Planning

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

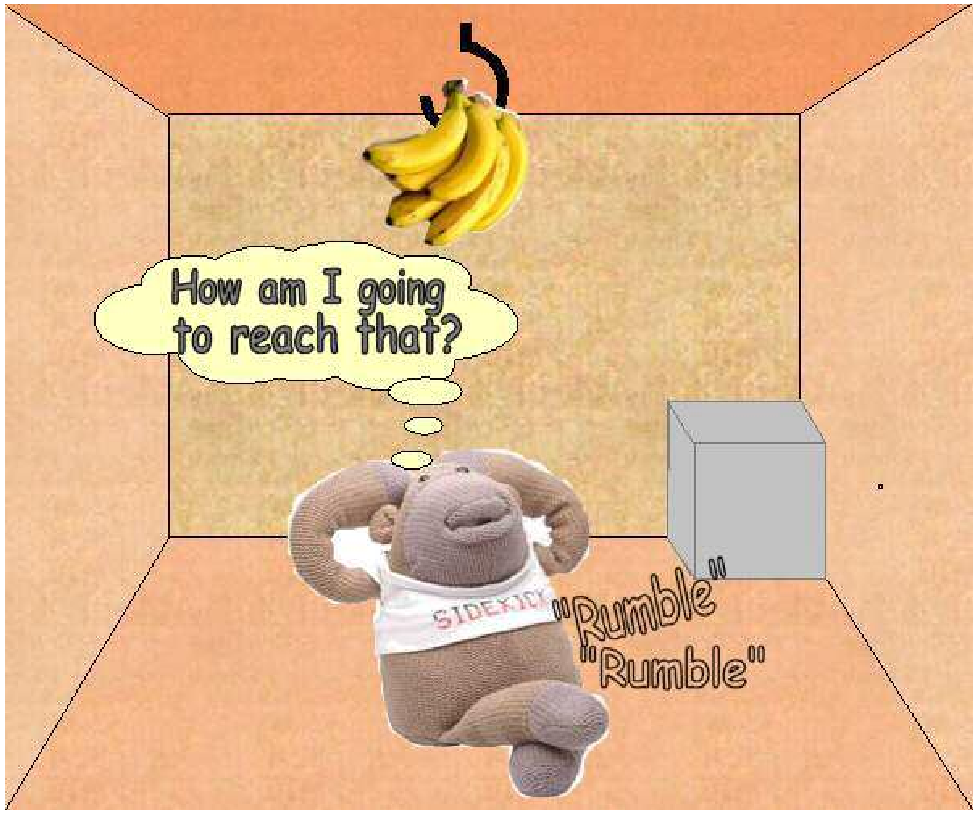
Lecture 15

18/11/04

Contents

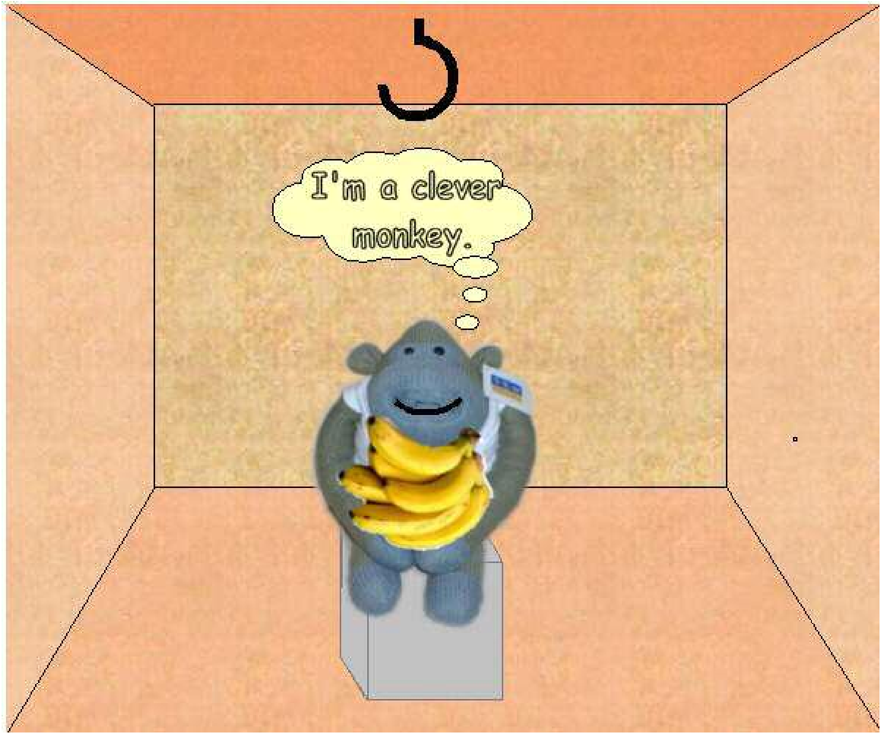
Monkey & Bananas

- A hungry monkey is in a room. Suspended from the roof, just out of his reach, is a bunch of bananas. In the corner of the room is a box. The monkey desperately wants the bananas but he can't reach them. What shall he do?



