Cuts, Negation, Arithmetic, and Classical Planning in Prolog

Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

Today

- Prolog Cuts
- Prolog Negation
- Prolog Arithmetic
- Classical Planning in Prolog

What is a Prolog Cut?

- A mechanism to restrict automatic backtracking
- Purpose is to increase efficiency by not exploring paths known in advance to be dead ends
- A cut is a special atom, denoted by **"!"**
  - a goal that always succeeds
  - 2 important side effects:
    - choice of rule now fixed
    - unifications in the rule made up to cut are now fixed

Example: Without Cut

```
:- module( cuts,
           [ a/1,b/1,c/1,d/1,e/1,f/1,p/1 ]
         ).
p(X) :- a(X).

p(X) :- b(X),c(X),d(X),e(X).

p(X) :- f(X).

a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

Q: What is the output for the query p(X) ?

Example: Without Cut

```
:- module( cuts,
           [ a/1,b/1,c/1,d/1,e/1,f/1,p/1 ]
         ).
p(X) :- a(X).

p(X) :- b(X),c(X),d(X),e(X).

p(X) :- f(X).

a(1).
b(1).  b(2).
c(1).  c(2).
d(2).
e(2).
f(3).
```

```
[trace]  ?- p(X).
   Call: (7) cuts:p(_G612) ? creep
   Call: (8) cuts:a(_G612) ? creep
   Exit: (8) cuts:a(1) ? creep
   Exit: (7) cuts:p(1) ? creep
X = 1 ;
   Redo: (7) cuts:p(_G612) ? creep
   Call: (8) cuts:b(_G612) ? creep
   Exit: (8) cuts:b(1) ? creep
   Call: (8) cuts:c(1) ? creep
   Exit: (8) cuts:c(1) ? creep
   Call: (8) cuts:d(1) ? creep
   Fail: (8) cuts:d(1) ? creep
   Redo: (8) cuts:b(_G612) ? creep
   Exit: (8) cuts:b(2) ? creep
   Call: (8) cuts:c(2) ? creep
   Exit: (8) cuts:c(2) ? creep
   Call: (8) cuts:d(2) ? creep
   Exit: (8) cuts:d(2) ? creep
   Call: (8) cuts:e(2) ? creep
   Exit: (8) cuts:e(2) ? creep
   Exit: (7) cuts:p(2) ? creep
X = 2 ;
   Redo: (7) cuts:p(_G612) ? creep
   Call: (8) cuts:f(_G612) ? creep
   Exit: (8) cuts:f(3) ? creep
   Exit: (7) cuts:p(3) ? creep
X = 3.
```

Q: What is the output for the query p(X) ?

cuts.pl

Example: Same, but with a cut

```
:- module( cuts,
           [ a/1,b/1,c/1,d/1,e/1,f/1,p/1 ]
         ).
p(X) :- a(X).

p(X) :- b(X),c(X),d(X),!,e(X).

p(X) :- f(X).

a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

Q: What is the output for the query p(X) ?

Example: With Cut

```
:- module( cuts,
           [ a/1,b/1,c/1,d/1,e/1,f/1,p/1 ]
         ).
p(X) :- a(X).

p(X) :- b(X),c(X),d(X),!,e(X).

p(X) :- f(X).

a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

