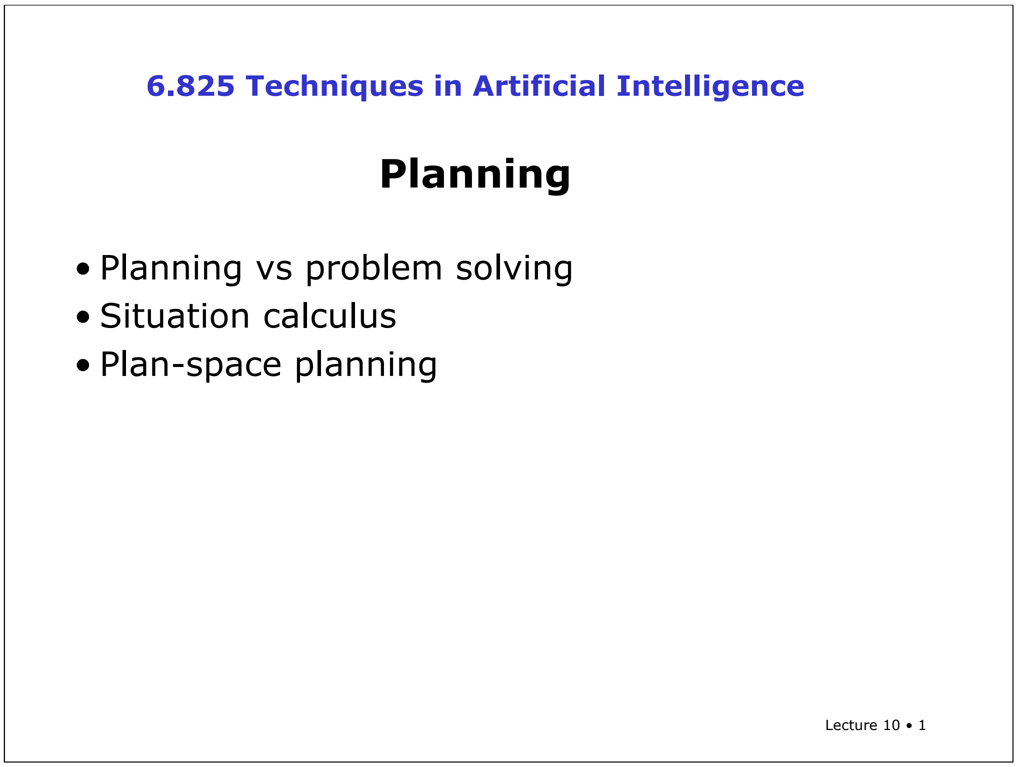


6.825 Techniques in Artificial Intelligence

# Planning

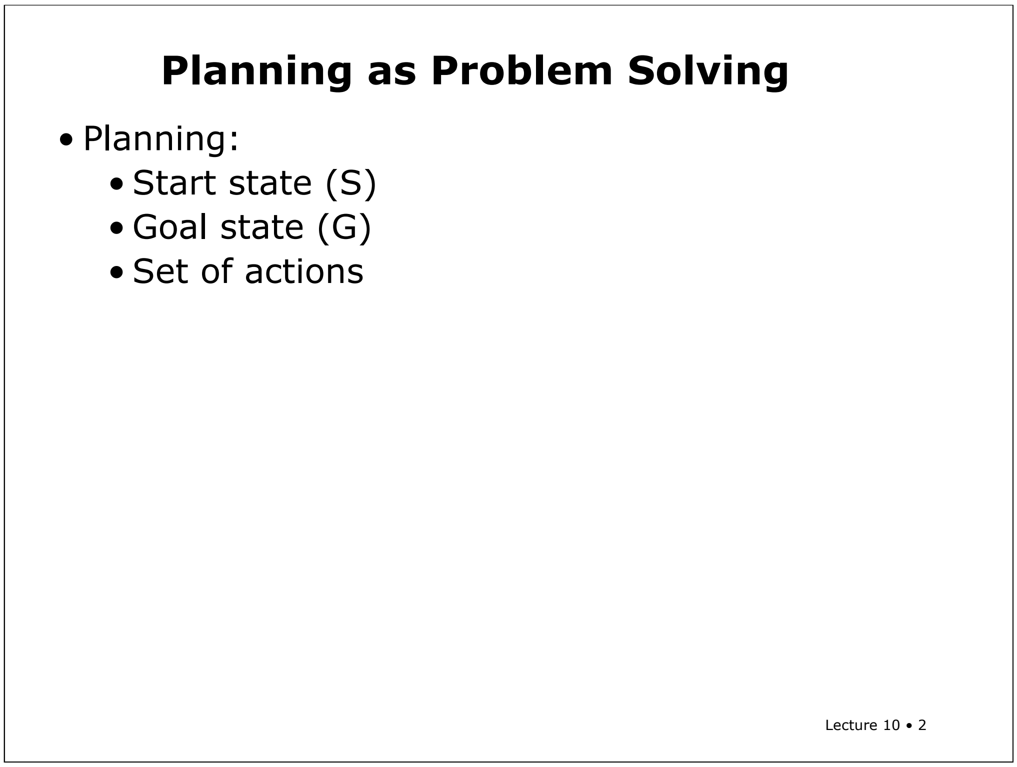
- Planning vs problem solving
- Situation calculus
- Plan-space planning

Lecture 10 • 1

We are going to switch gears a little bit now. In the first section of the class, we talked about problem solving, and search in general, then we did logical representations. The motivation that I gave for doing the problem solving stuff was that you might have an agent that is trying to figure out what to do in the world, and problem solving would be a way to do that.

And then we motivated logic by trying to do problem solving in a little bit more general way, but then we kind of really digressed off into the logic stuff, and so now what I want to do is go back to a section on planning, which will be the next four lectures.

Now that we know something about search and something about logic, we can talk about how an agent really could figure out how to behave in the world. This will all be in the deterministic case. We're still going to assume that when you take an action in the world, there's only a single possible outcome. After this, we'll back off on that assumption and spend most of the rest of the term thinking about what happens when the deterministic assumption goes away.

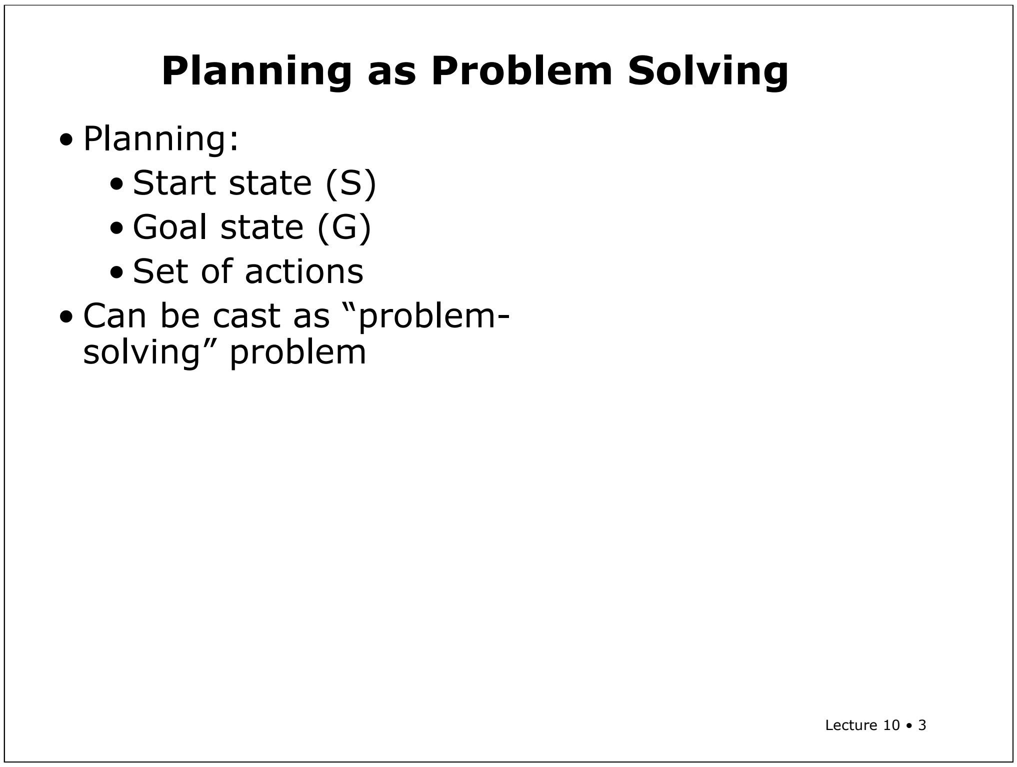


## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions

Lecture 10 • 2

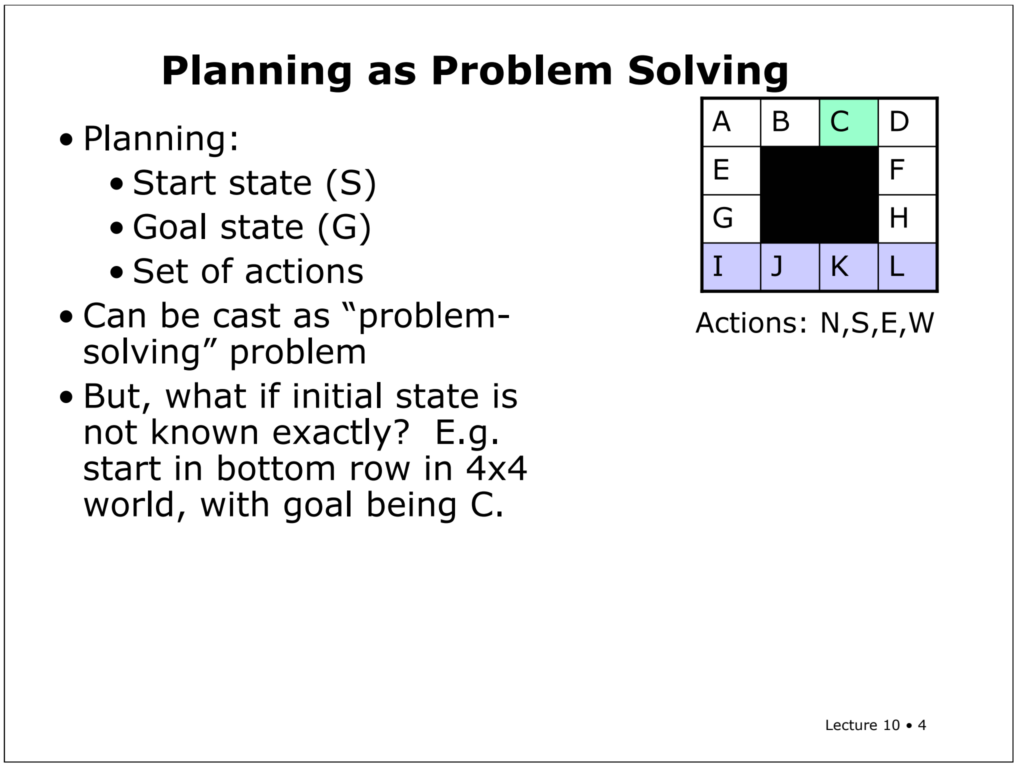
In planning, the idea is that you're given some description of a starting state or states; a goal state or states; and some set of possible actions that the agent can take. And you want to find the sequence of actions that get you from the start state to the goal state.



## Planning as Problem Solving

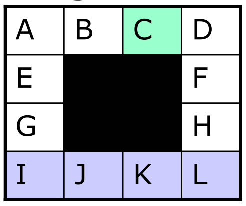
- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as "problem-solving" problem

It's pretty clear that you can cast this as a problem solving problem. Remember when we talked about problem solving, we were given a start state, and we searched through a tree that was the sequences of actions that you could take, and we tried to find a nice short plan. So, planning problems can certainly be viewed as problem-solving problems, but it may not be the best view to take.



## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as “problem-solving” problem
- But, what if initial state is not known exactly? E.g. start in bottom row in 4x4 world, with goal being C.

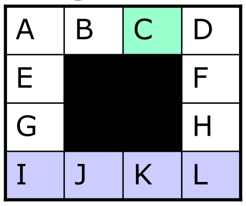


Actions: N,S,E,W

Just for fun, let's think about moving around on the squares of this grid. The goal is to get to state C, but all we know initially is that the agent is in the bottom row, one of states I, J, K, and L.

## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as “problem-solving” problem
- But, what if initial state is not known exactly? E.g. start in bottom row in 4x4 world, with goal being C.
- Do search over “sets” of underlying states to find a set of actions that will reach C for any starting state.

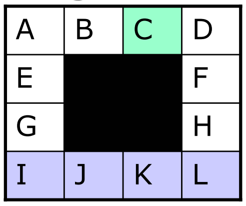


Actions: N,S,E,W

We could try to solve this using our usual problem solving search, but where our nodes would now represent sets of states, rather than single states. So, how would that go?

## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as “problem-solving” problem
- But, what if initial state is not known exactly? E.g. start in bottom row in 4x4 world, with goal being C.
- Do search over “sets” of underlying states to find a set of actions that will reach C for any starting state.



Actions: N,S,E,W

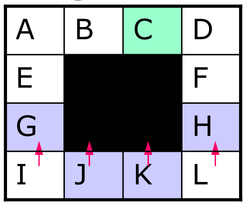
$$\begin{tikzcd} & \{I, J, K, L\} \arrow[dl, "N"'] \arrow[dr, "E"] & \\ \{G, J, K, H\} & & \{J, K, L\} \end{tikzcd}$$

The agent starts in the node corresponding to states I,J,K,L, because it doesn’t know exactly where it is.

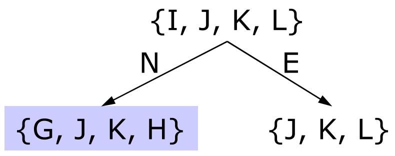
Now, let’s say that it can move North, South, East, or West, but if it runs into an obstacle or the edge of the world, it just stays where it was.

## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as "problem-solving" problem
- But, what if initial state is not known exactly? E.g. start in bottom row in 4x4 world, with goal being C.
- Do search over "sets" of underlying states.



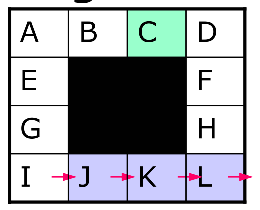
Actions: N,S,E,W



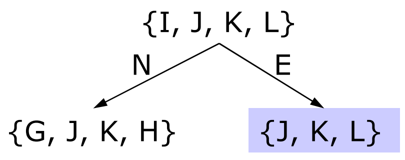
So, if it moves north, it could end up in states G, J, K, or H, so that gives us this result node.

## Planning as Problem Solving

- Planning:
  - Start state (S)
  - Goal state (G)
  - Set of actions
- Can be cast as "problem-solving" problem
- But, what if initial state is not known exactly? E.g. start in bottom row in 4x4 world, with goal being C.
- Do search over "sets" of underlying (atomic) states.



Actions: N,S,E,W



And if it moves East, it could end up in J, K, or L.

You can see how this process would continue. And probably the shortest plan is going to be to go east three times (so we could only possibly be in state L) and then continue north until we get to the top, and then go west to the goal.

## Planning as Logic

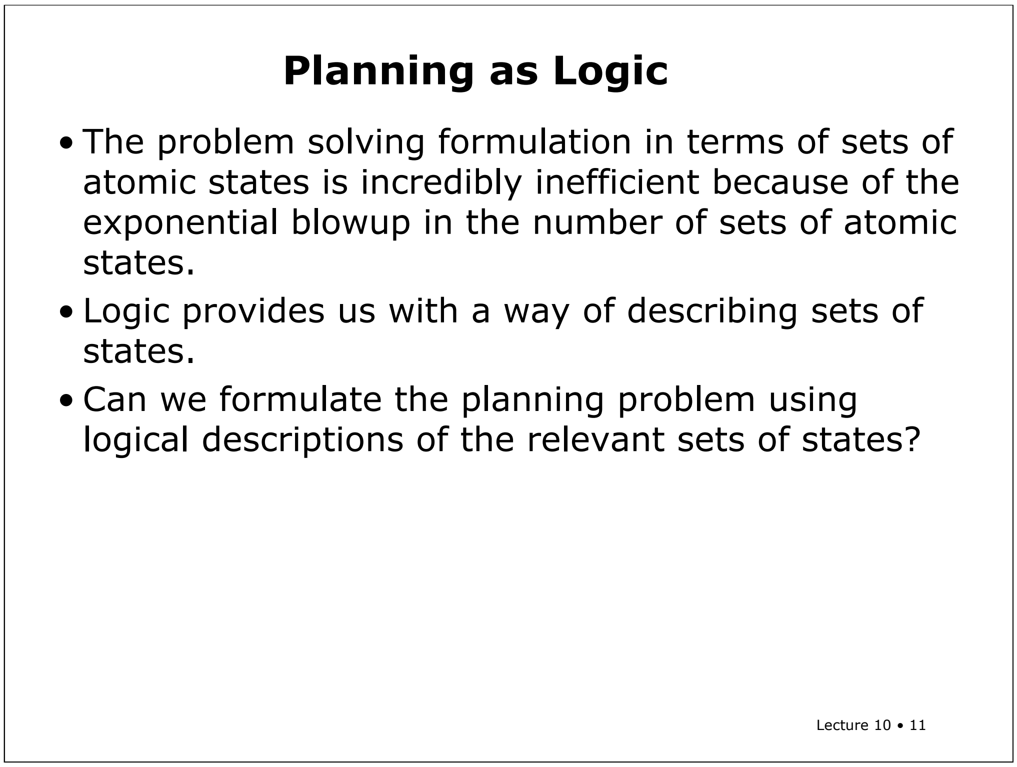
- The problem solving formulation in terms of sets of atomic states is incredibly inefficient because of the exponential blowup in the number of sets of atomic states.

So, we do have a way to think about planning in the case where you're not absolutely sure what your starting state is. But, it's really inefficient in any kind of a big domain because we're planning with sets of states at the nodes. Just to figure out how the state transition function works on sets of primitive states, we're having to go through each little atomic state in our set of states, and think about how it transitions to some other atomic state. And that could be really desperately inefficient.

## Planning as Logic

- The problem solving formulation in terms of sets of atomic states is incredibly inefficient because of the exponential blowup in the number of sets of atomic states.
- Logic provides us with a way of describing sets of states.

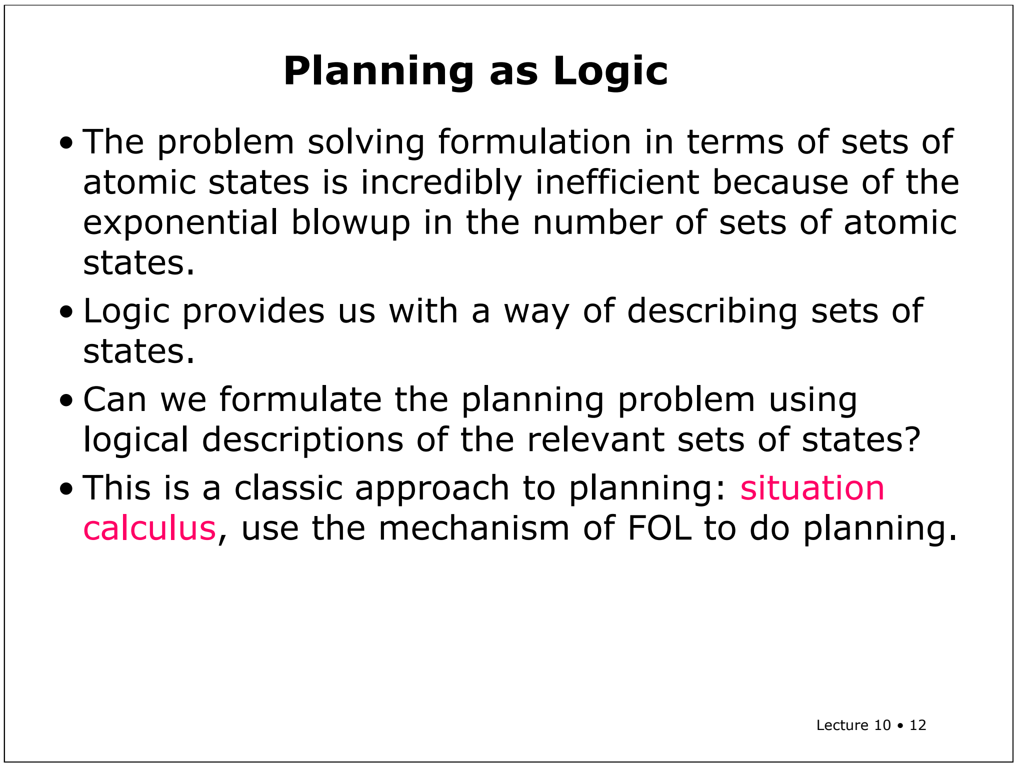
One of our arguments for moving to logical representations was to be able to have a compact way of describing a set of states. You should be able to describe that set of states IJKL by saying, the proposition "bottom row" is true and you might be able to say, well, if "bottom row", and no obstacles above me, then when I move, then I'll be in "next to bottom row", or something like that.



## Planning as Logic

- The problem solving formulation in terms of sets of atomic states is incredibly inefficient because of the exponential blowup in the number of sets of atomic states.
- Logic provides us with a way of describing sets of states.
- Can we formulate the planning problem using logical descriptions of the relevant sets of states?

