INST0072 Logic and Knowledge Representation

Lecture 6

Prolog: Recursion, Lists, and a Simple Planning Problem

In the Last Lecture

In the last lecture we saw that

- Prolog variables are not like JavaScript variables. They are not "pockets of computer memory", but instead are *logical variables*.
- Each Prolog clause is a statement in its own right, so that variables with the same name in different clauses do not refer to the same thing. But within a clause, variables with the same name do refer to the same thing. For example, the program

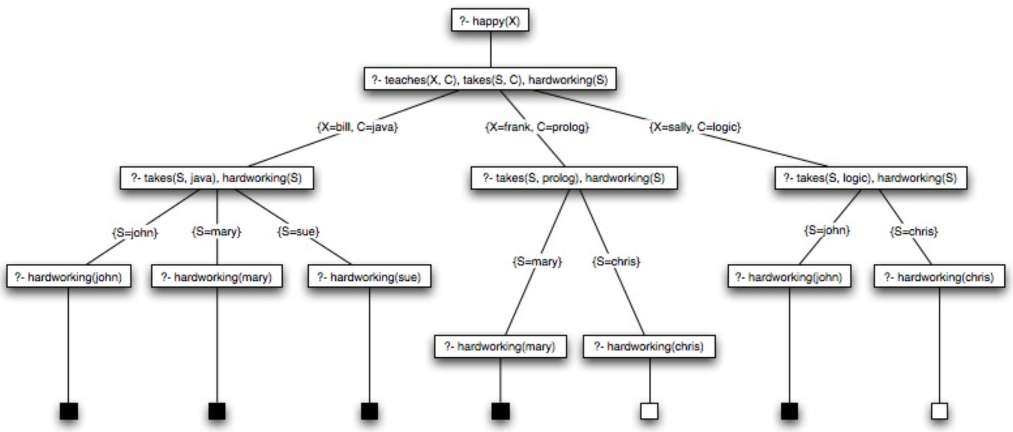
  ```
  grandfather(X, Z) :- father(X, Y), father(Y, Z).
  grandfather(X, Z) :- father(X, Y), mother(Y, Z).
  ```

  is exactly the same as the program

  ```
  grandfather(X, Z) :- father(X, Y), father(Y, Z).
  grandfather(A, C) :- father(A, B), mother(B, C).
  ```

- Prolog answers a query by mechanically and systematically exploring all the ways the query might be proved from the facts and rules (i.e. clauses) in the program. This can be visualised in a *search tree*.
- While searching through this tree, Prolog repeatedly *unifies* its current goal with the head of some clause in the program, and then substitutes that goal with the sub-goals in the body of the clause, before continuing its search.
- This PDF document illustrates the process for the program happyTeacher.pl.

The Search Tree for Happy Teacher



[happyTeacherSearchTree.pdf]

Recursion

- *Recursive definitions* are definitions that define a predicate in terms of itself (and maybe some other predicates too).
- For example, without recursion we would need an infinite set of clauses to define '`ancestor(X,Y)`':

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

- With recursion we need just two clauses:

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

- The first clause says that `X` is an ancestor of `Y` if `X` is a parent of `Y`. This clause itself is not recursive and is called the *base case*.
- The second clause says `X` is an ancestor of `Y` if there is a `Z` such that `X` is a parent of `Z` and `Z` is an ancestor of `Y`. It is recursive, as it defines 'ancestor' in terms of itself (and 'parent').
- For reasons of computational efficiency, it is often better to put the recursive sub-goal in a recursive clause last. Clauses written this way are known as *tail-recursive* clauses. But this is not always the case, and it is a good idea to experiment with re-ordering sub-goals in a clause if a program is not working or gets stuck in a loop.

