Planning

CIS 32

Functionalia

HW 4 is Up - Quesitons?

It is due in one (1) week, and is comprised of the Neural Network Problem; and a Planning Problem.

Today:

Planning (Brief Overview Again)

STRIPS Planning

Partial-Order Planning

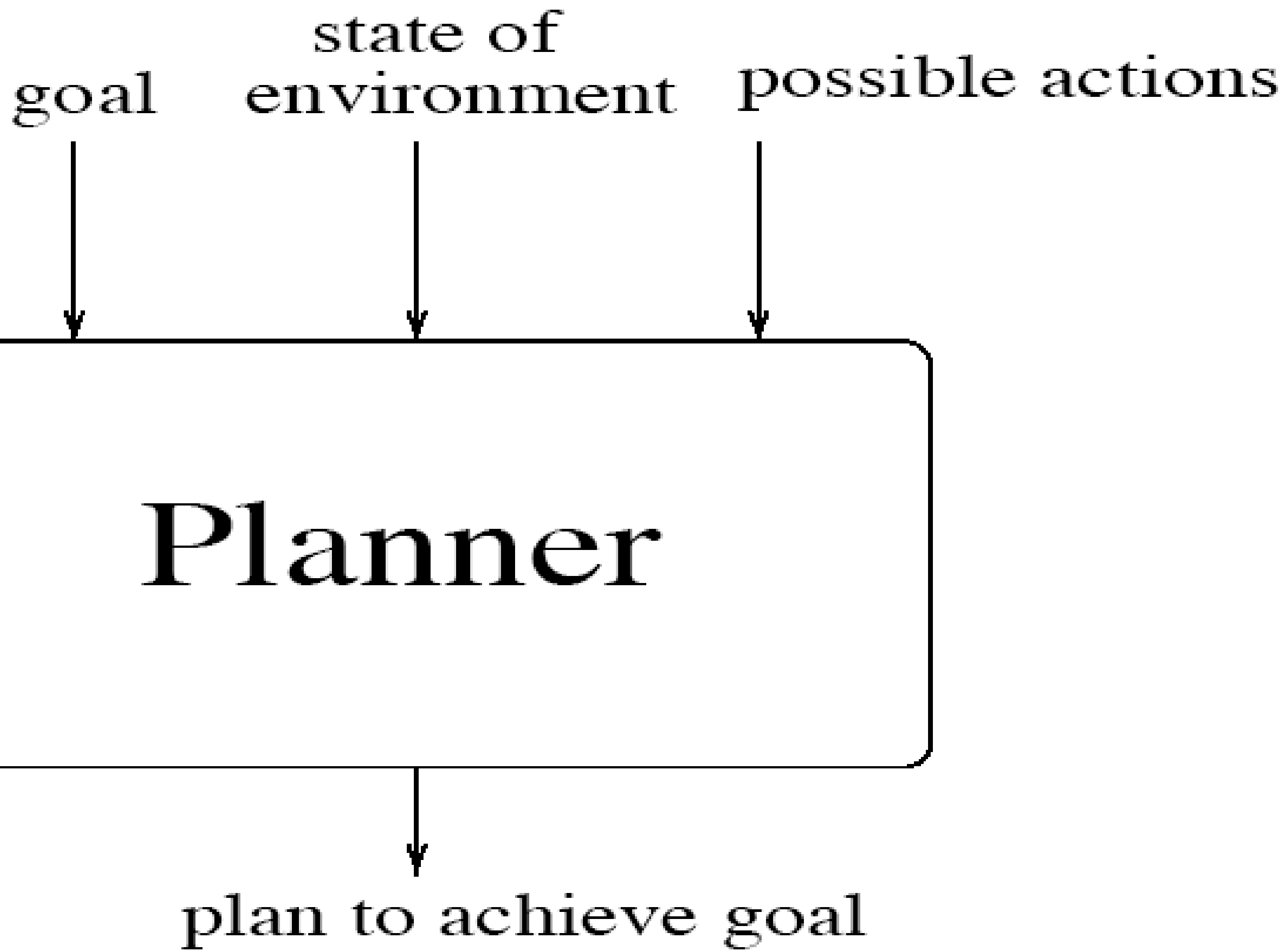
What is planning?

- 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

High Level Box View of a Planner



- **Question**: How do we *represent*. . .
  - goal to be achieved;
  - state of environment;
  - actions available to agent;
  - plan itself.

Language *expressive* enough to describe a wide variety of problems.

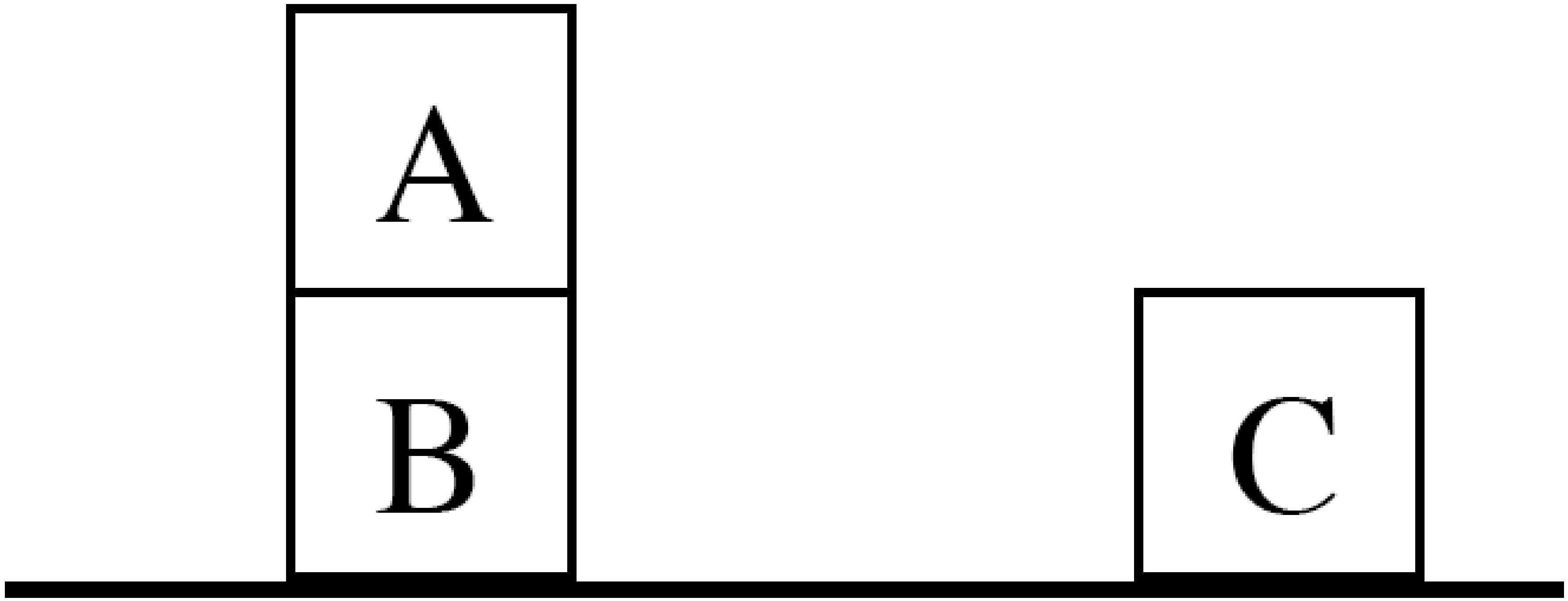
Language *restrictive* enough to allow efficient operation over them.

STRIPS Language (using First Order Logic).

STandford **R**esearch **I**nstitute **P**roblem **S**olver.