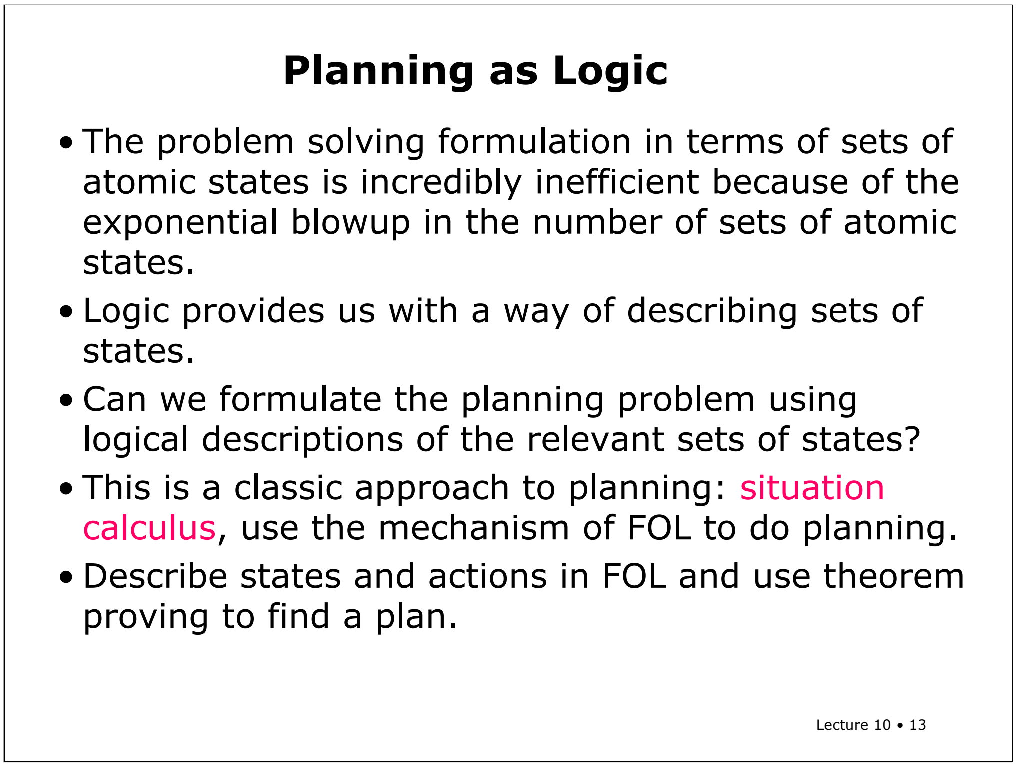
That might mean that if you can set up the logical description of your world in the right way, you can talk about these sets of states, not by enumerating them, but by giving logical descriptions of those sets of states. And the idea is that a logical formula stands for all the interpretations, or world situations, in which it is true. It stands for all the locations that the agent could be in that would make the formula true.



## Planning as Logic

- The problem solving formulation in terms of sets of atomic states is incredibly inefficient because of the exponential blowup in the number of sets of atomic states.
- Logic provides us with a way of describing sets of states.
- Can we formulate the planning problem using logical descriptions of the relevant sets of states?
- This is a classic approach to planning: situation calculus, use the mechanism of FOL to do planning.

So, this leads us to the first approach to planning. It's going to turn out not to be particularly practical, but it's worth mentioning because it's a kind of a classical idea, and you can get somewhere with it. It’s called **situation calculus**, and the idea is to use first-order logic, exactly as you know about it, to do planning.



## Planning as Logic

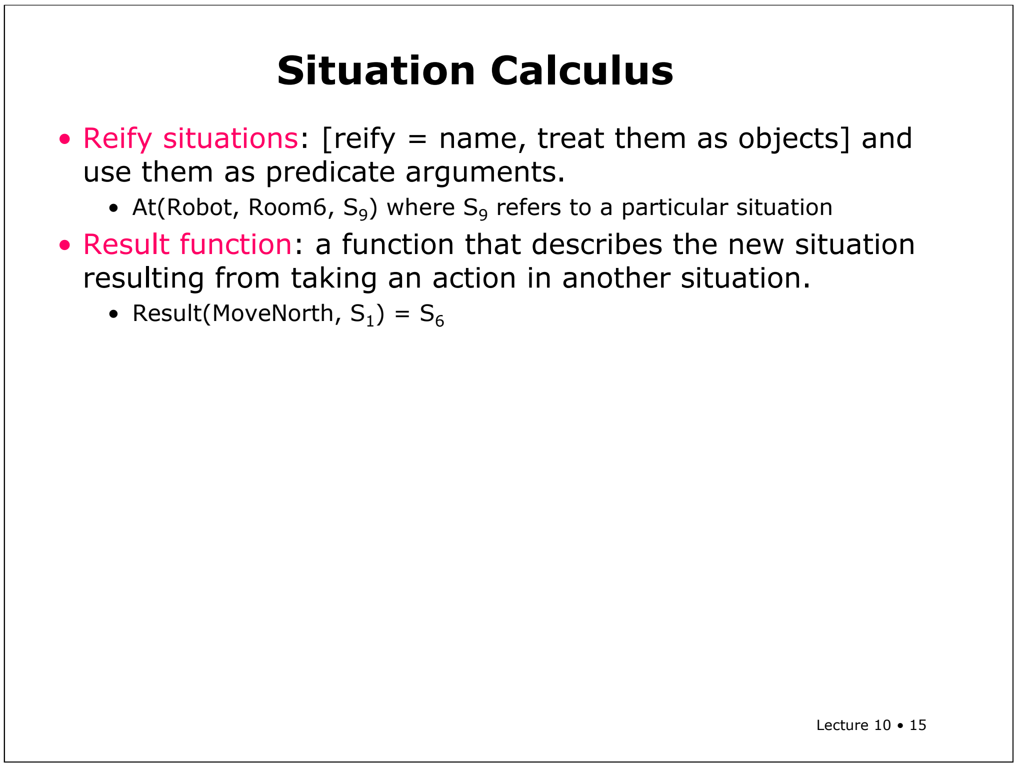
- The problem solving formulation in terms of sets of atomic states is incredibly inefficient because of the exponential blowup in the number of sets of atomic states.
- Logic provides us with a way of describing sets of states.
- Can we formulate the planning problem using logical descriptions of the relevant sets of states?
- This is a classic approach to planning: situation calculus, use the mechanism of FOL to do planning.
- Describe states and actions in FOL and use theorem proving to find a plan.

We have to figure out a way to talk about states of the world, and start states and goal states and actions in the language of first order logic, and then somehow use the mechanism of theorem proving to find a plan. That's the idea. It's appealing, right? It's always appealing, given a very general solution mechanism, if you can take a particular problem, and see how it's an instance of that general solution; then you can apply your general solution to the particular problem and get an answer out without inventing any more solution methods.

## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.

In situation calculus, the main idea is that we're going to "reify" situations. To reify something is to treat it as if it’s an object. So, we’re going to have variables and constants in our logical language that will range over the possible situations, or states of the world. This will allow us to say that world states have certain properties, but to avoid enumerating them all.

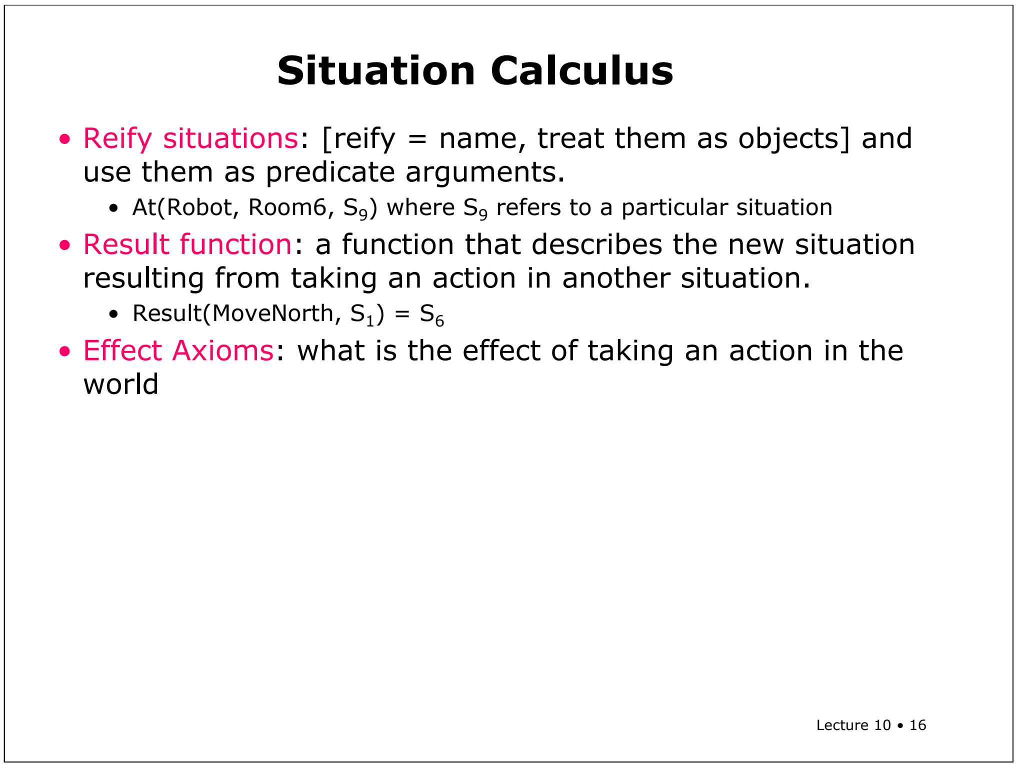


## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$

So we can say things like "at robot Room Six" in "situations 9". More ordinarily, we would have said at(robot,room6), implicitly saying something about the current state of the world. But in planning, we need to be thinking about a bunch of different situations at the same time, and so we have to be able to name them.

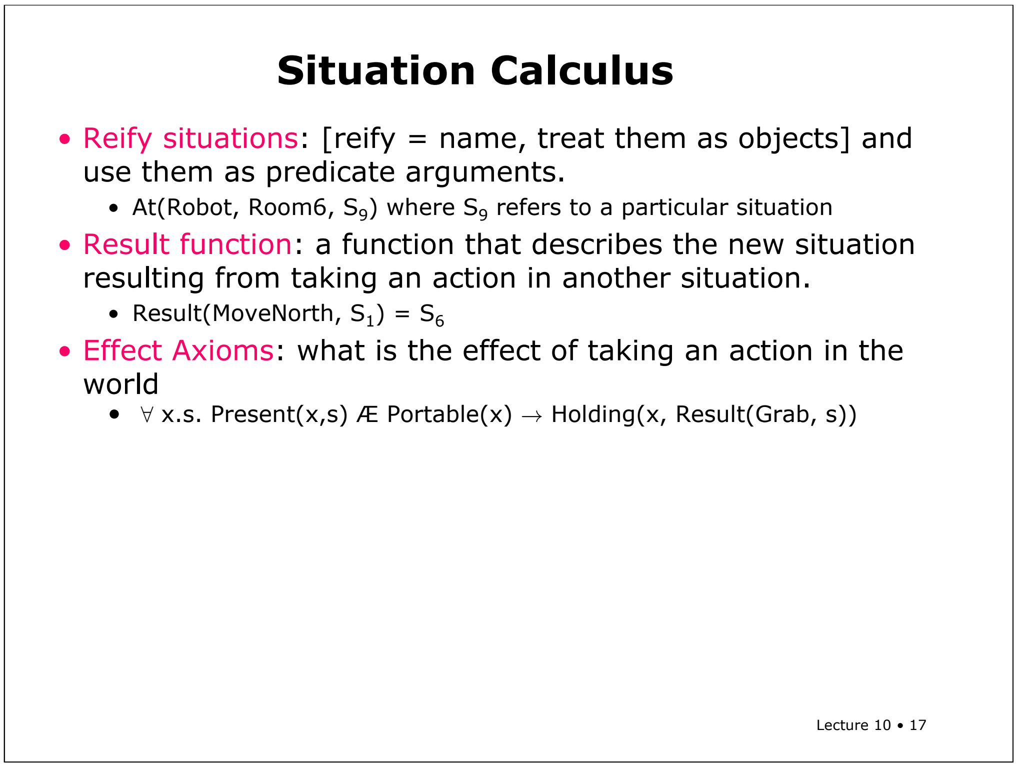
For any proposition that can change its truth value in different situations, we'll add an extra argument, which is the situation in which we're asserting that the proposition holds. Then we can talk about how the world changes over time, by having the predicate or function "result". We can say the result of doing action "move north" in situation one is situation six.



## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world

Now we need to write down what we know about S6 because of the fact that we got it by moving north from S1. So, what do we know about S6? In the textbook, Russell and Norvig use the wumpus world to illustrate situation calculus. We can describe the dynamics of the world using **effect axioms**. They say what the effects in the world are of taking particular actions.

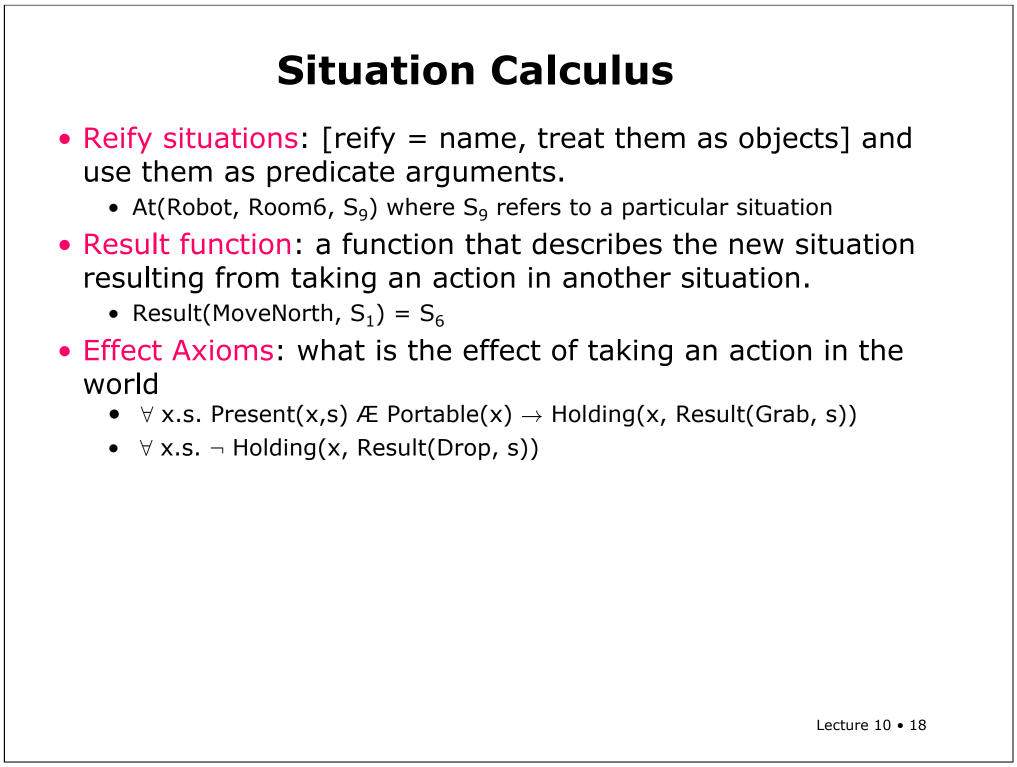


## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world
  - ∀ x.s. Present(x,s) Æ Portable(x) → Holding(x, Result(Grab, s))

So, we might say that, for all objects x and situations s, if object x is present in situation s, and if object s is portable, then in the situation that results from executing the grab action in situation s, the agent is holding the object x.

Just to be clear, Grab is an action. So Result(Grab, s) is a term that names the situation that results if the agent does a grab action in situation s. And what we're saying in the consequent here is that the agent will be holding the object x in that resulting situation.



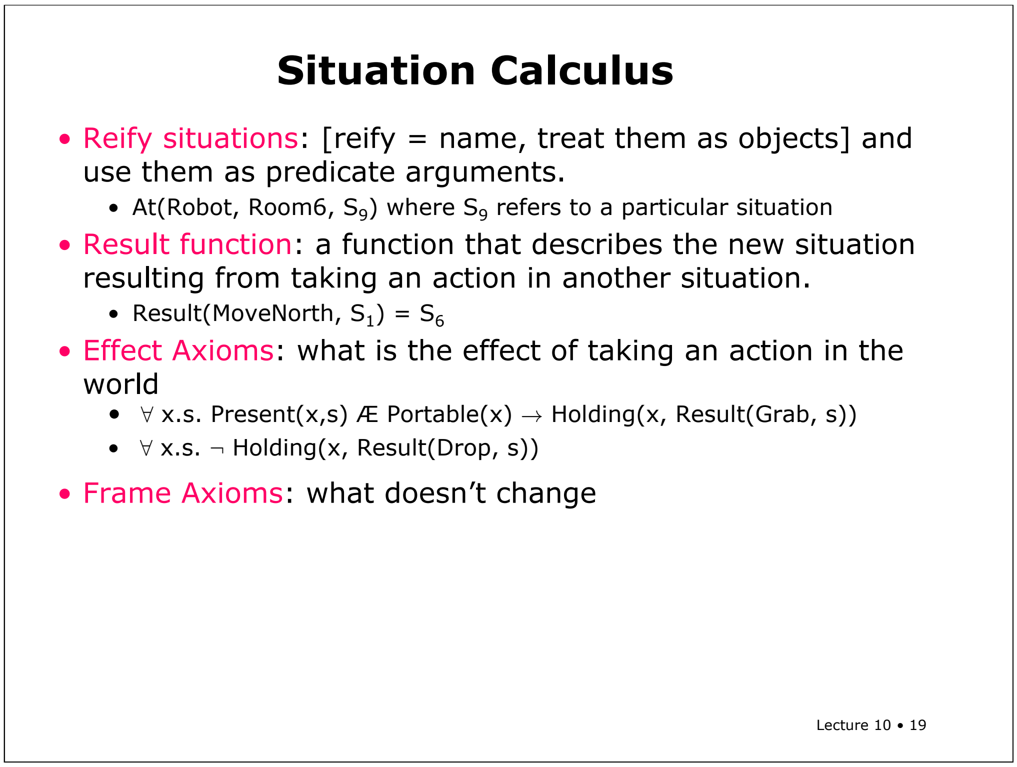
## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world
  - ∀ x.s. Present(x,s) Æ Portable(x) → Holding(x, Result(Grab, s))
  - ∀ x.s. ¬ Holding(x, Result(Drop, s))

Lecture 10 • 18

Or, we might say that, for all objects x and situations s, that the agent is not holding object x in a situation that results from doing the drop action in situation s.

You can see the power of the logical representation in these axioms. We're able to say things about a very large, or possibly infinite set of situations, without enumerating them.



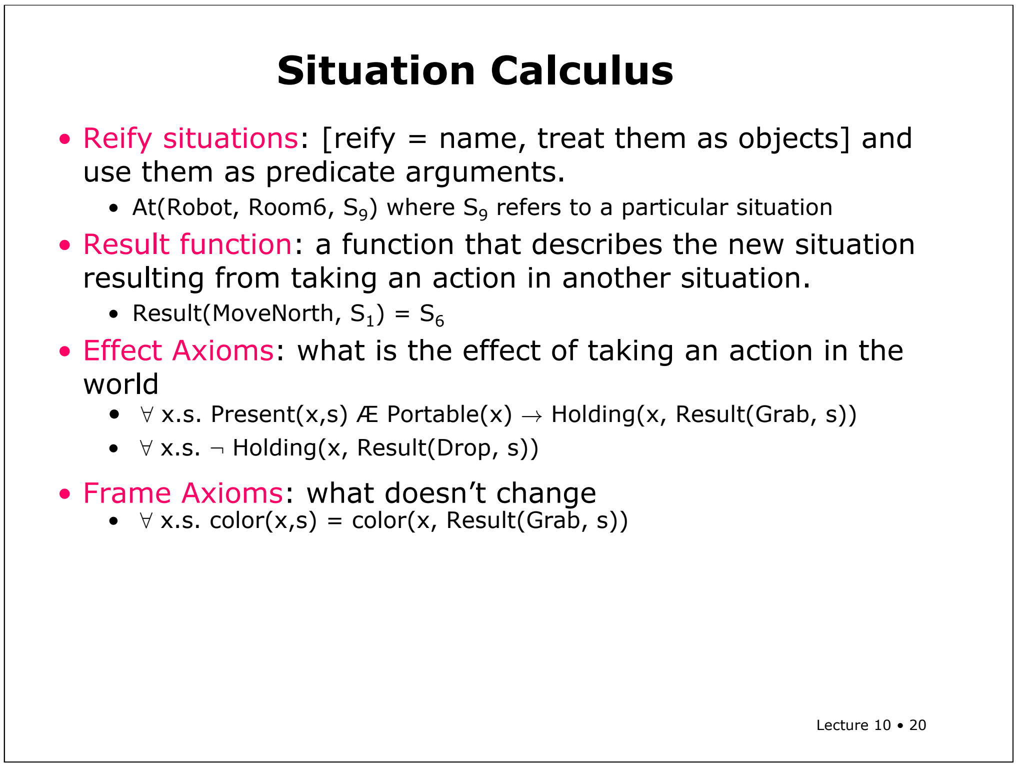
## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world
  - $\forall$ x.s. Present(x,s) Æ Portable(x) $\rightarrow$ Holding(x, Result(Grab, s))
  - $\forall$ x.s. $\neg$ Holding(x, Result(Drop, s))
- Frame Axioms: what doesn't change

Lecture 10 • 19

It's not enough to just specify these positive effects of taking actions. You also have to describe what doesn't change. The problem of having to specify all the things that don't change is called the "frame problem." I think the name comes from the frames in movies. Animators know that most of the time, from frame to frame, things don't change.

