INST0072 Logic and Knowledge Representation

Lecture 7

Prolog: Negation-as-failure, Cut and Fail, and Command Line Input/Output

---

In the Last Lecture

In the last lecture we saw that

- *Recursive definitions* are definitions that define a predicate in terms of itself (and maybe some other predicates too). For example, the following is a recursive definition of 'ancestor/2':

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

- The first clause above is *base case* and says that `X` is an ancestor of `Y` if `X` is a parent of `Y`. 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/2' in terms of itself (and 'parent/2').

- Lists in Prolog 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).

- Lists are actually *binary* data structures, 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

```
[Head | Tail]
```

- Both recursion and lists are useful in writing *planning problems* in Prolog, as illustrated with the 'Monkey and Banana' problem.

An Example from the Last Lecture

- The 'Monkey and Banana' planning problem:

```
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)).

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).
```

- We can run this program in SWI Prolog with the query

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

[monkeyAndBanana.pl]

Negation As Failure

- In Prolog, we can prefix a sub-goal in the body of a clause by a 'not' to represent the negation of that sub-goal. 'Not' in this case is short for 'not provable', and the symbol for this in SWI Prolog is '`\+`'.
- For example, the following could be included in our program about family relationships, as a definition of 'cousin':

```
cousin(X,Y) :-
    grandparent(Z,X),
    grandparent(Z,Y),
    \+ brother(X,Y),
    \+ sister(X,Y),
    \+ X == Y.
```

- Prolog uses a rule called *negation as failure* to evaluate a negative sub-goal. If it encounters the query

```
?- \+ p(...)
```

during some execution, it attempts to evaluate the query

```
?- p(...)
```

If '`?- p(...)`' succeeds then '`?- \+ p(...)`' fails, but if '`?-p(...)`' fails then '`?- \+ p(...)`' succeeds.

[cousins.pl]

An Example Program With Negation As Failure

- Suppose we have the following program:

```
happy(X) :-
    rich(X),
    \+ stressed(X).

stressed(X) :-
    overworked(X).
stressed(X) :-
    tired(X).

rich(sally).
rich(jim).
rich(ann).

overworked(sally).

tired(jim).
```

- If we query this program with

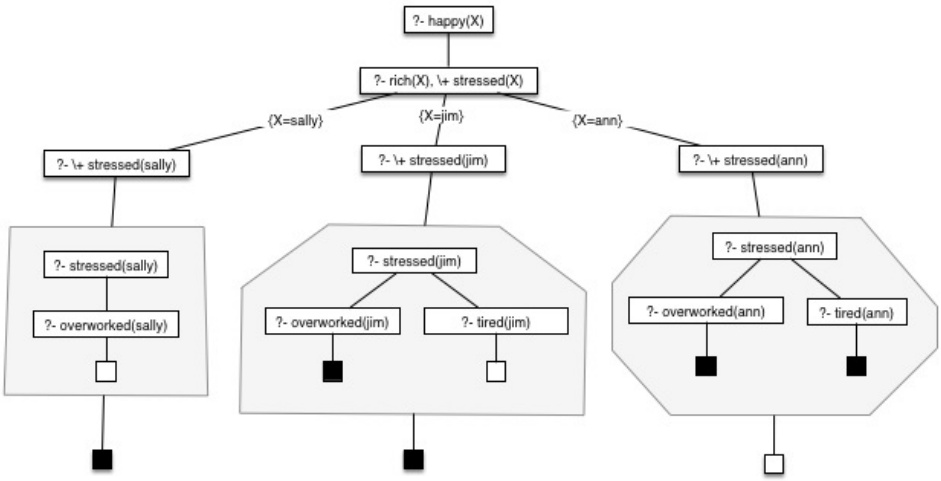
```
?- happy(X)
```

Prolog will answer with the single solution

```
X = ann.
```

[happyNotStressed.pl]

A Search Tree For '`?- happy(X)`'



[happyNotStressedSearchTree.pdf]

Issues Concerning Negation As Failure

- The meaning of negation-as-failure is not always obvious when it is used with sub-goals or queries containing variables. For example, consider the definition of the predicate 'element/2' (predefined in SWI Prolog as 'member/2') from the last lecture:

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

  The query

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

  is taken to mean '**does there exist an x** such that `X` is an element of the list `[a, b, c]`?' (Prolog answers '`X=a`, `X=b` or `X=c`').

-
```
?- element(d, [a, b, c])
```

  is taken to mean 'is `d` an element of the list `[a, b, c]`?' (Prolog answers '`false`').

-
```
?- \+ element(d, [a, b, c])
```

  is taken to mean 'is it not the case that `d` is an element of the list `[a, b, c]`?' (Prolog answers '`true`').

-
```
?- \+ element(X, [a, b, c])
```

  is taken to mean '**for all x** is it not the case that `X` is an element of the list `[a, b, c]`?' (Prolog answers '`false`').