Blocks World

- 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 first-order logic representation of the blocks world described above:

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

• Uses *ground* literals (function-free)

not Clear(x,y), or Clear(OnTopOf(B))

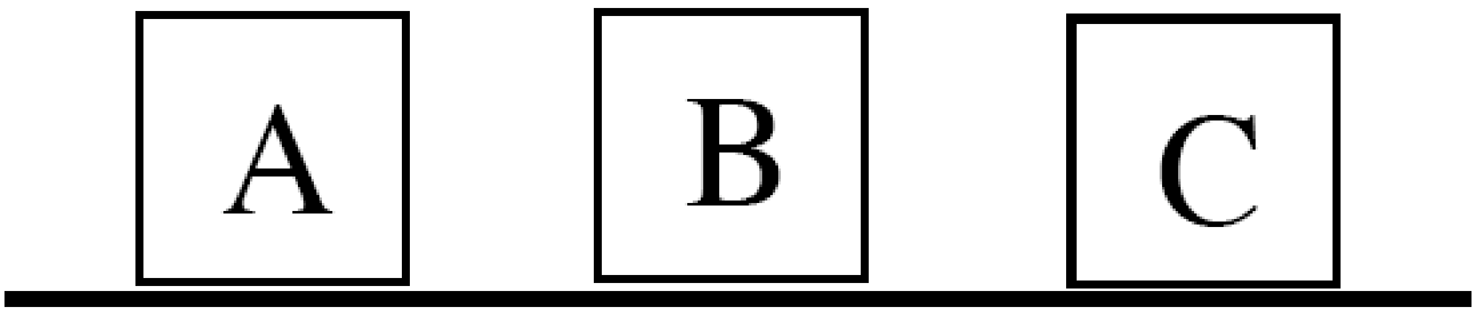
• Use the *closed world assumption*: anything not stated is assumed to be *false*

- A *goal* is represented as a first-order logic formula. (Conjunction of positive ground literals)

- Here is a goal:

$$OnTable(A) \wedge OnTable(B) \wedge OnTable(C)$$

- 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

Stack

- **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	$ArmEmpty \wedge On(x, y)$

Unstack

- **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.

Pickup

- **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	$Holding(x)$

Putdown

- **Example 4**:

The *putdown* action occurs when the arm places the object *x* onto the table.

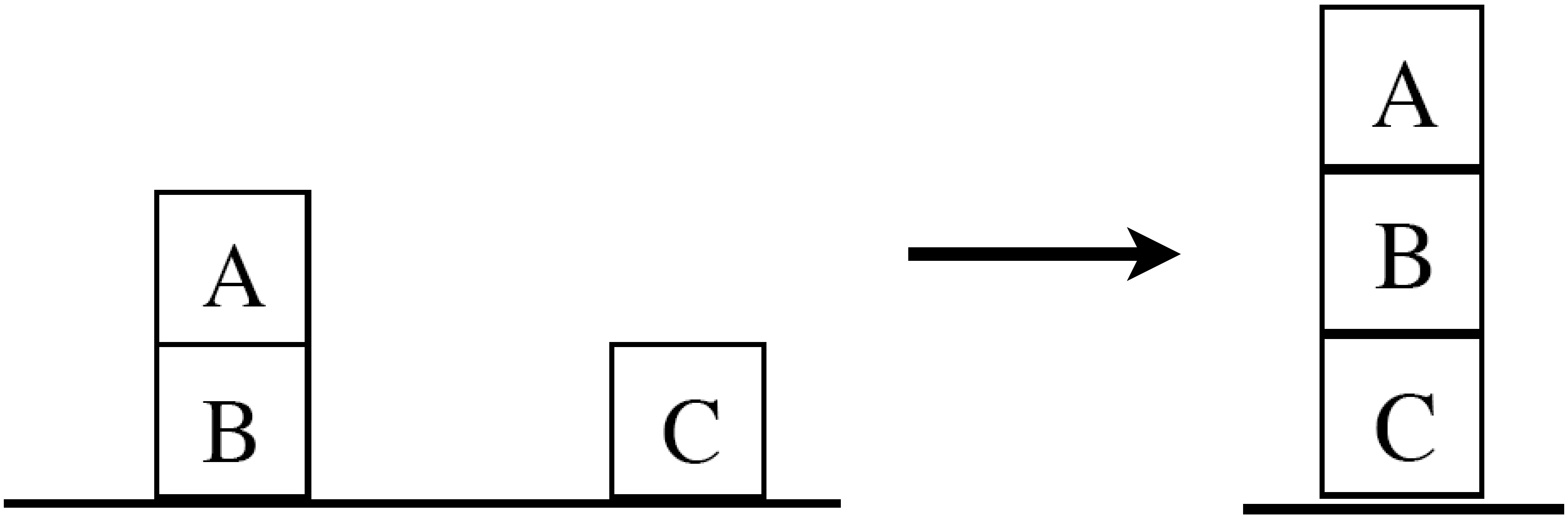
	$PutDown(x)$
pre	$Holding(x)$
del	$Holding(x)$
add	$OnTable(x) \wedge ArmEmpty$

- What is a plan?

  A sequence (list) of actions, with variables replaced by constants.

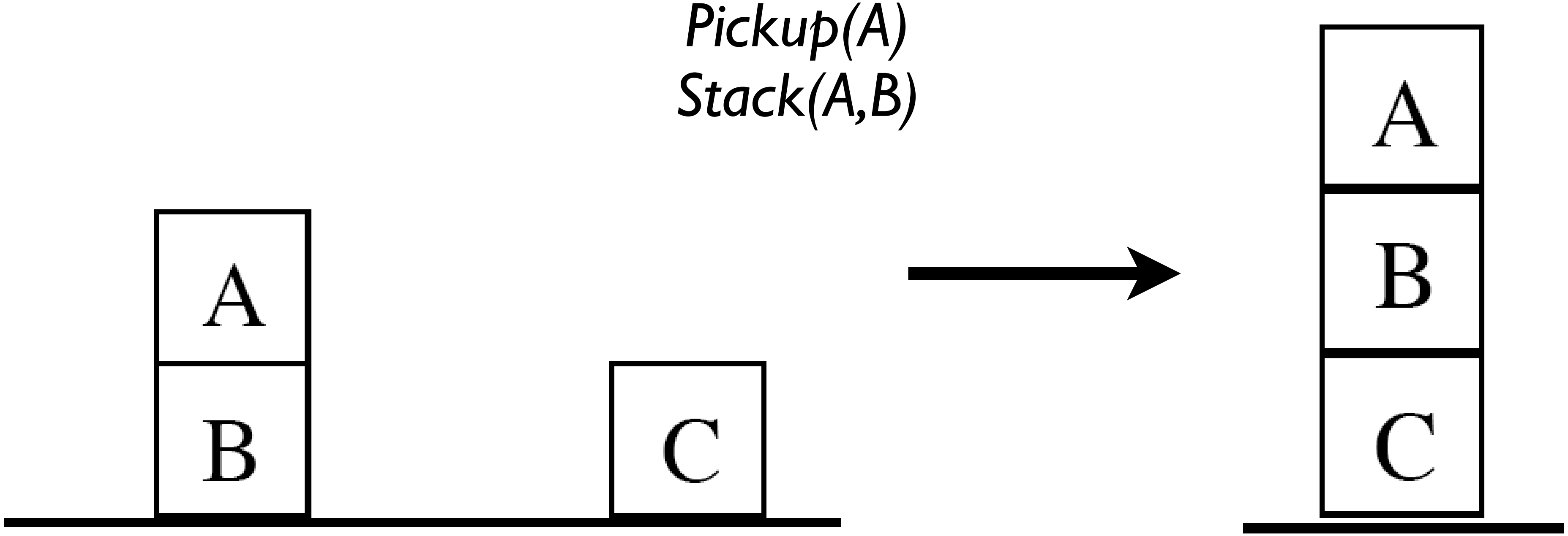
  Constants being: ***A, B, C, Floor***

- So, to get from:



- We need the set of actions:

Unstack(A)
Putdown(A)
Pickup(B)
Stack(B,C)
Pickup(A)
Stack(A,B)