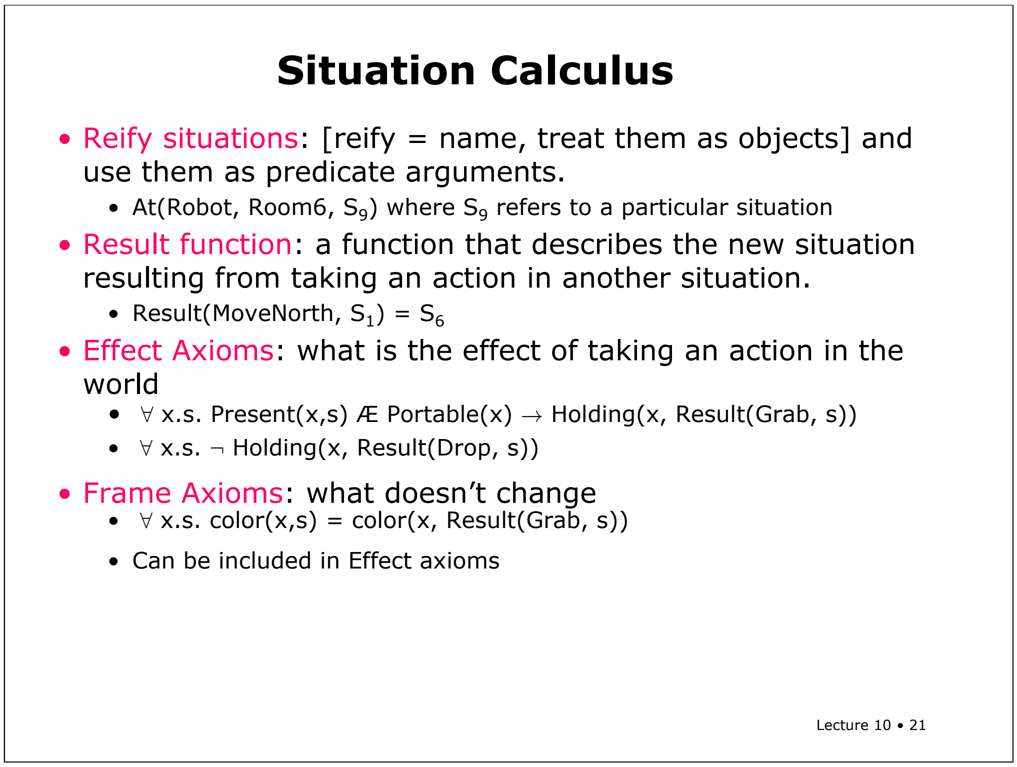
And when we write down a theory we're saying, well, if there's something here, and you do a grab, then you're going to be holding the thing. But, we haven't said, for instance, what happens to the colors of the objects, or whether your shoes are tied, or the weather, as a result of doing the grab action. But, you have to actually say that. At least, in the situation calculus formulation, you have to say, "and nothing else changes" somehow.



## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world
  - ∀ x.s. Present(x,s) Æ Portable(x) → Holding(x, Result(Grab, s))
  - ∀ x.s. ¬ Holding(x, Result(Drop, s))
- Frame Axioms: what doesn't change
  - ∀ x.s. color(x,s) = color(x, Result(Grab, s))

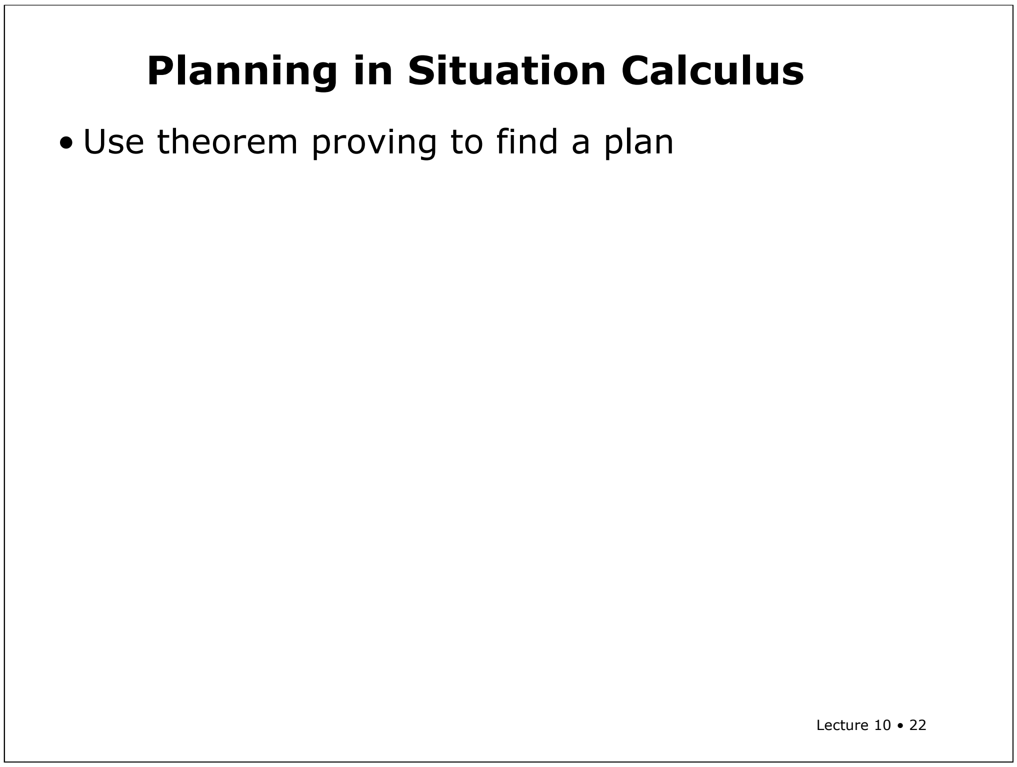
One way to do this, is to write frame axioms, where you say things like, "for all objects and situations, the color of X in situation S is the same as the color of X in the situation that is the result of doing grab. That is to say, picking things up doesn't change their color. Picking up other things doesn't change their color. But it can be awfully tedious to have to write all these out.



## Situation Calculus

- Reify situations: [reify = name, treat them as objects] and use them as predicate arguments.
  - At(Robot, Room6, $S_9$) where $S_9$ refers to a particular situation
- Result function: a function that describes the new situation resulting from taking an action in another situation.
  - Result(MoveNorth, $S_1$) = $S_6$
- Effect Axioms: what is the effect of taking an action in the world
  - ∀ x.s. Present(x,s) Æ Portable(x) → Holding(x, Result(Grab, s))
  - ∀ x.s. ¬ Holding(x, Result(Drop, s))
- Frame Axioms: what doesn't change
  - ∀ x.s. color(x,s) = color(x, Result(Grab, s))
  - Can be included in Effect axioms

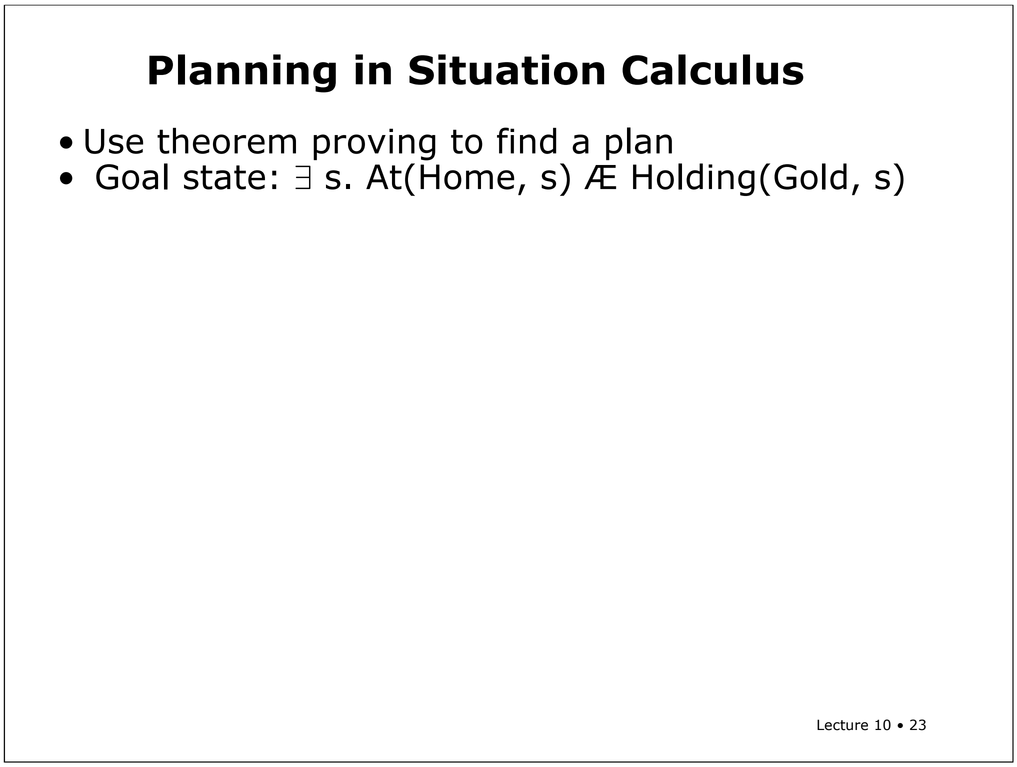
It turns out that there's a solution to the problem of having to talk about all the actions and what doesn't change when you do them. You can read it in the book where they're talking about successor-state axioms. You can, in effect, say what the conditions are under which an object changes color, for example, and then say that these are the **only** conditions under which it changes color.



## Planning in Situation Calculus

- Use theorem proving to find a plan

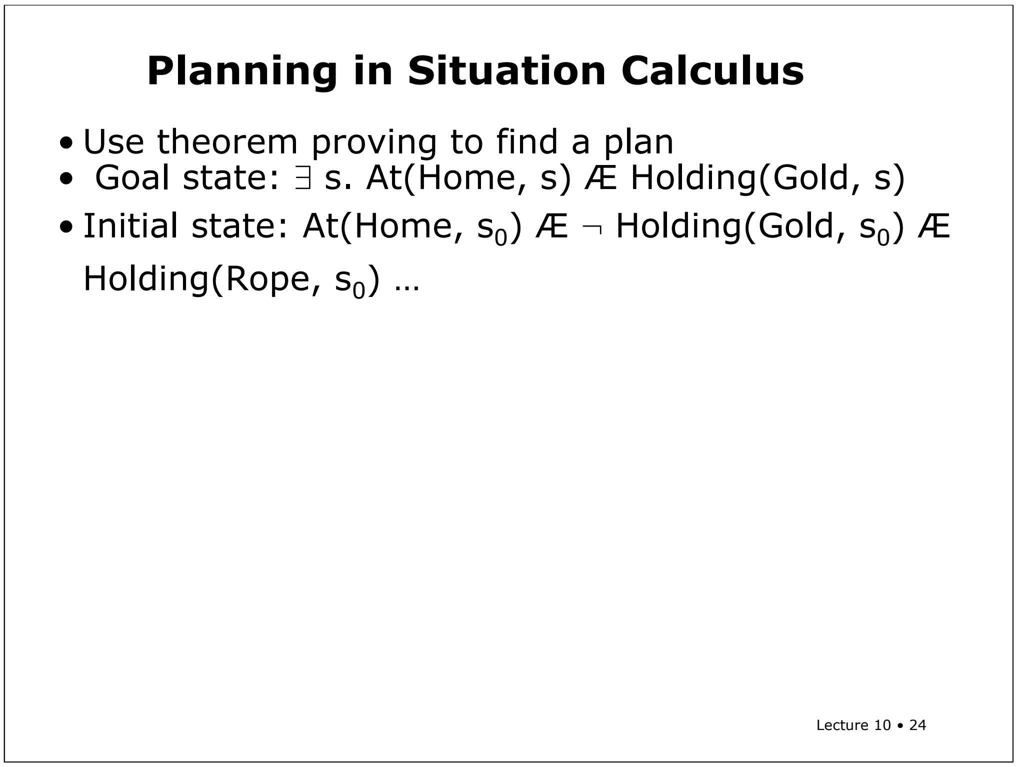
Assuming you write all of these axioms down that describe the dynamics of the world. They say how the world changes as you take actions. Then, how can you use theorem proving to find your answer?



## Planning in Situation Calculus

- Use theorem proving to find a plan
- Goal state: ∃ s. At(Home, s) Æ Holding(Gold, s)

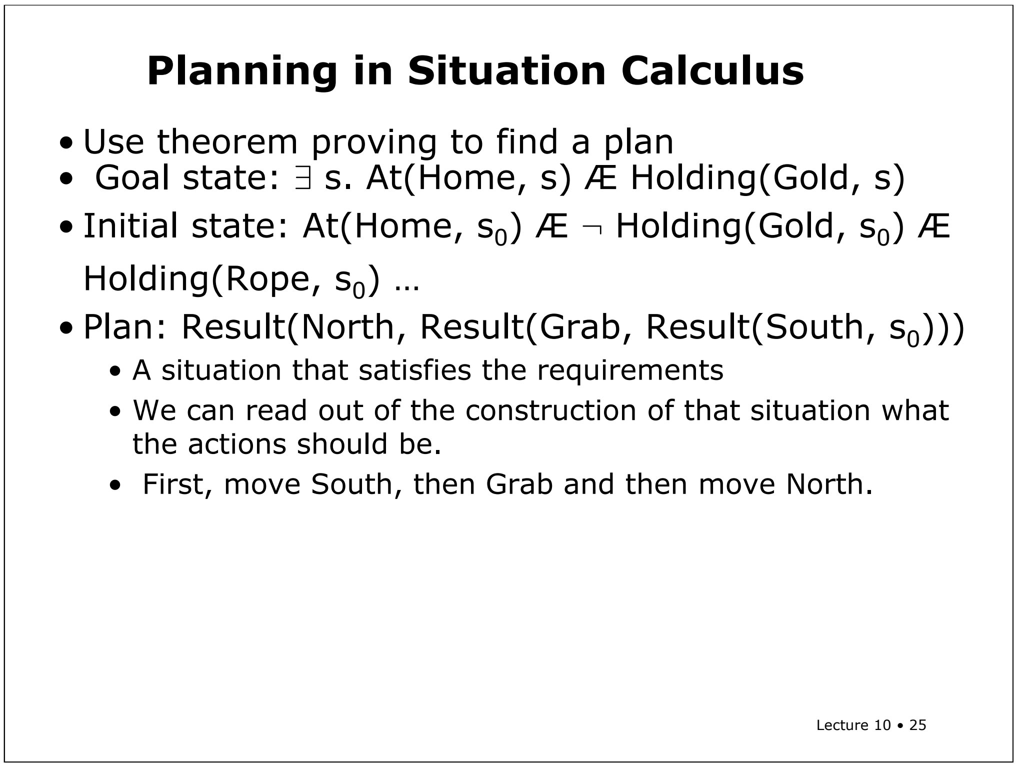
You can write the goal down somehow as the logical statement. For instance, you could say, well, I want to find a situation in which the agent is at home and is holding gold. But I don’t want to just prove that such a situation exists; I actually want to find it.



## Planning in Situation Calculus

- Use theorem proving to find a plan
- Goal state: ∃ s. At(Home, s) Æ Holding(Gold, s)
- Initial state: At(Home, $s_0$) Æ ¬ Holding(Gold, $s_0$) Æ Holding(Rope, $s_0$) …

You would encode your knowledge about the initial situation as some logical statements about some particular situation. We'll name it with the constant s0. We might say that the agent is at home in s0 and it's not holding gold, but it is holding rope, and so on.

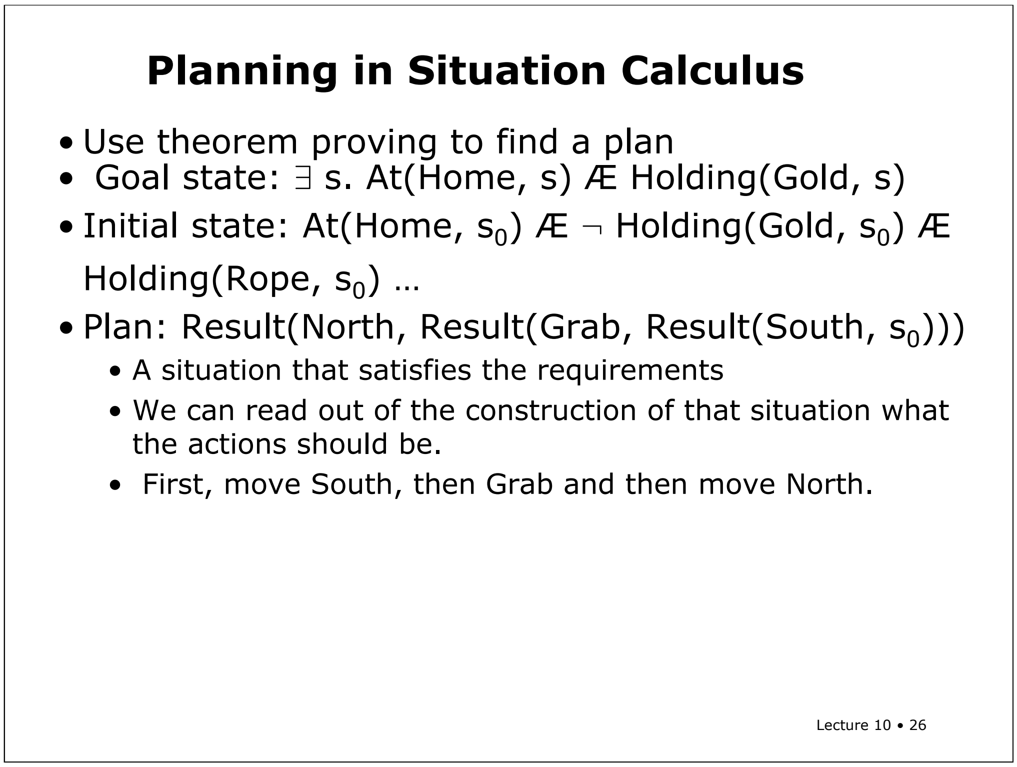


## Planning in Situation Calculus

- Use theorem proving to find a plan
- Goal state: ∃ s. At(Home, s) Æ Holding(Gold, s)
- Initial state: At(Home, $s_0$) Æ ¬ Holding(Gold, $s_0$) Æ Holding(Rope, $s_0$) …
- Plan: Result(North, Result(Grab, Result(South, $s_0$)))
  - A situation that satisfies the requirements
  - We can read out of the construction of that situation what the actions should be.
  - First, move South, then Grab and then move North.

Then, we could invoke a theorem prover on the initial state description, the effects axioms, and the goal description. It turns out that if you use resolution refutation to prove an existential statement, it's usually easy to extract a description of the particular s that the theorem prover constructed to prove the theorem (you can do it by keeping track of the substitutions that get made for s in the process of doing the proof).

So, for instance, in this example, we might find that the theorem prover found the goal conditions to be satisfied in the state that is the result of starting in situation s0, first going south, then doing a grab action, then going north. We can read the plan right out of the term that names the goal situation.



## Planning in Situation Calculus

- Use theorem proving to find a plan
- Goal state: ∃ s. At(Home, s) Æ Holding(Gold, s)
- Initial state: At(Home, $s_0$) Æ ¬ Holding(Gold, $s_0$) Æ Holding(Rope, $s_0$) …
- Plan: Result(North, Result(Grab, Result(South, $s_0$)))
  - A situation that satisfies the requirements
  - We can read out of the construction of that situation what the actions should be.
  - First, move South, then Grab and then move North.

Lecture 10 • 26

Note that we're effectively planning for a potentially huge class of initial states at once. We've proved that, no matter what state we start in, as long as it satisfies the initial state conditions, then following this sequence of actions will result in a new state that satisfies the goal conditions.

## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.

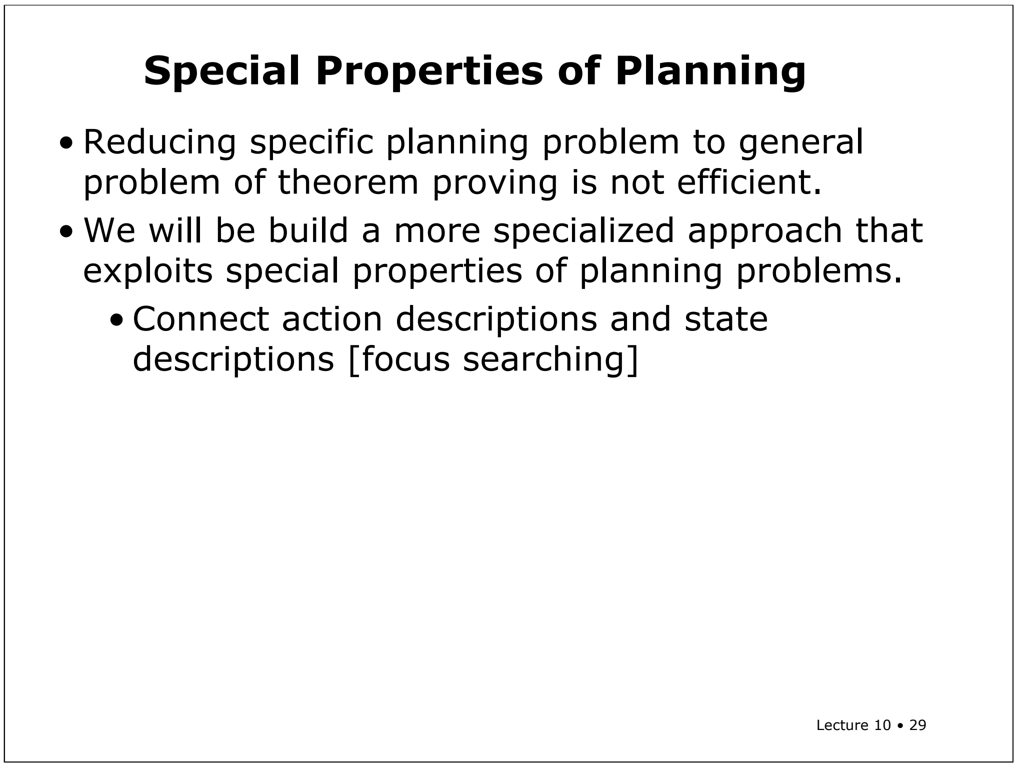
Way back when I was more naive than I am now, when I taught my first AI course, I tried to get my students to do this, to write down all these axioms for the Wumpus world, and to try to use a theorem prover to derive plans. It turned out that it was really good at deriving plans of Length 1, and sort of OK at deriving plans of Length 2, and, then, after that, forget it--it took absolutely forever. And it was not a problem with my students or the theorem prover, but rather with the whole enterprise of using the theorem prover to do this job.