Negation As Failure and Variable Quantification

- As we saw in the last slide

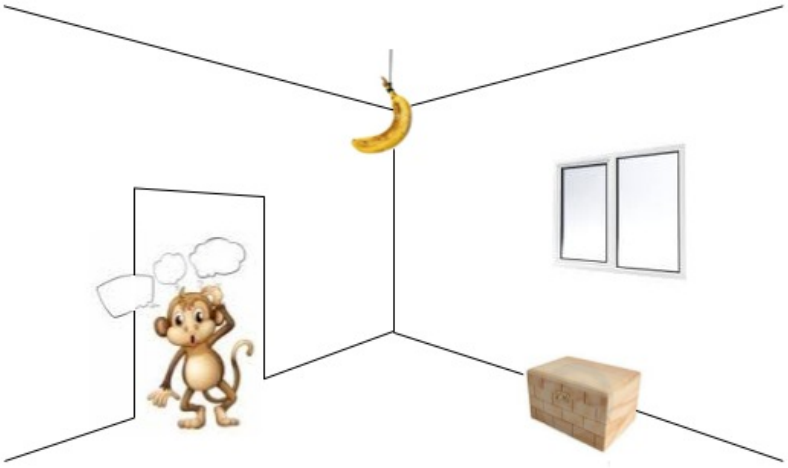
```
?- \+ element(X, [a, b, c])
```

  is taken to mean '**for all x** is it not the case that `X` is an element of the list `[a, b, c]`?'

- In logic terminology, variables with a 'for all' statement associated with them are called *universally quantified* variables.
- Variables with a 'there exists' statement associated with them are called *existentially quantified* variables.
- In general, *unground* variables in negated Prolog queries (i.e. variables that have not been assigned to a specific constant) are universally quantified rather than existentially quantified.
- Logicians are sometimes unhappy with negation as failure in Prolog and logic programming, because it does not mean the same thing as *classical negation* in standard logic.

Using Negation As Failure For Loop Checking

- In planning problems such as the Monkey and Banana problem (lecture 3), we can end up generating overly long plans which include unnecessary actions. Some of these actions may result in returning to a previous state of affairs. For example, the monkey could walk from the door to the window, but then walk back to the door.



- In order to disallow plans which result in the same state of affairs occuring more than once, we can include a *loop check* in the description of a problem. We do this by generating a list of states resulting from the actions in a plan (at the same time as generating the plan), and forbidding any actions which result in a state already in the list.
- In Prolog we can use negation as failure to say that a new state is not already in a list of previous states.

Places and Actions in the Monkey and Banana Program

```
place(atWindow).
place(middle).
place(atDoor).

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)) :-
    place(Position).

can_do(state(Position_before, onFloor, Position_before, X),
       push(Position_before, Position_after),
       state(Position_after, onFloor, Position_after, X)) :-
    place(Position_before),
    place(Position_after),
    Position_before \== Position_after.

can_do(state(Position_before, onFloor, X, Y),
       walk(Position_before, Position_after),
       state(Position_after, onFloor, X, Y)) :-
    place(Position_before),
    place(Position_after),
    Position_before \== Position_after.
```

[monkeyAndBananaWithLoopCheck.pl]

Loop Checking in the Planning Program

- To include loop checking in our planning program, we replace the predicate 'is_plan/3' with a predicate 'is_plan/4'. This predicate says that the agent (e.g. the monkey) can go from the state described by the first argument to the (goal) state described by the second argument by following the plan in the third argument, without re-visiting the states already visited and listed in the last argument.

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

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

- This program is better than the previous program because the ordering of the 'place/1' and 'can_do/3' clauses is no longer important. A re-ordering of these clauses just results in a different ordering of the solutions.

- We can avoid having to write the final argument of `is_plan` in queries by using a "wrapper" clause to express that the "history" of previous states at the start of a plan execution consists only of the start state itself:

```
is_plan_without_loop(State, Goal_state, Plan):-
    is_plan(State, Goal_state, Plan, [State]).
```

[monkeyAndBananaWithLoopCheck.pl]

