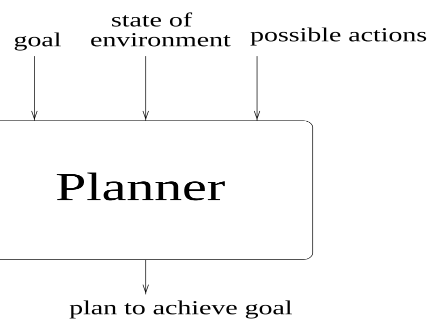
PLANNING

1 What is Planning?

- Key problem facing *agent* is *deciding what to do*.
- We want agents to be *taskable*: give them *goals* to achieve, have them decide for themselves how to achieve them.
- Basic idea is to give an agent:
  - representation of goal to achieve;
  - knowledge about what actions it can perform; and
  - knowledge about state of the world;

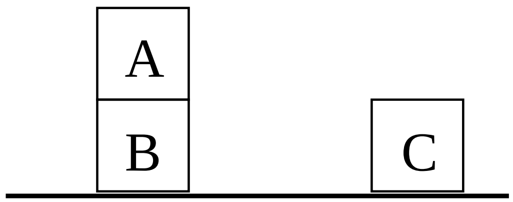
  and to have it generate a *plan* to achieve the goal.
- Essentially, this is

  automatic programming.



- Question: How do we *represent*...
  - goal to be achieved;
  - state of environment;
  - actions available to agent;
  - plan itself.
- We show how all this can be done in first-order logic...

- We'll illustrate the techniques with reference to the *blocks world*.
- Contains a robot arm, 3 blocks (A, B and C) of equal size, and a table-top.
- Initial state:



- To represent this environment, need an *ontology*.

$On(x, y)$	obj x on top of obj y
$OnTable(x)$	obj x is on the table
$Clear(x)$	nothing is on top of obj x
$Holding(x)$	arm is holding x

- Here is a FOL representation of the blocks world described above:

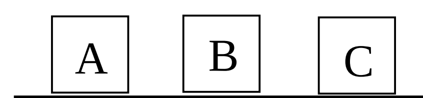
$$Clear(A)$$
$$On(A, B)$$
$$OnTable(B)$$
$$OnTable(C)$$
$$Clear(C)$$

- Use the *closed world assumption*: anything not stated is assumed to be *false*.

- A *goal* is represented as a FOL formula.
- Here is a goal:

$$OnTable(A) OnTable(B) OnTable(C)$$

- Which corresponds to the state:



- *Actions* are represented using a technique that was developed in the STRIPS planner.

- Each action has:
  - a *name*
    which may have arguments;
  - a *pre-condition list*
    list of facts which must be true for action to be executed;
  - a *delete list*
    list of facts that are no longer true after action is performed;
  - an *add list*
    list of facts made true by executing the action.

  Each of these may contain *variables*.

- Example 1:
  The *stack* action occurs when the robot arm places the object x it is holding is placed on top of object y.

	$Stack(x, y)$
pre	$Clear(y) \wedge Holding(x)$
del	$Clear(y) \wedge Holding(x)$
add	$ArmEmpty \wedge On(x, y)$

- Example 2:
  The *unstack* action occurs when the robot arm picks an object x up from on top of another object y.

	$UnStack(x, y)$
pre	$On(x, y) \wedge Clear(x) \wedge ArmEmpty$
del	$On(x, y) \wedge ArmEmpty$
add	$Holding(x) \wedge Clear(y)$

  Stack and UnStack are *inverses* of one-another.

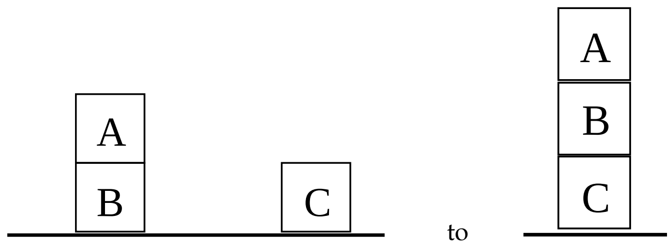
- Example 3:
  The *pickup* action occurs when the arm picks up an object x from the table.

	$Pickup(x)$
pre	$Clear(x) \wedge OnTable(x) \wedge ArmEmpty$
del	$OnTable(x) \wedge ArmEmpty$
add	$Holding(x)$

- Example 4:
  The *putdown* action occurs when the arm places the object x onto the table.

	$PutDown(x)$
pre	$Holding(x)$
del	$Holding(x)$
add	$Holding(x) \wedge ArmEmpty$

- What is a plan?
  A sequence (list) of actions, with variables replaced by constants.
- So, to get from:



- We need the set of actions:

$$
\begin{array}{c}
Unstack(A)\\
Putdown(A)\\
Pickup(B)\\
Stack(B, C)\\
Pickup(A)\\
Stack(A, B)
\end{array}
$$

- In "real life", plans contain conditionals (`IF .. THEN...`) and loops (`WHILE... DO...`), but most simple planners cannot handle such constructs — they construct *linear plans*.
- Simplest approach to planning: *means-ends analysis*.
- Involves backward chaining from goal to original state.
- Start by finding an action that has goal as post-condition.
  Assume this is the *last* action in plan.
- Then figure out what the previous state would have been.
  Try to find action that has *this* state as post-condition.
- *Recurse* until we end up (hopefully!) in original state.


```
function plan(
    d : WorldDesc,          // initial env state
    g : Goal,               // goal to be achieved
    p : Plan,               // plan so far
    A : set of actions      // actions available)
    if d ⊨ g then
        return p
    else
        choose a in A such that
            add(a) ⊨ g and
            del(a) ⊭ g
        set g = pre(a)
        append a to p
        return plan(d, g, p, A)
```

- How does this work on the previous example?