So, there's sort of a general lesson, which is that even if you have a very general solution, applying it to a particular problem is often very inefficient, because you are not taking advantage of special properties of the particular problem that might make it easier to solve. Usually, the fact that you have a particular specific problem to start with gives you some leverage, some insight, some handle on solving that problem more directly and more efficiently. And that's going to be true for planning.

## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.
- We will be build a more specialized approach that exploits special properties of planning problems.

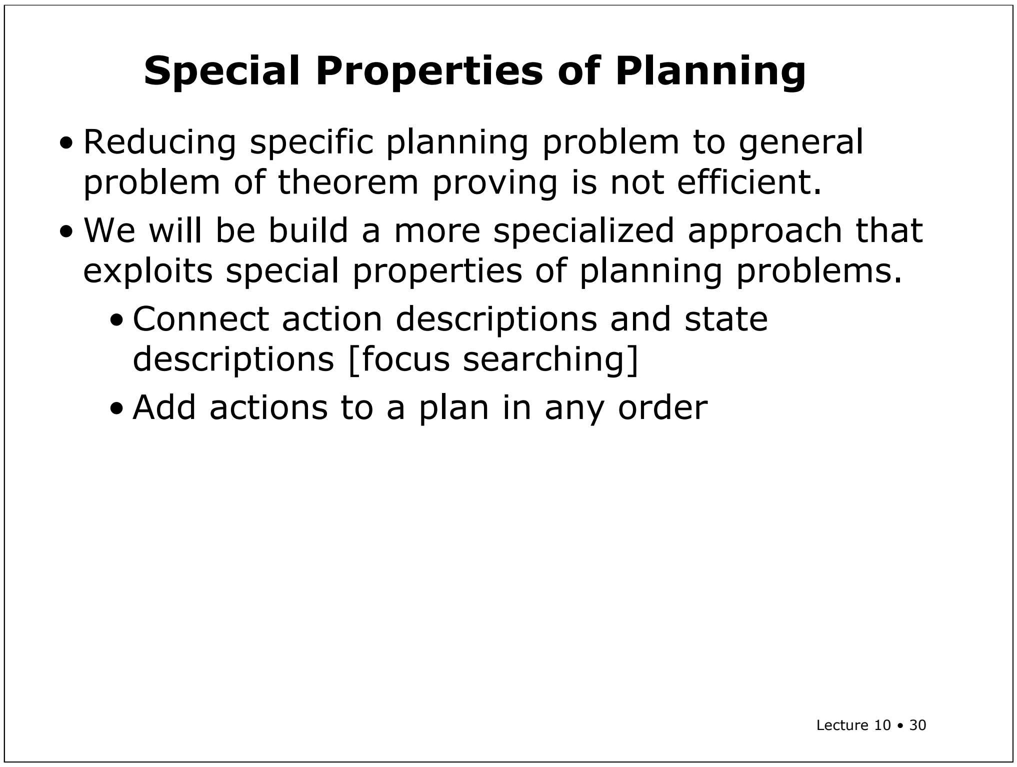
We're going to use a very restricted kind of logical representation in building a planner, and we're going to take advantage of some special properties of the planning problem to do it fairly efficiently. So, what special properties of planning can we take advantage of?



## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.
- We will be build a more specialized approach that exploits special properties of planning problems.
  - Connect action descriptions and state descriptions [focus searching]

One thing we can do is make explicit the connection between action descriptions and state descriptions. If I had an axiom that said "As a result of doing the grab action, we can make holding true." and we were trying to satisfy a goal statement that had “holding” as part of it, then it would make sense to consider doing the “grab” action. So, rather than just trying all the actions and searching blindly through the search space, you could notice, "Goodness, if I need holding to be true at the end, then let me try to find an action that would make it true." So, you can see these connections, and that can really focus your searching.

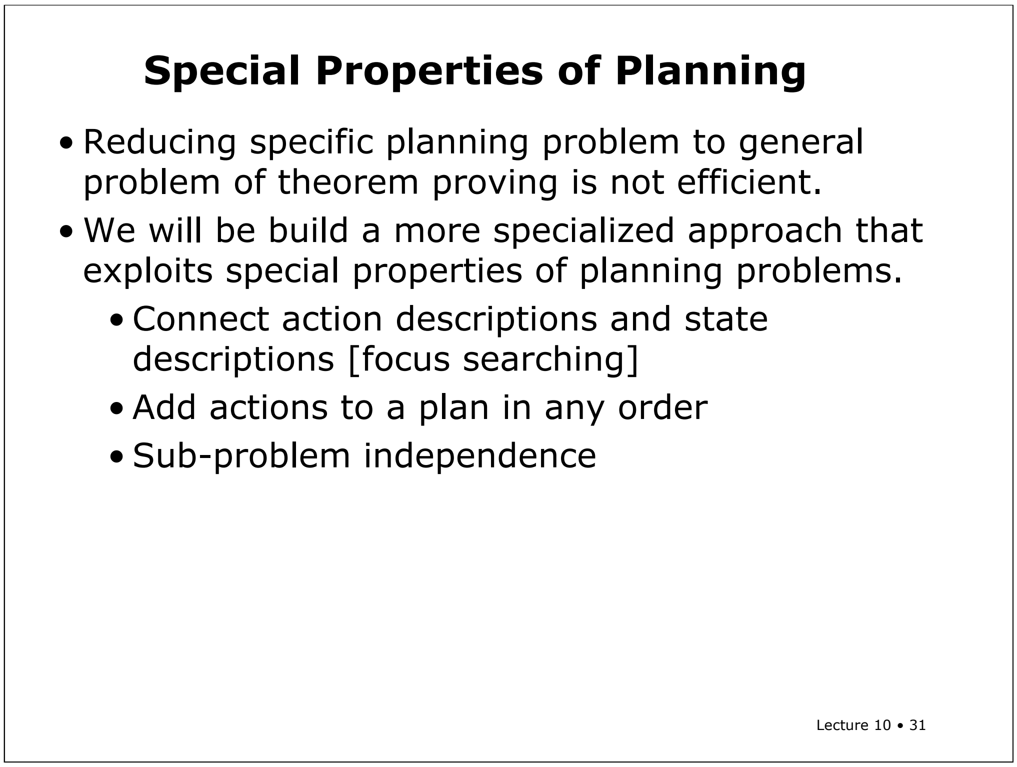


## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.
- We will be build a more specialized approach that exploits special properties of planning problems.
    - Connect action descriptions and state descriptions [focus searching]
    - Add actions to a plan in any order

Lecture 10 • 30

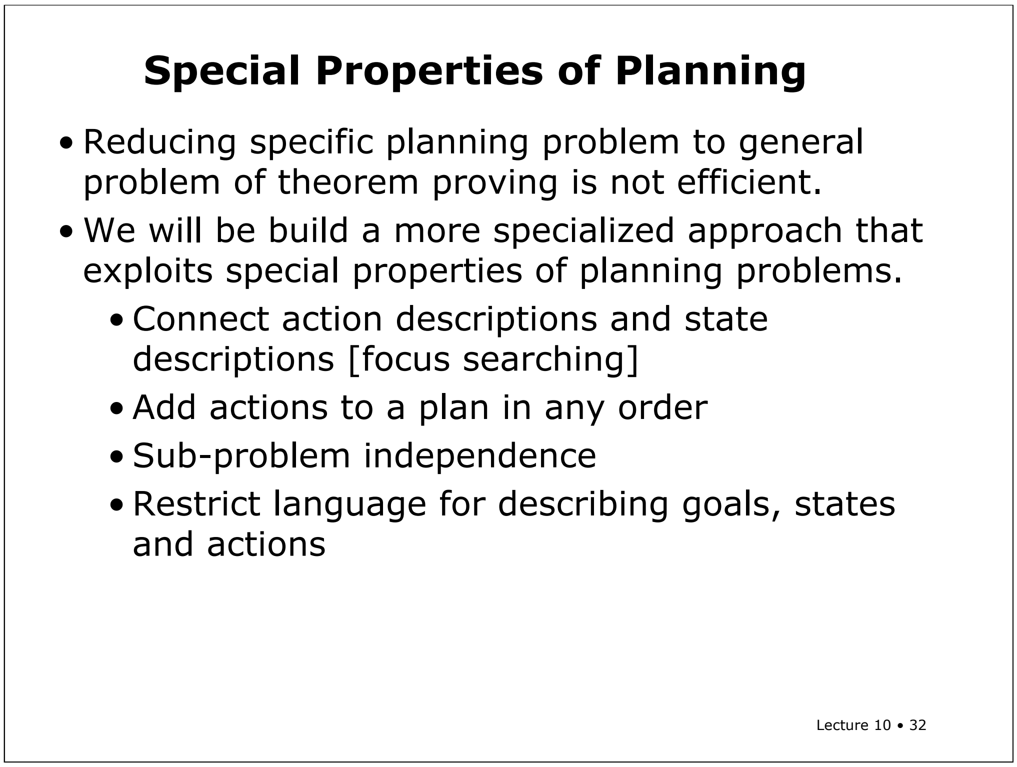
Another thing to notice, that's going to be really important, is that you can add actions to your plan in any order. Maybe you're trying to think about how best to go to Tahiti for spring break; you might think about which airline flight to take first. That maybe seems like the most important thing. You might think about that and really nail it down before you figure out how you're going to get to the airport or what hotel you're going to stay in or a bunch of other things like that. You wouldn't want to have to make your plan for going to Tahiti actually in the order that you're going to execute it, necessarily. Right? If you did that, you might have to consider all the different taxis you could ride in, and that would take you a long time, and then for each taxi, you think about, "Well, then how do I...." You don't want to do that. So, it can be easier to work out a plan from the middle out sometimes, or in different directions, depending on the constraints.



## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.
- We will be build a more specialized approach that exploits special properties of planning problems.
  - Connect action descriptions and state descriptions [focus searching]
  - Add actions to a plan in any order
  - Sub-problem independence

Another thing that we sometimes can take advantage of is sub-problem independence. If I have the goal to be at home this evening with a quart of milk and the dry cleaning, I can try to solve the quart of milk part and the dry cleaning part roughly independently and put those solutions together. So, to the extent that we can decompose a problem and solve the individual problems and stitch the solutions back together, we can often get a more efficient planning process. So, we're going to try take advantage of those things in the construction of a planning algorithm.



## Special Properties of Planning

- Reducing specific planning problem to general problem of theorem proving is not efficient.
- We will be build a more specialized approach that exploits special properties of planning problems.
  - Connect action descriptions and state descriptions [focus searching]
  - Add actions to a plan in any order
  - Sub-problem independence
  - Restrict language for describing goals, states and actions

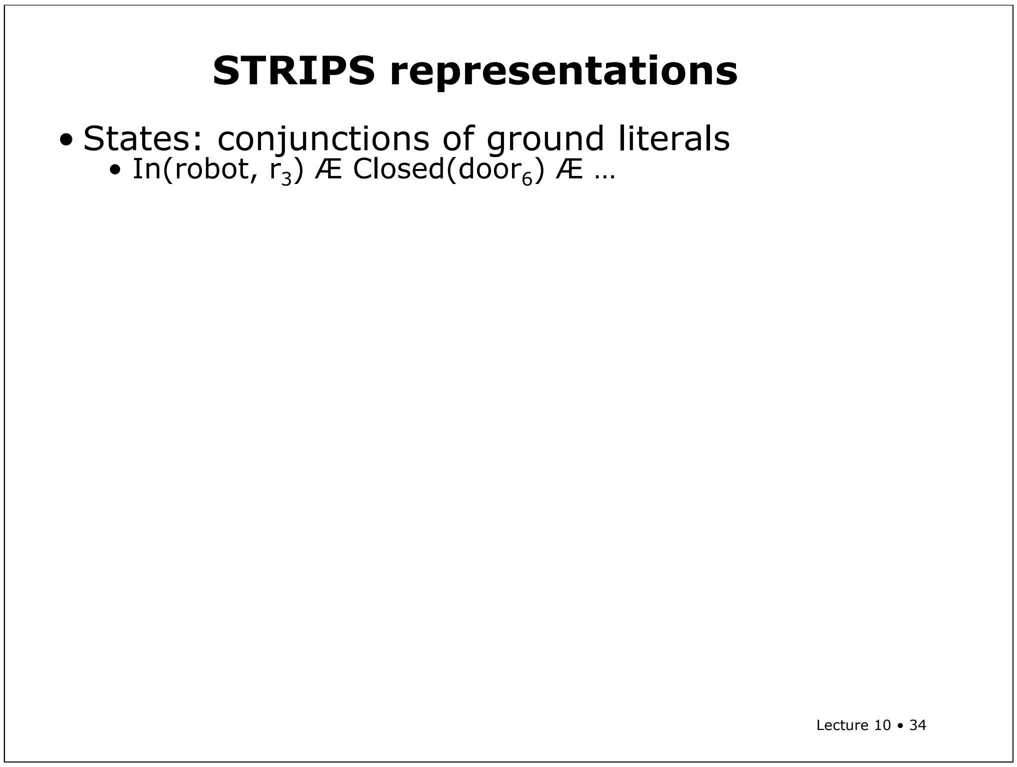
We're also going to scale back the kinds of things that we're able to say about the world. Especially for right now, just to get a very concrete planning algorithm. We'll be very restrictive in the language that we can use to talk about goals and states and actions and so on. We'll be able to relax some of those restrictions later on.

## STRIPS representations

Now we're going to talk about something called STRIPS representation. STRIPS is the name of the first real planning system that anybody built. It was made in a place called SRI, which used to stand for the Stanford Research Institute, so STRIPS stands for the Stanford Research Institute Problem Solver or something like that.

Anyway, they had a robot named Shakey, a real, actual robot that went around from room to room, and push boxes from here to there. I should say, it could do this in a very plain environment with nothing in the way. They built this planner, so that Shakey could figure out how to achieve goals by going from room to room, or pushing boxes around. It worked on a very old computer running in hardly any memory, so they had to make it as efficient as possible.

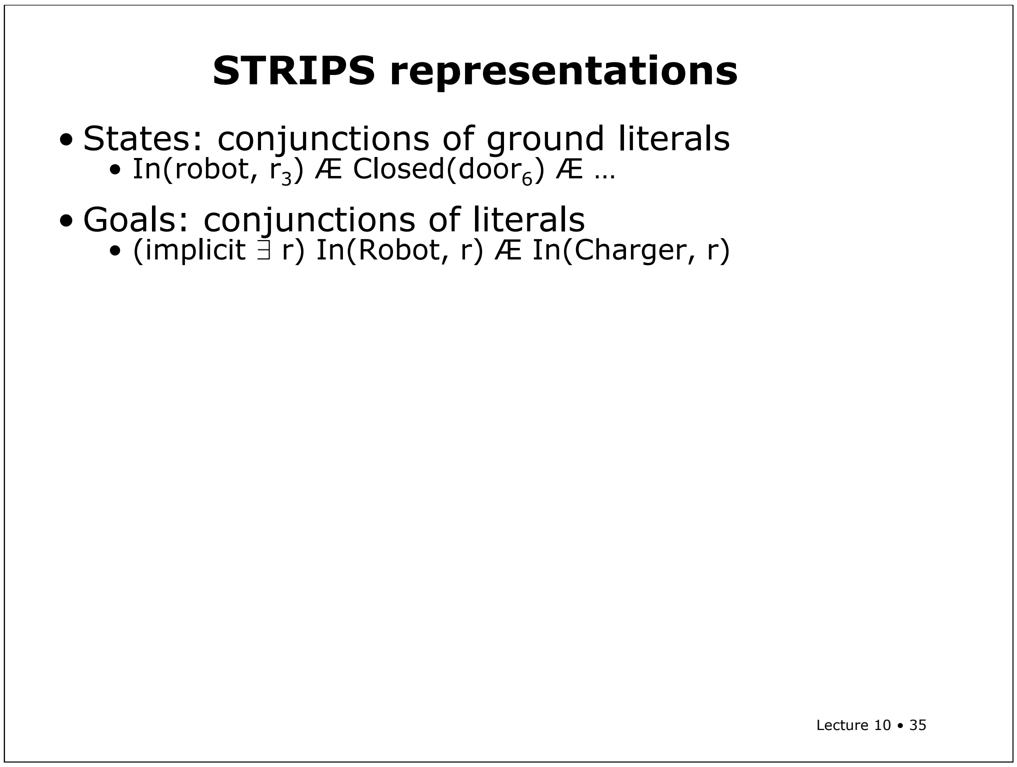
So, we're going to look at a class of planners that has essentially grown out from STRIPS. The algorithms have changed, but the basic representational scheme has been found to be quite useful.



## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …

So, we can represent a state of the world as a conjunction of ground literals. Remember, "ground" means there are no variables. And a "literal" can be either positive or negative. So, you could say, "In robot Room3" and "closed door6" and so on. That would be a description of the state of the world. Things that are not mentioned here are assumed to be unknown. You get some literals, and the conjunction of all those literals taken together describes the set of possible worlds; a set of possible configurations of the world. But we don't say whether Door 1 is open or closed, and so that means Door 1 could be either. And when we make a plan, it has to be a plan that would work in either case.

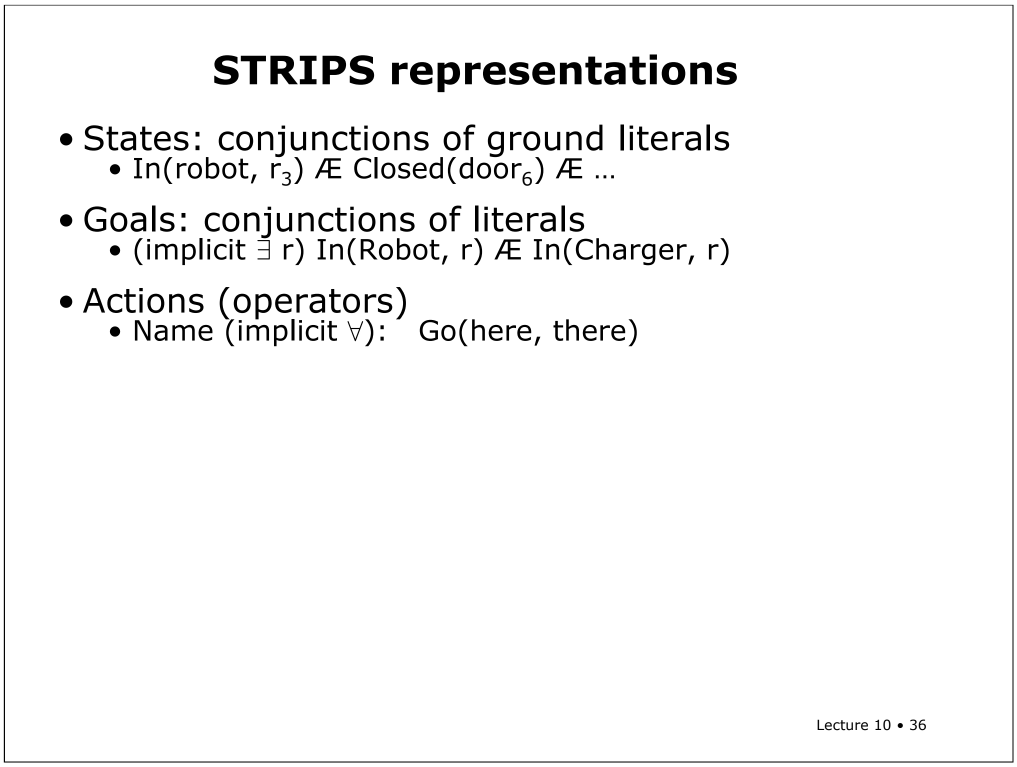


## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …
- Goals: conjunctions of literals
  - (implicit ∃ r) In(Robot, r) Æ In(Charger, r)

The goal is also a conjunction of literals, but they're allowed to have variables. You could say something like "I want to be in a room where the charger is." That could be your goal. You could say it logically by “In(Robot, r) and In(Charger, r)”.