```
[trace]  ?- p(X).
   Call: (7) cuts:p(_G612) ? creep
   Call: (8) cuts:a(_G612) ? creep
   Exit: (8) cuts:a(1) ? creep
   Exit: (7) cuts:p(1) ? creep
X = 1 ;
   Redo: (7) cuts:p(_G612) ? creep
   Call: (8) cuts:b(_G612) ? creep
   Exit: (8) cuts:b(1) ? creep
   Call: (8) cuts:c(1) ? creep
   Exit: (8) cuts:c(1) ? creep
   Call: (8) cuts:d(1) ? creep
   Fail: (8) cuts:d(1) ? creep
   Redo: (8) cuts:b(_G612) ? creep
   Exit: (8) cuts:b(2) ? creep
   Call: (8) cuts:c(2) ? creep
   Exit: (8) cuts:c(2) ? creep
   Call: (8) cuts:d(2) ? creep
   Exit: (8) cuts:d(2) ? creep
   Call: (8) cuts:e(2) ? creep
   Exit: (8) cuts:e(2) ? creep
   Exit: (7) cuts:p(2) ? creep
X = 2.
```

Q: What is the output for the query p(X) ?

cuts2.pl

A More Complicated Example

- Let's consider the set_diff predicate from utils.pl

```
:- module(diff,[
              member/2,
              member_set/2,
              set_diff/3
          ]).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

member_set(E,S) :- member(E,S).

set_diff([], _, []).

set_diff([H|T], S, T_new) :-
        member_set(H, S),
        set_diff(T, S, T_new),!.

set_diff([H|T], S, [H|T_new]) :-
        set_diff(T, S, T_new).
```

Q: First, what does set_diff do?

diff.pl

A More Complicated Example

- Let's look at the query result with the cut:

```
:- module(diff,[
            member/2,
            member_set/2,
            set_diff/3
        ]).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

member_set(E,S) :- member(E,S).

set_diff([], _, []).

set_diff([H|T], S, T_new) :-
        member_set(H, S),
        set_diff(T, S, T_new),!.

set_diff([H|T], S, [H|T_new]) :-
        set_diff(T, S, T_new).
```

```
[trace]  ?- set_diff([a,b,c,e],[a,c,d],X).
   Call: (7) diff:set_diff([a, b, c, e], [a, c, d], _G353) ? creep
   Call: (8) diff:member_set(a, [a, c, d]) ? creep
   Call: (9) diff:member(a, [a, c, d]) ? creep
   Exit: (9) diff:member(a, [a, c, d]) ? creep
   Exit: (8) diff:member_set(a, [a, c, d]) ? creep
   Call: (8) diff:set_diff([b, c, e], [a, c, d], _G353) ? creep
   Call: (9) diff:member_set(b, [a, c, d]) ? creep
   Call: (10) diff:member(b, [a, c, d]) ? creep
   Call: (11) diff:member(b, [c, d]) ? creep
   Call: (12) diff:member(b, [d]) ? creep
   Call: (13) diff:member(b, []) ? creep
   Fail: (13) diff:member(b, []) ? creep
   Fail: (12) diff:member(b, [d]) ? creep
   Fail: (11) diff:member(b, [c, d]) ? creep
   Fail: (10) diff:member(b, [a, c, d]) ? creep
   Fail: (9) diff:member_set(b, [a, c, d]) ? creep
   Redo: (8) diff:set_diff([b, c, e], [a, c, d], _G353) ? creep
   Call: (9) diff:set_diff([c, e], [a, c, d], _G447) ? creep
   Call: (10) diff:member_set(c, [a, c, d]) ? creep
   Call: (11) diff:member(c, [a, c, d]) ? creep
   Call: (12) diff:member(c, [c, d]) ? creep
   Exit: (12) diff:member(c, [c, d]) ? creep
   Exit: (11) diff:member(c, [a, c, d]) ? creep
   Exit: (10) diff:member_set(c, [a, c, d]) ? creep
   Call: (10) diff:set_diff([e], [a, c, d], _G447) ? creep
   Call: (11) diff:member_set(e, [a, c, d]) ? creep
   Call: (12) diff:member(e, [a, c, d]) ? creep
   Call: (13) diff:member(e, [c, d]) ? creep
   Call: (14) diff:member(e, [d]) ? creep
   Call: (15) diff:member(e, []) ? creep
   Fail: (15) diff:member(e, []) ? creep
   Fail: (14) diff:member(e, [d]) ? creep
   Fail: (13) diff:member(e, [c, d]) ? creep
   Fail: (12) diff:member(e, [a, c, d]) ? creep
   Fail: (11) diff:member_set(e, [a, c, d]) ? creep
   Redo: (10) diff:set_diff([e], [a, c, d], _G447) ? creep
   Call: (11) diff:set_diff([], [a, c, d], _G450) ? creep
   Exit: (11) diff:set_diff([], [a, c, d], []) ? creep
   Exit: (10) diff:set_diff([e], [a, c, d], [e]) ? creep
   Exit: (9) diff:set_diff([c, e], [a, c, d], [e]) ? creep
   Exit: (8) diff:set_diff([b, c, e], [a, c, d], [b, e]) ? creep
   Exit: (7) diff:set_diff([a, b, c, e], [a, c, d], [b, e]) ? creep
X = [b, e].
```

- And without (giving incorrect results):