- 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 **start state**.
    1. Start by finding an action that has goal as post-condition. (Assume this is the *last* action in plan.)
    2. Then figure out what the previous state would have been.
    3. Try to find action that has *this* state as post-condition.
- *Recurse* until we end up (hopefully!) in original state

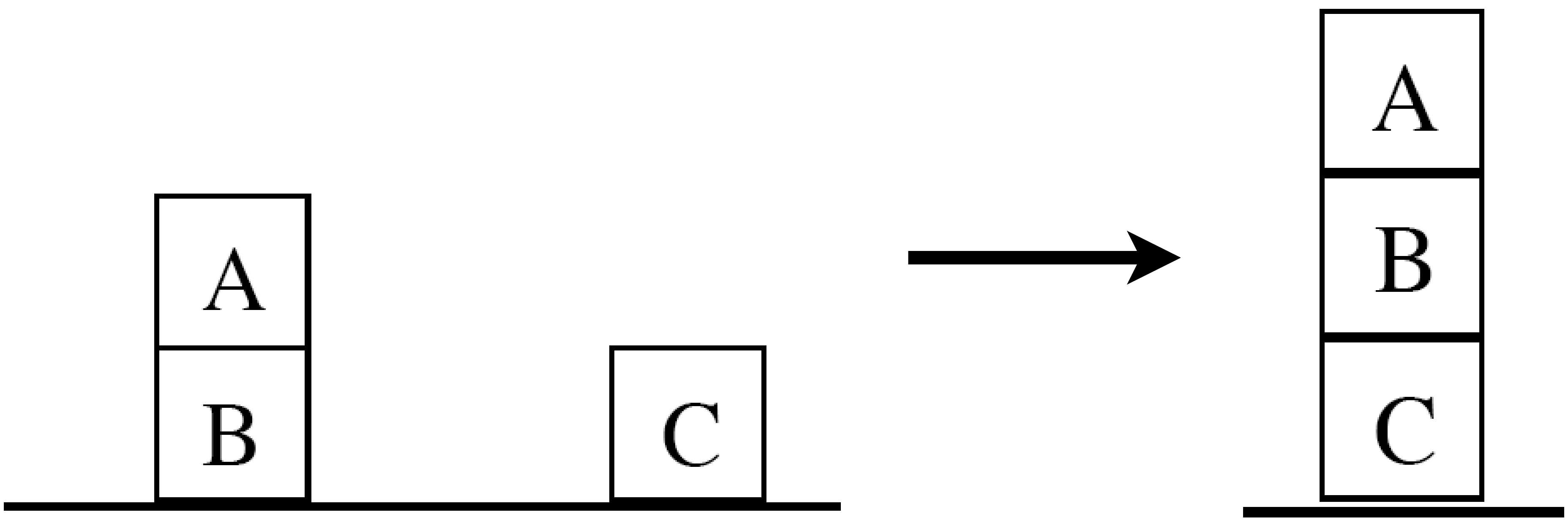
```
function plan(

    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)
```

All positive effects of ***a*** that appear in ***g*** are deleted.
Each precondition literal of ***a*** is added, unless it already appears.

How does this work on the previous example?



How does this work on the previous example?

Start		**Goal**
OnTable(C)		*OnTable(C)*
OnTable(B)		*On(B,C)*
On(A,B)	→	*On(A,B)*
Clear(A)		*Clear(A)*
Clear(C)		*Clear(Floor)*
Clear(Floor)		*ArmEmpty*
ArmEmpty		

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

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

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

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

Start

OnTable(C)
OnTable(B)
On(A,B)
Clear(A)
Clear(C)
Clear(Floor)
ArmEmpty

Goal

OnTable(C)
On(B,C)
On(A,B)
Clear(A)
Clear(Floor)
ArmEmpty

	Stack(x, y)
pre	*Clear(y) ∧ Holding(x)*
del	*Clear(y) ∧ Holding(x)*
add	*ArmEmpty ∧ On(x, y)*

	UnStack(x, y)
pre	*On(x, y) ∧ Clear(x) ∧ ArmEmpty*
del	*On(x, y) ∧ ArmEmpty*
add	*Holding(x) ∧ Clear(y)*

	Pickup(x)
pre	*Clear(x) ∧ OnTable(x) ∧ ArmEmpty*
del	*OnTable(x) ∧ ArmEmpty*
add	*Holding(x)*

	PutDown(x)
pre	*Holding(x)*
del	*Holding(x)*
add	*OnTable(x) ∧ ArmEmpty*

Start

OnTable(C)
OnTable(B)
On(A,B)
Clear(A)
Clear(C)
Clear(Floor)
ArmEmpty

→

Goal

OnTable(C)
On(B,C)
On(A,B)
Clear(A)
Clear(Floor)
ArmEmpty

- This algorithm not guaranteed to find the plan. . .

. . . but it is *sound*: If it finds a plan, that plan is correct.

- Some problems:
  - negative goals;
  - maintenance of goals(Frame Problem, Clobbering);
  - conditionals & loops;
  - exponential search space;

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 color stay the same?

- One solution is to write *frame axioms*.

Here is a frame axiom, which states that *CHIPP's* hair color is the same in all the situations *(symbolized by s and s')* that result from performing *Pickup(x)* in situation *s* as it is in *s'*.

$$\forall s, s'. \mathit{Result}(\mathit{CHIPP}, \mathit{Pickup}(x), s) = s' \Rightarrow$$

$$\mathit{HairColor}(\mathit{CHIPP}, s) = \mathit{HairColor}(\mathit{CHIPP}, s')$$

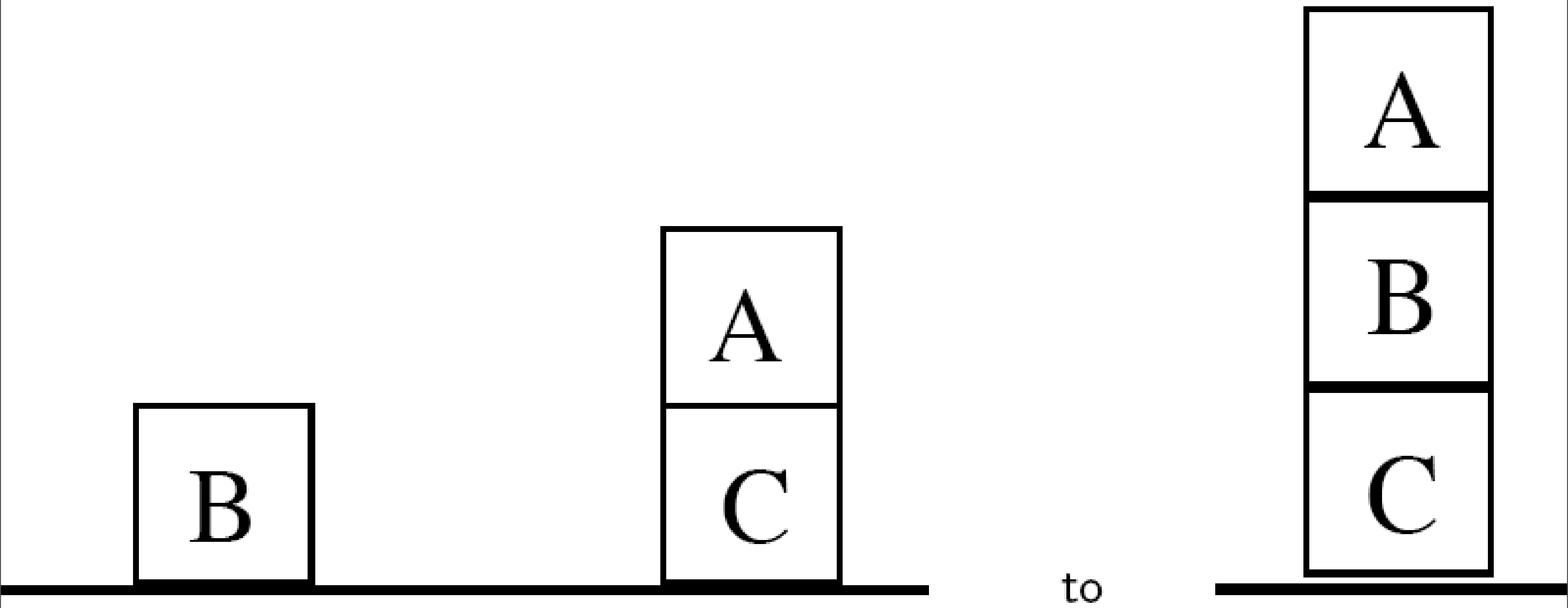
STRIPS Planning

- 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.
- The price we pay for this is that we lose the advantages of using logic:
  - – Semantics goes out of the window
- However, more recent work has effectively solved the frame problem (using clever second-order approaches).

