**Logic Programming 1**

(Facts, Relations, Rules, Queries)

**Case Study:** Family Relationship

**Defining Relations by Facts**

***Task 1:*** *Represent the facts in prolog for the following family tree.*

***Perform the following Query and wrte ther syntax-***

***T1: Fnd all X and Y such that X s a parent of Y.***

PROLOG CLAUSE:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***T2: Express grandparent relation base on the following two queries-***

1. Who is parent of Z? Assume that this is some Y.
2. Who is parent of Y? Assume that this is some X.

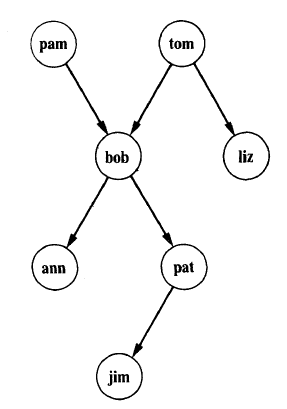
PROLOG CLAUSE:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***T3: Does Z has the same parent?***

PROLOG CLAUSE:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



**Task 2:** *Formulate in Prolog the following questions about the parent relation:*

(a) Who is Pat's parent? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(b) Does Liz have a child? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(c) Who is Pat's grandparent? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Defining Relations by Rules**

Add the following facts to our program:

female( pam).

male( tom).

male( bob).

female( liz).

female( pat).

female( ann).

male( jim).

**Task 3:** *Define* ***offspring*** *relations base on the following logical statement:*

For all X and Y,

Y is an offspring of X if

X is a parent of Y.

PROLOG CLAUSE: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Task 4:** *Define* ***mother*** *relation and write the logical statement:*

**Statement:** **PROLOG CLAUSE:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For all X and Y,

X is the mother of Y if,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Task 5:** *Define sister relation and illustrate with diagram:*

***Statement***

For any X and Y,

X is a sister of Y if,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Diagram***

**PROLOG CLAUSE:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Task 6:** *Translate the following statements in to prolog rules:*

(a) Everybody who has a child is happy (introduce a one-argument relation **happy**).

(b) For all X, if X has a child who has a sister then X has two children

*(Introduce new relation* ***hastwochildren****).*

**PROLOG CLAUSE:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Task 6:** *Define the relation* ***grandchild*** *using the* ***parent*** *relation.*

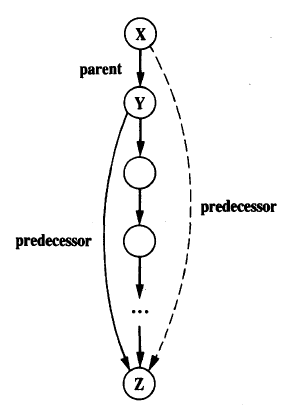
Hint: It will be similar to the grandparent relation.

**PROLOG CLAUSE:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Task 7:** Define the relation **aunt(X, Y)** in terms of the relations **parent** and **sister**.

**PROLOG CLAUSE:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**A recursive rule definition**



Add one more relation to our family program, the **predecessor** relation.

This relation will be defined in terms of the **parent** relation.

The whole definition can be expressed with two rules.

1. The first rule will define the direct (immediate) predecessors
2. The second rule the indirect predecessors.

We say that some **X** is an indirect predecessor of some **Z** if.

There is a parentship chain of people between **X** and **Z**.

The first rule is simple and can be formulated as:

For all X and Z,

X is a predecessor of Z if

X is a parent of Z.

**PROLOG SYNTAX: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

The key idea is to define the predecessor relation in terms of itself.

For all X and Z.

X is a predecessor of Z if there is a Y such that

(1) X is a parent of Y and

(2) Y is a predecessor of Z.

**PROLOG CLAUSE: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Put together all the pieces of our family program and Write it down below.**

**Goal Execut**i**on trace:** *Write the following prolog program and do the goal execution trace:*

PROGRAM

**big( bear).** % Clause 1

**big( elephant).** % Clause 2

**small( cat).** % Clause 3

**brown( bear).** % Clause 4

**black( cat).** % Clause 5

**gray( elephant).** % Clause 6

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

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

**QUESTION:? - dark( X), big( X).** % Who is dark and big?

**EXECUTION TRACE**

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

*Execution Steps*

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

Found: - Clause No: \_\_\_\_\_\_\_ Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) Replace the first goal by the instantiated body of clause no \_\_, giving a new goal list. **black( X), big( X)**

(3) Scan the program to find a match with **black(X)**.