```
?- set_diff([a,b,c,e],[a,c,d],X).
X = [b, e] ;
X = [b, c, e] ;
X = [a, b, e] ;
X = [a, b, c, e].
```

diff.pl

Today

- Prolog Cuts
- Prolog Negation
- Prolog Arithmetic
- Classical Planning in Prolog

Negation as Failure

- The issue: How to express a rule with an exception in Prolog
- Suppose: Vincent loves burgers, but not Big Kahuna Burgers.
- There are two ways to do this:
  - Cut-Fail:

```
enjoys(vincent,X)  :-  big_kahuna_burger(X),!,fail.
enjoys(vincent,X)  :-  burger(X).
```

  - Not (or “\+”)

```
enjoys(vincent,X)  :-  burger(X), not(big_kahuna_burger(X)).
```

or

```
enjoys(vincent,X)  :-  burger(X), \+ big_kahuna_burger(X).
```

Example: Cut-Fail

```
:-  module( negation,
            [ enjoys/2, burger/1, big_mac/1,
              big_kahuna_burger/1, whopper/1 ]
          ).
enjoys(vincent,X)   :-   big_kahuna_burger(X),!,fail.
enjoys(vincent,X)   :-   burger(X).

burger(X)   :-   big_mac(X).
burger(X)   :-   big_kahuna_burger(X).
burger(X)   :-   whopper(X).

big_mac(a).
big_mac(b).
big_kahuna_burger(c).
whopper(d).
```

Queries and results:

```
?- enjoys(vincent,a).
true .

?- enjoys(vincent,b).
true .

?- enjoys(vincent,c).
false.

?- enjoys(vincent,d).
true.

?- enjoys(vincent,X).
false.
```

With "cut-fail", the general query fails

Example: Not

```
:-  module( negation2,
            [ enjoys/2, burger/1, big_mac/1,
              big_kahuna_burger/1, whopper/1 ]
          ).
enjoys(vincent,X)  :-  burger(X),not(big_kahuna_burger(X)).

burger(X)  :-  big_mac(X).
burger(X)  :-  big_kahuna_burger(X).
burger(X)  :-  whopper(X).

big_mac(a).
big_mac(b).
big_kahuna_burger(c).
whopper(d).
```

Queries and results:

```
?- enjoys(vincent,a).
true .

?- enjoys(vincent,b).
true .

?- enjoys(vincent,c).
false.

?- enjoys(vincent,d).
true.