- This algorithm not guaranteed to find the plan...
- ... but it is *sound*: If it finds the plan is correct.
- Some problems:
  - negative goals;
  - maintenance goals;
  - conditionals & loops;
  - exponential search space;
  - logical consequence tests;

The Frame Problem

- A general problem with representing properties of actions:

  How do we know exactly what changes as the result of performing an action?

  If I pick up a block, does my hair colour stay the same?
- One solution is to write *frame axioms*.

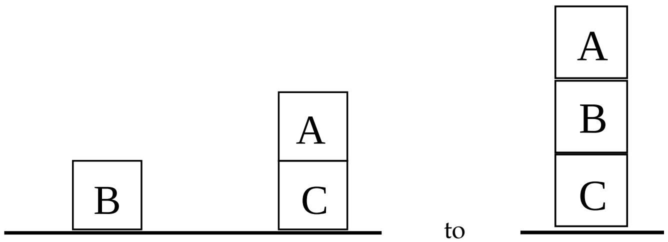
  Here is a frame axiom, which states that SP's hair colour is the same in all the situations s' that result from performing $Pickup(x)$ in situation s as it is in s.

$$\forall s, s'.Result(SP, Pickup(x), s) = s' \Rightarrow$$
$$HCol(SP, s) = HCol(SP, s')$$

- Stating frame axioms in this way is unfeasible for real problems.
- (Think of all the things that we would have to state in order to cover all the possible frame axioms).
- STRIPS solves this problem by assuming that everything not explicitly stated to have changed remains unchanged.
- We will revisit this problem in a few lectures' time.
- It connects with the general problem of handling incomplete information, and non-monotonic reasoning.

Sussman's Anomaly

- Consider we have the following initial state and goal state:



- What operations will be in the plan?

- Clearly we need to $Stack$ B on C at some point, and we also need to $Unstack$ A from C and $Stack$ it on B.
- Which operation goes first?
- Obviously we need to do the $UnStack$ first, and the $Stack\ B$ on C, but the planner has no way of knowing this.
- It also has no way of "undoing" a partial plan if it leads into a dead end.
- So if it chooses to $Stack(A, C)$ after the $Unstack$, it is sunk.
- This is a big problem with linear planners
- How could we modify our planning algorithm?

- Modify the middle of the algorithm to be:

```
1.  if d ⊨ g then
2.      return p
3.  else
4.      choose a in A such that
5.          add(a) ⊨ g and
6.          del(a) ⊭ g
6a.         no_clobber(add(a), del(a), rest_of_plan)
7.      set g = pre(a)
8.      append a to p
9.      return plan(d, g, p, A)
```


Partial Order Planning

- So we check before adding an action to the plan that it doesn't mess up the rest of the plan.
- The problem is that in this recursive process, we don't know what the rest of the plan is.
- We also have little idea which things will clobber what things.
- We need to do two things:
  - Add information to the plan representation.
  - Think about plans in a different way.

- Planning can be though of as a search problem.
- As we have viewed it so far, it is a search through a space of possible situations.
- We have a start situation and an end situation, and each operation takes us from one situation to another.
- We can also think of it as a search through a space of possible plans.
- Each operation added then reduces the space of possible plans in which the plan we are constructing can lie.
- In the middle of planning we have a *partial plan* with some steps filled in with operations, and other steps still to be filled in.

- What we do is to instantiate this partial plan step by step.
- But the crucial thing is that we don't have to put the steps in a particular order.
- Thus before adding a step we can check that it doesn't clobber other steps.
- To guide the way we put plans together, we have information about which steps come before which other steps.
- This is the extra information we have to add.
- We also record all the partial plans.
- Then, if we find that adding a new step clobbers some step which currently comes later in the plan, we can *backtrack* and try and find a different ordering.

Planning and Acting

- How do we fit in carrying out actions with building plans?
- The most naive view is:
  1. Build plan
  2. Execute plan
- This is fine if the plan is guaranteed to succeed.
- If not, we may find we have got to the end of the plan and not achieved the goal.

- So we can modify the sequence:
  1. Build plan
  2. Execute plan
  3. Check if plan succeeded
  4. If yes, hurrah
  5. Else, go back to 1.
- This kind of process is sound, it will get to the goal eventually, but it is wasteful.
- Actions typically cost, so we want to minimise the number of useless actions we carry out.

- So we check if actions have succeeded:
  1. Build plan
  2. Execute action.
  3. Check if action succeeded
  4. If yes, go back to 2.
  5. Else, go back to 1.
- This approach has its own costs—checking that an action has succeeded can be hard.
- It also only covers the last action.

- What we really want to do is to check that the plan will still work.
- So we actually want:
  1. Build plan
  2. Execute action.
  3. Check if plan will still achieve goal
  4. If yes, go back to 2.
  5. Else, go back to 1.
- This checking is even harder, especially if we want to do it well.
- As we will see next lecture, what we really need is a *policy*—a plan that tells us what to do in all possible cases.

Summary

- This lecture has looked at planning.
- We looked mainly at a logical view of planning, using STRIPS operators.
- We also discussed the frame problem, and Sussman's anomaly.
- Sussman's anomaly motivated some thoughts about partial-order planning.
- Thinking about actions that can fail suggested we need to think of plans as more than just linear sequences of actions.