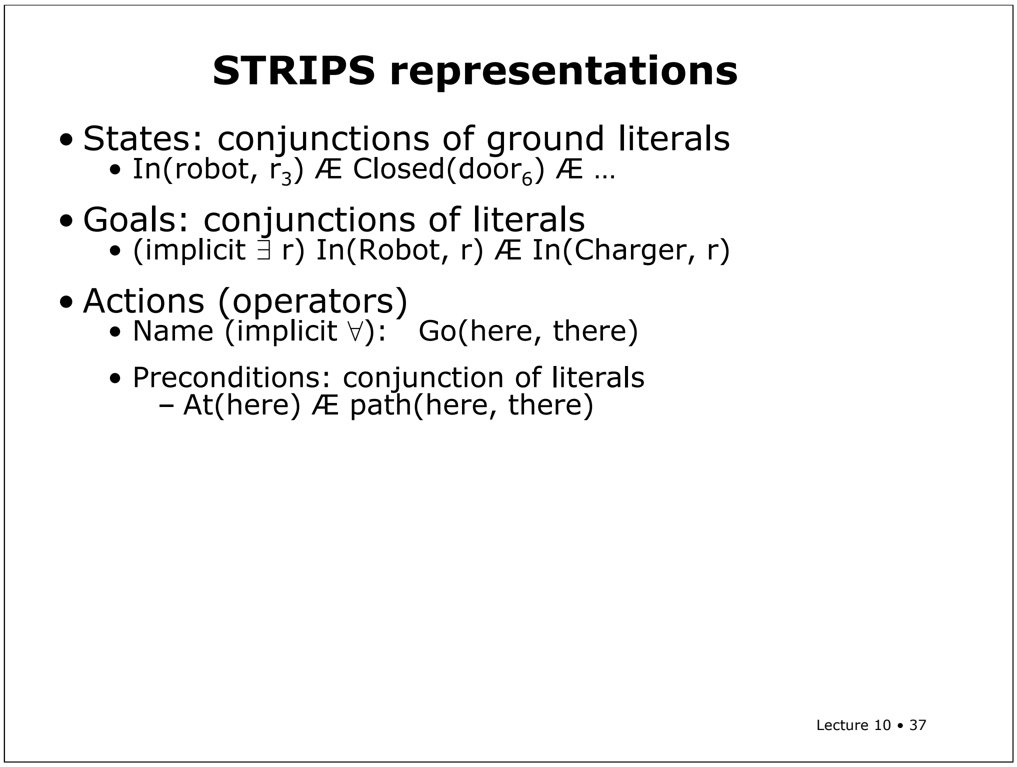
And maybe you'd make that true by going to the room where the charger is or maybe you'd make that true by moving the charger into some other room. It's implicitly, existentially quantified; we just have to find some r that makes it true.



## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …
- Goals: conjunctions of literals
  - (implicit ∃ r) In(Robot, r) Æ In(Charger, r)
- Actions (operators)
  - Name (implicit ∀): Go(here, there)

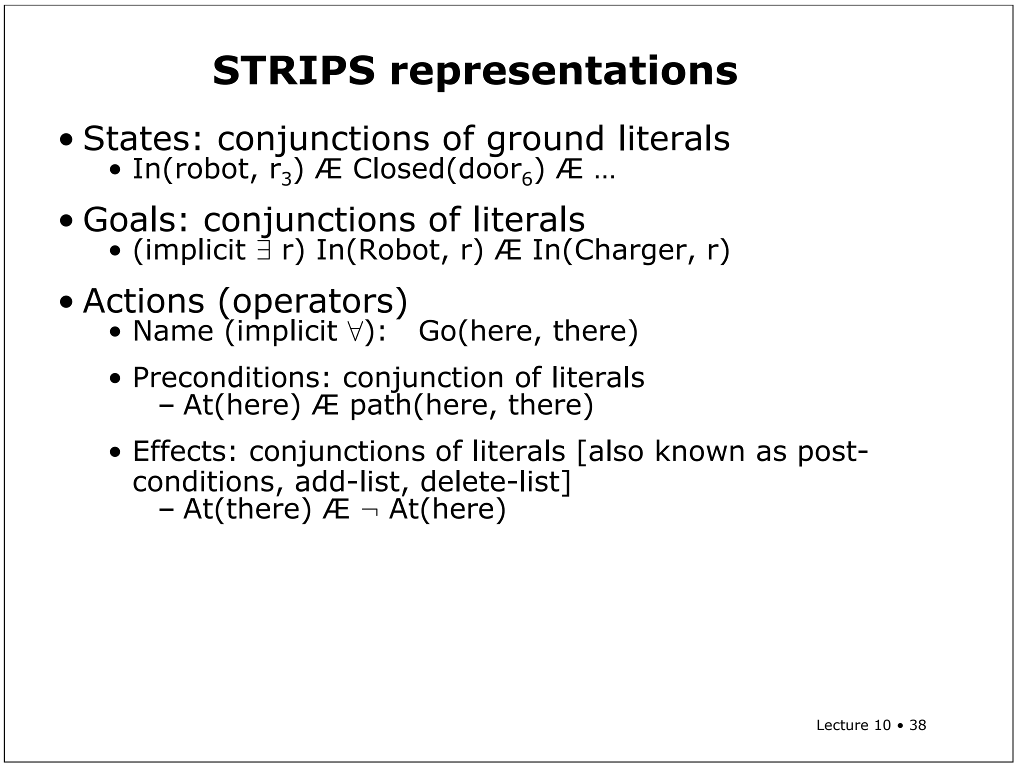
Actions are also sometimes also called operators. They have a name, and sometimes it's parameterized with variables that are implicitly universally quantified. So, we might have the operator Go with parameters here and there. This operator will tell us how to go from here to there for any values of here and there that satisfy the preconditions.



## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …
- Goals: conjunctions of literals
  - (implicit ∃ r) In(Robot, r) Æ In(Charger, r)
- Actions (operators)
  - Name (implicit ∀): Go(here, there)
  - Preconditions: conjunction of literals
    - At(here) Æ path(here, there)

The preconditions are also a conjunction of literals. A precondition is something that has to be true in order for the operator to be successfully applicable. So, in order to go from here to there, you have to be at here, and there has to be a path from here to there.



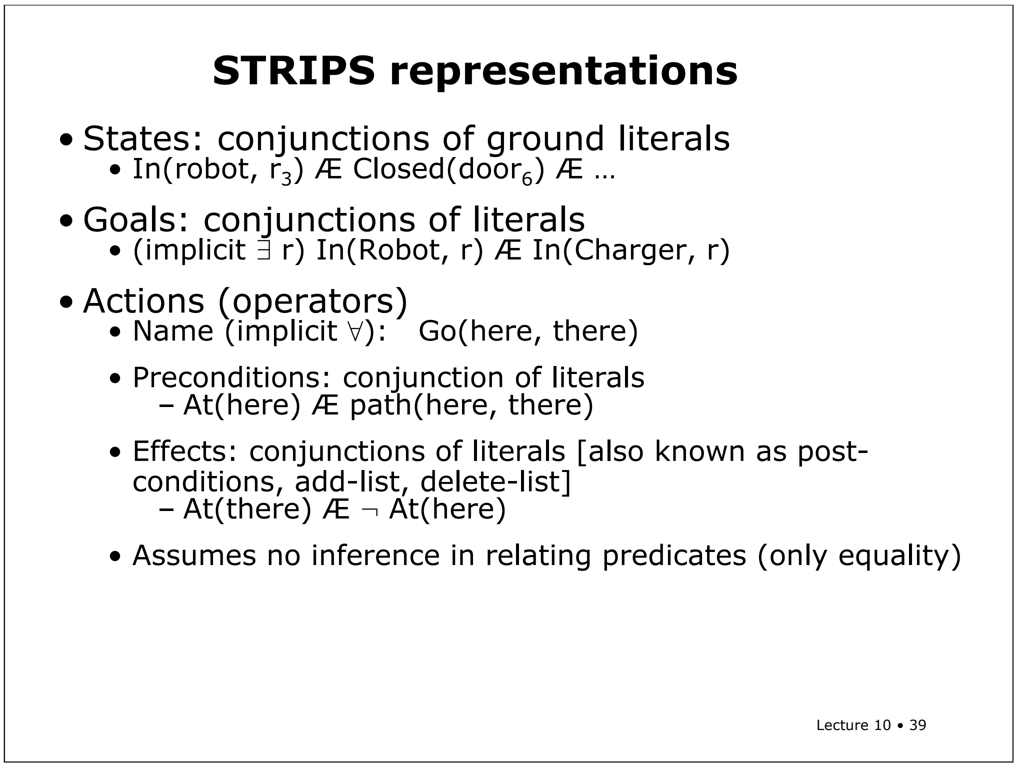
## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …
- Goals: conjunctions of literals
  - (implicit $\exists$ r) In(Robot, r) Æ In(Charger, r)
- Actions (operators)
  - Name (implicit $\forall$): Go(here, there)
  - Preconditions: conjunction of literals
    - At(here) Æ path(here, there)
  - Effects: conjunctions of literals [also known as post-conditions, add-list, delete-list]
    - At(there) Æ $\neg$ At(here)

Lecture 10 • 38

And then you have the "effect" which is a conjunction of literals. In our example, the effects would be that you're at there and not at here. The effects say what's true now is a result of having done this action. The effects are sometimes known as "post-conditions". In the original STRIPS, the positive effects were known as the add-list and the negative effects as the delete list (and the preconditions and goals could only be positive).

Now, since we're not working in a very general, logical framework, we're using the logic in a very special way, we don't have to explicitly, have arguments that describe the situation, because, implicitly, an operator description says, "If these things are true, and I do this action, then these things will be true in the resulting situation." But the idea of a situation as a changing thing is really built into the algorithm that we're going to use in the planner.



## STRIPS representations

- States: conjunctions of ground literals
  - In(robot, $r_3$) Æ Closed($door_6$) Æ …
- Goals: conjunctions of literals
  - (implicit ∃ r) In(Robot, r) Æ In(Charger, r)
- Actions (operators)
  - Name (implicit ∀): Go(here, there)
  - Preconditions: conjunction of literals
    - At(here) Æ path(here, there)
  - Effects: conjunctions of literals [also known as post-conditions, add-list, delete-list]
    - At(there) Æ ¬ At(here)
  - Assumes no inference in relating predicates (only equality)

Lecture 10 • 39

Now they built this system when, as I say, all they had was an incredibly small and slow computer, and so they absolutely had to just make it as easy as they could for the system. As time has progressed and computers have gotten fancier, people have added more richness back into the planners. Typically, you don't have to go to quite this extreme. They can do a little bit of inference about how predicates are related to each other. And then you might be allowed to have, say, "not(open)" here as the precondition, rather than "closed." But, in the original version, there was no inference at all. All you had were these operator descriptions, and whatever they said to add or delete, you added and deleted, and that was it. And to test if the pre-conditions were satisfied, there was no theorem proving involved; you would just look the preconditions up in your database and see if they were there or not. STRIPS is at one extreme of simplicity in inference, and situation calculus is at the other, complex extreme. Now, people are building systems that live somewhere in the middle. But it's useful to see what the simple case is like.

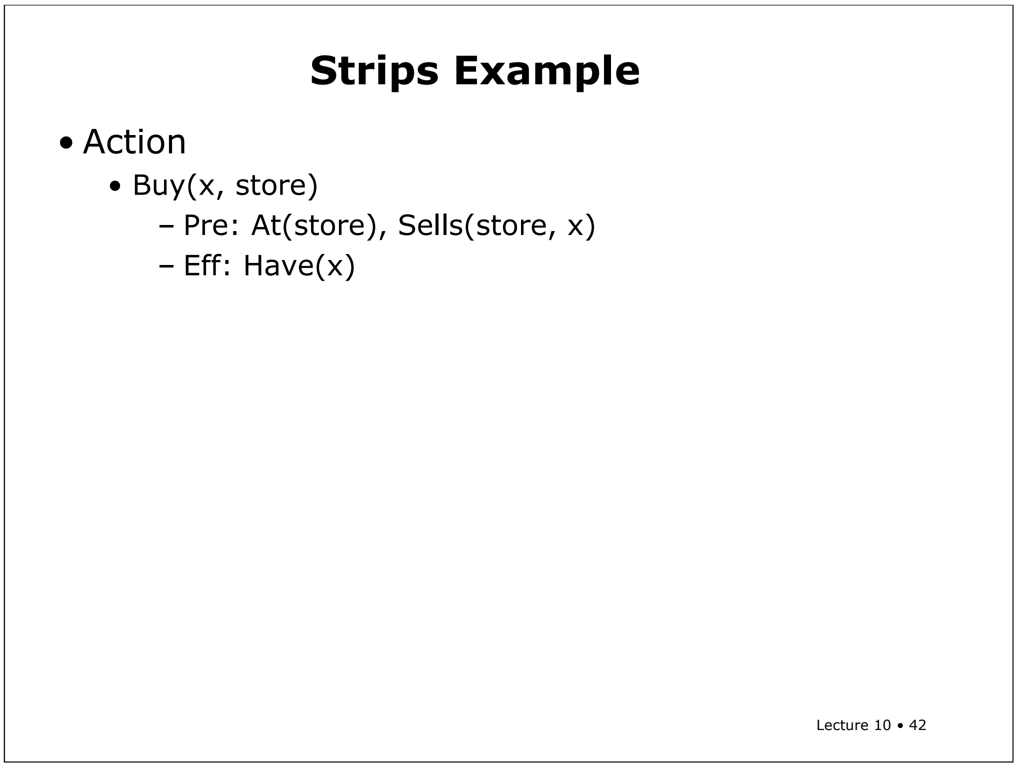
## Strips Example

Let's talk about a particular planning problem in STRIPS representation. It involves doing some errands.

## Strips Example

- Action

There are two actions.



## Strips Example

- Action
  - Buy(x, store)
    - Pre: At(store), Sells(store, x)
    - Eff: Have(x)

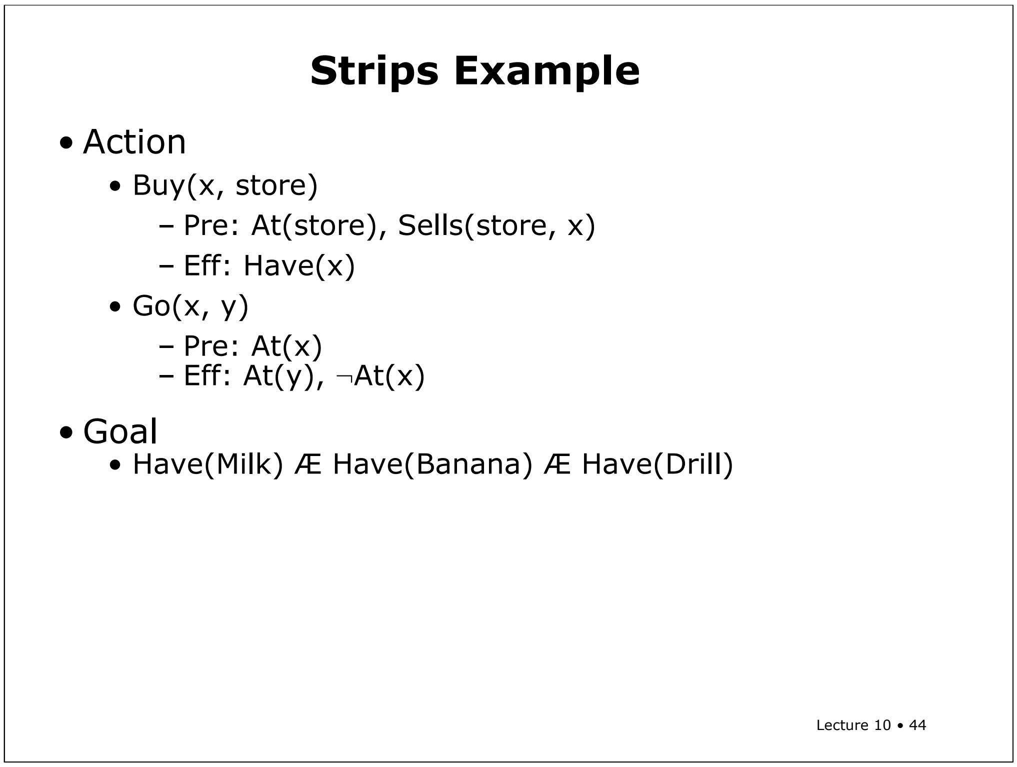
The “buy” action takes two arguments, the product, x, to be bought and the store. The preconditions are that we are at the store, and that the store sells the desired product. The effect is that we have x. Nothing else changes. We assume that we’re still at the store, and that, at least in this problem, we haven’t bought the last item, so that the store still sells x.

There's a nice problem in the book about extending this domain to deal with the case where you have money and how you would add money in as a pre- condition and a post- condition of the buy operator. It gets a little bit complicated because you have to have amounts of money and then money usually declines, and you go to the bank and so on.

## Strips Example

- Action
  - Buy(x, store)
    - Pre: At(store), Sells(store, x)
    - Eff: Have(x)
  - Go(x, y)
    - Pre: At(x)
    - Eff: At(y), ¬At(x)

The “go” action is just like the one we had before, except we’ll leave off the requirement that there be a path between x and y. We’ll assume that you can get from anywhere to anywhere else.

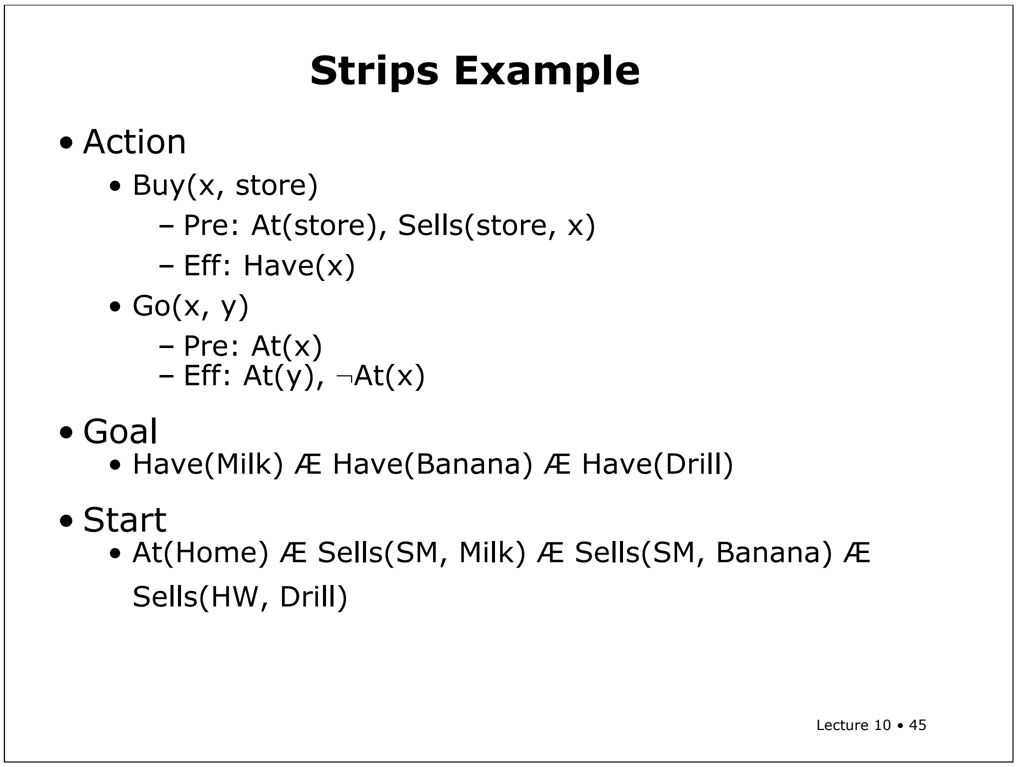


## Strips Example

- Action
  - Buy(x, store)
    - Pre: At(store), Sells(store, x)
    - Eff: Have(x)
  - Go(x, y)
    - Pre: At(x)
    - Eff: At(y), ¬At(x)
- Goal
  - Have(Milk) Æ Have(Banana) Æ Have(Drill)

Lecture 10 • 44

OK, now let's imagine that your goal is to have milk and have bananas and have a variable speed drill.



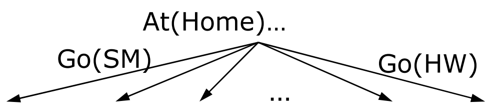
## Strips Example

- Action
  - Buy(x, store)
    - Pre: At(store), Sells(store, x)
    - Eff: Have(x)
  - Go(x, y)
    - Pre: At(x)
    - Eff: At(y), ¬At(x)
- Goal
  - Have(Milk) Æ Have(Banana) Æ Have(Drill)
- Start
  - At(Home) Æ Sells(SM, Milk) Æ Sells(SM, Banana) Æ Sells(HW, Drill)

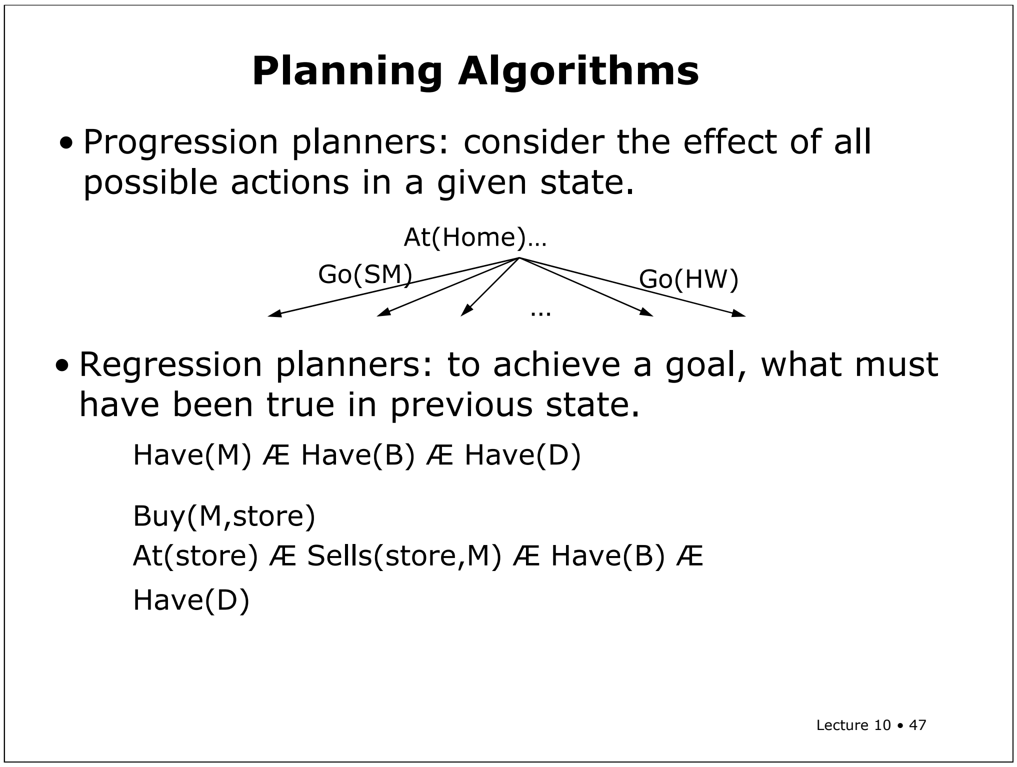
We're going to start at home and knowing that the supermarket sells milk, and the supermarket sells bananas, and the hardware store sells drills.

