

# COMP219: Artificial Intelligence

## Lecture 6: Recursion in Prolog

1

## Overview

- Last time
  - Introduction to Prolog: facts, rules, queries; family tree program
- Today:
  - Recursive rules in Prolog; writing and evaluating them
  - Structures in Prolog
  - Declarative and procedural meanings for Prolog programs
- Learning outcome covered today:  
Understand and write Prolog code to solve simple knowledge-based problems.

2

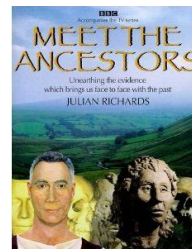
## Recap - Last Week

- Prolog programs comprised of facts and rules
- **Facts** describe things that are true without conditions, like data in a database
- **Rules** describe things that hold depending on certain conditions
- Prolog programs can be queried using **questions**
- Prolog **clauses** are facts, rules and questions
- Queries are answered by **instantiating variables**, creating new **sub-goals** from rules, and **matching** with facts

3

## The Ancestor Relation

- Consider the family tree from the previous Prolog lecture. We'd like to be able to define **ancestor**:
  - a parent, or
  - a parent of a parent, or
  - a parent of a parent of a parent, or
  - ...



4

# First Attempt

```
ancestor(X,Z):-
    parent(X,Z).
ancestor(X,Z):-
    parent(X,Y),
    parent(Y,Z).
ancestor(X,Z):-
    parent(X,Y1),
    parent(Y1,Y2),
    parent(Y2,Z).
```

- Problems
  - Lengthy (not so important)
  - Using only the first two rules (or more) we get a **finite depth** on our search for ancestors

# Recursion

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

Annotations:

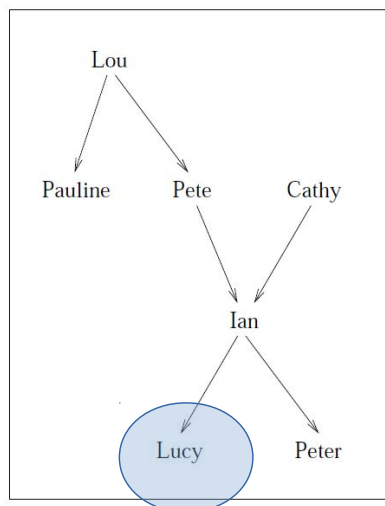
- Base case (points to the first clause)
- Recursive Call (points to the recursive call in the second clause)
- Predicate in head and body (points to the recursive call in the second clause)

- This type of definition is called a **recursive definition**.
- Recursion is very important in Prolog. A set of clauses referring to the same relation is known as a **procedure**. This is a **recursive procedure**.

5

6

# Example: A Family Tree



# Questioning Recursive Definitions

- The question *Who are the ancestors of Lucy?* is posed as follows
  - ?- ancestor(X,lucy).
  - X=ian ;
  - X=pete ;
  - X=lou ;
  - X=cathy ;
  - false.

Semi-colon for 'next answer', full stop for 'enough'.

7

8

## How this Works (I)

```

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

- parent(pete,ian).
- parent(ian,pete).
- parent(ian,lucy).
- parent(cathy,ian).
- parent(lou,peter)
- parent(lou,pauline)

```

- ancestor(X,lucy)?
- {try first clause}
- parent(X, lucy)?
- match: X=ian
- success
- X=ian
- ;
- {no other matches for parent(X, lucy) → try second clause}
- parent(X,Y)?
- match: X=pete, Y=ian
- ancestor(ian,lucy)?
- {yes, via first clause!} succes
- X=pete
- ;
- {retry second clause}