A Program With Recursion

- The program parentsAndAncestors.pl contains the same 'father/2' and 'mother/2' definitions as the program mothersAndFathers.pl from lecture 1, plus the following definitions for 'parent/2' and 'ancestor/2':

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

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

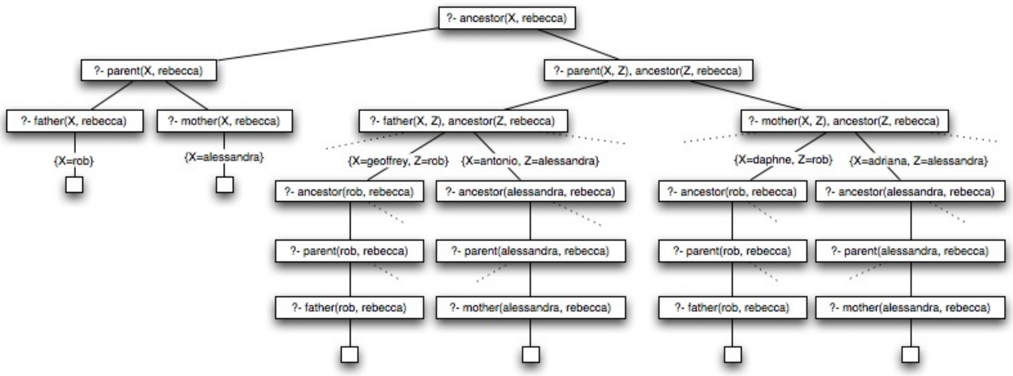
- It gives six solutions for the query '`?- ancestor(X, rebecca)`':

```
?- ancestor(X, rebecca).
X = rob ;
X = alessandra ;
X = geoffrey ;
X = antonio ;
X = daphne ;
X = adriana ;
false.
```

[parentsAndAncestors.pl]

A Partial Search Tree for an 'Ancestor' Query

The diagram below shows the six successful branches of the search tree for the query '`?- ancestor(X, rebecca)`'. The dotted lines indicate where there are other failed branches.


?- ancestor(X, rebecca)
?- parent(X, rebecca)
?- parent(X, Z), ancestor(Z, rebecca)
?- father(X, rebecca)
?- mother(X, rebecca)
?- father(X, Z), ancestor(Z, rebecca)
?- mother(X, Z), ancestor(Z, rebecca)
{X=rob}
{X=alessandra}
{X=geoffrey, Z=rob}
{X=antonio, Z=alessandra}
{X=daphne, Z=rob}
{X=adriana, Z=alessandra}
?- ancestor(rob, rebecca)
?- ancestor(alessandra, rebecca)
?- ancestor(rob, rebecca)
?- ancestor(alessandra, rebecca)
?- parent(rob, rebecca)
?- parent(alessandra, rebecca)
?- parent(rob, rebecca)
?- parent(alessandra, rebecca)
?- father(rob, rebecca)
?- mother(alessandra, rebecca)
?- father(rob, rebecca)
?- mother(alessandra, rebecca)

[parentsAndAncestorsSearchTree.pdf | parentsAndAncestors.pl]

Lists

- Lists are structures that provide a useful way of representing data. Lists are written inside two square brackets. For example

  ```
  [1, 3, 5, 7, 9]
  [john, X, [sue, eric], sally]
  ```

  Are both lists. The first list has five elements, the second list has four elements (the third of which is itself a list).

- Although lists can have as many elements as we like, they are actually *binary* data structures. That is, they are made up of two components; an element called the *head*, and another list called the *tail*. These two components can be symbolised with the notation

  ```
  [H | T]
  ```

  where `H` is the element which forms the head of the list, and `T` is list which is the tail.

Lists as Recursive, Binary Data Structures

- As one of the two components of a list (i.e. the tail) is itself a list, we see that lists are recursive data structures. In fact we can give a definition in Prolog of a predicate "list" which is true if its single argument is a list:

  ```
  list([]).
  list([Head|Tail]) :-
      list(Tail).
  ```

  As lists are recursive data structures, we will very often need to write recursive programs to operate on them.

