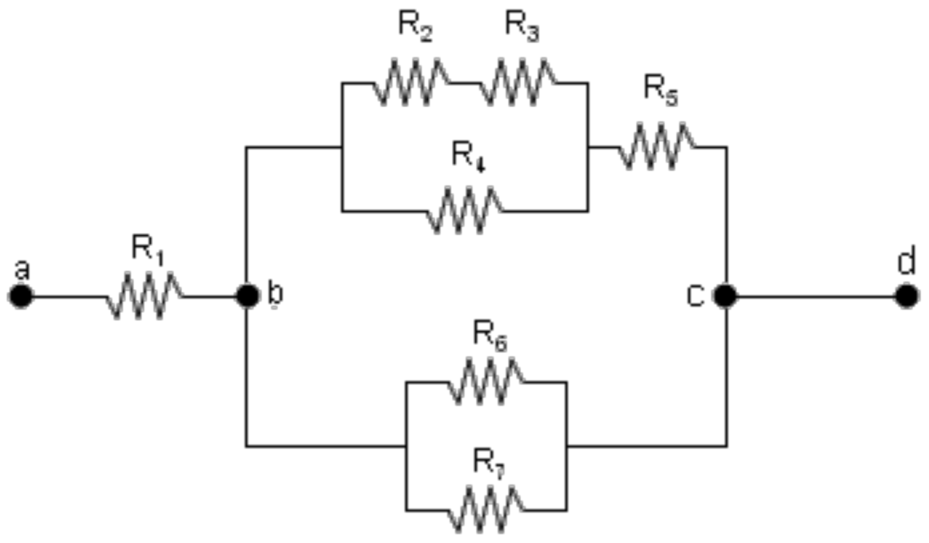
The atoms **r1**, **r2**, **r3** and **r4** are the names of the resistors.

The functor **par** and **seq** denote the parallel and the sequential compositions of resistors respectively. The corresponding Prolog terms are:

seq(r1, r2)
par(r1, r2)
par(r1, par(r2, r3))
par(r1, seq(par(r2, r3), r4))



Prolog and Electric Circuits



$R_1 = 7\ \Omega$ $R_2 = 5\ \Omega$ $R_3 = 11\ \Omega$ $R_4 = 8.67\ \Omega$ $R_5 = 5\ \Omega$ $R_6 = 7\ \Omega$ $R_7 = 35\ \Omega$

In parallel e.g., $\frac{R6 * R7}{R6 + R7}$

In series e.g., R2+R3

Finding Equivalent Resistance of a Resistive Circuit

Matching

Matching

The most important operation on terms is *matching*.

Given two terms, we say that they *match* if:

1. They are identical, or
2. The variables 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.

Matching

Matching takes as input two terms and checks whether they match.

E.g., we can ask Prolog: **?- date(D, M, 2001) = date(D1, may, Y1).**

D = D1,	D is instantiated to D1
M = may,	M is instantiated to may
Y1 = 2001.	Y1 is instantiated to 2001

Matching

Matching takes as input two terms and checks whether they match.

E.g., we can ask Prolog: **?- date(D, M, 2001) = date(D1, may, Y1).**

D = 1	D = third
D1 = 1	D1 = third
M = may	M = may
Y1 = 2001	Y1 = 2001

These instantiations also make both terms identical. But they are *less general*.

Prolog always results in the *most general* instantiation.

Matching

Consider the following question:

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

To satisfy the first goal, Prolog instantiates the variables as follows:

D = D1

M = may

Y1 = 2001

After having satisfied the second goal, the instantiation becomes more specific as follows:

D = 15

D1 = 15

M = may

Y1 = 2001

Y = 2001

Matching

The general rules to decide whether two terms, S and T, match are as follows:

1. If S and T are constants then S and T match only if they are the same object.

2. If S is a variable and T is anything, then they match,and S is instantiated to T. Conversely, if T is a variable then T is instantiated to S.

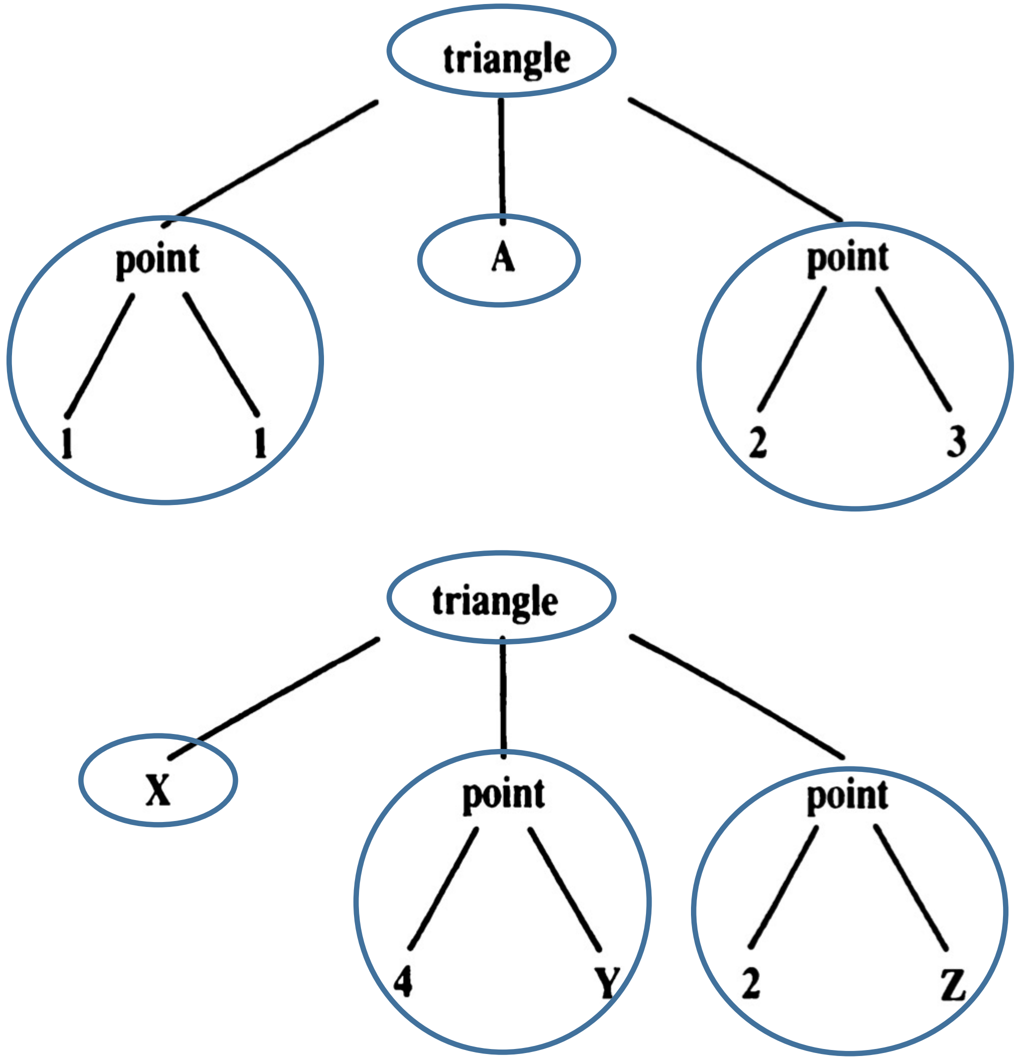
3. If S and T are structures then they match only if
(a) S and T have the same principal functor, and
(b) All their corresponding components match.
The resulting instantiation is determined by the matching of the components