9



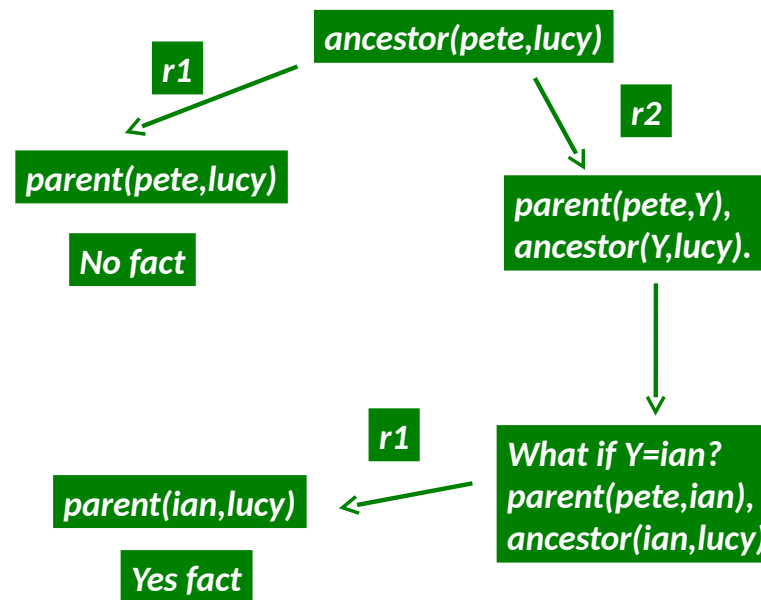
## Recursion and Search



- Recursion is a powerful construct essential to Prolog
- Prolog **searches** for an answer through several possibilities using **depth first** search
- This search method can be applied to search problems
- The lexical scope of a variable is the clause; the 'same' variable in different clauses is different (Bratko book section 2.1.2)
- But first we have to learn about structures

11

## How this Works (II)



10

## Structures



- **Structures** are Prolog's way to represent structured data
- They are objects that have several components and a name (**functor**) that associates them together
  - date(5, february, 2002).
  - location(depot1, manchester).
  - id\_no(rajeev, gore, 02571).
  - state(onTable,onBlock).

12

# Example: Monkey and Banana

- “A monkey is 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 the banana, but cannot stretch high enough from the floor. At the window there is a box the monkey can climb on to get the banana.”
- The monkey can: walk, climb, push the box (if at box), grasp the banana (if standing on box under banana)
- Can the monkey get the banana?

Example from section 2.5 of Bratko book.



## States

- Goal state: `state(_,_,_,banana)`
- The underscore `_` stands for an *anonymous variable* – could be any value, we do not care what it is
- The program has to search the state space for a solution given the available moves

13

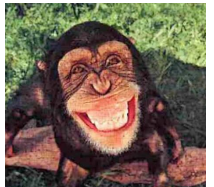


15

# States

- States are represented by the structure `state(X,Y,Z,U)`
  - X: Horizontal position of monkey
    - door, middle, window
  - Y: Vertical position of monkey
    - onFloor, onBox
  - Z: Horizontal position of box
    - door, middle, window
  - U: Monkey has banana or doesn't have banana
    - banana, noBanana

14



## Moves

- Monkey can: grasp, climb, push, walk
- Moves change states, for example:

```
move (  
    state(middle, onBox, middle, noBanana),  
    grasp,  
    state(middle, onBox, middle, banana)).
```

- Where the state is that monkey is in the middle, on the box, the box is in the middle, and the monkey doesn't have the banana, the monkey grasps, then the state is that the monkey is in the middle, on the box, the box is in the middle, and the monkey has the banana – Joy!

16

# Problem Analysed with Recursive Rules

- Write rules to move the monkey and box around, have the monkey doing actions, till the monkey and box are in position, and the right action is executed. No strategy (except), just search
- Base case: monkey has the banana

```
canGet (state (_, _, _, banana)) .
```

- Recursive rule: move until monkey gets the banana

```
canGet (State1) :-
```

```
    move (State1, Move, State2),
    canGet (State2) .
```