Monkey & Bananas (2)

- After several unsuccessful attempts to reach the bananas, the monkey walks to the box, pushes it under the bananas, climbs on the box, picks the bananas and eats them.



- The hungry monkey is now a happy monkey.

Planning

- To solve this problem the monkey needed to devise a plan, *a sequence of actions that would allow him to reach the desired goal.*
- Planning is a topic of traditional interest in Artificial Intelligence as it is an important part of many different AI applications, such as robotics and intelligent agents.
- To be able to plan, a system needs to be able to reason about the individual and cumulative effects of a series of actions. This is a skill that is only observed in a few animal species and only mastered by humans.
- The planning problems we will be discussing today are mostly Toy-World problems but they can be scaled up to real-world problems such as a robot negotiating a space.

Planning vs. Problem Solving

- Planning and problem solving (Search) are considered as different approaches even though they can often be applied to the same problem.
- Basic problem solving (as discussed in the Search lectures) searches a *state-space* of possible *actions*, starting from an *initial state* and following any path that it believes will lead it the *goal state*.
- Planning is distinct from this in three key ways:
  1. Planning "opens up" the representation of states, goals and actions so that the planner can deduce direct connections between states and actions.
  2. The planner does not have to solve the problem in order (from initial to goal state) it can suggest actions to solve any sub-goals at anytime.
  3. Planners assume that most parts of the world are independent so they can be stripped apart and solved individually (turning the problem into practically sized chunks).

Planning using STRIPS

- The "classical" approach most planners use today is derived from the STRIPS language.
- STRIPS was devised by SRI in the early 1970s to control a robot called Shakey.
- Shakey's task was to negotiate a series of rooms, move boxes, and grab objects.
- The STRIPS language was used to derive plans that would control Shakey's movements so that he could achieve his goals.
- The STRIPS language is very simple but expressive language that lends itself to efficient planning algorithms.
- The representation we will use in Prolog is derived from the original STRIPS representation.

Shakey

- Shakey.ram

STRIPS Representation

- Planning can be considered as a logical inference problem:
  - a plan is inferred from facts and logical relationships.
- STRIPS represented planning problems as a series of *state descriptions* and *operators* expressed in first-order predicate logic.

State descriptions represent the state of the world at three points during the plan:

- *Initial state*, the state of the world at the start of the problem;
- *Current state*, and
- *Goal state*, the state of the world we want to get to.

Operators are actions that can be applied to change the state of the world.

- Each operator has *outcomes* i.e. how it affects the world.
- Each operator can only be applied in certain circumstances. These are the *preconditions* of the operator.

Planning in Prolog

- As STRIPS uses a logic based representation of states it lends itself well to being implemented in Prolog.
- To show the development of a planning system we will implement the Monkey and Bananas problem in Prolog using STRIPS.
- When beginning to produce a planner there are certain *representation considerations* that need to be made:
  - How do we represent the state of the world?
  - How do we represent operators?
  - Does our representation make it easy to:
    - check preconditions;
    - alter the state of the world after performing actions; and
    - recognise the goal state?

Representing the World

- In the M&B problem we have:
  - *objects*: a monkey, a box, the bananas, and a floor.
  - *locations*: we'll call them a, b, and c.
  - *relations of objects to locations*. For example:
    - the monkey is at location a;
    - the monkey is on the floor;
    - the bananas are hanging;
    - the box is in the same location as the bananas.
- To represent these relations we need to choose appropriate predicates and arguments:
    - at(monkey,a).
    - on(monkey,floor).
    - status(bananas,hanging).
    - at(box,X), at(bananas,X).