- Internally, Prolog computes with lists as binary data structures, so that for example it understands

  ```
  [1, 2, 3, 4]
  ```

  to mean

  ```
  [1|[2|[3|[4|[]]]]]
  ```

Unification of Lists and the '=' Predicate

- We can unify list structures just as we can with other terms. For example, the two lists

  ```
  [1|Rest]
  ```

  and

  ```
  [1, 3, 5, 7, 9]
  ```

  unify via the substitution

  ```
  {Rest = [3, 5, 7, 9]}
  ```

- In SWI Prolog, the symbol '=' can be used as a predicate symbol that means 'unifies with'. So we can demonstrate the example above like this:

  ```
  ?- =([1|Rest], [1, 3, 5, 7, 9]).
  Rest = [3, 5, 7, 9].
  ```

  or, because '=' is also an *infix operator*, like this:

  ```
  ?- [1|Rest] = [1, 3, 5, 7, 9].
  Rest = [3, 5, 7, 9].
  ```

Example Programs for List Manipulation

- The predicate 'element/2' is true if the second argument is a list and the first argument is an element of that list. Its definition is

  ```
  element(X, [X|Y]).
  element(X, [Y|Z]) :-
      element(X,Z).
  ```

  In SWI Prolog (and many other versions of Prolog) the inbuilt predicate 'member/2' is equivalent to 'element/2'.

- The predicate 'join/3' is true if the third argument is a list consisting of the lists in the first two arguments joined together. Its definition is

  ```
  join([], X, X).
  join([H|T], X, [H|Y]) :-
      join(T, X, Y).
  ```

  In SWI Prolog (and most other versions of Prolog) the inbuilt predicate 'append/3' is equivalent to 'join/3'.

- Example queries:

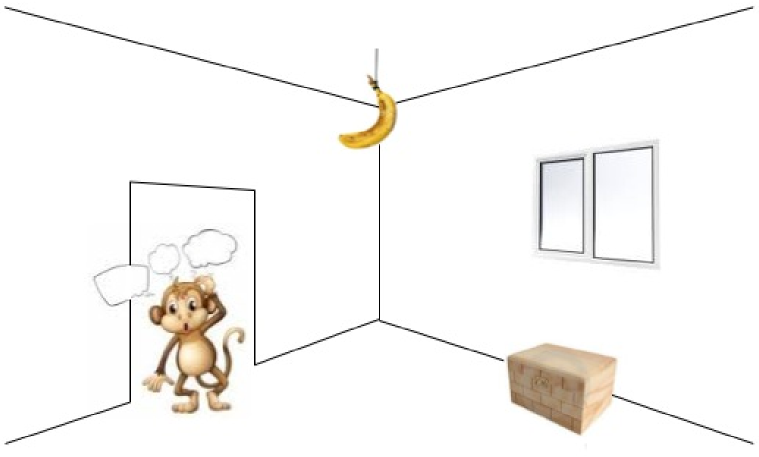
  ```
  ?- element(X, [a, n, y]).
  X = a ;
  X = n ;
  X = y ;
  false.

  ?- join([a, n, y], [b, o, d, y], X).
  X = [a, n, y, b, o, d, y].
  ```

[elementAndJoin.pl]

An Example Planning Problem: The Monkey and Banana Problem

Prolog, with its search strategy and unification algorithm, can be used very effectively to tackle classic A.I. problems to do with generating 'plans' to get from an initial state of affairs to a 'goal' state of affairs. Such a plan might be constructed as a Prolog list of 'actions' as illustrated in the example below (From Bratko, edition 1, pages 49 - 55).

There is a monkey at the door of a room. In the middle of the room a banana is hanging from the ceiling. The monkey 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. What sequence of actions does the monkey have to perform in order to get the banana?



Representing the Monkey's Environment

- To tackle the monkey and banana problem, the first thing we need is a way of representing a particular state of affairs (or 'state of the world'). In this problem, there are four important parameters, and so we use a structured object with four arguments, labelled with the functor '`state`'.