?- enjoys(vincent,X).
X = a ;
X = b ;
X = d.
```

With "not", we can use the general query

Today

- Prolog Cuts
- Prolog Negation
- Prolog Arithmetic
- Classical Planning in Prolog

Prolog Arithmetic

- Basic ideas:
  1. Arithmetic expressions are goals, like any other
  2. Prolog understands ordinary arithmetic syntax

- Examples:

```
?- 8 is 6+2.
yes

?- R is mod(7,2).
R = 1
```

Using Arithmetic

- Length of a list:

```
len([],0).
len([_|T],N)  :-  len(T,X),  N  is  X+1.
```

Example: ?- len([a,b,c,d,e,[a,b],g],X).
X = 7

Note how the number is an *argument* in a predicate

- Increment operation:

```
increment( X, Y ) :- Y is X + 1, Y > 0.
```

Example: ?- increment(6,X).
X = 7.

Today

- Prolog Cuts
- Prolog Negation
- Prolog Arithmetic
- Classical Planning in Prolog

Planning Concepts

- **Plan**: a sequence of actions for achieving a goal

- Actions have **preconditions (antecedents)**:
  - what the current state of the world must have for an action to be available

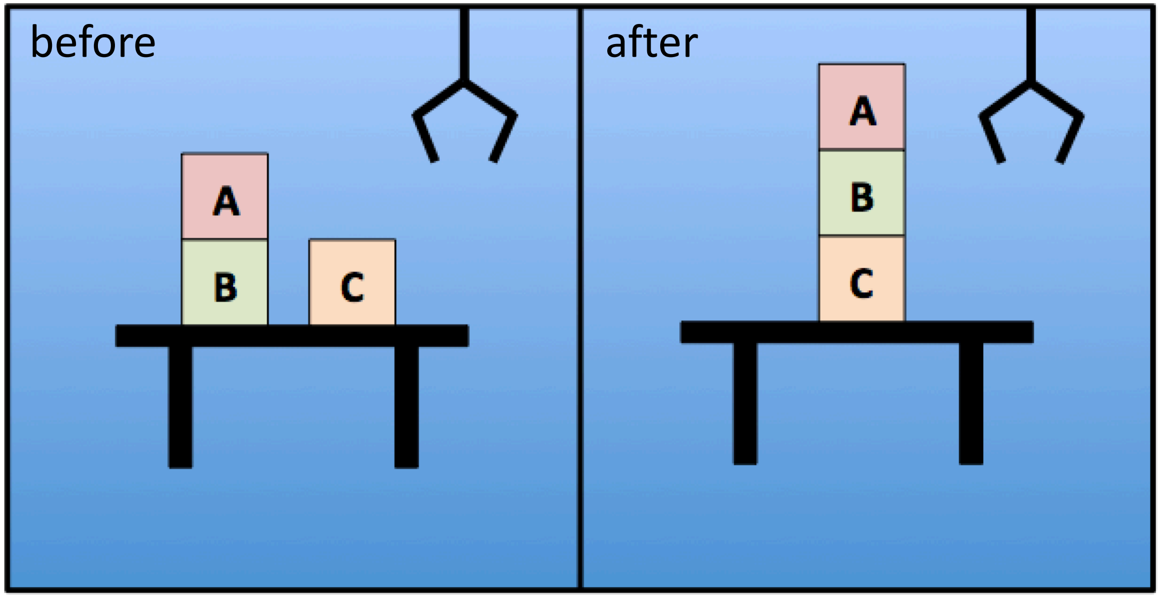
- Actions have **effects (consequences)**:
  - the state of the world changes based on the action taken
  - actions typically leave most aspects of the world the same

- Assumptions made by Prolog
  - **closed-world** – everything that is true in the world is stated in the KB
  - **unique names** – if we have 2 different names, then we're talking about 2 different objects

- *Prolog rules can easily associate preconditions and effects with particular actions*

Blocks World

- **Problem domain:** Set of cube-shaped blocks on a table; some stacked
  Robot arm that can pick up, move, and put down
- **Goal:** Rearrange blocks into a particular arrangement



Prolog Design

- State predicates:
  - robot arm — handempty
    holding(X)
  - blocks — ontable(X)
    on(X, Y) ← *block X is on top of block Y*
    clear(X) ← *nothing on top of block X*
- Rules:
  - for actions, of the form: action(name(), [preconditions], [effects]).
  - for verifying preconditions
  - for changing state based on action effects

Dissecting a Prolog Blocks World Planner

```
%%%%%%%%% Simple Prolog Planner %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% Based on one of the sample programs in:
%%%
%%% Artificial Intelligence:
%%% Structures and strategies for complex problem solving
%%%
%%% by George F. Luger and William A. Stubblefield
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- module( planner,
        [
            plan/4,change_state/3,conditions_met/2,member_state/2,
            move/3,go/2,test/0,test2/0
        ]).