Matching

We can visualize rule n°3

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

The result is:
X = point(1,1)
A = point(4,Y)
Z = 3



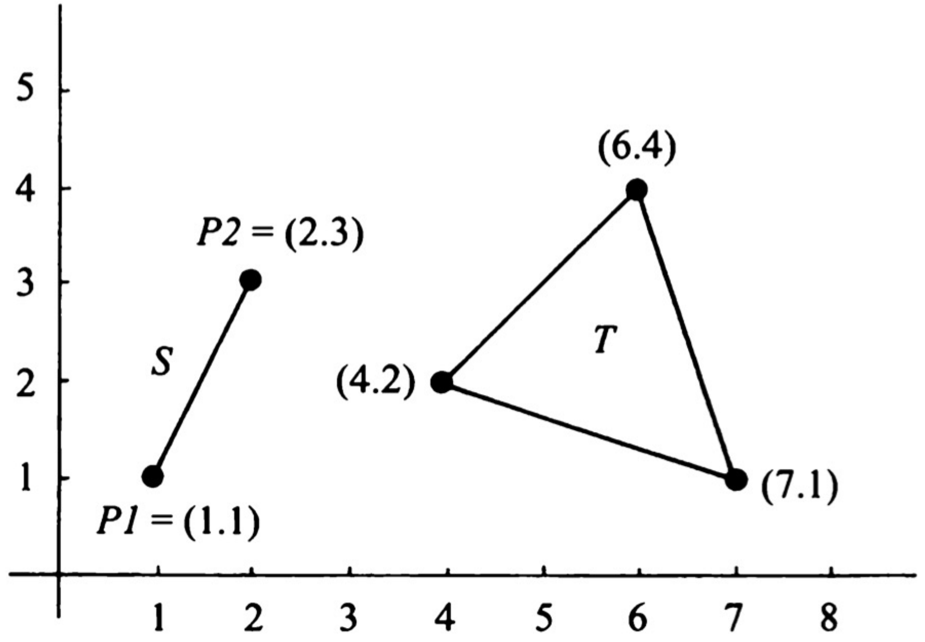
Matching

triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z))

Matching

Geometric objects

```
point(1,1).
point(2,3).
seg( point(1,1), point(2,3)).
triangle( point(4,2), point(6,4), point(7,1)).
```



Let us define a piece of program for recognizing horizontal and vertical line segments.

Matching

Let us define a piece of program for recognizing horizontal and vertical line segments.

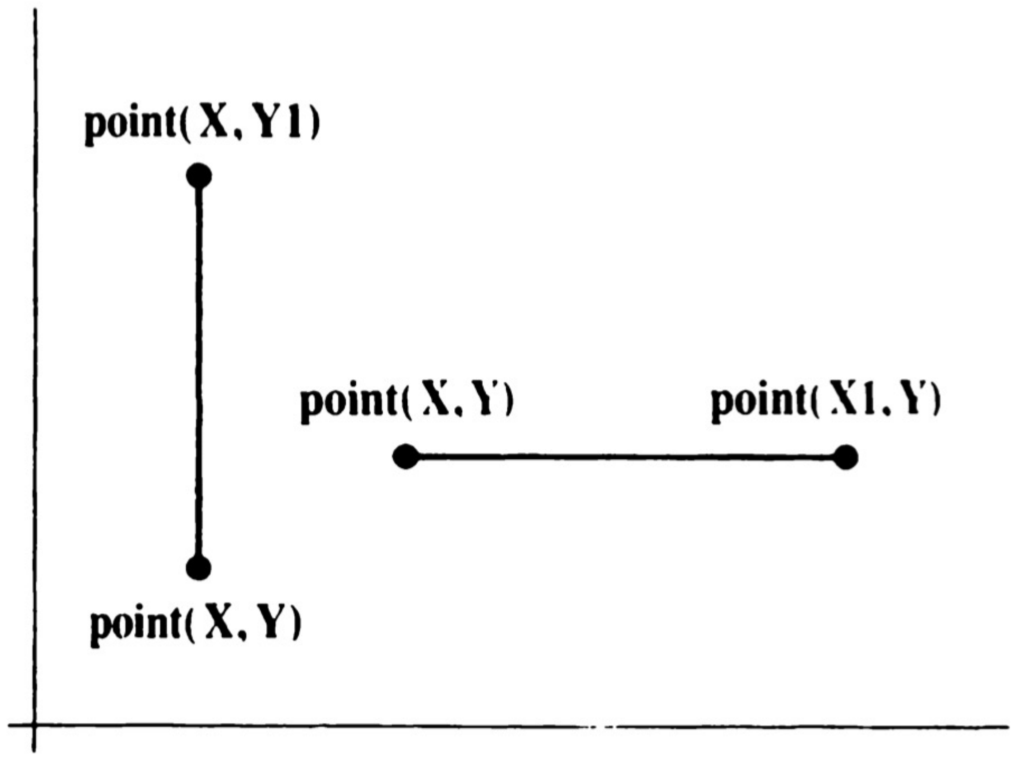
```
?- vertical( seg( point(1,1), point(1,2) ) ).
?- vertical( seg( point(1,1), point(2,Y) ) ).
?- horizontal( seg( point(1,1), point(2,Y) ) ).
```

'Vertical' is a property of segments, so it can be formalized as a unary relation.

```
vertical( seg( point(X1,Y1), point(X1,Y2) ) ).
```

'Horizontal' is a property of segments, so it can be formalized as a unary relation.

```
horizontal( seg( point(X1,Y1), point(X2,Y1) ) ).
```