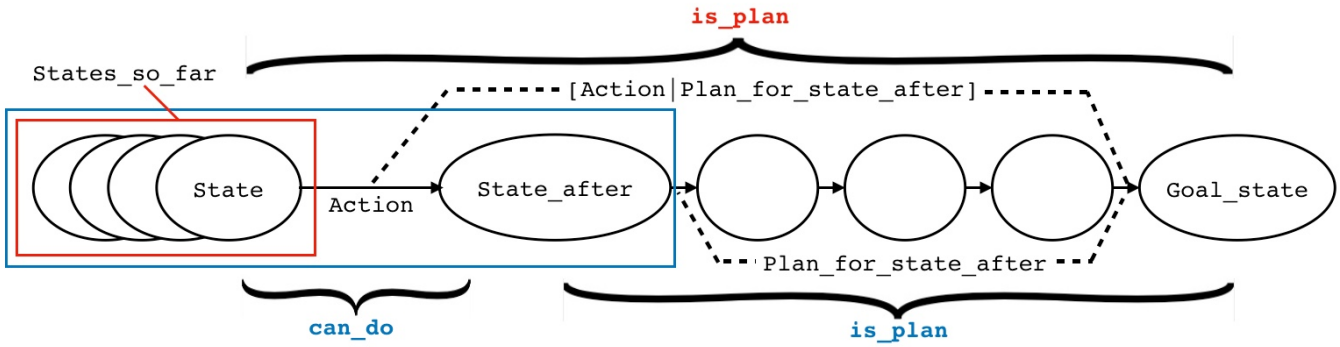
Querying the New Monkey and Banana Program

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

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

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

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

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

false.


?- is_plan_without_loop(state(middle, onFloor, atDoor, hasNot), state(_, _, _, has), Plan).

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

[monkeyAndBananaWithLoopCheck.pl]

New 'is_plan' Recursive Clause Diagram

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




```
    can_do(State, Action, State_after),
    \+ member(State_after, States_so_far)
    is_plan(State_after, Goal_state, Plan_for_state_after, [State_after|States_so_far]).
```

[is_plan_without_loop.pdf]

Input and Output

- Standard Prolog (e.g. SWI Prolog) provides pre-defined predicates to enable data to be read in from the keyboard (or a file) during the execution of a query, and for data to be output to the screen (or to a file). These input and output activities are regarded as *side effects* and do not affect the success or failure of a query.
- The goal

  `write(X)`

  always succeeds and as a side effect displays the term `X` on the screen.
- The goal

  `read(X)`

  always succeeds and as a side effect instantiates (i.e. assigns) `X` with whatever is subsequently typed at the keyboard.
- SWI Prolog will not backtrack over the goal `read(X)`.

An Abstract Program with 'write/1'

If we have the following program:

```
p(X) :- q(X), write('hello '), r(X), write('somebody ').
p(X) :- s(X), write('everybody ').

q(a).
r(b).
s(a).
s(c).
```

we can have the following dialogue with Prolog:

```
?- p(a).
hello everybody
true.

?- p(b).
false.

?- p(c).
everybody
true.
```

[abstractReadAndWrite.pl]

A More Useful Program with 'write/1' and 'read/1'

If we have the following program:

```
go :-
    write('Whose grandfather would you like to discover?\n'),
    write('Type an all-lower-case name, followed by a full stop and RETURN: '),
    read(P),
    grandfather(G, P),
    write(G),
    write(' is '),
    write(P),
    write('\'s grandfather.\n'),
    false.
go.

father(geoffrey, sylvia).
....
```

we can have the following dialogue with Prolog:

```
?- go.
Whose grandfathers would you like to discover?
Type an all-lower-case name, followed by a full stop and RETURN: natascia.
geoffrey is natascia's grandfather.
antonio is natascia's grandfather.
true.
```

[grandfathersWithPrompt.pl]

Control Primitives

- **The 'Cut' Primitive**

  Prolog provides a means of 'pruning' the search space of a program through an operator called the *cut*, written (with good reason) '`!`'. A cut may be included as a sub-goal in the body of a clause. Its effect is to prevent backtracking as follows. Once a cut has been reached in the evaluation of a query, Prolog no longer considers any alternative ways of satisfying the sub-goals that precede the cut in the clause in which it appears, and also will not consider any further clauses for the same predicate as the clause containing the cut.

- **The 'Fail' Primitive**

  Prolog allows the programmer to force a failure along a particular branch of a proof tree by inclusion of the pre-defined primitive '`fail`'. As might be expected, '`fail`' always fails.

- If you decide to use the cut and fail primitives at all, you should use them with great care, because they can easily destroy the declarative nature of a Prolog program.

An Example Program with the Cut Primitive

The following program:

```
happy(X) :-
    takes(X, Course),
    !,
    interesting(Course).

takes(jim, pascal).
takes(sally, prolog).
takes(eric, ai).

interesting(prolog).
interesting(ai).
```

gives the following example input/output:

```
?- happy(X).
false.
```

[happyWithCut.pl]

An Example Program with the Fail Primitive

The following program:

```
p(X) :-
    write(1),
    fail.
p(X) :-
    write(2),
    fail.
p(X) :-
    write(3),
    fail.
```

gives the following example input/output:

```
?- p(X).
123
false.
```

[abstractWithFail.pl]