Second-order Logic (BTW) - involved quantification across different sets

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 $d \models g$ then
2. return p
3. else
4. choose a in A such that
5. $add(a) \models g$ and
6. $del(a) \not\models g$

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)*

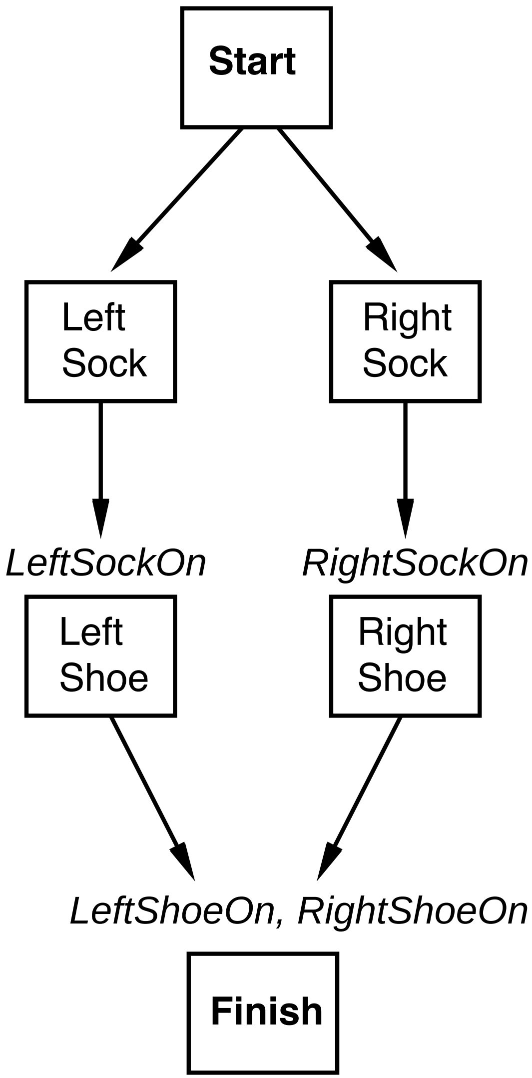
- We do this with *partial-order planning*.

Partial Order Planning

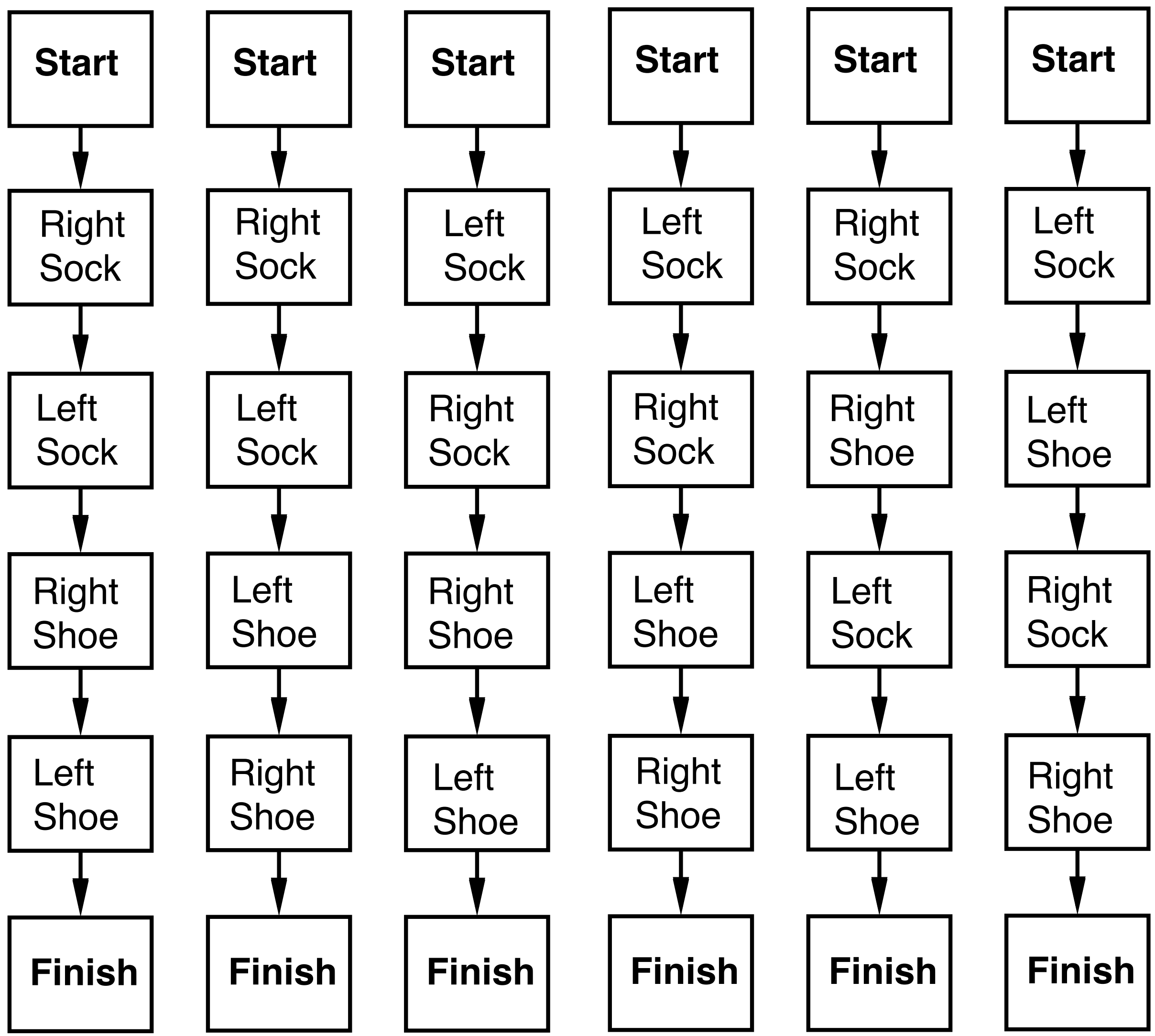
- The answer to the problem is to use partial order planning.
- Basically this gives us a way of checking before adding an action to the plan that it doesn't mess up the rest of the plan.
- The problem is that in this recursive process, we don't know what the rest of the plan is.
- Need a new representation *partially ordered plans*.