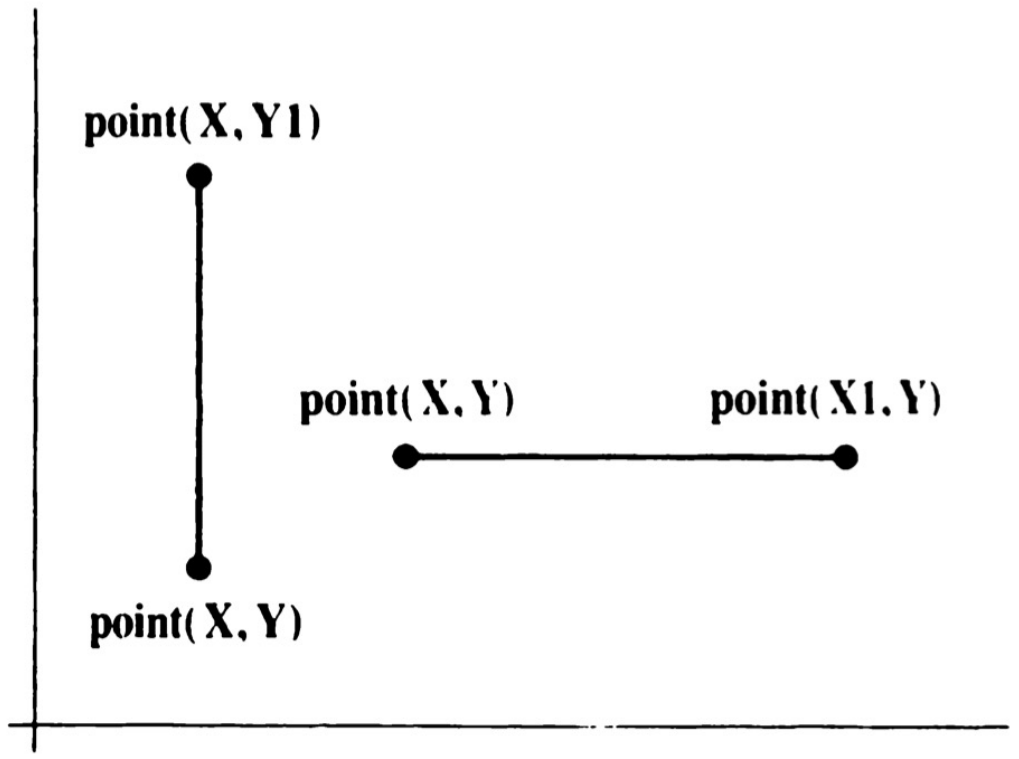
Matching

Let us define a piece of program for recognizing horizontal and vertical line segments.

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

?- vertical( seg( point(1,1), point(2,Y) ) ).

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



What are we asking?

Are there any vertical segments that start at the point(2,3)?

```
?- vertical( seg( point(2,3), P) ).

P = point(2, _).
```

Matching

Let us define a piece of program for recognizing horizontal and vertical line segments.

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

?- vertical( seg( point(1,1), point(2,Y) ) ).
?- horizontal( seg( point(1,1), point(2,Y) ) ).
```

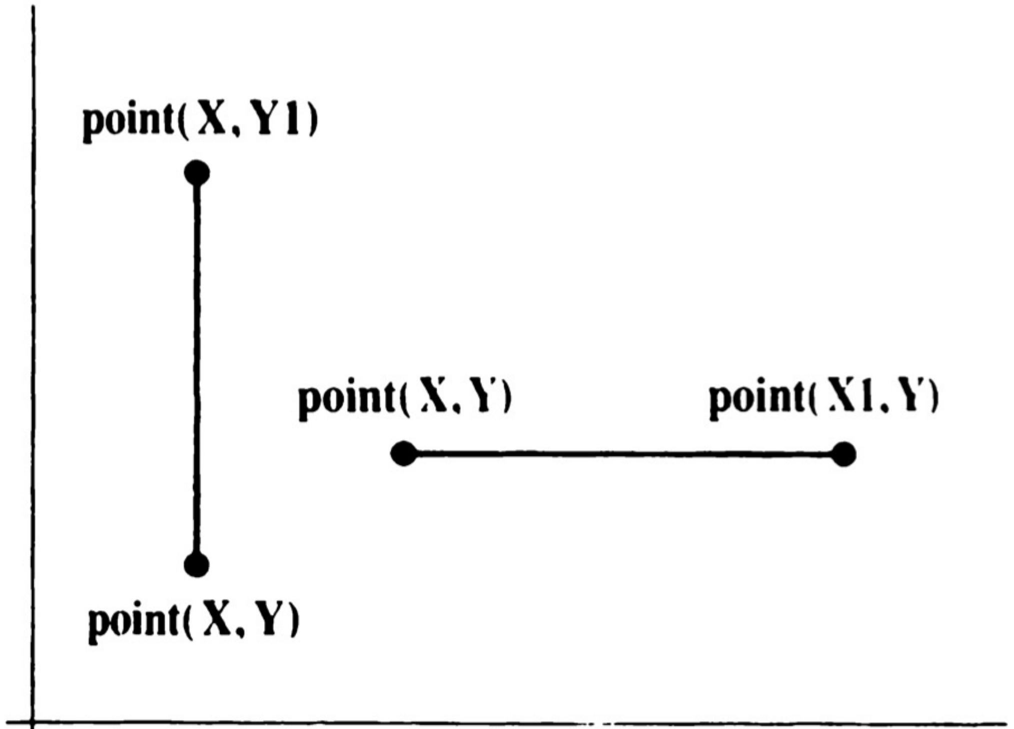
What are we asking?

Is there a segment that is both vertical and horizontal?

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

S = seg(point(_A, _B), point(_A, _B)).
```

A single point



Declarative and Procedural meaning of Prolog programs

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

Declarative readings:	P is true if Q and R are ture.
	From Q and R follows P.

Alternative procedural readings:

To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.
To satisfy P, *first* satisy Q and *then* R.

The difference is that the latter also defines the *order* in which the goals are processed.

Declarative meaning of Prolog programs

Determines whether a given goal is true, and if so, for what values of variables it is true.

Given a program and a goal G, the declarative meaning says the following.

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if:

(1) there is a clause C in the pogram 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.

Declarative meaning of Prolog programs

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if:

(1) there is a clause C in the pogram 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.