:- [utils].

plan(State, Goal, _, Moves) :-  equal_set(State, Goal),
                                write('moves are'), nl,
                                reverse_print_stack(Moves).
plan(State, Goal, Been_list, Moves) :-
                                move(Name, Preconditions, Actions),
                                conditions_met(Preconditions, State),
                                change_state(State, Actions, Child_state),
                                not(member_state(Child_state, Been_list)),
                                stack(Child_state, Been_list, New_been_list),
                                stack(Name, Moves, New_moves),
                        plan(Child_state, Goal, New_been_list, New_moves),!.

change_state(S, [], S).
change_state(S, [add(P)|T], S_new) :-   change_state(S, T, S2),
                                        add_to_set(P, S2, S_new), !.
change_state(S, [del(P)|T], S_new) :-   change_state(S, T, S2),
                                        remove_from_set(P, S2, S_new), !.
conditions_met(P, S) :- subset(P, S).
```

```
member_state(S, [H|_]) :-       equal_set(S, H).
member_state(S, [_|T]) :-       member_state(S, T).

/* move types */

move(pickup(X), [handempty, clear(X), on(X, Y)],
                [del(handempty), del(clear(X)), del(on(X, Y)),
                                 add(clear(Y)), add(holding(X))]).

move(pickup(X), [handempty, clear(X), ontable(X)],
                [del(handempty), del(clear(X)), del(ontable(X)),
                                 add(holding(X))]).

move(putdown(X), [holding(X)],
                [del(holding(X)), add(ontable(X)), add(clear(X)),
                                  add(handempty)]).

move(stack(X, Y), [holding(X), clear(Y)],
                [del(holding(X)), del(clear(Y)), add(handempty), add(on(X, Y)),
                                  add(clear(X))]).
/* run commands */

go(S, G) :- plan(S, G, [S], []).

test :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
              [handempty, ontable(c), on(a,b), on(b, c), clear(a)]).

test2 :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
              [handempty, ontable(a), ontable(b), on(c, b), clear(a), clear(c)]).
```

We will discuss the details in the following slides

Planner Dissection: Preliminaries

- First line of planner loads Utils.pl, which contains our abstract data types

- Utils.pl contains:

```
%%% List %%%

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

writelist([ ]) :- nl.
writelist([H|T]) :- write(' '), write(H), writelist(T).

%%% Stack %%%

reverse_print_stack(S) :- empty_stack(S).
reverse_print_stack(S) :-
  stack(E, Rest, S),
  reverse_print_stack(Rest),
  write(E), nl.

empty_stack([ ]).

stack(Top, Stack, [Top|Stack]).

member_stack(Element, Stack) :- member(Element, Stack).

%%% Queue %%%

empty_queue( [ ] ).

enqueue(E,[],[E]).
enqueue(E,[H|T],[H|W]) :- enqueue(E,T,W).

dequeue(E,[E|T],T).
dequeue(E,[E|_],_).

member_queue(E,Q) :- member(E,Q).
```

```
%%% Set %%%

empty_set( [ ] ).

member_set(E,S) :- member(E,S).

add_to_set(X, S, S) :- member(X, S), !.
add_to_set(X, S, [X|S]).

remove_from_set(_, [], []).
remove_from_set(E, [E|T], T) :- !.
remove_from_set(E, [H|T], [H|T_new]) :-
        remove_from_set(E, T, T_new), !.