## Planning Algorithms

- Progression planners: consider the effect of all possible actions in a given state.

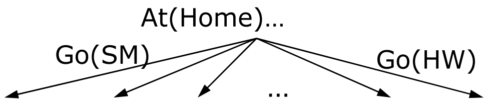


Now, if we go back to thinking of planning, as problem solving, then you might say, "All right, I'm going to start in this state." You'd right down the logical description for the start state, which really stands for a set of states. Then, you could consider applying each possible operator. But remember that our operators are parameterized. So from Home, we could go to any possible location in our domain. There might be a lot of places you could go, which would generate a huge branching factor. And then you can buy stuff. Think of all the things you could buy -- right? – there's potentially an incredible number of things you could buy. So, the branching factor, when you have these operators with variables in them is just huge, and if you try to search forward, you get into terrible trouble. Planners that search directly forward are called "progression planners."



## Planning Algorithms

- Progression planners: consider the effect of all possible actions in a given state.

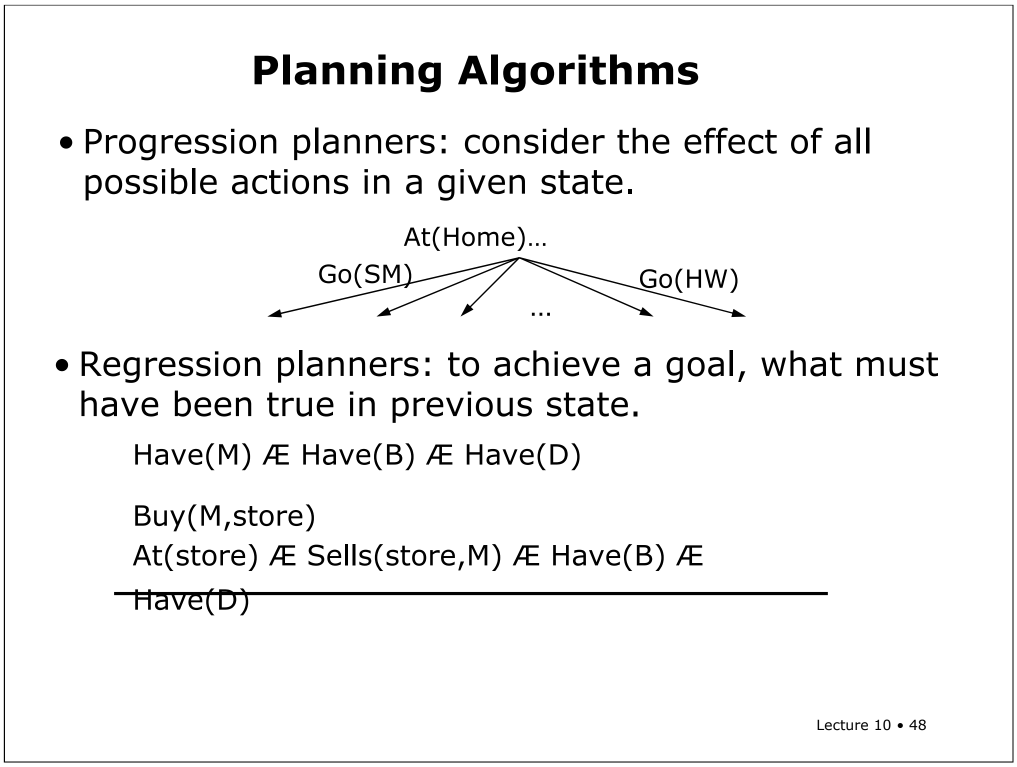


- Regression planners: to achieve a goal, what must have been true in previous state.

Have(M) Æ Have(B) Æ Have(D)

Buy(M,store)
At(store) Æ Sells(store,M) Æ Have(B) Æ
Have(D)

But, that doesn't work so well because you can’t take advantage of knowing where you're trying to go. They're not very directed. So, you say, "OK, well, I need to make my planner more directed. Let me drive backwards from the goal state, rather than driving forward from the start state." So, let's think about that. Rather than progression, we can do regression. So, now we're going to start with the goal state. Our goal state is this: We have milk and we have bananas and we have a drill. OK. So now we can do goal regression, which is sort of an interesting idea. We're going to go backward. If we wanted to make this true in the world and the last action we took before doing this was a particular action -- let's say it was "buy milk" -- then you could say, "All right, what would have to have been true in the previous situation in order that "buy milk" would make these things true in this situation?" So, if I want this to be true in the last step, what had to have been true on the next to the last step, so that "buy milk" would put me in that circumstance? "Well, we have to be at a store, and that store has to sell milk. And we also have to already have bananas and a drill.”



## Planning Algorithms

- Progression planners: consider the effect of all possible actions in a given state.

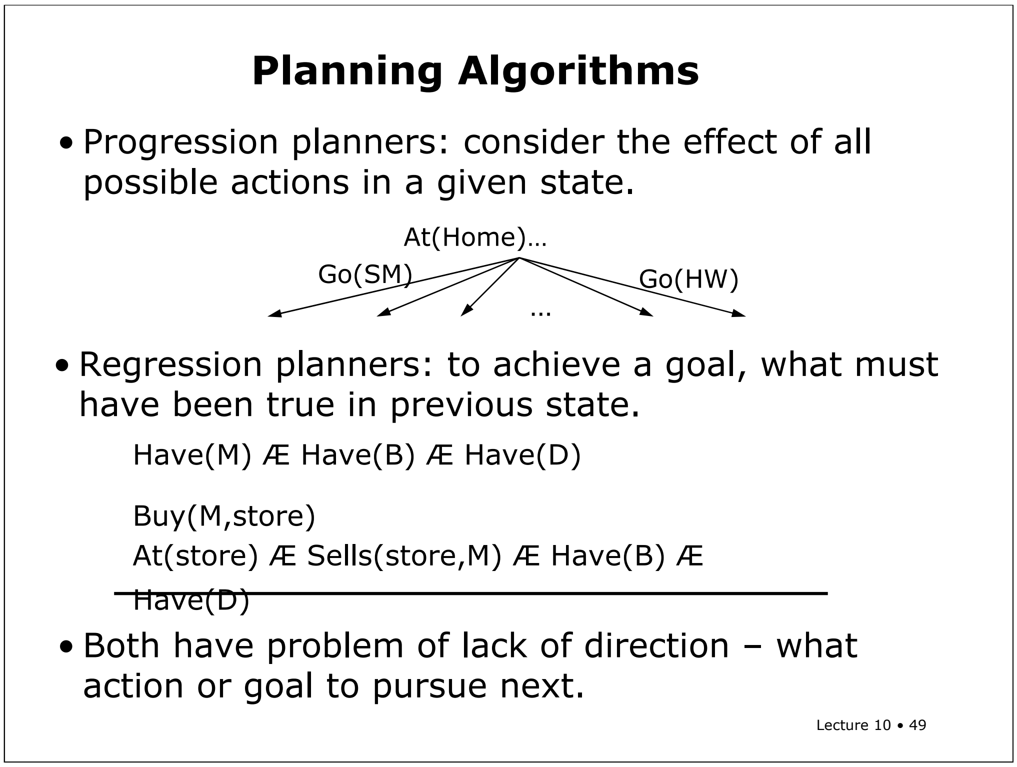
$$\begin{tikzcd}
 & & \text{At(Home)...} \arrow[dll, "\text{Go(SM)}"'] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr, "\text{Go(HW)}"] & & \\
 \, & \, & \cdots & \, & \,
\end{tikzcd}$$

- Regression planners: to achieve a goal, what must have been true in previous state.

  Have(M) Æ Have(B) Æ Have(D)

  Buy(M,store)
  At(store) Æ Sells(store,M) Æ Have(B) Æ Have(D)

So, now you're left with this thing that you're trying to make true. And now it seems a little bit harder -- right? -- because, again, you could say, "All right, well, what can I do? I could put in a step for buying bananas. So, then, that's going to be requiring me to be somewhere. Or, I could put in a step for buying the drill right now. You guys know that putting the drill-buying step here probably isn't so good because we can't buy it in the same place. But the planning algorithm doesn't really know that yet. You could also kind of go crazy. If you pick this "at store," you could go nuts because "store" could be anything. So, you could just suddenly decide that you want to be anywhere in the world.



## Planning Algorithms

- Progression planners: consider the effect of all possible actions in a given state.

At(Home)…
Go(SM)
…
Go(HW)

- Regression planners: to achieve a goal, what must have been true in previous state.

  Have(M) Æ Have(B) Æ Have(D)

  Buy(M,store)
  At(store) Æ Sells(store,M) Æ Have(B) Æ Have(D)

- Both have problem of lack of direction – what action or goal to pursue next.

But anyway, you're going to see that if you try to build the planner based on this principle, that, again, you're going to get into trouble because it's hard to see, of these things, what to do next -- how they ought to hook together. You can do it. And you can do search just like usual, and you would try a whole sequence of actions backwards until it didn't work, and then you'd go up and back-track and try again.  And so the same search procedures that you know about would apply here. But again, it feels like they're going to have to do a lot of searching, and it's not very well directed.

## Plan-Space Search

- Situation space – both progressive and regressive planners plan in space of situations

Progression and regression search in what we'll call "situation space". It's kind of like state space, but each node stands for a set of states. You're moving around in a space of sets of states. Your operations are doing things in the world, changing the set of states you're in.

## Plan-Space Search

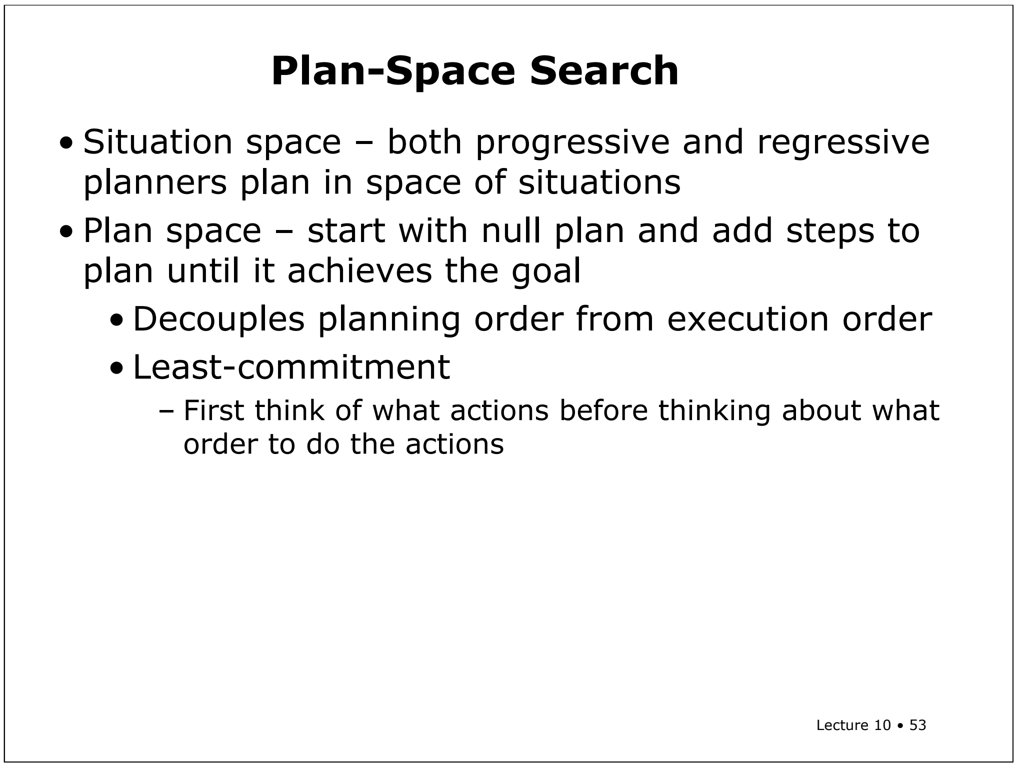
- Situation space – both progressive and regressive planners plan in space of situations
- Plan space – start with null plan and add steps to plan until it achieves the goal

A whole different way to think about planning is that you're moving around in a space of plans. Your operations are adding steps to your plan, changing the plan. You don't ever think explicitly about states or sets of states. The idea is you start out with an empty plan, and then there are operations that you can do to a plan, and you want to do operations to your plan until it's a satisfactory plan.

## Plan-Space Search

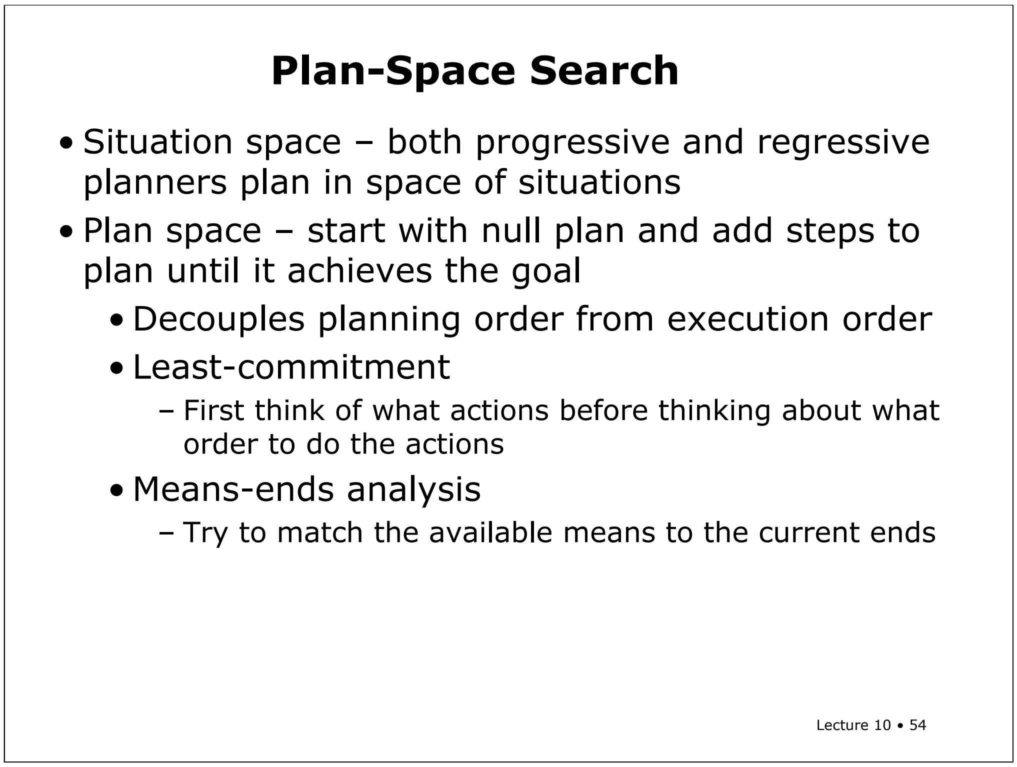
- Situation space – both progressive and regressive planners plan in space of situations
- Plan space – start with null plan and add steps to plan until it achieves the goal
    - Decouples planning order from execution order

Now you can de- couple the order in which you do things to your plan from the order in which the plan steps will eventually be executed in the world. You can decouple planning order from execution order.


## Plan-Space Search

- Situation space – both progressive and regressive planners plan in space of situations
- Plan space – start with null plan and add steps to plan until it achieves the goal
  - Decouples planning order from execution order
  - Least-commitment
    - – First think of what actions before thinking about what order to do the actions

Plan-space search also let us take what's typically called a "least commitment approach." By "least commitment," what we mean is that we don’t decide on particular details in our plan, like what store to go to for milk, until we are forced to. This keeps us from making premature, uninformed choices that we have to back out of later. You can say, "Well, I'm going to have to go somewhere and get the drill. I'm going to have to go somewhere and get the bananas. Can I figure that out and then think about what order they have to go in? Maybe then I can think about which stores would be the right stores to go to in order to do that." And so the idea is that you want to keep working on your plan, but never commit to doing a particular thing unless you're forced to.



## Plan-Space Search

- Situation space – both progressive and regressive planners plan in space of situations
- Plan space – start with null plan and add steps to plan until it achieves the goal
    - Decouples planning order from execution order
    - Least-commitment
        - First think of what actions before thinking about what order to do the actions
    - Means-ends analysis
        - Try to match the available means to the current ends

Plan-space search also lets us do means-end analysis. That is to say, that you can look at what the plan is trying to do, look at the means that you have available to you, and try to match them together. Simon built an early planner that tried to look at the goal state, and the current state, and find the biggest difference. So, trying to fly from one place to another, you would say, "The operator that will reduce the difference most between where I know I am and where I know I want to be is the flying operator. So, let me put that one in first. And then, I still have some other differences on either end. Like, there's a difference between being at my house and being at the airport, and so I'll put in some operators to deal with that. And there's another difference at the other end. I'll put those in." So, maybe addressing the biggest difference first is a way to organize your planning in a way that'll make it more effective. That's another idea at the higher level.

## Partially Ordered Plan

A partially ordered plan (or PO plan, for short), is kind of a complicated object. It's made up of 4 parts.

## Partially Ordered Plan

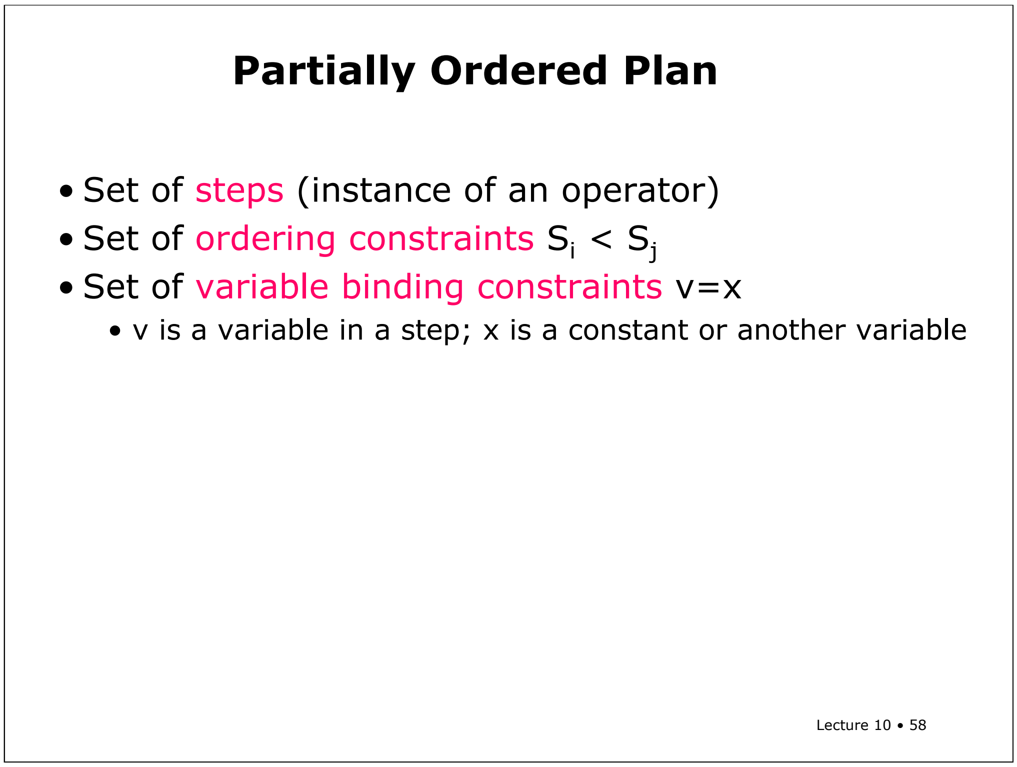
- Set of steps (instance of an operator)

The first part is a set of steps. A step is an operator, perhaps with some of the variables filled in with constants.