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

Instances of this clause are:

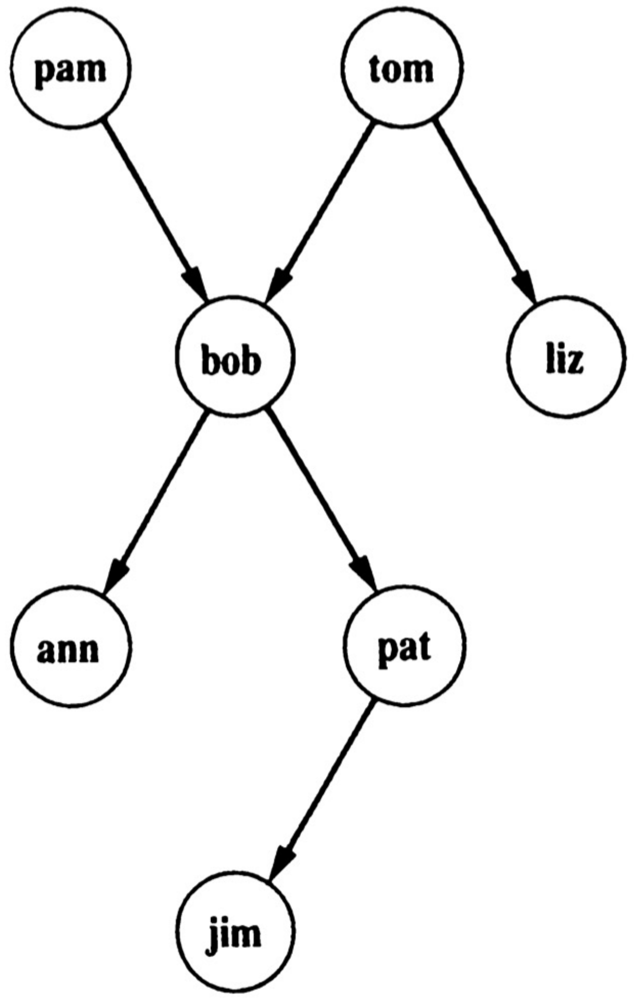
```
ancestor( pam, Z) :- parent( pam, Z).
ancestor( pam, bob) :- parent( pam, bob ).
```

Declarative meaning of Prolog programs

Define the *relatives* relation

Two people are relatives if

(a) one is an ancestor of the other, or
(b) they have a common ancestor, or
(c) they have a common successor.

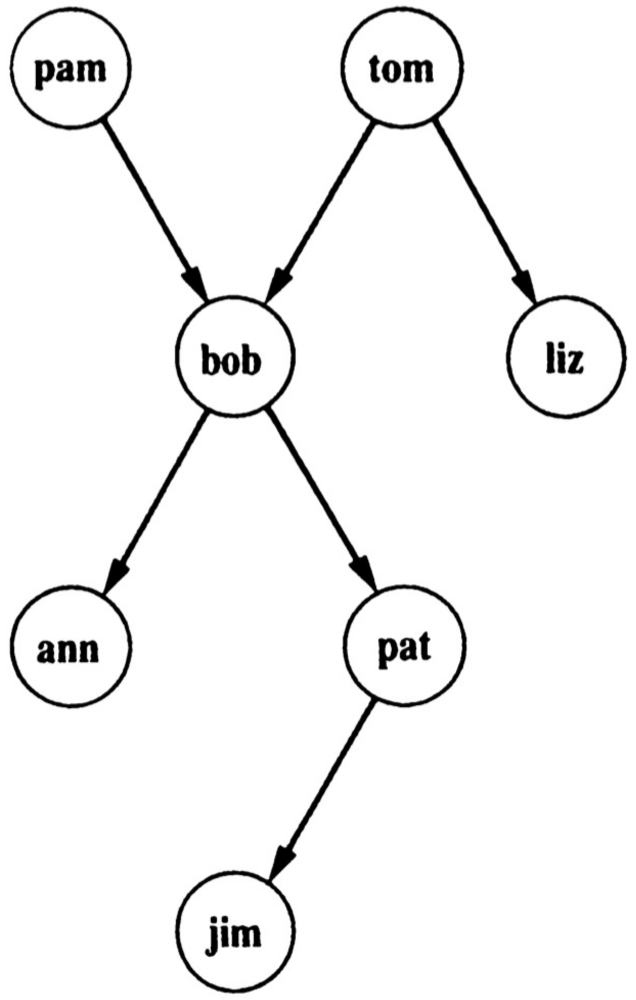


Declarative meaning of Prolog programs

Two people are relatives if

(a) one is an ancestor of the other, or
(b) they have a common ancestor, or
(c) they have a common successor.

?- relatives(pam, bob).
?- relatives(pam, tom).
?- relatives(pam, liz).



Declarative meaning of Prolog programs

Two people are relatives if

(a) one is an ancestor of the other, or
(b) they have a common ancestor, or
(c) they have a common successor.

```
?- relatives(pam, bob).
?- relatives(pam, tom).
?- relatives(pam, liz).
```

```
relatives( X, Y) :-  % X is ancestor of Y
ancestor( X, Y).

relatives( X, Y) :-  % Y is an ancestor of X
ancestor( Y, X).

relatives( X, Y) :- % X and Y have a common ancestor
ancestor( Z, X),
ancestor( Z, Y).

relatives( X, Y) :- % X and Y have a common successo
ancestor( X, Z),
ancestor( Y, Z).
```

Declarative meaning of Prolog programs

A comma between goals denotes the *conjunction* of goals:

they *all* have to be true.

Prolog also accepts the *disjunction* of goals:

any one of the goals in a disjunction has to be true. Disjunction is indicated by a semicolon “;”.

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

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

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

Declarative meaning of Prolog programs

Simplify the *relative* relation by using “;”

(a) one is an ancestor of the other, or
(b) they have a common ancestor, or
(c) they have a common successor.

```
?- relatives(pam, bob).
?- relatives(pam, tom).
?- relatives(pam, liz).
```

```
relatives( X, Y) :-  % X is ancestor of Y
ancestor( X, Y).

relatives( X, Y) :-  % Y is an ancestor of X
ancestor( Y, X).

relatives( X, Y) :- % X and Y have a common ancestor
ancestor( Z, X),
ancestor( Z, Y).

relatives( X, Y) :- % X and Y have a common successor
ancestor( X, Z),
ancestor( Y, Z).
```

Declarative meaning of Prolog programs

The *relative* relation by using “;”