Representation

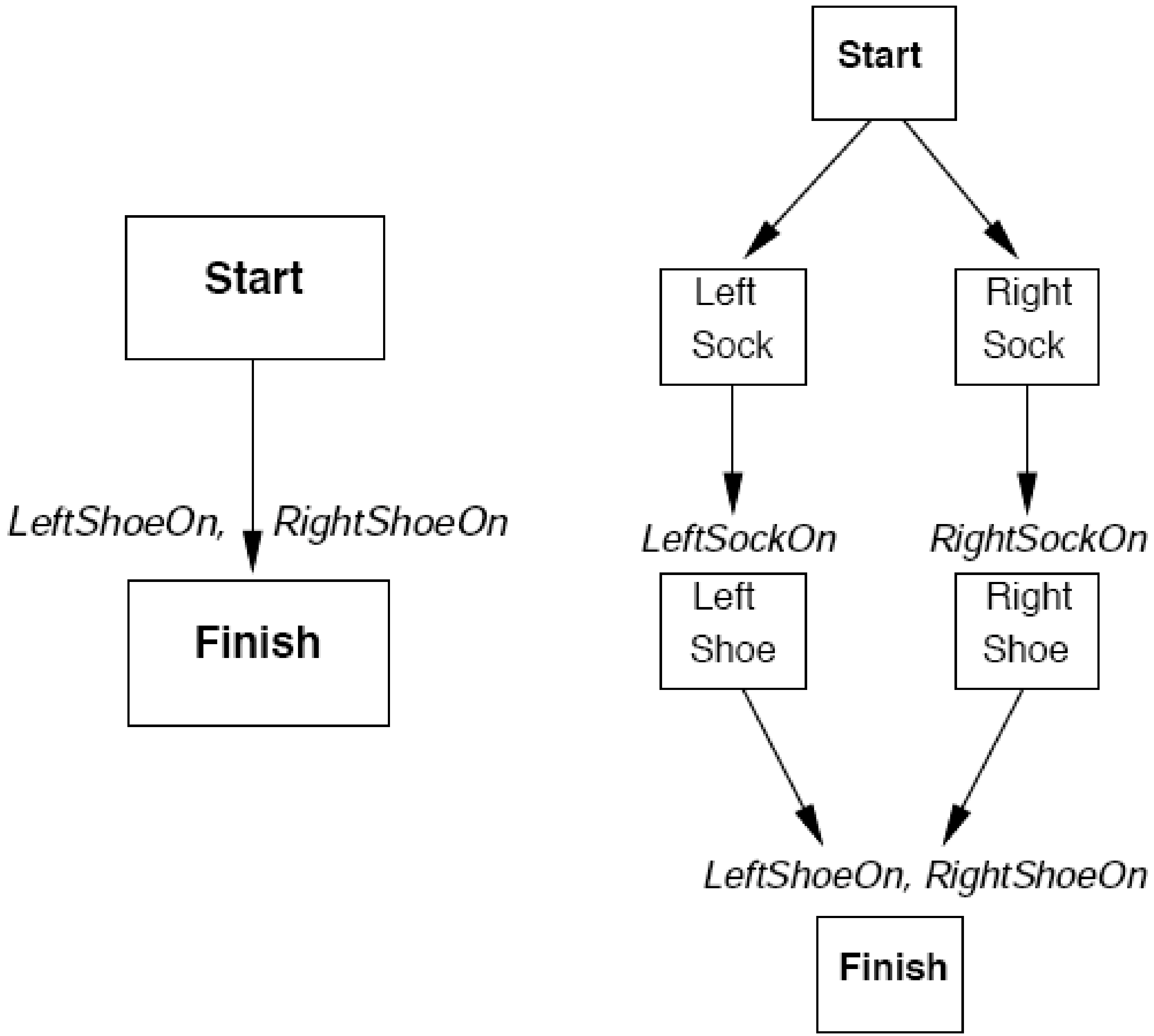
Partial-Order Plan:



Total-Order Plans:



Representation

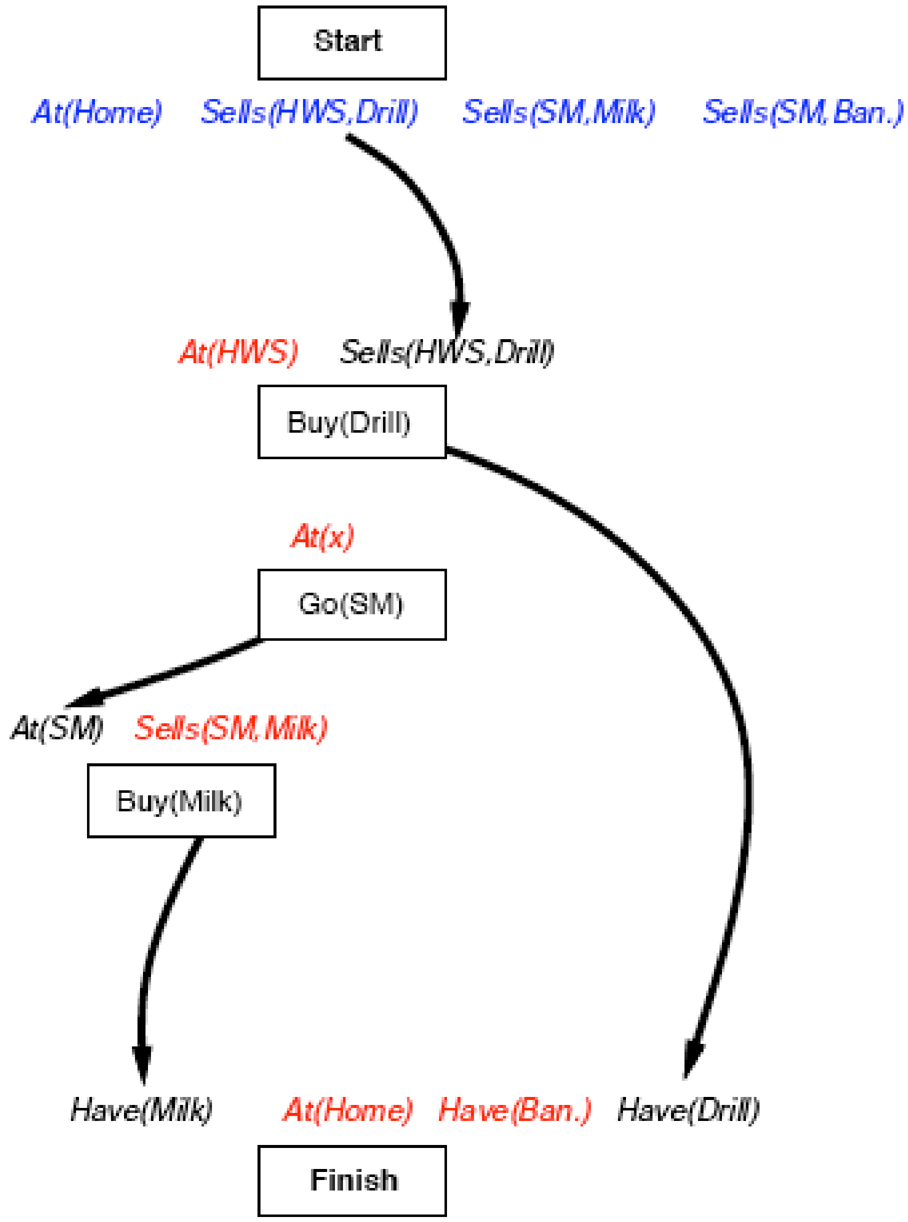


Partially ordered plans

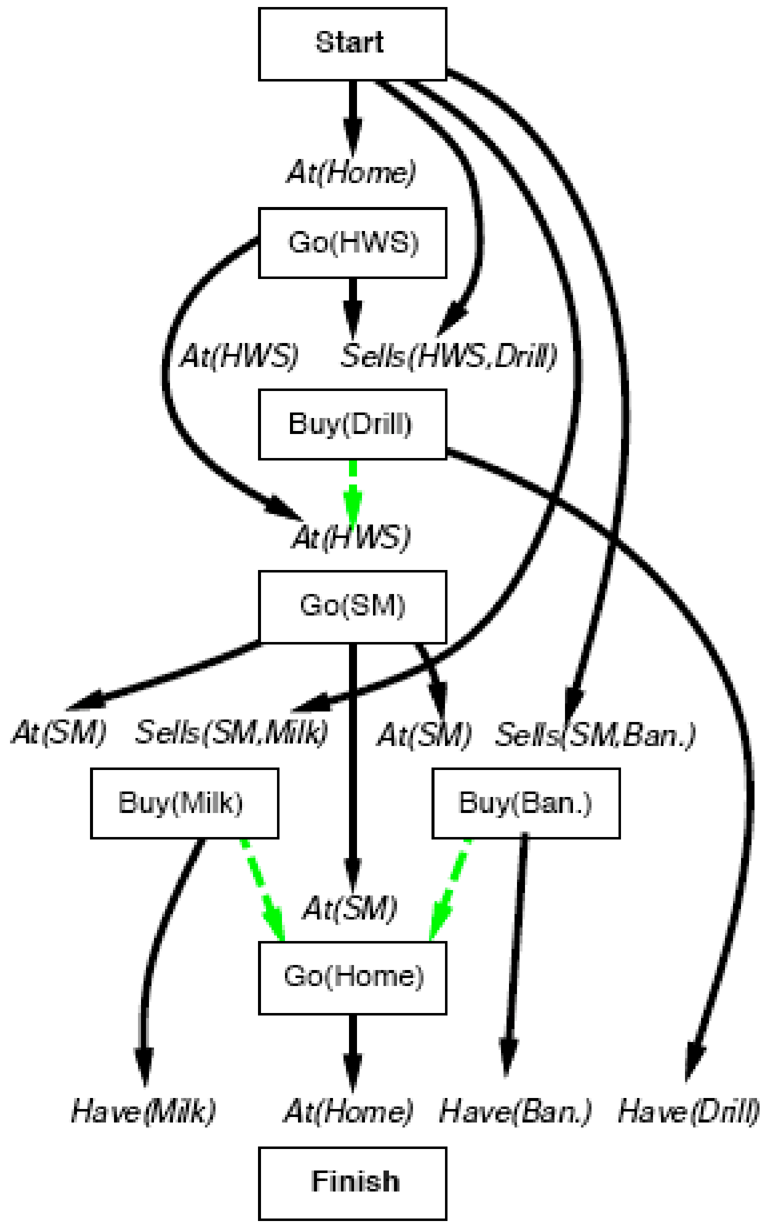
- *Partially ordered* collection of steps with
  - *Start* step has the initial state description as its effect
  - *Finish* step has the goal description as its precondition
  - *causal links* from outcome of one step to precondition of another
  - *temporal ordering* between pairs of steps
- *Open condition* = precondition of a step not yet causally linked
- A plan is *complete* iff every precondition is achieved
- A precondition is *achieved* iff it is the effect of an earlier step and no *possibly intervening* step undoes it

Plan Construction

Plan construction (2)



Plan construction (3)



Planning process

- Operators on partial plans:
  - *add a link* from an existing action to an open condition
  - *add a step* to fulfill an open condition
  - *order* one step wrt another to remove possible conflicts
- Gradually move from incomplete/vague plans to complete, correct plans
- Backtrack if an open condition is unachievable or if a conflict is unresolvable

POP algorithm

function POP (*initial, goal, operators*) returns *plan*
    plan $\leftarrow$ MAKE-MINIMAL-PLAN(*initial,goal*)
    loop do
        if SOLUTION?(*plan*) then return *plan*
        $S_{need}, c \leftarrow$ SELECT-SUBGOAL(*plan*)
        CHOOSE-OPERATOR(*plan,operators,*S_{need}, c)
        RESOLVE-THREATS(*plan*)
    end loop
end function

function SELECT-SUBGOAL(*plan*) returns S_{need}, c
    pick a plan step S_{need} from STEPS(*plan*)
        with a precondition c that has not been achieved
    return S_{need}, c
end function

POP algorithm, continued

procedure CHOOSE-OPERATOR(*plan*,*operators*,S_{need}, c)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect

if there is no such step then fail   add the causal link $S_{add} \xleftarrow{c} S_{need}$ to LINKS(*plan*)

add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)

if S_{add} is a newly added step from *operators* then

add S_{add} to STEPS(*plan*)

add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

end if

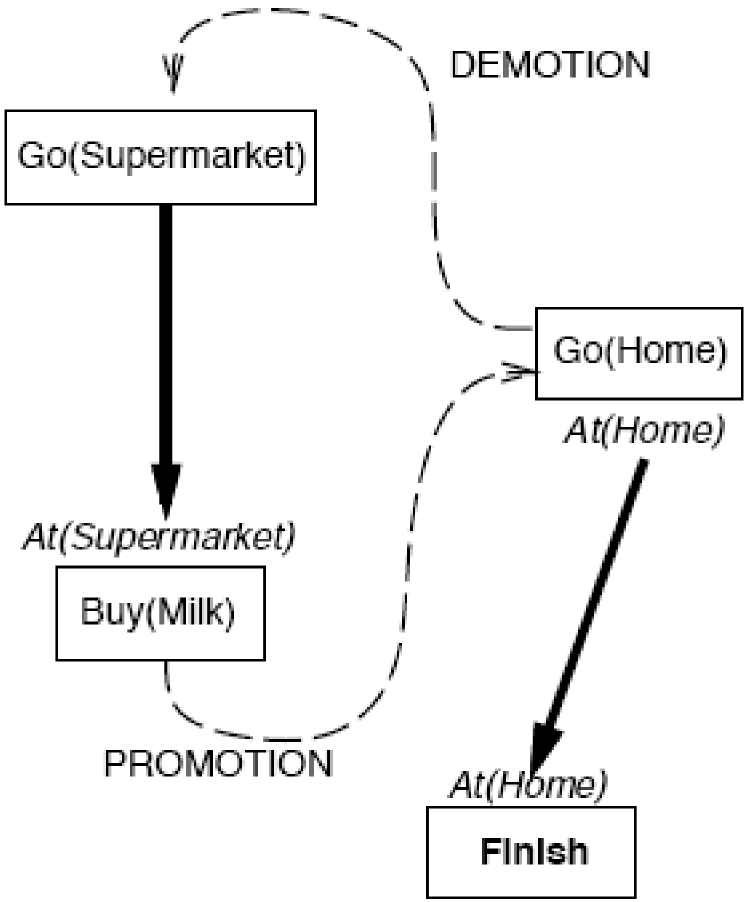
end procedure

POP algorithm, continued

procedure RESOLVE-THREATS(*plan*)
  for each S_{threat} that threatens a link $S_i \leftarrow^c S_j$ in LINKS(*plan*) do
    choose either
      Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
      Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
    if not CONSISTENT(*plan*) then fail
  end for each
end procedure

Clobbering

- A *clobberer* is a potentially intervening step that destroys the condition achieved by a causal link. E.g., *Go(Home)* clobbers *At(Supermarket)*:



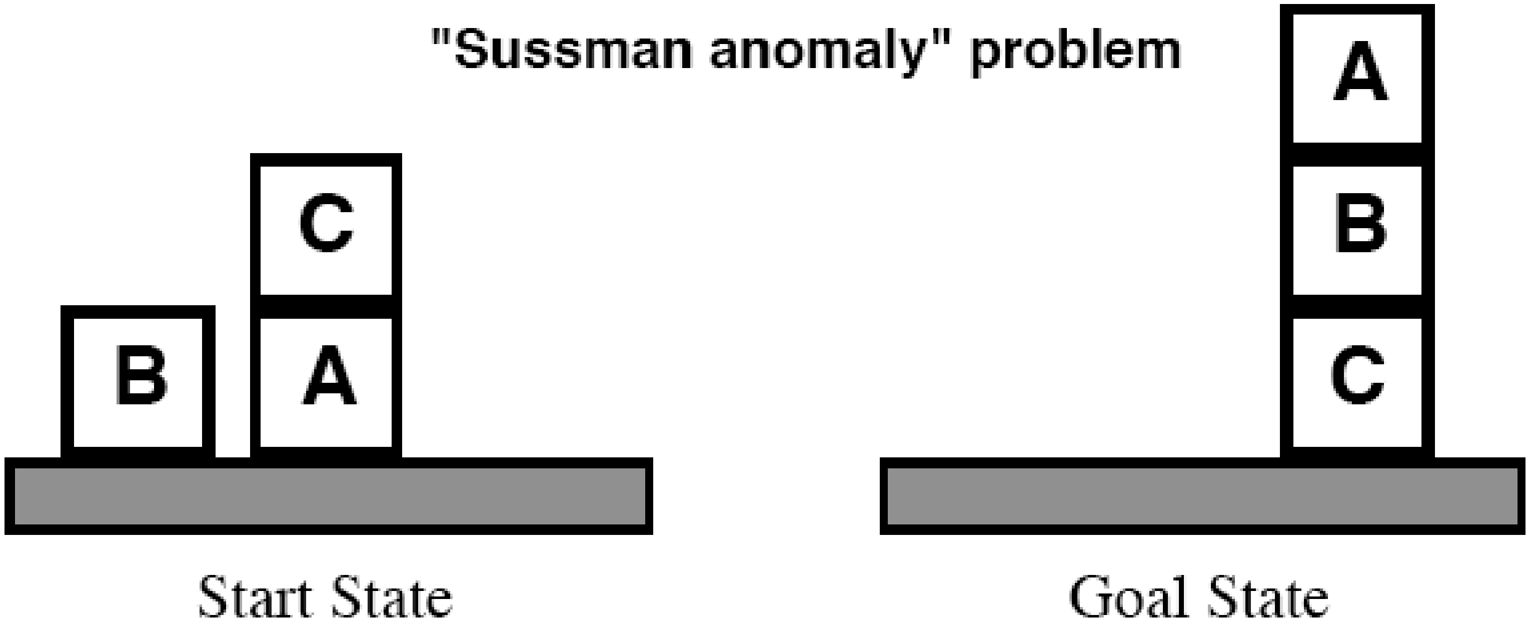
Demotion: put before *Go(Supermarket)*

Promotion: put after *Buy(Milk)*

Properties of POP

- Nondeterministic algorithm: backtracks at *choice* points on failure:
  - choice of S_{add} to achieve S_{need}
  - choice of demotion or promotion for clobberer
  - selection of S_{need} is irrevocable
- POP is sound, complete, and *systematic* (no repetition)
- Extensions for disjunction, universals, negation, conditionals
- Can be made efficient with good heuristics derived from problem description
- Particularly good for problems with many loosely related subgoals

Example



Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

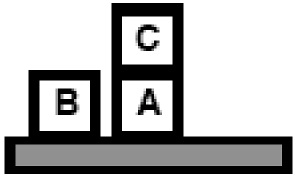
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example (2)

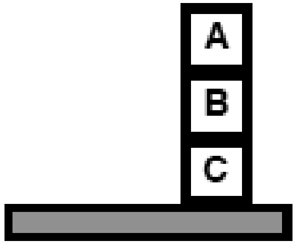
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