## Partially Ordered Plan

- Set of steps (instance of an operator)
- Set of ordering constraints $S_i < S_j$

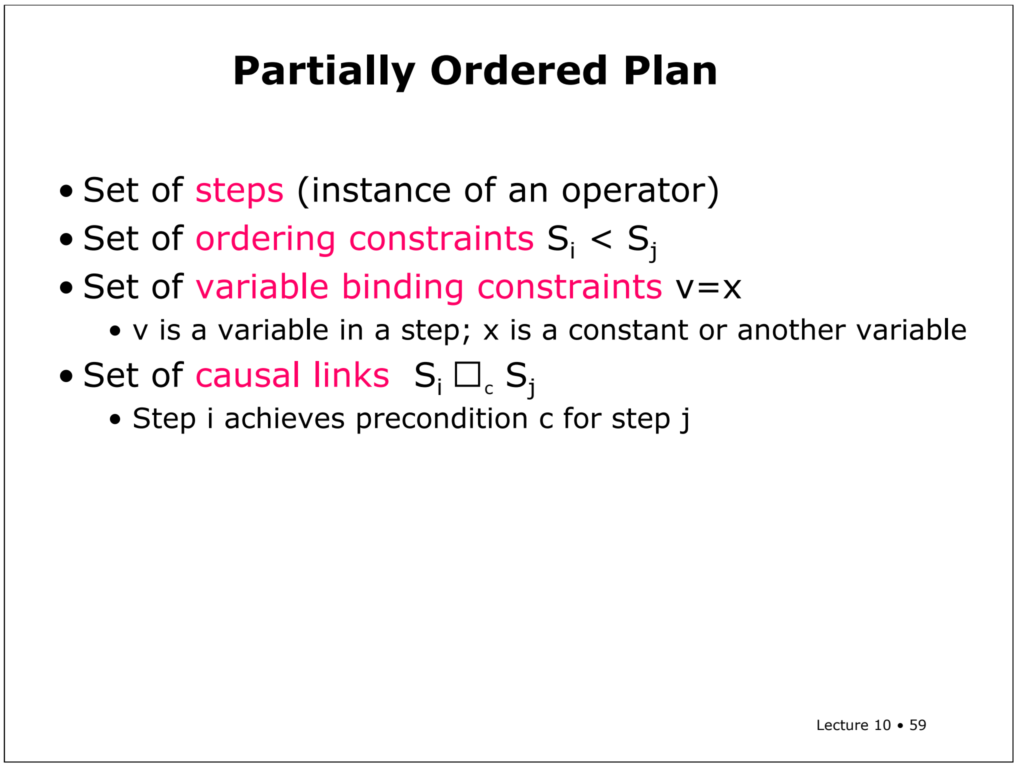
The second part is a set of ordering constraints between steps. They say that step I has to come before step j in the plan. This may just be a partial order (that is, it doesn't have to specify whether j comes before k or k comes before j), but it does have to be consistent (so it can't say that j comes before k and k comes before j).



## Partially Ordered Plan

- Set of steps (instance of an operator)
- Set of ordering constraints $S_i < S_j$
- Set of variable binding constraints $v=x$
  - $v$ is a variable in a step; $x$ is a constant or another variable

Then there is a set of variable binding constraints. They have the form "Some variable equals some value." V is a variable in one of the steps, and the X is a constant or another variable. So, for example, if you're trying to make a cake, and you have to have the eggs and the flour in the same bowl. You'd have an operator that says, "Put the eggs in [something]" and another that says "Put the flour in [something]." Let's say, you don't want to commit to which bowl you're going to use yet. Then, you might say, "Well, whatever the bowl is that I put the eggs into, it has to be the same as the bowl that I put the flour into, but it's not yet Bowl32." So, that's the case where you would end up having a partial plan where you had a variable constraint, that this variable equal that variable.



## Partially Ordered Plan

- Set of steps (instance of an operator)
- Set of ordering constraints $S_i < S_j$
- Set of variable binding constraints v=x
  - v is a variable in a step; x is a constant or another variable
- Set of causal links $S_i \square_c S_j$
  - Step i achieves precondition c for step j

And the last thing is really for internal bookkeeping, but it's pretty important. We also have a set of causal links. A causal link might be "Step I achieved pre-condition C for Step J." So, if I have to have money in order to buy the bananas, then I might have a "go to the bank" action and a "buy bananas" action and I would put, during the bookkeeping and planning, I would put this link in there that says: The reason I have the go-to-the-bank action in here is because it achieves the "have(money)" pre-condition that allows me to buy bananas.

So, the way that we do planning is: we add actions that we hope will achieve either parts of our goal, or pre-conditions of our other actions. And to keep track of what preconditions we have already taken care of, we put these links in so that we can remember why we're doing things.

So, a plan is just this. It's the set of steps with these constraints and the bookkeeping stuff.

## Initial Plan

The way we initialize the planning process is to start with a plan that looks like this.

## Initial Plan

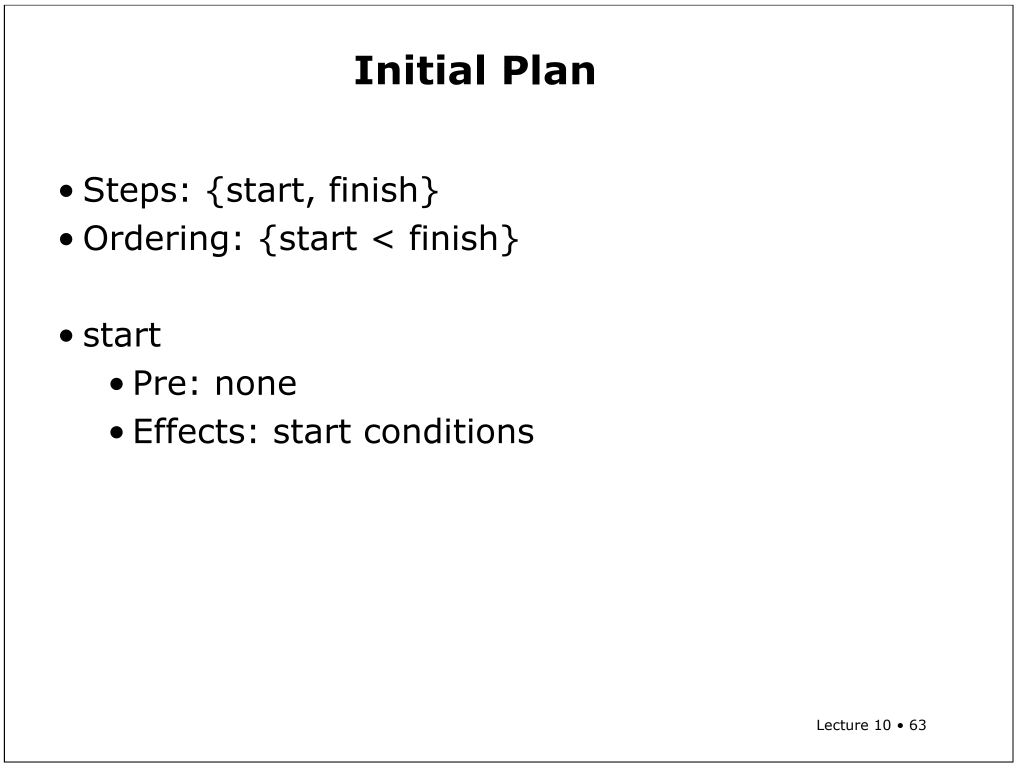
- Steps: {start, finish}

It has two steps: Start and Finish.

## Initial Plan

- Steps: {start, finish}
- Ordering: {start < finish}

We constrain the start step to happen before the finish step.



## Initial Plan

- Steps: {start, finish}
- Ordering: {start < finish}

- start
  - Pre: none
  - Effects: start conditions

Lecture 10 • 63

Start is a special operator that has no pre-condition and it has as its effect the starting conditions of the problem. So, in our "going to the supermarket" example, the starting conditions of that problem were that we were at home and we didn't have any milk and we didn't have the drill, and so on. In order to make the planning process uniform, we just assume this starting step has happened, to set things up.

## Initial Plan

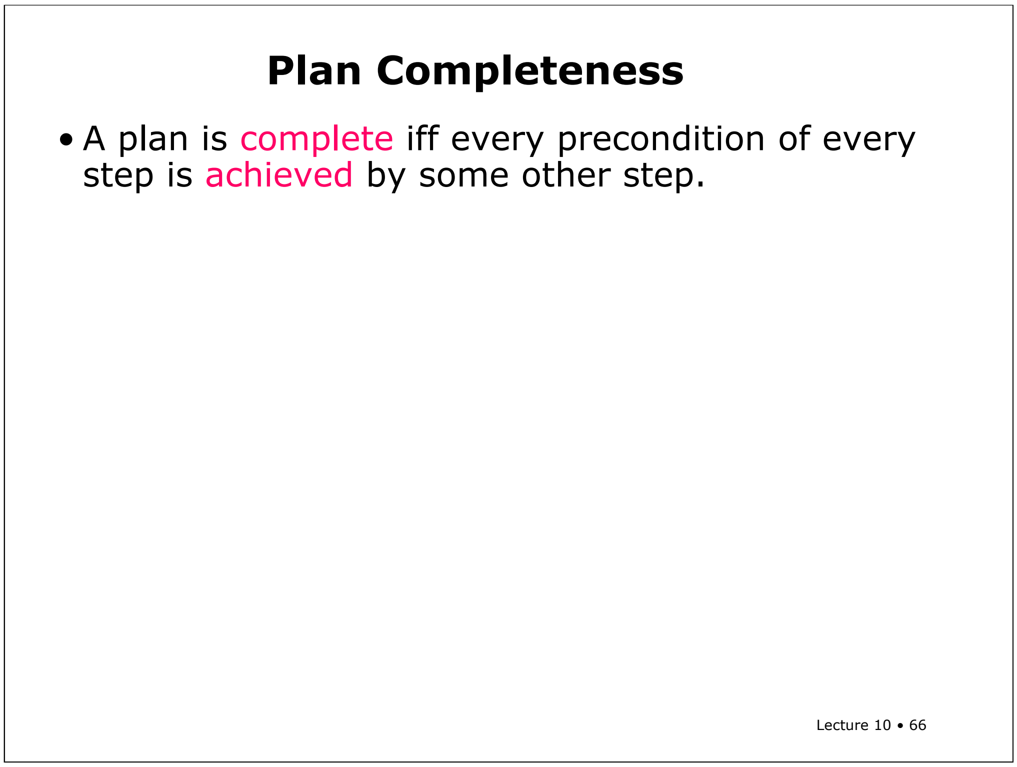
- Steps: {start, finish}
- Ordering: {start < finish}

- start
  - Pre: none
  - Effects: start conditions
- finish
  - Pre: goal conditions
  - Effects: none

And there's a special final action, "finish", that has as its pre-conditions the goal condition, and it has no effect. So, this is our initial plan.

## Plan Completeness

And now we're going to refine the plan by adding steps, ordering constraints, variable binding constraints, and causal links until it's a satisfactory plan. And so the question is, "Well, what is it that makes a plan satisfactory?" Here is a set of formal conditions on a plan that makes it a solution, that makes it a complete, correct plan. So, for a plan to be a solution, it has to be complete and consistent.

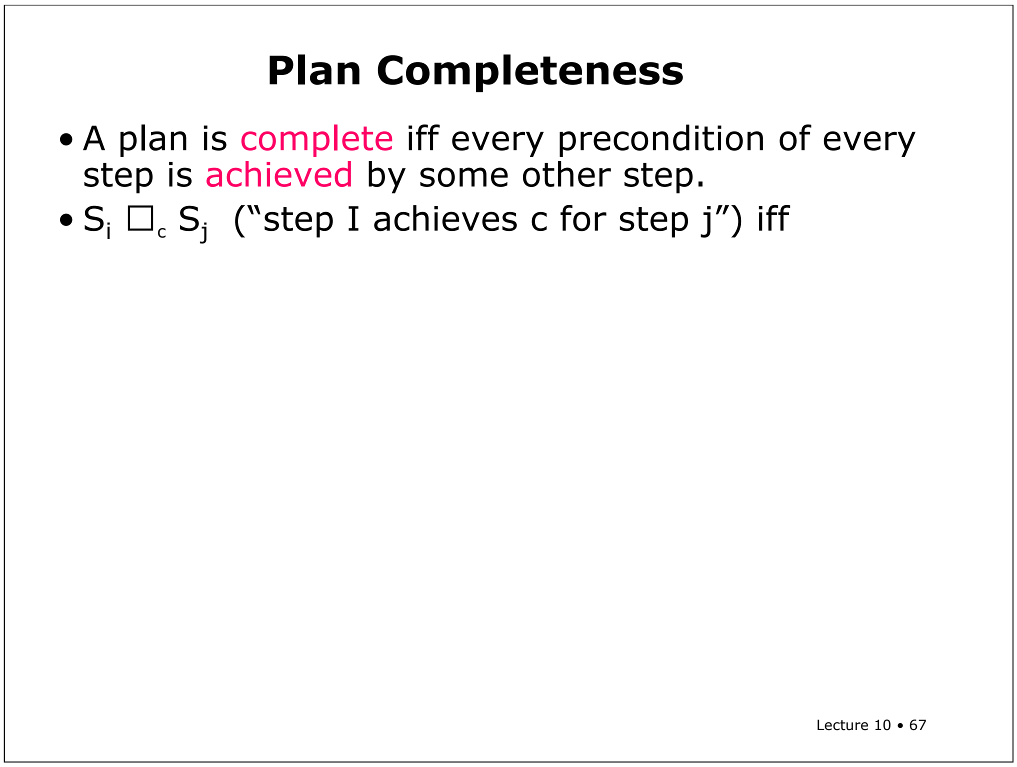


## Plan Completeness

- A plan is complete iff every precondition of every step is achieved by some other step.

So, let's talk about when a plan is complete. A plan is complete if every pre-condition of every step is achieved by some other step.

So basically, if you look at our starting plan there, we see that there's a big list of pre-conditions for the final step, and so what we're going to have to do is somehow add enough other steps to achieve all the pre-conditions of the final step, as well as any of the pre-conditions of those other steps that we've added.



## Plan Completeness

- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i \square_c S_j$ (“step I achieves c for step j”) iff

Lecture 10 • 67

So, then, let's understand what "achieve" means.

## Plan Completeness

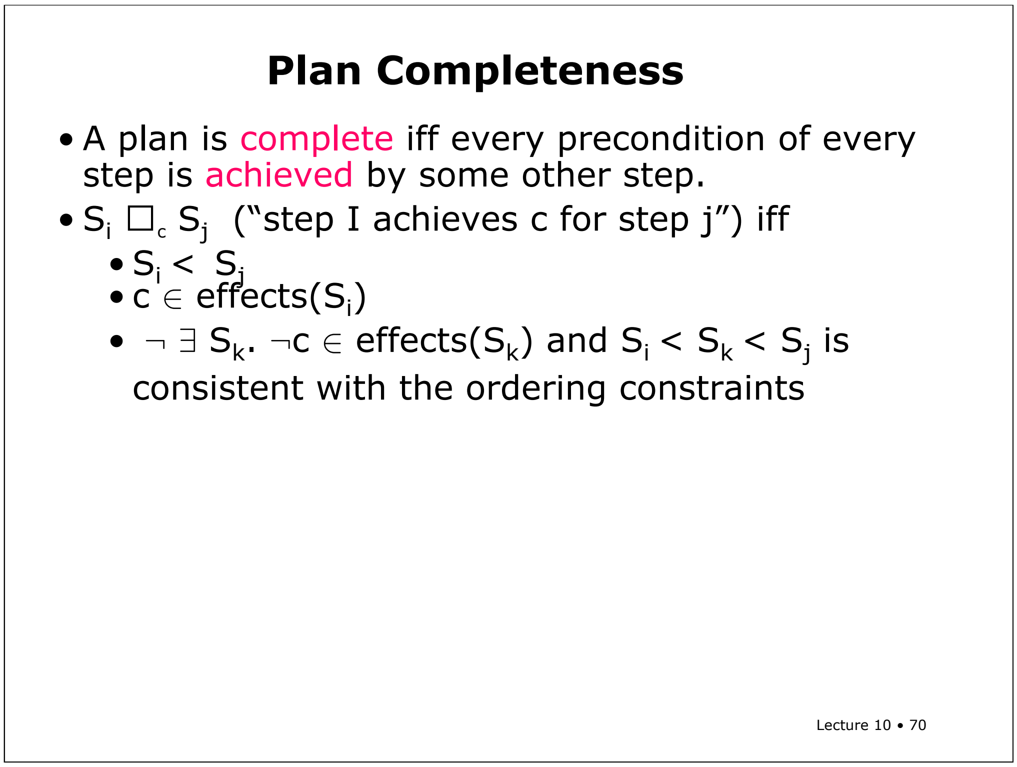
- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i$ □$_c$ $S_j$ ("step I achieves c for step j") iff
  - $S_i < S_j$

For Step I to achieve C for step J, SI has to come before SJ, right? It's part of the notion of achievement,

## Plan Completeness

- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i \square_c S_j$ (“step I achieves c for step j”) iff
  - $S_i < S_j$
  - $c \in \text{effects}(S_i)$

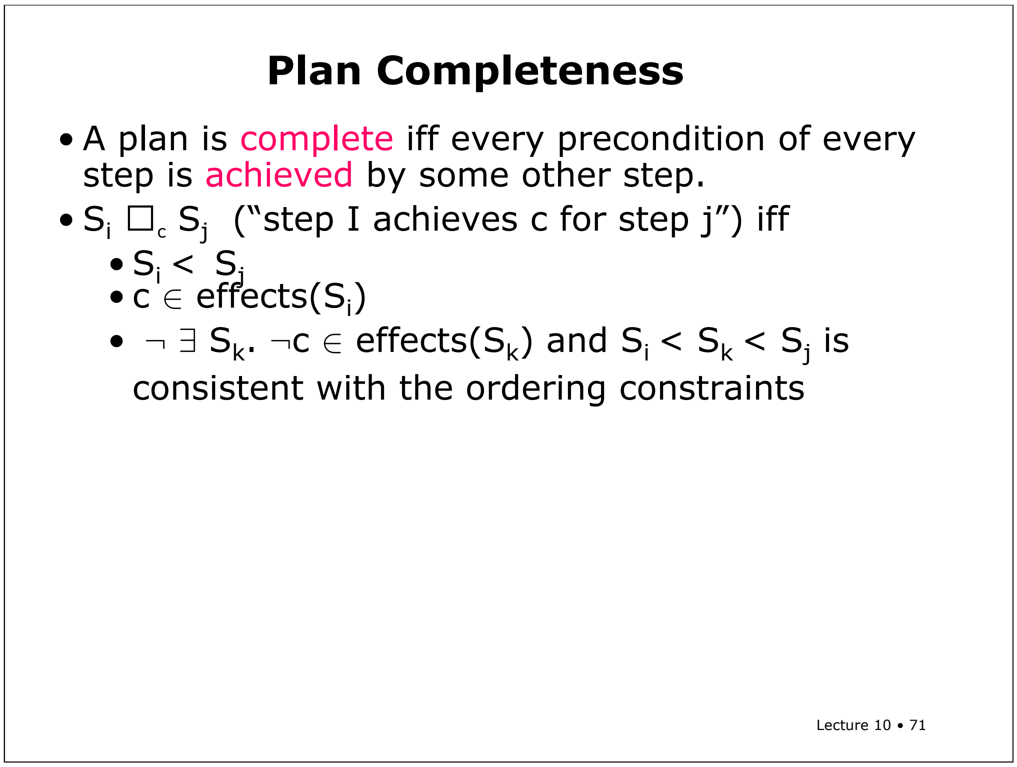
And C has to be a member of the effects of SI.



## Plan Completeness

- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i \square_c S_j$ ("step I achieves c for step j") iff
  - $S_i < S_j$
  - $c \in \text{effects}(S_i)$
  - $\neg \exists S_k.\ \neg c \in \text{effects}(S_k)$ and $S_i < S_k < S_j$ is consistent with the ordering constraints

What else? Is there anything else that you imagine we might need here? What if going to the bank step achieves "have money" for "buy milk." If we go to the bank before we buy the milk, and going to the bank achieves having money – there's still something that could go wrong. You could go to the bank. Between going to the bank and buying milk, all kinds of things could happen. Right? You could be robbed, or you could spend it all on something more fun or who knows what? So, you also have to say "And there's no intervening event that messes things up." "There's no Sk such that, not C is in the effects of Sk." "And SI < SK < Sj is consistent with the ordering constraint." So, there's no step that can mess up our effect that could possible intervene between when we achieve it with si and when we need it at sj.



## Plan Completeness

- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i \square_c S_j$ ("step I achieves c for step j") iff
  - $S_i < S_j$
  - $c \in \text{effects}(S_i)$
  - $\neg \exists S_k.\ \neg c \in \text{effects}(S_k)$ and $S_i < S_k < S_j$ is consistent with the ordering constraints

The idea is that these ordering constraints don't necessarily specify a total order on the plan steps. So, it might be that the plan is very laissez-faire. It's says, "Oh, go to the bank sometime. Buy a mink coat sometime. Buy milk sometime." And so it doesn't say that buying the mink coat is between the bank and the milk, but buying the mink coat could be between the bank and the milk. And that's enough for that to be a violation. So, if we have a way of achieving every condition so that the thing that's doing the achieving is constrained to happen before we need it -- we get the money before we buy the milk -- it does achieve the effect that we need and there's nothing that comes in between to clobber it, then our plan is good.