# The Program

```
% move( State1, Move, State2): making Move in State1 results in State2;
% a state is represented by a structure:
% state( MonkeyHorizontal, MonkeyVertical, BoxPosition, HasBanana)

move( state( middle, onBox, middle, noBanana), % Before move
     grasp, % Grasp banana
     state( middle, onBox, middle, banana) ). % After move

move( state( P, onFloor, P, H), % Climb box
     climb,
     state( P, onBox, P, H) ).

move( state( P1, onFloor, P1, H), % Push box from P1 to P2
     push( P1, P2),
     state( P2, onFloor, P2, H) ).

move( state( P1, onFloor, B, H),| % Walk from P1 to P2
     walk( P1, P2),
     state( P2, onFloor, B, H) ).

canGet( state( _, _, _, banana) ). % canGet 1: Monkey already has it
canGet( State1) :- % canGet 2: Do some work to get it
    move( State1, Move, State2), % Do something
    canGet( State2), % Get it now
```

Tries moves in order:  
grasp, climb, push, walk

Note variables  
and predicates

```
?- trace, canGet(state(door,onFloor>window,noBanana)).
Call: (9) canGet(state(door, onFloor, window, noBanana)) ? creep
Call: (10) move(state(door, onFloor, window, noBanana), _L247, _L230) ? creep
Exit: (10) move(state(door, onFloor, window, noBanana), walk(door, _G493), state(_G493, onFloor, window, noBanana)) ? creep
Call: (10) canGet(state(_G493, onFloor, window, noBanana)) ? creep
Call: (11) move(state(_G493, onFloor, window, noBanana), _L264, _L247) ? creep
Exit: (11) move(state(window, onFloor, window, noBanana), climb, state(window, onBox, window, noBanana)) ? creep
Call: (11) canGet(state(window, onBox, window, noBanana)) ? creep
Call: (12) move(state(window, onBox, window, noBanana), _L301, _L284) ? creep
Fail: (12) move(state(window, onBox, window, noBanana), _L301, _L284) ? creep
Fail: (11) canGet(state(window, onBox, window, noBanana)) ? creep
Redo: (11) move(state(_G493, onFloor, window, noBanana), _L264, _L247) ? creep
Exit: (11) move(state(window, onFloor, window, noBanana), push(window, _G501), state(_G501, onFloor, _G501, noBanana)) ? creep
Call: (11) canGet(state(_G501, onFloor, _G501, noBanana)) ? creep
Call: (12) move(state(_G501, onFloor, _G501, noBanana), _L302, _L285) ? creep
Exit: (12) move(state(_G501, onFloor, _G501, noBanana), climb, state(_G501, onBox, _G501, noBanana)) ? creep
Call: (12) canGet(state(_G501, onBox, _G501, noBanana)) ? creep
Call: (13) move(state(_G501, onBox, _G501, noBanana), _L339, _L322) ? creep
Exit: (13) move(state(middle, onBox, middle, noBanana), grasp, state(middle, onBox, middle, banana)) ? creep
Call: (13) canGet(state(middle, onBox, middle, banana)) ? creep
Exit: (13) canGet(state(middle, onBox, middle, banana)) ? creep
Exit: (12) canGet(state(middle, onBox, middle, noBanana)) ? creep
Exit: (11) canGet(state(middle, onFloor, middle, noBanana)) ? creep
Exit: (10) canGet(state(window, onFloor, window, noBanana)) ? creep
Exit: (9) canGet(state(door, onFloor, window, noBanana)) ? creep
true
```