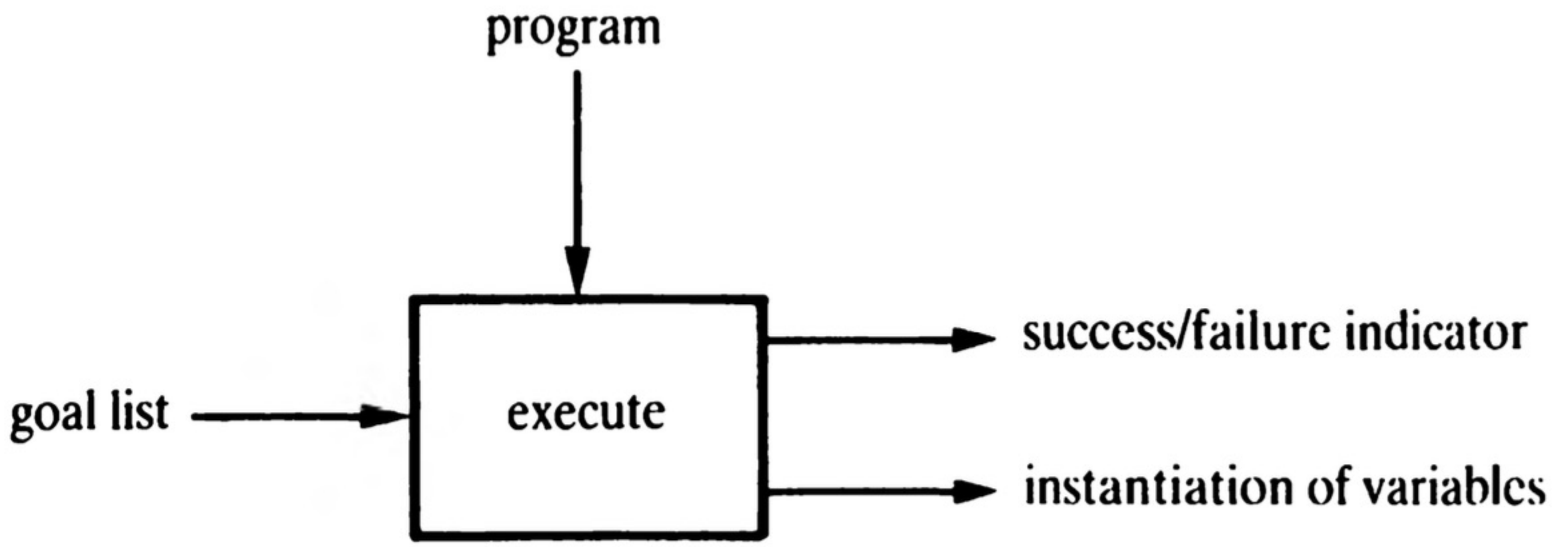
```
relatives( X, Y) :-
   ancestor( X, Y) % X is ancestor of Y
   ;
   ancestor( Y, X) % Y is an ancestor of X
   ;
   ancestor( Z, X), % X and Y have a common ancestor
   ancestor( Z, Y)
   ;
   ancestor( X, Z), % X and Y have a common successor
   ancestor( Y, Z).
```

```
?- relatives(pam, bob).
?- relatives(pam, tom).
?- relatives(pam, liz).
```

Procedural meaning of Prolog programs

The procedural meaning specifies *how* Prolog answers questions.

To answer a questions means to try to satisfy a list of goals. Thus the porcedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To «execute goals» means: try to satisfy them.



Let us write a program

Write a program that describes the following.

bears, and elephants are big, cats are small.

bears are brown, cats are black and elephants are gray.

Anything black is dark; anything brown is dark.

Who is dark and big?

Let us write a program

Our program

bears, and elephants are big, cats are small.

```
big( bears).
big( elephants).
small( cats).
```

bears are brown, cats are black and elephants are gray.

```
brown( bears).
black( cats).
gray( elephants).
```

Let us write a program

Anything black is dark; anything brown is dark.

```
dark( Z) :-
    black( Z).


dark( Z) :-
    brown( Z).
```

Who is dark and big?

```
?- dark( X), big( X).          X = bears.
```

Let us write a program

Our Program

```
big( bears).         % Clause 1
big( elephants).     % Clause 2
small( cats).        % Clause 3

brown( bears).       % Clause 4
black( cats).        % Clause 5
gray( elephants).    % Clause 6
```

```
dark( Z) :-     % Clause 7: Anything black is dark
   black( Z).

dark( Z) :-     % Clause 8: Anything brown is dark
   brown( Z).
```

?- dark(X), big(X). Which is the execution trace for answering such a question?

Execution Trace

1. Initial goal list: **dark(X), big(X).**

2. Scan the program from top to bottom looking for a clause whose head matches the first goal **dark(X)**. Found clause 7:

    dark(Z) :- black(Z).

  Replace the first goal by the instantiated body of clause 7

    black(X), big(X)

3. Scan the program to find a match with **black(X)**. Clause 5 found: **black(cats)**. This clause has no body, so the goal list, properly instantiated, shrinks to:

    big(cats)

Execution Trace

4. Scan the program for the goal **big(cats)**. No clause found. Therefore backtrack to step 3 and undo the istantiation **X = cats**. Now the goal list is again:

black(X), big(X)

Continue scanning the prgram below clause 5. No clause found. Therefore backtrack to step 2 and continue scanning below clause 7. Clause 8 is found:

dark(Z) :- brown(Z).

Replace the first goal in the goal list by **brown(X)**, giving:

brown(X), big(X)

Execution Trace

5. Scan the program to match **brown(X)**, finding **brown(bear).** This clause has no body, so the goal list shrinks to:

big(bears)

6 Scan the program and find clause **big(bear)**. It has no body so the goa list shrinks to empty. This indicates successful termination, and the corresponding variable instantiation is:

X = bear

Execution Procedure

procedure *execute* (*Program*, *GoalList*, *Success*);

Input arguments:
  Program: list of clauses
  GoalList: list of goals
Output argument:
  Success: truth value; *Success* will become true if *GoalList* is true with respect to *Program*
Local variables:
  Goal: goal
  OtherGoals: list of goals
  Satisfied: truth value
  MatchOK: truth value
  Instant: instantiation of variables
  H, *H'*, *B1*, *B1'*, . . . , *Bn*, *Bn'*: goals
Auxiliary functions:
  empty(L): returns true if *L* is the empty list
  head(L): returns the first element of list *L*
  tail(L): returns the rest of *L*
  append(L1,L2): appends list *L2* at the end of list *L1*
  match(T1,T2,MatchOK,Instant): tries to match terms *T1* and *T2*; if succeeds
    then *MatchOK* is true and *Instant* is the corresponding instantiation of variables
  substitute(Instant, Goals): substitutes variables in *Goals* according to instantiation *Instant*

```
begin
  if empty(GoalList) then Success := true
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      Satisfied := false;
      while not Satisfied and "more clauses in program" do
        begin
          Let next clause in Program be
            H :- B1, . . . , Bn.
          Construct a variant of this clause
            H' :- B1', . . . , Bn'.
          match(Goal,H',MatchOK,Instant);
          if MatchOK then
            begin
              NewGoals := append([B1', . . . ,Bn'], OtherGoals);
              NewGoals := substitute(Instant,NewGoals);
              execute(Program,NewGoals,Satisfied)
            end
        end;
      Success := Satisfied
    end
end;
```

Order of clauses and goals

Write a program with the following clause.

```
p :- p.
```

“p is true if p is true”