## Plan Completeness

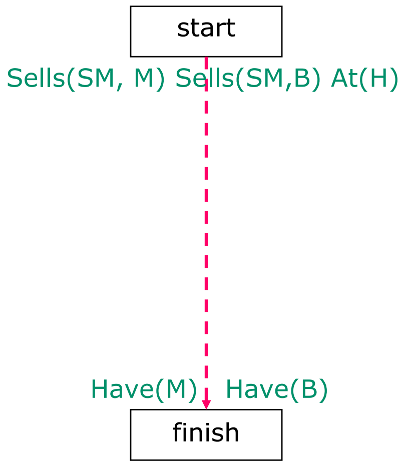
- A plan is complete iff every precondition of every step is achieved by some other step.
- $S_i$ □$_c$ $S_j$ ("step I achieves c for step j") iff
  - $S_i < S_j$
  - $c \in \text{effects}(S_i)$
  - $\neg \exists S_k.\ \neg c \in \text{effects}(S_k)$ and $S_i < S_k < S_j$ is consistent with the ordering constraints
- A plan is consistent iff the ordering constraints are consistent and the variable binding constraints are consistent.

For a plan to be consistent, it is enough for the temporal ordering constraints to be consistent (we can't have I before j and j before I) and for the variable binding constraints to be consistent (we can't require two constants to be equal).

## PO Plan Example

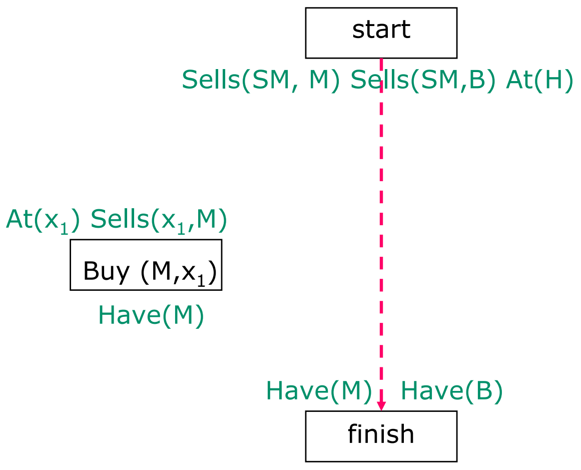
Let us just spend a few minutes doing an example plan for the milk case, very informally, and next time we'll go through the algorithm and do it again, more formally.

## PO Plan Example



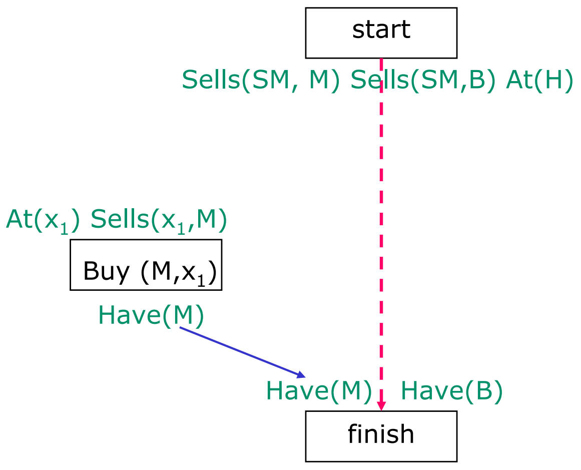
Here’s our initial plan. With the start and finish steps. We’ll draw ordering constraints using dashed red lines. We put the effects of a step below it, and the preconditions of a step above it. We’ll do a simplified version of the whole problem, deleting the requirement for a drill, and just having as our goal that we have milk and bananas.

## PO Plan Example



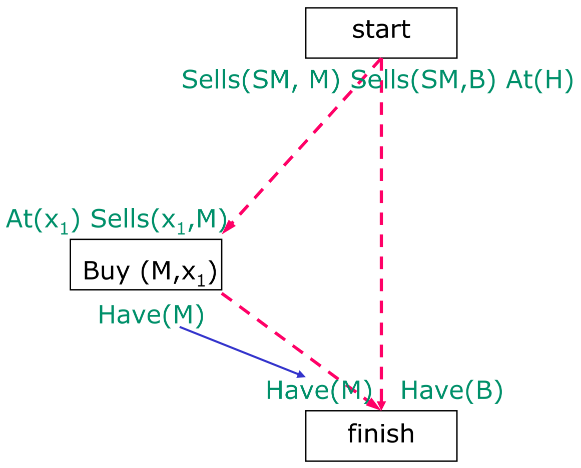
It doesn't seem useful to add any further constraints at this point, so let's buy milk. We'll start by adding a step that says we're going to by milk at some place called x1. It has preconditions at(x1) and sells(x1, milk); and it has the effect have(milk).

## PO Plan Example



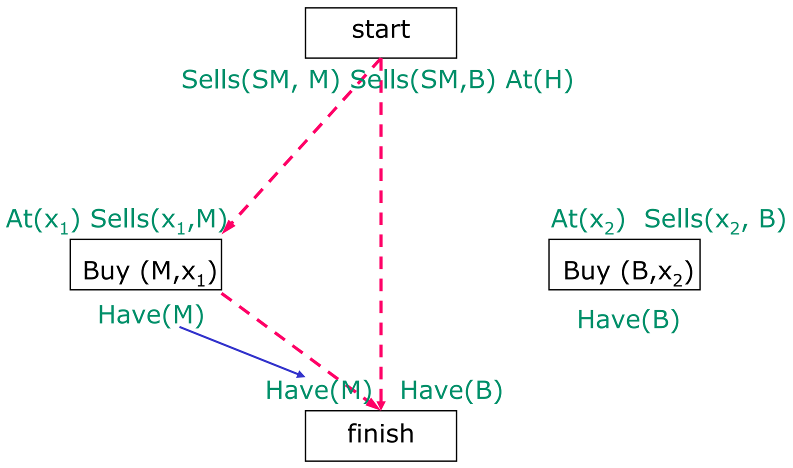
Now, we can add a blue causal link that says we are going to use this effect of have(Milk) to satisfy the precondition have(Milk) in the finish step.

## PO Plan Example



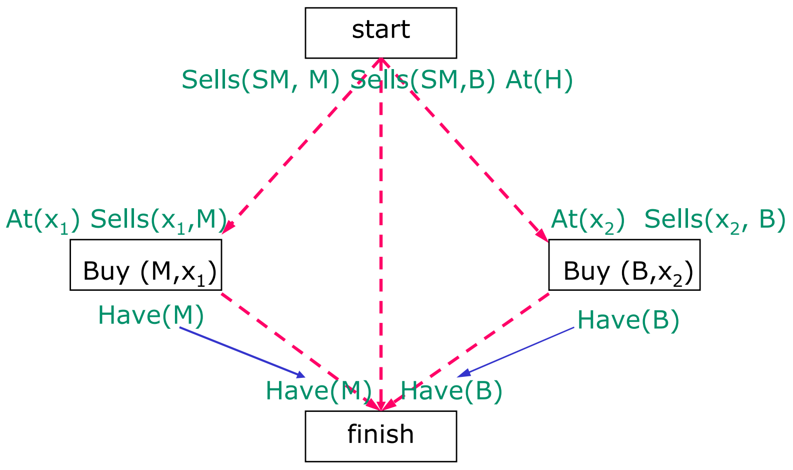
Every causal link also implies an ordering link, so we’ll add an ordering link as well between this step and finish. And we should also require that this step happen after start.

## PO Plan Example



Now, let's buy bananas. We add a step to buy bananas at location x2, including its preconditions and effects.

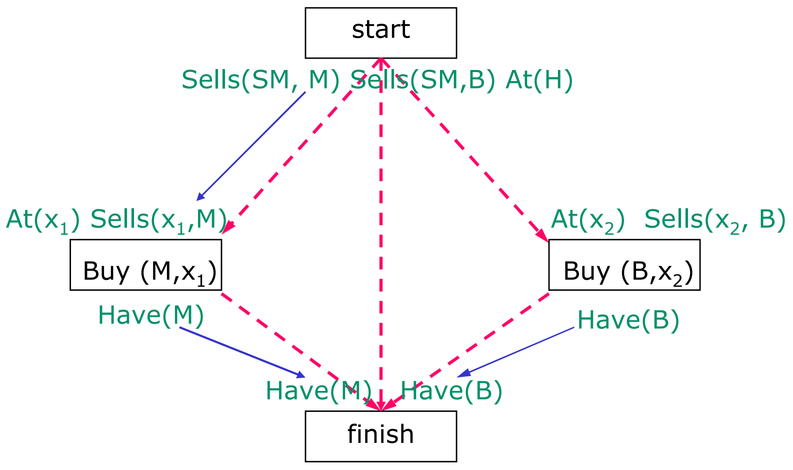
## PO Plan Example



Now we add a causal link and temporal constraints, just as we did for the buy milk step.

## PO Plan Example

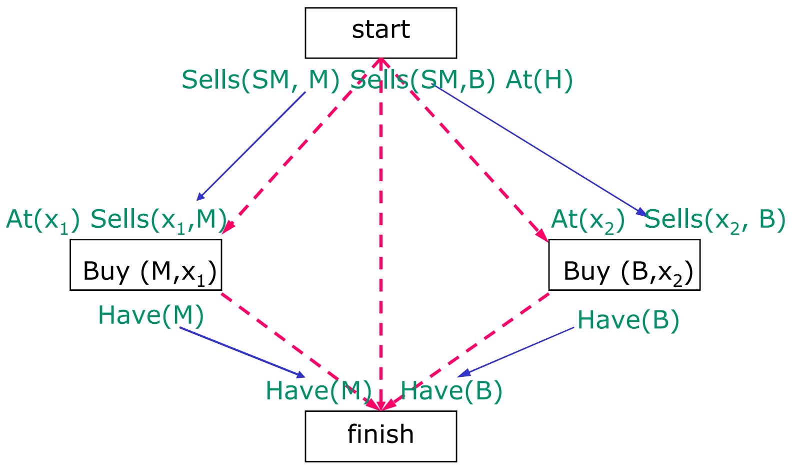
$x_1$ = SM



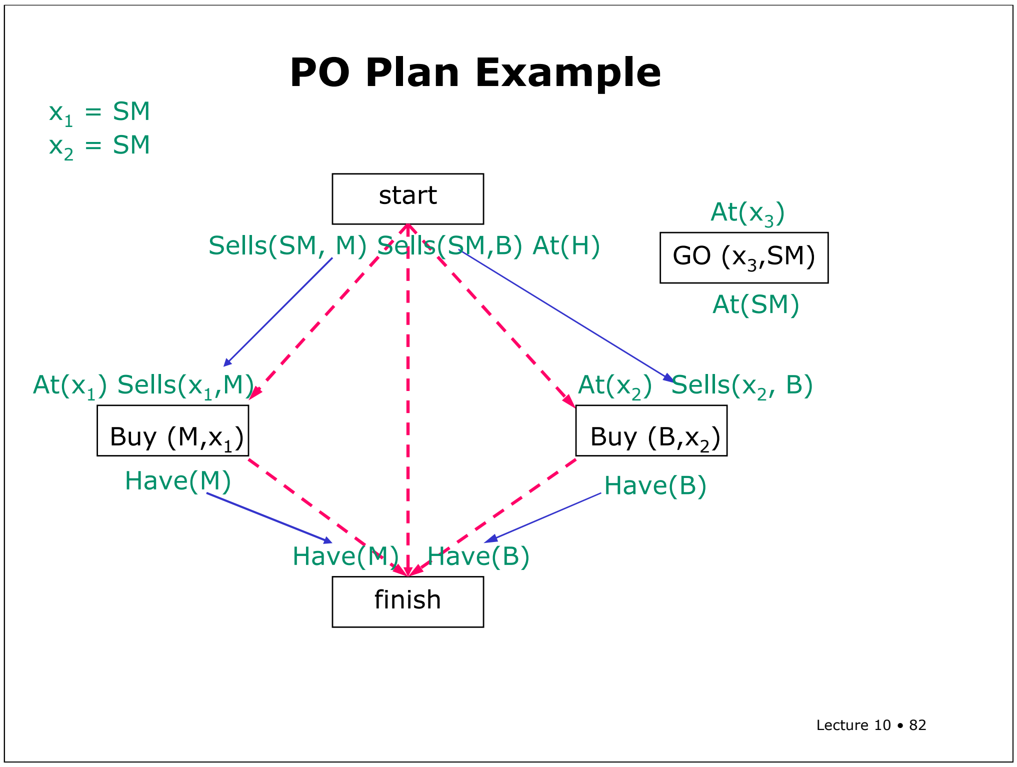
Now, a relatively straightforward thing to do is satisfy sells(x1,Milk) by constraining x1 to be the supermarket. We add a variable binding constraint, saying that x1 is equal to the supermarket. And that allows us to put a causal link between Sells(SM,M) in the effects of start, and the precondition here on buy.

## PO Plan Example

$x_1$ = SM
$x_2$ = SM



Similarly, we can satisfy sells(x2,Bananas) by adding a variable binding constraint that x2 must be the supermarket, and adding a causal link.

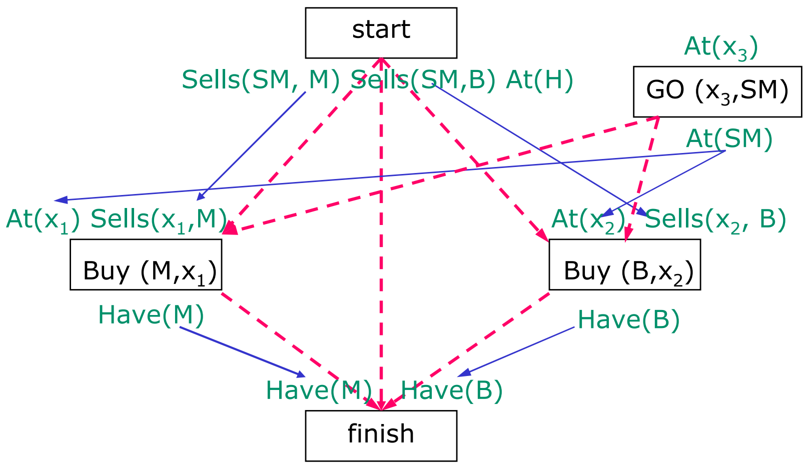


## PO Plan Example

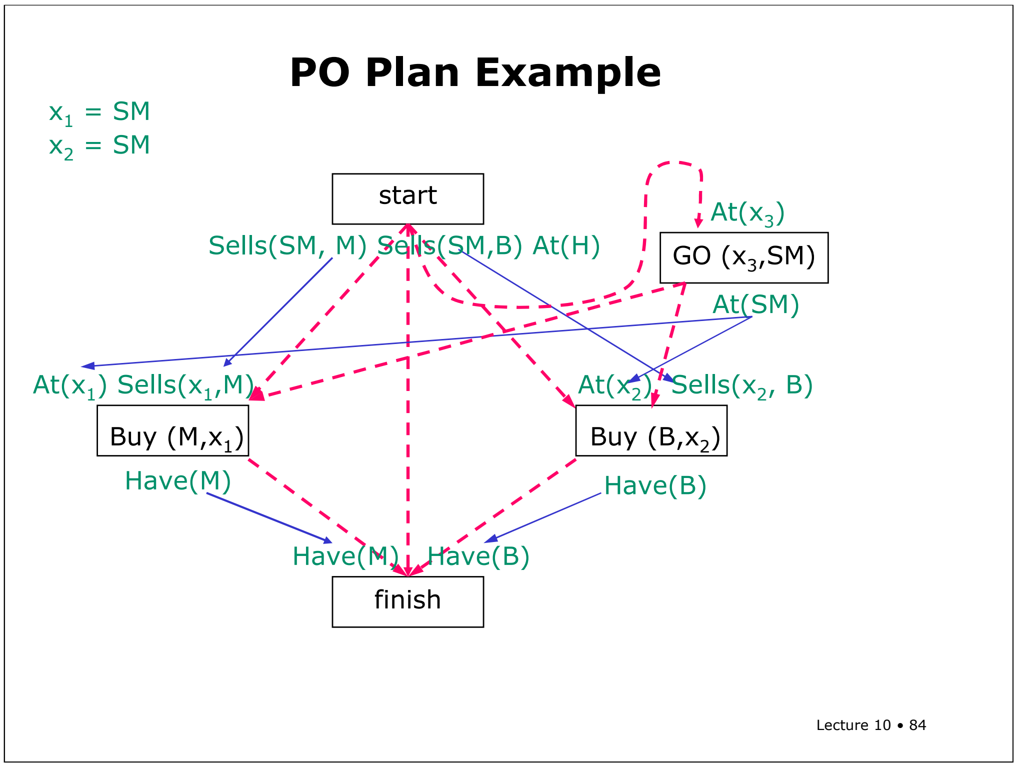
Now, the only preconditions that remain unsatisfied are at(x1) and at(x2). Since x1 and x2 are both constrained to be the supermarket, it seems like we should add a step to go to the supermarket.

## PO Plan Example

$x_1$ = SM
$x_2$ = SM



The effect of at(SM) can be used to satisfy both preconditions, so we add causal links to the at preconditions and temporal links to the buy actions.



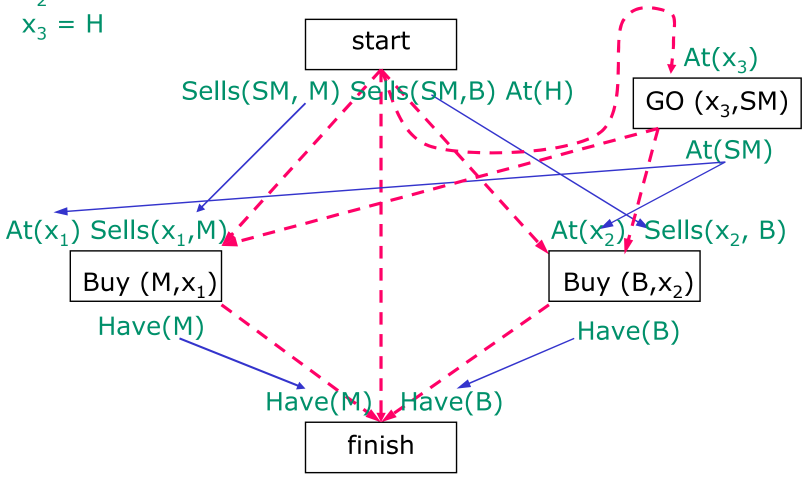
## PO Plan Example

$x_1$ = SM
$x_2$ = SM

And we add another temporal constraint to force this step to come after start.

## PO Plan Example

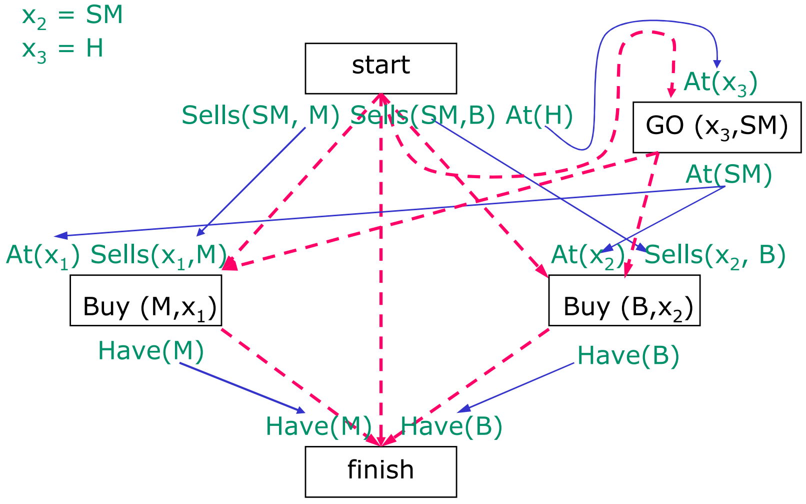
$x_1$ = SM
$x_2$ = SM
$x_3$ = H



Now, the At(x3) precondition can be satisfied by adding a variable binding constraint to force x3 to be home.

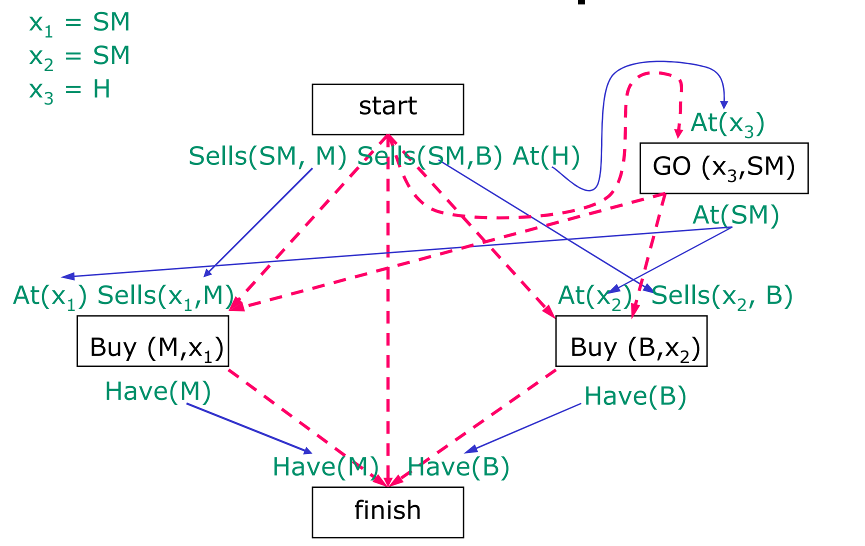
## PO Plan Example

$x_1$ = SM
$x_2$ = SM
$x_3$ = H



We can add a causal link from the At(home) effect of start to this precondition, and we’re done!

## PO Plan Example



If you look at this plan, you find that it has in fact followed the "least commitment" strategy and left open the question of what order to buy the milk and bananas in.

If the plan is complete, according to the definition we saw above, but the temporal constraints do not represent a partial order, then any total order of the steps that is consistent with the steps will be a correct plan that satisfies all the goals.

Next time, we'll write down the algorithm for doing partial order planning and go through a couple of examples.