union([], S, S).
union([H|T], S, S_new) :-
        union(T, S, S2),
        add_to_set(H, S2, S_new).

intersection([], _, []).
intersection([H|T], S, [H|S_new]) :-
        member_set(H, S),
        intersection(T, S, S_new),!.
intersection([_|T], S, S_new) :-
        intersection(T, S, S_new),!.

set_diff([], _, []).
set_diff([H|T], S, T_new) :-
        member_set(H, S),
        set_diff(T, S, T_new),!.
set_diff([H|T], S, [H|T_new]) :-
        set_diff(T, S, T_new), !.

subset([], _).
subset([H|T], S) :-
        member_set(H, S),
        subset(T, S).

equal_set(S1, S2) :-
        subset(S1, S2),
        subset(S2, S1).
```

Planner: Plan

- Plan is defined recursively
- Terminate search when current state is goal state
- Uses a "visited" list called “been” to check if we have seen a child state before

```
plan(State, Goal, _, Moves) :-  equal_set(State, Goal),
                                write('moves are'), nl,
                                reverse_print_stack(Moves).
plan(State, Goal, Been_list, Moves) :-
                                move(Name, Preconditions, Actions),
                                conditions_met(Preconditions, State),
                                change_state(State, Actions, Child_state),
                                not(member_state(Child_state, Been_list)),
                                stack(Child_state, Been_list, New_been_list),
                                stack(Name, Moves, New_moves),
                        plan(Child_state, Goal, New_been_list, New_moves),!.
```

Planner: Administrative Rules

- Rules needed:
  - determining when preconditions are satisfied
  - changing the state based on taking an action
  - determining whether a state is in the “been” list

```
change_state(S, [], S).
change_state(S, [add(P)|T], S_new) :-   change_state(S, T, S2),
                                        add_to_set(P, S2, S_new), !.
change_state(S, [del(P)|T], S_new) :-   change_state(S, T, S2),
                                        remove_from_set(P, S2, S_new), !.
conditions_met(P, S) :- subset(P, S).

member_state(S, [H|_]) :-       equal_set(S, H).
member_state(S, [_|T]) :-       member_state(S, T).
```

Planner: Actions

- Action predicates determine the state changes for each possible situation
- They also provide a move "name" for use in recording the move

```
move(pickup(X), [handempty, clear(X), on(X, Y)],
                [del(handempty), del(clear(X)), del(on(X, Y)),
                                 add(clear(Y)), add(holding(X))]).

move(pickup(X), [handempty, clear(X), ontable(X)],
                [del(handempty), del(clear(X)), del(ontable(X)),
                                 add(holding(X))]).

move(putdown(X), [holding(X)],
                [del(holding(X)), add(ontable(X)), add(clear(X)),
                                  add(handempty)]).

move(stack(X, Y), [holding(X), clear(Y)],
                [del(holding(X)), del(clear(Y)), add(handempty), add(on(X, Y)),
                                  add(clear(X))]).
```

Planner: Launching and Testing

- Program launch predicate
- Test predicates

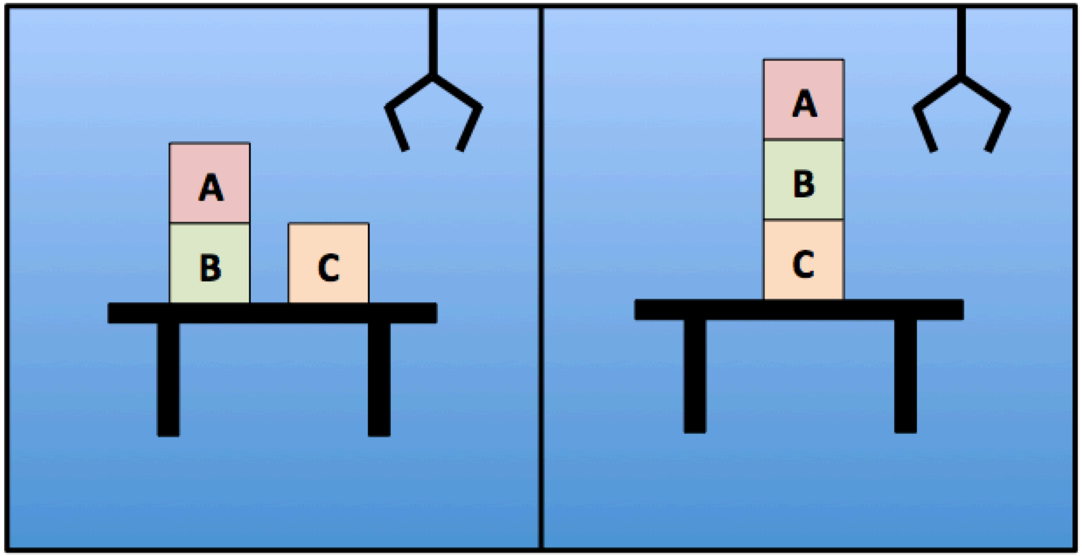
```
go(S, G) :- plan(S, G, [S], []).