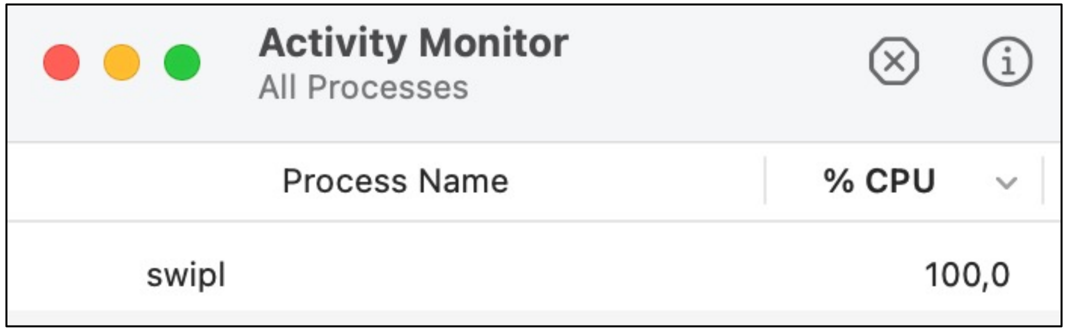
It is declaratively correct.

ask the following question.

```
?- p.
```

Which is the answer?

Open task manager or activity monitor



Process Name	% CPU
swipl	100,0

Infinite loop

Write a program with the following clause.

```
p :- p.
```

“p is true if p is true”

Is is procedurally useless

It results in an infinite loop

Infinite loops are not unusual in other programming languages.

A Prolog program may be declaratively correct, but at the same time be procedurally incorrect.

It may be not able to produce an answer to a question although the answer exists.

Prolog might choose a wrong path and the path could be infinite.

Infinite Loop

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
   parent( X, Z).

ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
   parent(X, Y),
   ancestor( Y, Z).
```

Infinite Loop

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
   ancestor( Y, Z),
   parent(X, Y).

ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
   parent( X, Z).
```

Which is the execution trace of **ancestor(tom, pat)**?

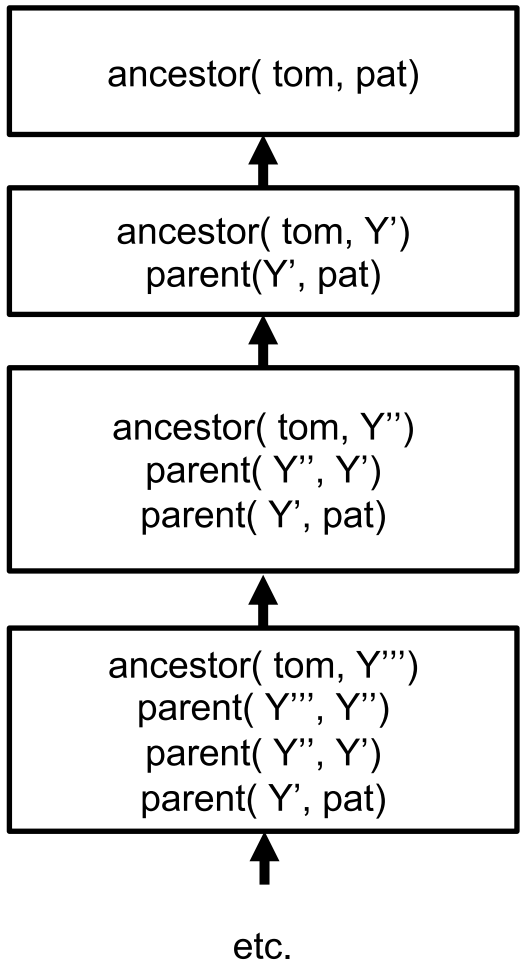
Try to graphically design the execution trace.

Infinite Loop

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

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

ancestor(tom, pat) execution trace.



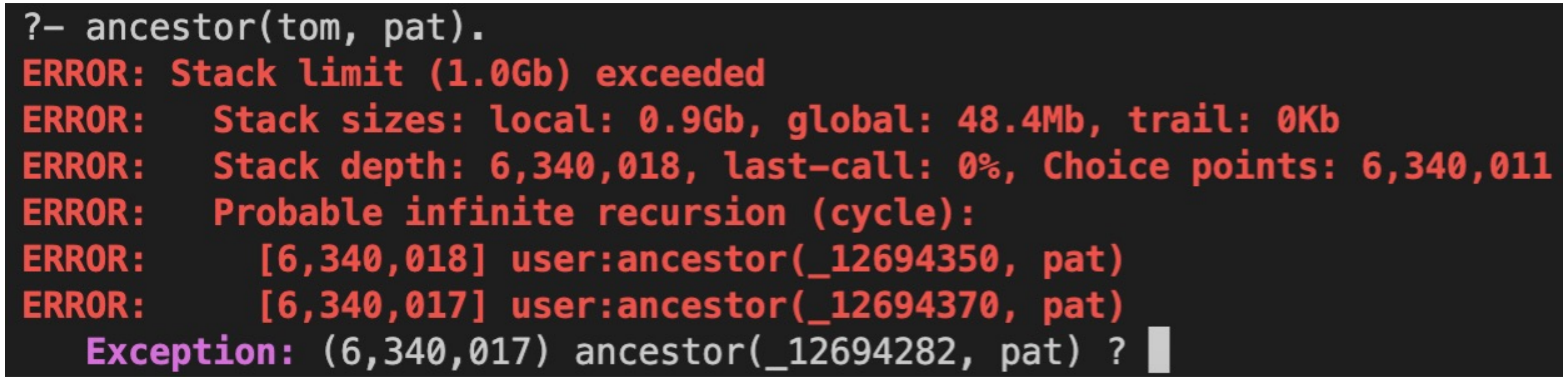
Infinite Loop

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

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

What if we ask ancestor(tom, pat) to Prolog?

```
?- ancestor( tom, pat)
```



```
?- ancestor(tom, pat).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 48.4Mb, trail: 0Kb
ERROR:   Stack depth: 6,340,018, last-call: 0%, Choice points: 6,340,011
ERROR:   Probable infinite recursion (cycle):
ERROR:     [6,340,018] user:ancestor(_12694350, pat)
ERROR:     [6,340,017] user:ancestor(_12694370, pat)
   Exception: (6,340,017) ancestor(_12694282, pat) ?
```

A general heuristic in problem solving

It is usually best to try the simplest idea first.

1. the simpler idea is to check whether the two arguments of the **ancestor** relation satisfy the **parent** relation;

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
    parent( X, Z).
```

2. the more complicated idea is to find somebody “between” both people (somebody who is related to them by the **parent** and **ancestor** relations).