- The arguments of "state" are as follows:
  - 1st argument: horizontal position of monkey
    possible values: '`atDoor`', '`atWindow`' or '`middle`'
  - 2nd argument: vertical position of monkey
    possible values: '`onFloor`' or '`onBox`'
  - 3rd argument: position of box
    possible values: '`atDoor`', '`atWindow`' or '`middle`'
  - 4th argument: whether the monkey has got the banana or not
    possible values: '`has`' or '`hasNot`'

- For example, the term

  ```
  state(atDoor, onFloor, atWindow, hasNot)
  ```

  represents a state of affairs in which the monkey is at the door on the floor, the box is at the window, and the monkey doesn't have the banana.

- Like all functors, the functor '`state`' can just be used in the definition of a predicate - it doesn't have to be defined anywhere separately.

Representing Actions the Monkey Can Do

- To describe all of the possible things the monkey can do, we define the predicate '`can_do`' with four Prolog facts. In each case the first argument of this predicate represents the state before the action, the second argument is the name of the action, and the third argument is the state after the action:

```
can_do(state(middle, onBox, middle, hasNot),
                grab,
                state(middle, onBox, middle, has)).

can_do(state(Position, onFloor, Position, X),
                climbOn,
                state(Position, onBox, Position, X)).

can_do(state(Position_before, onFloor, Position_before, X),
                push(Position_before, Position_after),
                state(Position_after, onFloor, Position_after, X)).

can_do(state(Position_before, onFloor, X, Y),
                walk(Position_before, Position_after),
                state(Position_after, onFloor, X, Y)).
```

[monkeyAndBanana.pl]

Defining Plans

- We can describe when (and how) an agent (such as the monkey) can get from a starting state to a goal state with a recursively defined predicate '`is_plan`'. This predicate says that it is possible to get from the state described by the first argument to the state described by the second argument by following the plan (sequence of actions) in the third argument. The base case of this predicate corresponds to the fact that it's not necessary to do anything if already in the goal state:

```
is_plan(Goal_state, Goal_state, []).

is_plan(State, Goal_state, [Action|Plan_for_state_after]):-
    can_do(State, Action, State_after),
    is_plan(State_after, Goal_state, Plan_for_state_after).
```

- If we run this program in SWI Prolog with the query

```
?- is_plan(state(atDoor, onFloor, atWindow, hasNot), state(_, _, _, has), Plan).
```

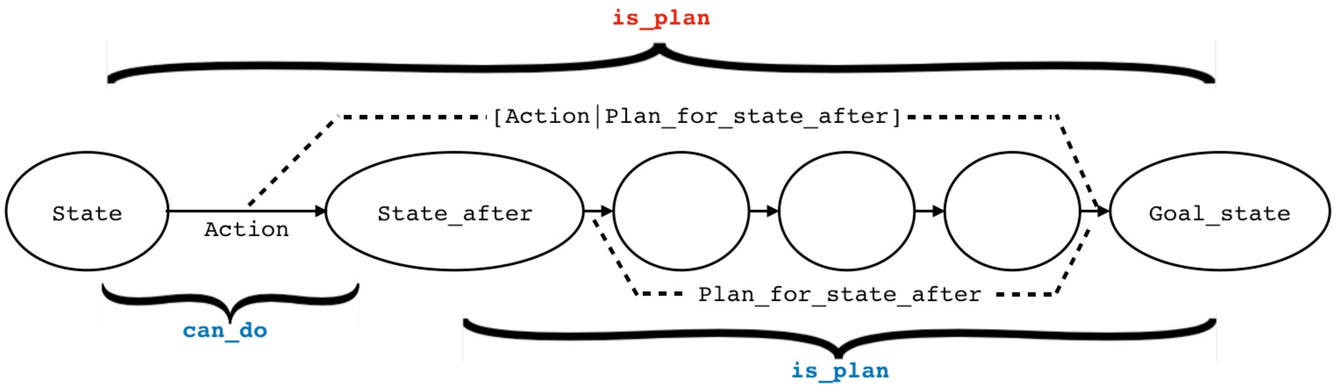
we get output similar to the following:

```
Plan = [walk(atDoor, atWindow), push(atWindow, middle), climbOn, grab] ;
Plan = [walk(atDoor, atWindow), push(atWindow, _4054), push(_4054, middle), climbOn, grab] ;
.....
```