test :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
                  [handempty, ontable(c), on(a,b), on(b, c), clear(a)]).

test2 :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
                  [handempty, ontable(a), ontable(b), on(c, b), clear(a), clear(c)]).
```

Planner: “test” demo

```
test :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
           [handempty, ontable(c), on(a,b), on(b, c), clear(a)]).
```

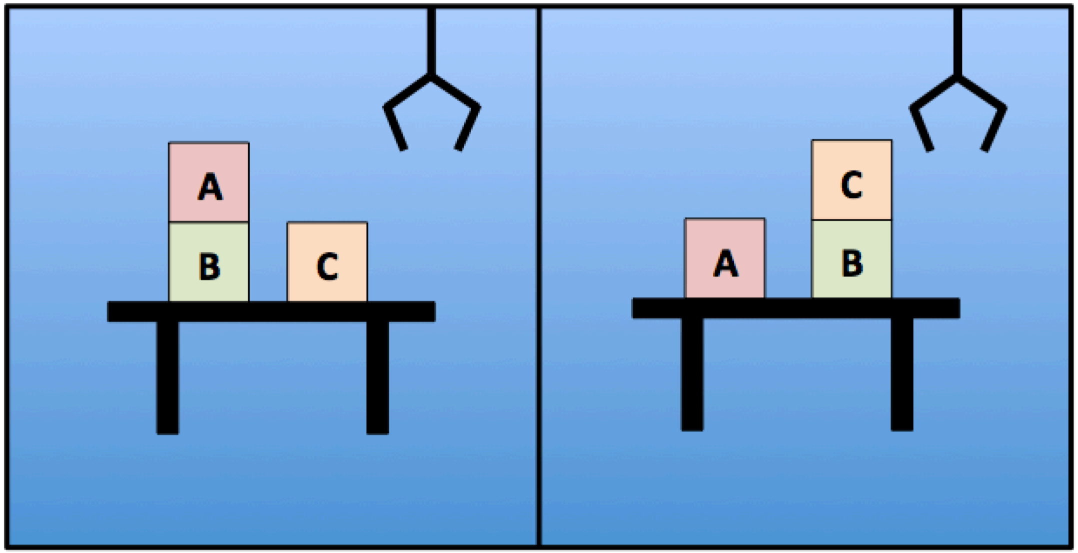



```
?- test.
moves are
pickup(a)
putdown(a)
pickup(b)
stack(b,c)
pickup(a)
stack(a,b)
true.
```

demo: planner.pl

Planner: “test2” demo

```
test2 :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
            [handempty, ontable(a), ontable(b), on(c, b), clear(a), clear(c)]).
```




```
?- test2.
moves are
pickup(a)
putdown(a)
pickup(c)
stack(c,b)
true.
```