```
ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
    parent(X, Y),
    ancestor( Y, Z).
```

Summary

What we covered until now is called “pure Prolog” since it corresponds cosely to formal logic.

We will cover extensions for tailoring the language toward practical needs in the upcoming lectures.

Simple objects in Prologs are: *atoms*, *variables*, and *numbers*. Structured objects are used to represent objects that have several components.

Structures are constructed by means of *functors*. Each functor is defined by its name and arity (e.g., point(X1, Y1) and point(X, Y, Z))

Summary

The type of object is recognized by its syntactic form.

The *lexical* scope of variables is one clause. Thus the same variable name in two clauses means two different variables.

Structures can be naturally pictured as trees. Prolog can be viewed as a language for processing trees.

The *matching* operation takes two terms and tries to make them identical by instantiating the variables in both terms.

Matching, if it succeeds, results in the *most general* instantiation of variables.

Summary

The *declarative semantics* of Prolog defines whether a goal is true with respect to a given program, and if it is true, for what instantiation of variables it is true.

A comma between goals means the conjunction of goals. A semicolon between goals means the disjunction of goals.

The *procedural semantics* of Prolog is a procedure for satisfying a list of goals in the context of a given program. The procedure outputs the truth or falsity of the goal list and the corresponding instantiations of variables. The procedure automatically backtracks to examine alternatives.

Summary

The declarative meaning of programs in «pure Prolog» does not depend on the order of clauses and the order of goals in clauses.

The procedural meaning does depend on the order of goals and clauses. Thus the order can affect the efficiency of the program; an unsuitable order may even lead to infinite recursive calls

Given a declarative correct program, changing the order of clauses and goals can improve the program’s efficiency while retaining its declarative correctness. Reordering is one method of preventing indefinite looping.

Representation of lists

The *list* is a simple data structure widely used in non-numeric programming.

A sequence list is a sequence of any number of items, such as:

[ann, tennis, tom, skiing]

Remember: all structured objects in Prolog are trees. Lists are no exception to this.

Representation of lists

A list can be empty or non-empty.

Empty: [ ] .

Non-Empty:

1. the first items is called the *head* of the list;
2. the remaining part of the list is called the *tail*.

$$\textbf{[}\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ \textbf{]}$$

Representation of lists

$$[\ \underbrace{\textbf{ann}}_{\text{Head}},\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ ]$$

The tail has to be a list. The head and the tail are combined into a structure by the functor “.”:

.(Head, Tail)

Since **Tails** is in turn a list, it is either empty or it has its own head and tail

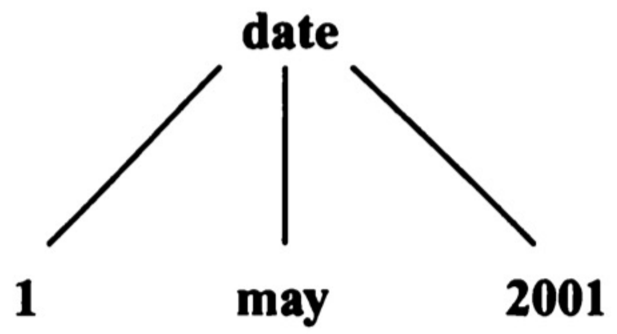
Representation of lists

$$\textbf{[}\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ \textbf{]}$$

.(Head, Tail)

Which is the graphical representation of the tree structure?

Remember the examples we made:
date(1, may, 2001)



Representation of lists

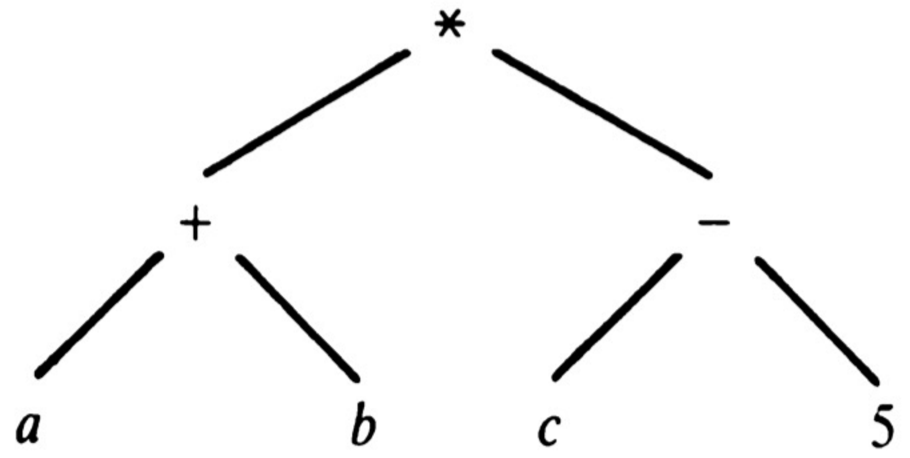
$$[\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ ]$$

.(Head, Tail)

Which is the graphical representation of the tree structure?

Remember the examples we made:

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

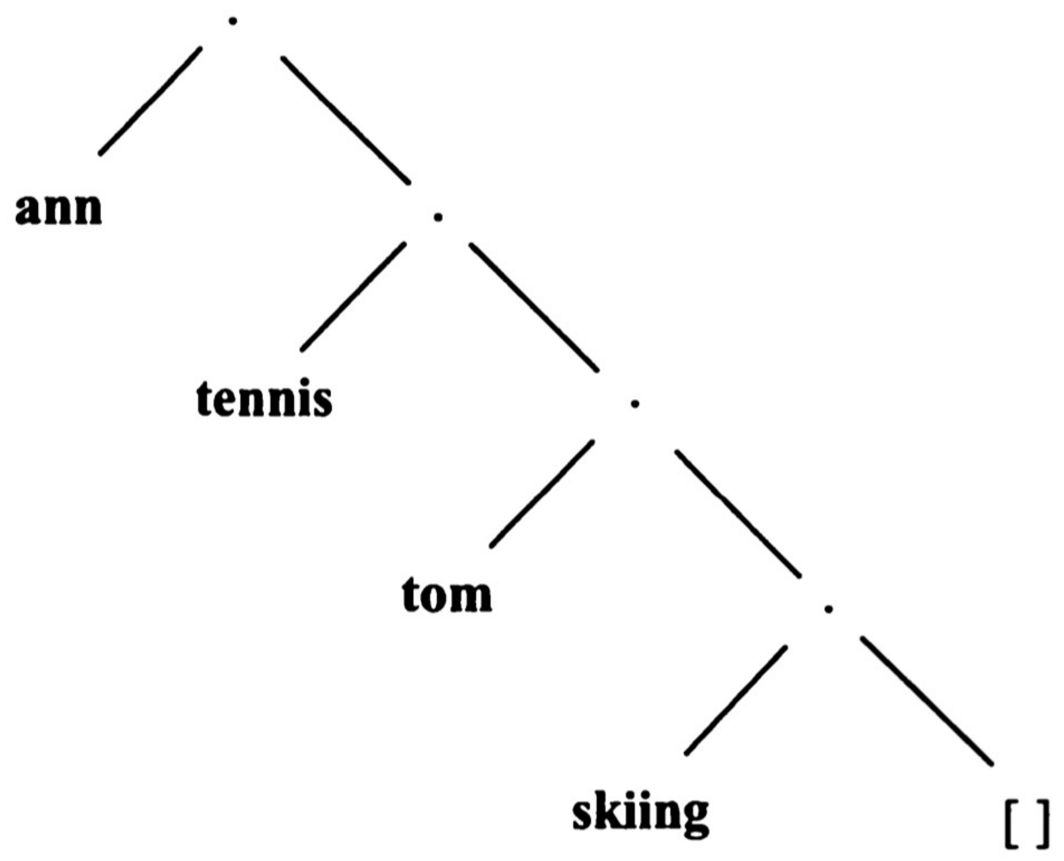


Representation of lists

$$[\ \underbrace{\text{ann,}}_{\text{Head}}\ \underbrace{\text{tennis, tom, skiing]}}_{\text{Tail}}$$

$$.(\ \text{ann},\ .(\text{tennis},\ .(\ \text{tom},\ .(\ \text{skiing},\ \underbrace{[\ ]}_{\text{Empty list}}\ )\ )\ )\ )$$

.(Head, Tail)

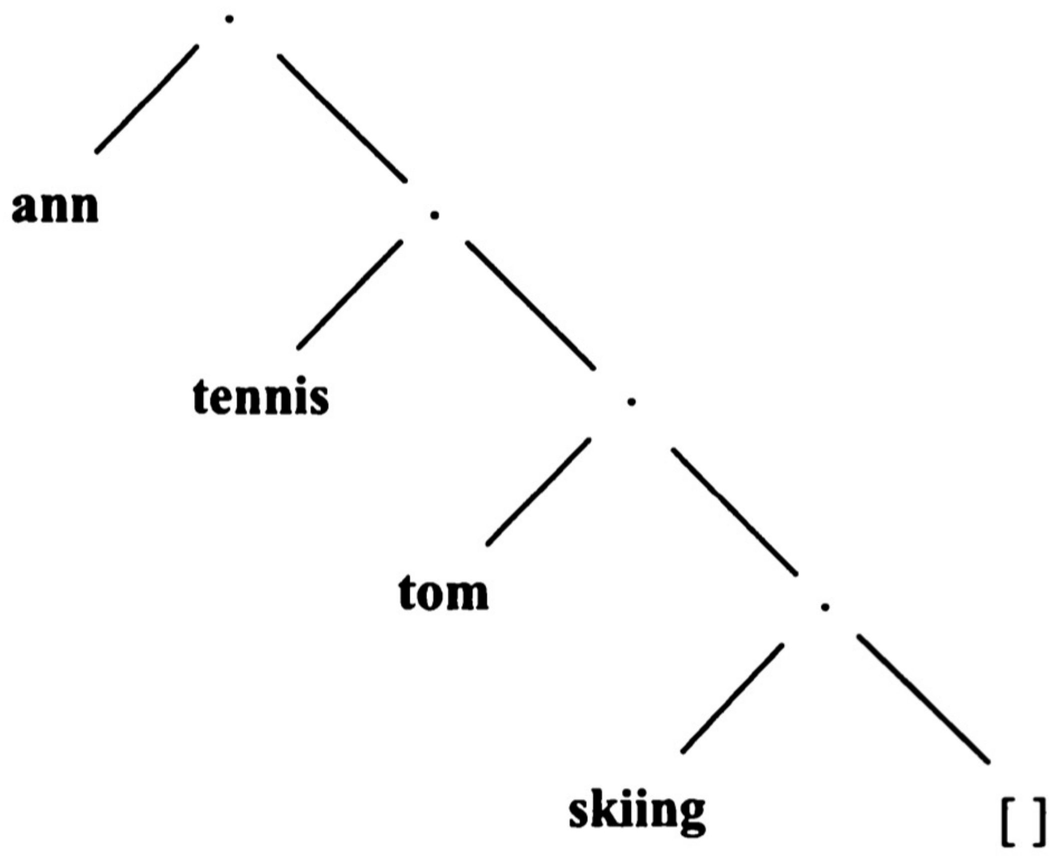


Representation of lists

Which one do you prefer?

[ann, tennis, tom, skiing]

.(ann, .(tennis, .(tom, .(skiing, [ ]))))



Operations on lists

Checking whether some object is an element of a list

Concatenation of two lists, obtaining a third one

Adding a new object to a list, or deleting an object from it.

Membership

Let us implement the membership relation as:

member(X, L)

where X is an object and L is a list. The goal **member(X, L)** is true if X occurs in L.

Some examples:

member(b, [a,b,c]) is true

member(b, [a,[b,c]]) is true

member([b,c], [a,[b,c]]) is true

Membership

The program for the membership relation can be based on the observation:

X is a member of L if either:

(1) X is the head of L, or

(2) X is a member of the tail of L.

This can be written in two clauses:

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

member( X, [Head | Tail] ) : -
   member( X, Tail).
```