Found: - Clause No: \_\_\_\_\_\_\_ Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This clause has no body, so the goal list, properly instantiated, shrinks to Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(4) Scan the program for the goal **big( cat).**

No clause found. Therefore backtrack to step (3) and undo the instantiation X: cat.

Now the goal list is again Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Continue scanning the program 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 Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(5) Scan the program to match **brown(X)**, finding **brown(bear)**.

This clause has no body, so the goal list shrinks to Clause: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(6) Scan the program and find clause **big( bear).**

It has no body so the goal list shrinks to empty.

This indicates successful termination, and the corresponding variable instantiation is:

X = \_\_\_\_\_\_\_\_\_\_\_\_\_

**Case Study: Monkey and Banana**

(Matching & Backtracking)

The monkey and banana problem is often used as a simple example of problem solving. Our Prolog program for this problem will show how the mechanism of matching and backtracking can be used in such exercises. We will develop the program in the non-procedural way, and then study its procedural behavior in detail. The program will be compact and illustrative.

We will use the following variation of the 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* (if it is already at the box) and *grasp the banana* if standing on the box directly under the banana.

**Can the monkey get the banana?**

One important task in programming is that of finding a representation of the problem in terms of concepts of the programming language used. In our case we can think of the **'monkey world'** as always being in some state that can change in time. The current state is determined by the positions of the objects.

For example, the initial state of the world is determined by:

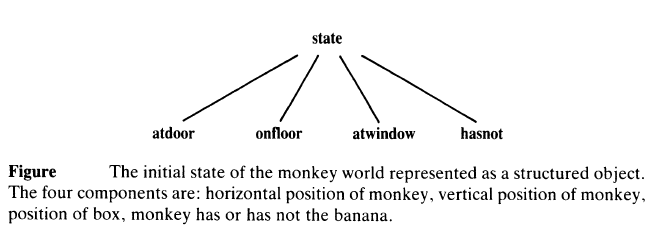
(1) Monkey is at door.

(2) Monkey is on floor.

(3) Box is at window.

(4) Monkey does not have banana.

It is convenient to combine all of these four pieces of information into one structured object. Let us choose the word 'state' as the functor to hold the four components together. Figure below shows the initial state represented as a structured object.



Our problem can be viewed as a one-person game. Let us now formalize the **rules** of the game.

**First**, 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)**

**Second**, what are the allowed moves that change the world from one state to another?

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.

For example, 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.

Such rules can be formalized in Prolog as a three-place relation named **move**: **move (State1, M, State2)**

The three arguments of the relation specify a move thus: State1 ----------🡪 State2

**M**

**Statel** is the state before the move. **M** is the move executed and **State2** is the state after the move.

The move **'grasp'**, with its necessary precondition on the state before the **move**, can be defined by the clause:

**move( state( middle, onbox, middle, hasnot),** % Before move

**grasp,** % Move

**state( middle, onbox, middle, has)).** % After move

This fact says that after the move the monkey has the banana and he has remained on the box in the middle of the room.

In a similar way we can express the fact that the money on the floor can walk from any horizontal position P1 to any position P2. The monkey can do this regardless of the position of the box and whether it has the banana or not.

All this can be defined by the following Prolog fact:

**move( state( P1, onfloor, B, H),**

**walk( P1, P2),**

**state( P2, onfloor, B, H) ).**

Note that this clause says many things, including, for example:

1. the move executed was 'walk from some position Pl to some position P2’;
2. the monkey is on the floor before and after the move;
3. the box is at some point B which remained the same after the move;
4. the 'has banana' status remains the same after the move.

The clause actually specifies a whole set of possible moves because it is applicable to any situation that matches the specified state before the move. Such a specification is therefore sometimes also called a ***move schema***. Due to the concept of Prolog variables such schemas can be easily programmed in Prolog.

The other two types of moves, **'push'** and **'climb'**, can be similarly specified.