[monkeyAndBanana.pl]

'is_plan' Recursive Clause Diagram

```
is_plan(State, Goal_state, [Action|Plan_for_state_after]):-
```

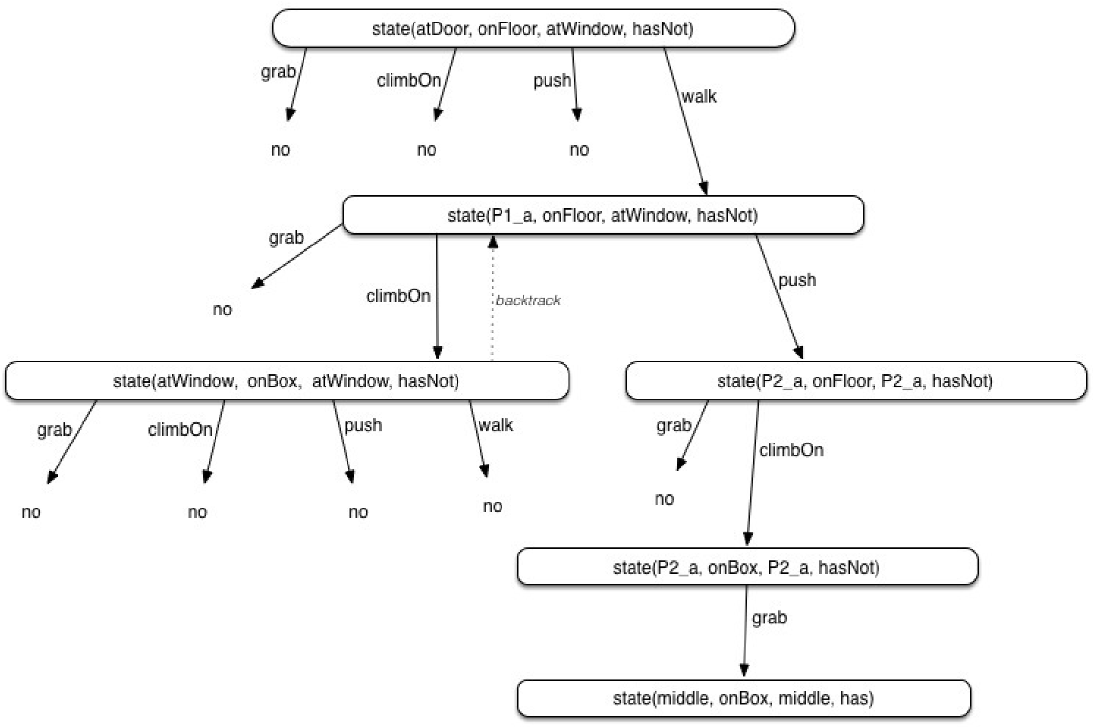


```
        can_do(State, Action, State_after),
        is_plan(State_after, Goal_state, Plan_for_state_after).
```

[is_plan.pdf]

Monkey and Banana Search Tree Diagram

Planning problems such as the Monkey and Banana Problem can often be visualised with search tree diagrams, similar to Prolog's own search trees. The following diagram is part of such a search tree and illustrates how the monkey generates its first plan to get the banana.


state(atDoor, onFloor, atWindow, hasNot)
grab
climbOn
push
walk
no
no
no
state(P1_a, onFloor, atWindow, hasNot)
grab
no
climbOn
backtrack
push
state(atWindow, onBox, atWindow, hasNot)
grab
climbOn
push
walk
no
no
no
no
state(P2_a, onFloor, P2_a, hasNot)
grab
climbOn
no
state(P2_a, onBox, P2_a, hasNot)
grab
state(middle, onBox, middle, has)

[MonkeyAndBananaSearchTree.pdf]