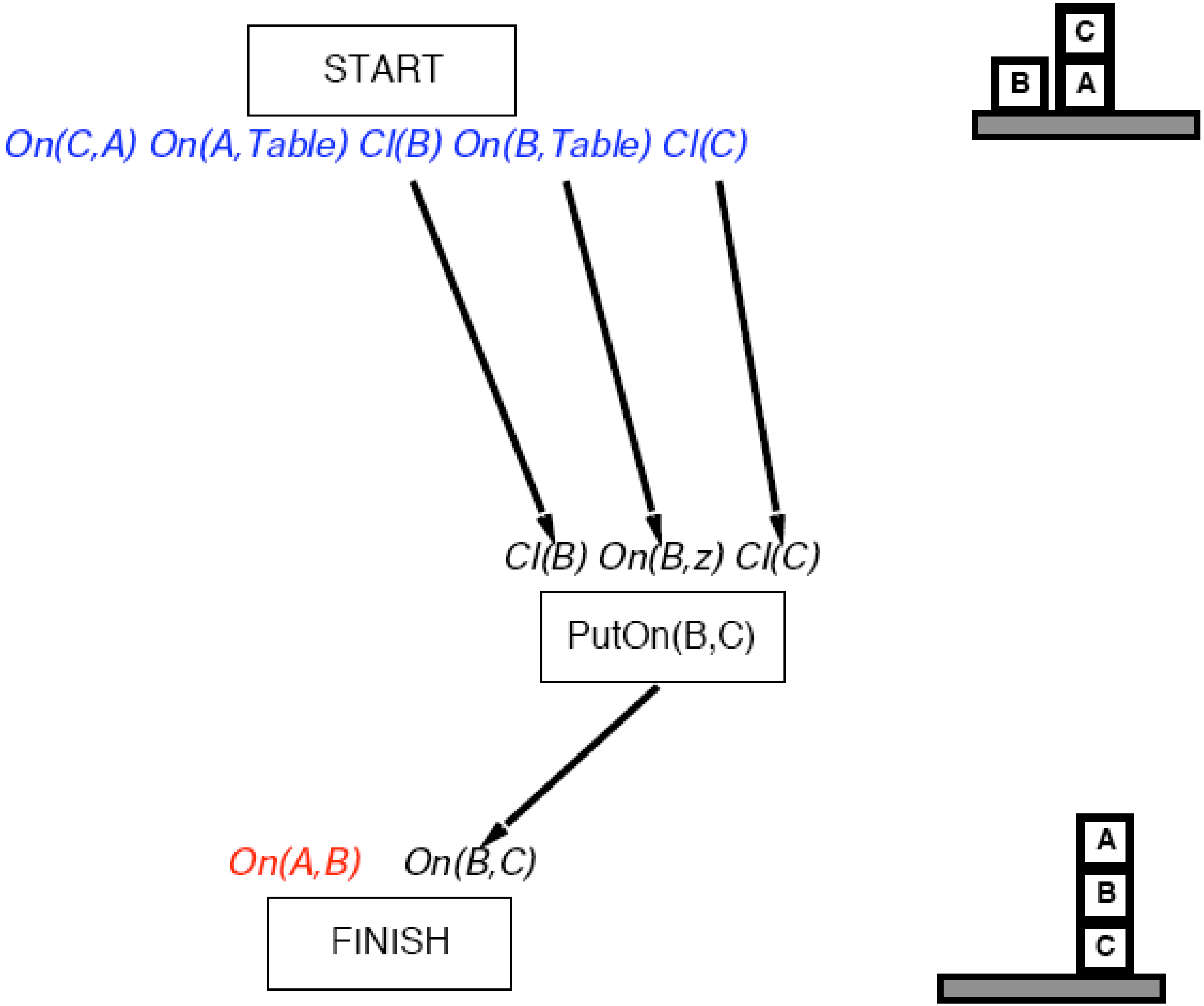


On(A,B) *On(B,C)*

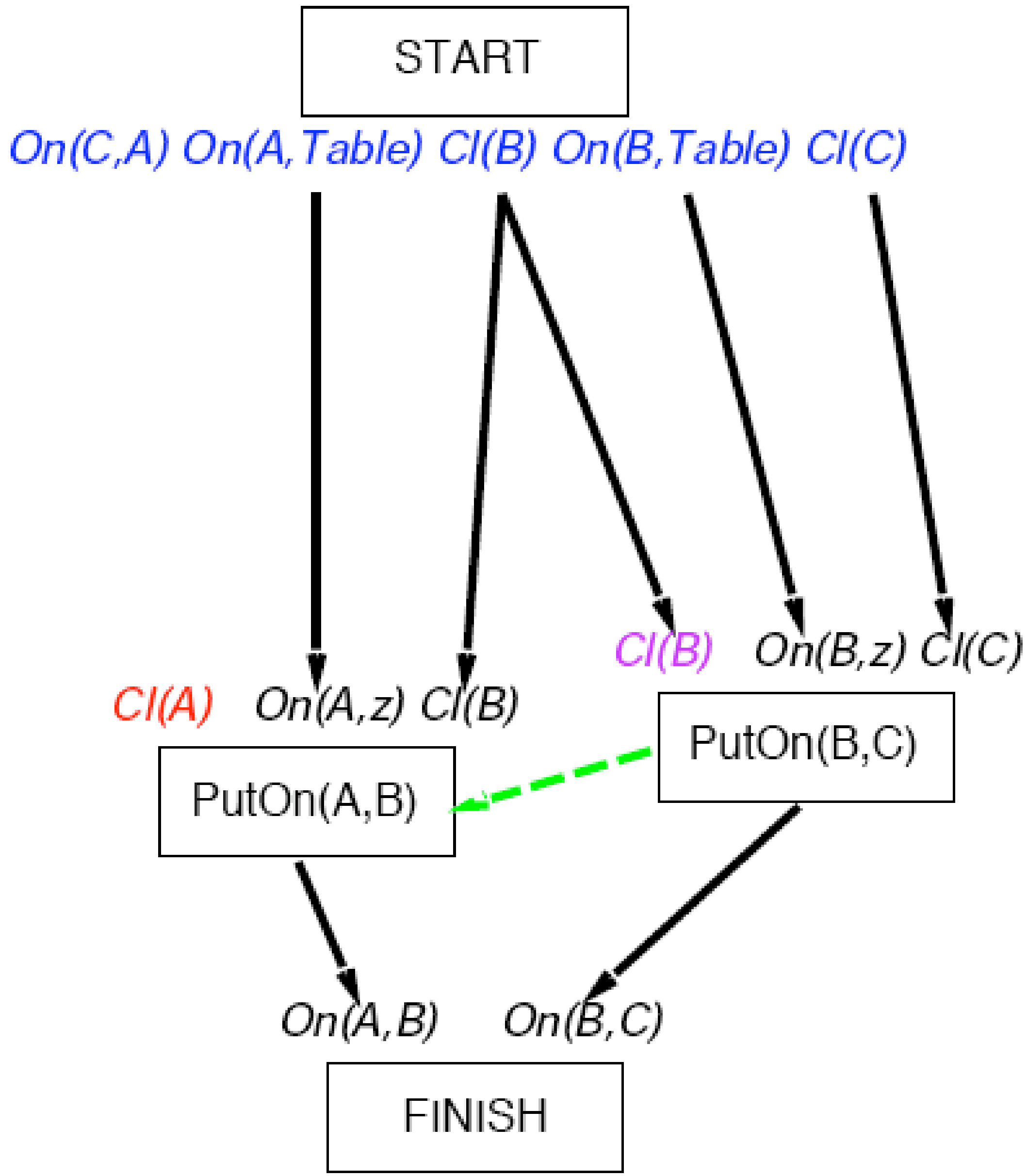
FINISH

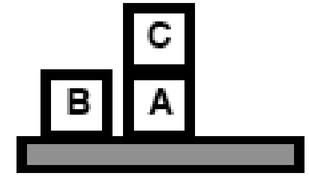


Example (3)

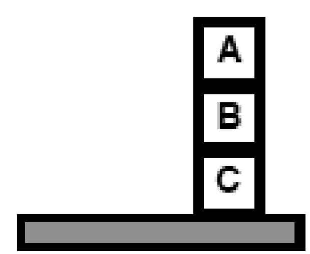

START
On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)
C
B
A
Cl(B) On(B,z) Cl(C)
PutOn(B,C)
On(A,B)
On(B,C)
FINISH
A
B
C

Example (4)

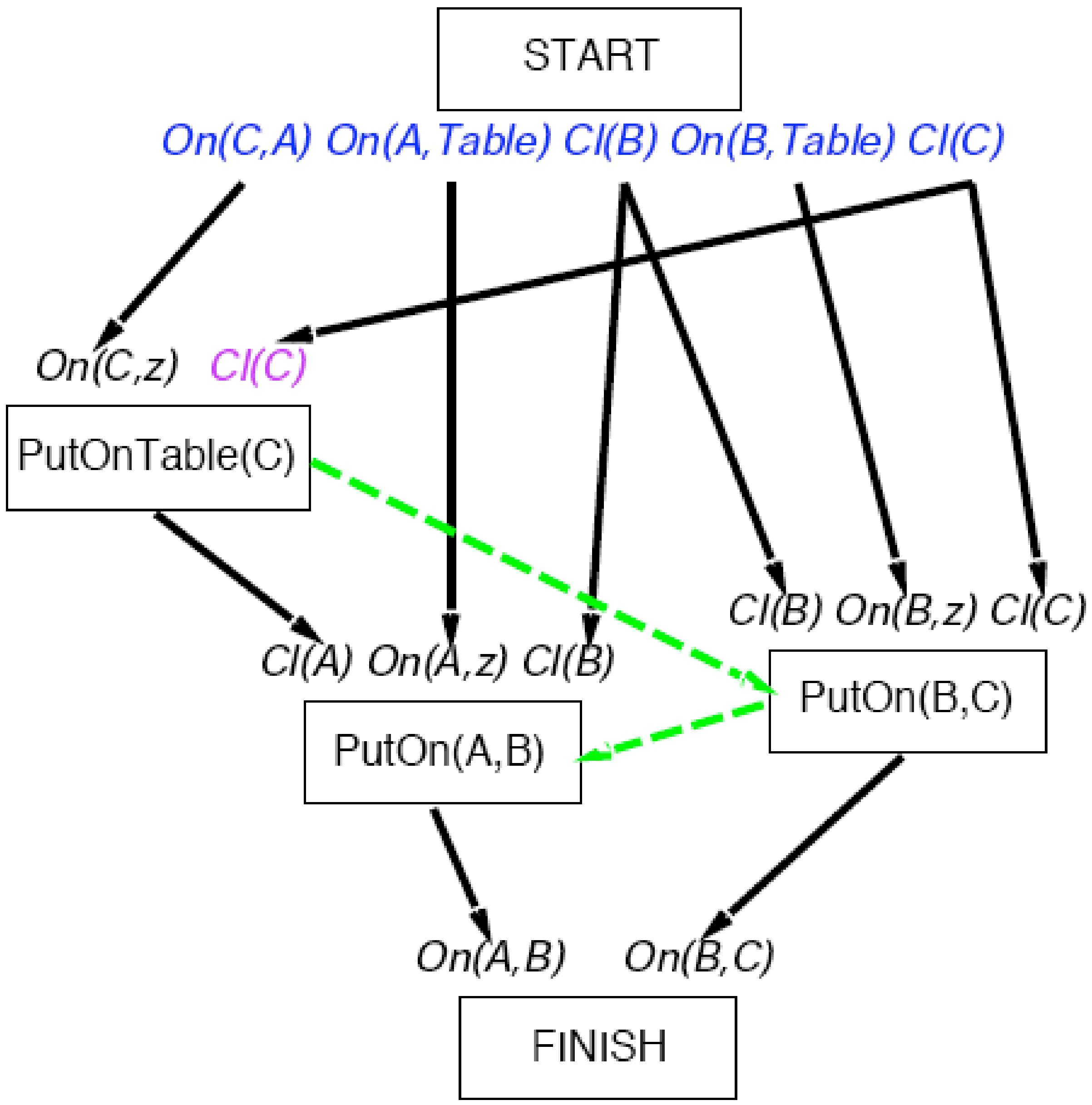


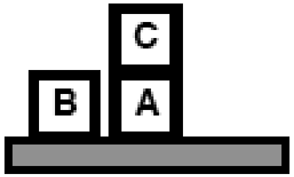


PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)



Example (5)





PutOn(A,B)
clobbers Cl(B)
=> order after
PutOn(B,C)

PutOn(B,C)
clobbers Cl(C)
=> order after
PutOnTable(C)