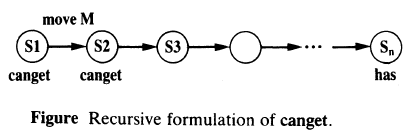
The main kind of question that our program will have to answer is: Can the monkey in some initial state **S** get the banana? This can be formulated as a predicate **canget( S)** where the argument S is a state of the monkey world.

The program for **canget** can be based on two observations:

1. 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. This corresponds to the Prolog fact:

**cange( state( -, -, -, has) ).**

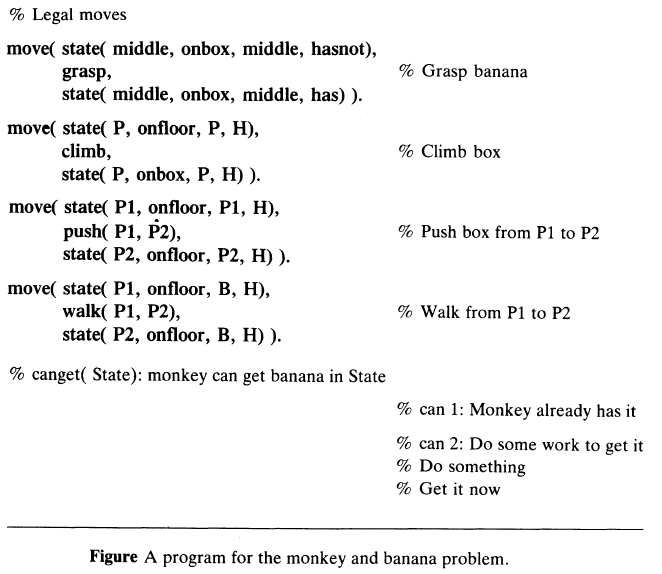
1. In other cases one or more moves are necessary. The monkey can get the banana in any state S1 if there is some move M from state S1 to some state S2, such that the monkey can then get the banana in state S2 (in zero or more moves).This principle is illustrated in the following Figure. A Prolog clause that corresponds to this rule is:



**canget( S1) :- move( s1, M, s2), canget( S2).**

The formulation of **canget** is recursive and is similar to that of the predecessor relation we have done in our Family Tree Case Study. This principle is used in Prolog again and again.

This completes our program which is shown in Figure below-

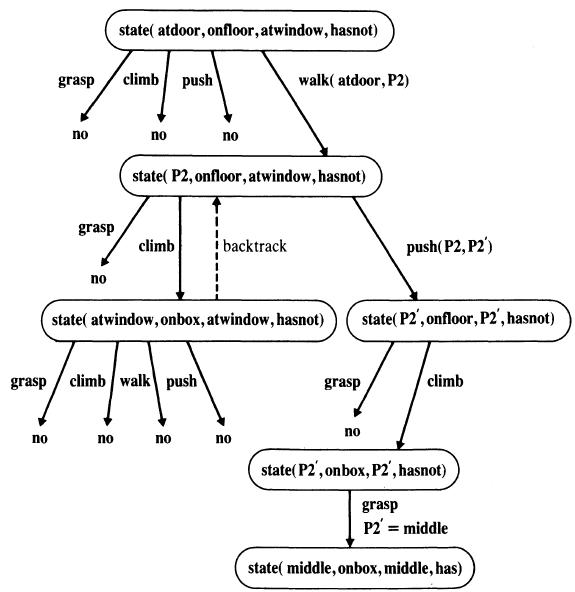


Perform the following query which should answer ‘yes/true’.

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

The process carried out by Prolog to reach this answer proceeds according to the procedural semantics of Prolog, through a sequence of goal lists. It involves some search for right moves among the possible alternative moves .At some point this search will take a wrong move leading to a dead branch. At this stage, backtracking will help it to recover.

Now consider the above question and following diagram (which illustrate backtracking), and discuss ***how prolog had to backtrack to answer the query?***



**Figure** The monkey's search for the banana. The search starts at the top node and proceeds downwards, as indicated. Alternative moves are tried in the left-to-right order.

**Discussion:**

**What is your assumption about how many times the backtracking happens?**