## Trace

```
?- trace, canGet(state(door,onFloor>window,noBanana)).
Call: (9) canGet(state(door, onFloor, window, noBanana)) ? creep
Call: (10) move(state(door, onFloor, window, noBanana), _L247, _L230) ? creep
Exit: (10) move(state(door, onFloor, window, noBanana), walk(door, _G493), state(_G493, onFloor, window, noBanana)) ? creep
Call: (10) canGet(state(_G493, onFloor, window, noBanana)) ? creep
Call: (11) move(state(_G493, onFloor, window, noBanana), _L264, _L247) ? creep
Exit: (11) move(state(window, onFloor, window, noBanana), climb, state(window, onBox, window, noBanana)) ? creep
Call: (11) canGet(state(window, onBox, window, noBanana)) ? creep
Call: (12) move(state(window, onBox, window, noBanana), _L301, _L284) ? creep
Fail: (12) move(state(window, onBox, window, noBanana), _L301, _L284) ? creep
Fail: (11) canGet(state(window, onBox, window, noBanana)) ? creep
Redo: (11) move(state(_G493, onFloor, window, noBanana), _L264, _L247) ? creep
Exit: (11) move(state(window, onFloor, window, noBanana), push(window, _G501), state(_G501, onFloor, _G501, noBanana)) ? creep
Call: (11) canGet(state(_G501, onFloor, _G501, noBanana)) ? creep
Call: (12) move(state(_G501, onFloor, _G501, noBanana), _L302, _L285) ? creep
Exit: (12) move(state(_G501, onFloor, _G501, noBanana), climb, state(_G501, onBox, _G501, noBanana)) ? creep
Call: (12) canGet(state(_G501, onBox, _G501, noBanana)) ? creep
Call: (13) move(state(_G501, onBox, _G501, noBanana), _L339, _L322) ? creep
Exit: (13) move(state(middle, onBox, middle, noBanana), grasp, state(middle, onBox, middle, banana)) ? creep
Call: (13) canGet(state(middle, onBox, middle, banana)) ? creep
Exit: (13) canGet(state(middle, onBox, middle, banana)) ? creep
Exit: (12) canGet(state(middle, onBox, middle, noBanana)) ? creep
Exit: (11) canGet(state(middle, onFloor, middle, noBanana)) ? creep
Exit: (10) canGet(state(window, onFloor, window, noBanana)) ? creep
Exit: (9) canGet(state(door, onFloor, window, noBanana)) ? creep
true
```

Can't grasp, climb  
or push, so walks

Tries climbing

Not under banana

Pushes instead

Can't grasp so  
climbs

Can grasp now

Unwind the  
recursion

# Comments

- Prolog backtracked only once
- Good ordering of clauses in the move procedure
- With a different order may never terminate!
  - Example: if push is the first line in the move procedure, the monkey will go to the box, and then push it around aimlessly
  - The monkey always needs to try to grasp (first action in order), and see whether climbing helps (second action in order)
  - There is some strategy built into the rule order

21

# Infinite Loops

- Avoid infinite recursive loops!
- Suppose a rule which means that people are silly if they are silly:

```
silly(X) :- silly(X)
```

- Querying this program

```
?- silly(katie).
```

matched the head of the above rule, X is instantiated by [katie](#), and the new sub-goal is [silly\(katie\)](#). This is matched a second time to the head of the above clause, and a new sub-goal is generated, and so on. But we never get an answer because there are no base facts.

22

# Infinite Loops with predecessor

- If we change the definition of 'ancestor' to 'predecessor' and put the clauses in the following order, we have problems

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

We never hit the base case!

We call predecessor(.,.) forever



23

# Avoiding Infinite Looping

- A general rule to avoid such problems is to ensure that the [base case is the first clause](#) - try the simplest idea first
- Must hit a fact to terminate. Look at our nice ancestor example...

24

# Nice Order

```

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

```

?- trace, ancestor(pete,lucy).
Call: (9) ancestor(pete, lucy) ? creep
Call: (10) parent(pete, lucy) ? creep
Fail: (10) parent(pete, lucy) ? creep
Redo: (9) ancestor(pete, lucy) ? creep
Call: (10) parent(pete, _L233) ? creep
Exit: (10) parent(pete, ian) ? creep
Call: (10) ancestor(ian, lucy) ? creep
Call: (11) parent(ian, lucy) ? creep
Hits base case first
Exit: (11) parent(ian, lucy) ? creep
Exit: (10) ancestor(ian, lucy) ? creep
Exit: (9) ancestor(pete, lucy) ? creep
true .
    
```

25

# Avoiding Infinite Looping

- Must hit a fact to terminate
 

```

predecessor2(X, Z) :-
    parent(X, Y) ,
    predecessor2(Y, Z) .
predecessor2(X, Z) :-
    parent(X, Z) .
            