Membership

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

member( X, [Head | Tail] ) : -
   member( X, Tail).
```

Let us aks whether b is in [a,b,c]

```
?- member( b, [ a, b, c]).            % Check whether b is in [a,b,c]
```

What if we ask the following? What do we obtain?

```
?- member( X, [ a, b, c]).
```

We can generate through backtracking all the members of a given list

Membership

We may also reverse the question: Find lists that contain a given item, e.g., “apple”

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

L = [ apple | _A];               % Any list that has “apple” as the head
L = [ _A, apple | _B];           %First item is anything, second is “apple”
L = [ _A, _B, apple | _C];
```

Membership

Find lists that contain **a**, **b**, and **c**:

```
?- member( a, L), member(b, L), member( c, L).

L = [ a, b, c | _A];
L = [ a, b, _A, c | _B];
L = [ a, b, _A, _B, c | _C];
L = [ a, b, _A, _B, _C, c | _D];
```

List with any length

Membership

Permutations of **a**, **b**, and **c**.   % L is any list with exactly three elements

```
?- L = [ _, _, _], member( a, L), member( b, L), member( c, L).

L = [ a, b, c];
L = [ a, c, b];
L = [ b, a, c];
L = [ c, a, b];
L = [ b, c, a];
L = [ c, b, a];
false.
```