```
- Does this cause infinite looping or *other badness*?

26

Works, but very inefficient.  
 Explores whether Peter is Lucy's predecessor

Would be worse if Peter had children

And whether Lucy is her own predecessor

Would be worse if Lucy had children

Before it reaches the base case.

```

?- predecessor2(pete,lucy).
Call: (9) predecessor2(pete, lucy) ? creep
Call: (10) parent(pete, _L196) ? creep
Exit: (10) parent(pete, ian) ? creep
Call: (10) predecessor2(ian, lucy) ? creep
Call: (11) parent(ian, _L214) ? creep
Exit: (11) parent(ian, peter) ? creep
Call: (11) predecessor2(peter, lucy) ? creep
Call: (12) parent(peter, _L249) ? creep
Fail: (12) parent(peter, _L249) ? creep
Redo: (11) predecessor2(peter, lucy) ? creep
Call: (12) parent(peter, lucy) ? creep
Fail: (12) parent(peter, lucy) ? creep
Fail: (11) predecessor2(peter, lucy) ? creep
Redo: (11) parent(ian, _L214) ? creep
Exit: (11) parent(ian, lucy) ? creep
Call: (11) predecessor2(lucy, lucy) ? creep
Call: (12) parent(lucy, _L232) ? creep
Fail: (12) parent(lucy, _L232) ? creep
Redo: (11) predecessor2(lucy, lucy) ? creep
Call: (12) parent(lucy, lucy) ? creep
Fail: (12) parent(lucy, lucy) ? creep
Fail: (11) predecessor2(lucy, lucy) ? creep
Redo: (10) predecessor2(ian, lucy) ? creep
Call: (11) parent(ian, lucy) ? creep
Exit: (11) parent(ian, lucy) ? creep
Exit: (10) predecessor2(ian, lucy) ? creep
Exit: (9) predecessor2(pete, lucy) ? creep
true .
    
```

27

# Declarative and Procedural Meaning

- Two interpretations of the meaning of Prolog programs
  - declarative (logical)
  - procedural
- The declarative meaning determines **what will count** as an answer. It is concerned **only** with the relations that have been defined in the program.
- The procedural meaning also involves **how** this output is obtained. This means that the **order** of clauses is **significant**.

28

## Declarative and Procedural Meaning

- Consider the query `parent(X,ian)`.
- The declarative meaning tells us that **both** cathy and pete can be successfully instantiated as X.
- Declaratively, `(cathy,ian)` and `(pete,ian)` are the same
- The procedural meaning tells us that if the fact `parent(pete,ian)` occurs in the program **before** `parent(cathy,ian)`, then the first answer returned is based on `parent(pete,ian)`.
- Answer will be `X=pete` (but we can also obtain the second answer, `X=cathy`, by typing a semi-colon)

29

## Declarative and Procedural Meaning

- The `ancestor` and `predecessor2` programs are **declaratively the same** – they return the same answers. But, `predecessor2` is **procedurally inefficient**
- We have shown that `predecessor` does not return an answer, i.e. it is **not procedurally correct**

31

## Declarative and Procedural Meaning

- Remember Prolog uses **depth first search** to order its goals
- If the wrong node is chosen, the path may be **infinite** (so the program does not terminate)
- Or the path may be **very long** and ultimately (perhaps) **unsuccessful**
- So it is important that we order the clauses so the best path will be tried first. This is a matter of good programming style.

30

## Summary

- Recursion is a powerful construct essential to Prolog
- Take care with recursive rules to avoid an infinite sequence of recursive calls
- The **order** of clauses and goals **does matter**
- Programs that are declaratively correct may not be procedurally correct (and so will not work in practice)
- **Next time**
  - Improving on blind search

32



# Reminder

- No Lecture on Monday!
  
- There will be no lectures on:
  - 16-10-2017
  - 27-10-2017
  - 30-10-2017