Initial and Goal State

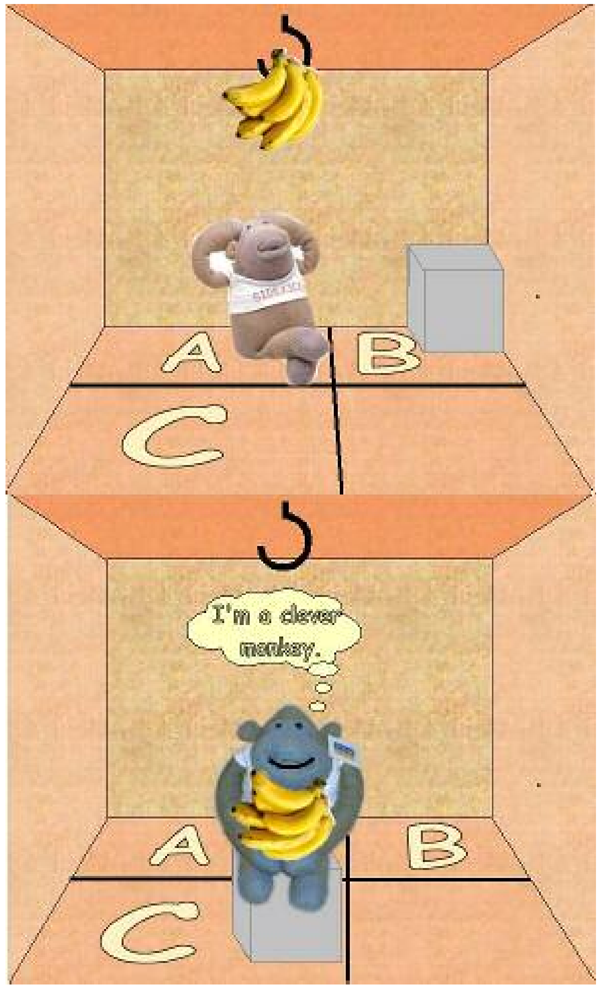
- Once we have decided on appropriate state predicates we need to represent the Initial and Goal states.
- Initial State:

```
on(monkey, floor),
on(box, floor),
at(monkey, a),
at(box, b),
at(bananas, c),
status(bananas, hanging).
```

- Goal State:

```
on(monkey, box),
on(box, floor),
at(monkey, c),
at(box, c),
at(bananas, c),
status(bananas, grabbed).
```

- Only this last state can be known without knowing the details of the Plan (i.e. how we're going to get there).



Representing Operators

- STRIPS operators are defined as:
  - **NAME**: How we refer to the operator e.g. go(Agent, From, To).
  - **PRECONDITIONS**: What states need to hold for the operator to be applied. e.g. [at(Agent, From)].
  - **ADD LIST**: What new states are added to the world as a result of applying the operator e.g. [at(Agent, To)].
  - **DELETE LIST**: What old states are removed from the world as a result of applying the operator. e.g. [at(Agent, From)].
- We will specify operators within a Prolog predicate `opn/4`:

`opn( go(Agent,From,To),`	← Name
`     [at(Agent, From)],`	← Preconditions
`     [at(Agent, To)],`	← Add List
`     [at(Agent, From)] ).`	← Delete List

The Frame Problem

- When representing operators we make the assumption that the only effects our operator has on the world are those specified by the add and delete lists.
- In real-world planning this is a hard assumption to make as we can never be absolutely certain of the extent of the effects of an action.
  - This is known in AI as the *Frame Problem.*
- Real-World systems, such as Shakey, are notoriously difficult to plan for because of this problem. Plans must constantly adapt based on incoming sensory information about the new state of the world otherwise the operator preconditions will no longer apply.
- The planning domains we will be working in our Toy-Worlds so we can assume that our framing assumptions are accurate.

All Operators

Operator	Preconditions	Delete List	Add List
go(X,Y)	at(monkey,X)	at(monkey,X)	at(monkey,Y)
	on(monkey, floor)		
push(B,X,Y)	at(monkey,X)	at(monkey,X)	at(monkey,Y)
	at(B,X)	at(B,X)	at(B,Y)
	on(monkey,floor)		
	on(B,floor)		
climb_on(B)	at(monkey,X)	on(monkey,floor)	on(monkey,B)
	at(B,X)		
	on(monkey,floor)		
	on(B,floor)		
grab(B)	on(monkey,box)	status(B,hanging)	status(B,grabbed)
	at(box,X)		
	at(B,X)		
	status(B,hanging)		

Finding a solution

1. Look at the state of the world:
   - Is it the goal state? If so, the list of operators so far is the plan to be applied.
   - If not, go to Step 2.
2. Pick an operator:
   - Check that it has not already been applied (to stop looping).
   - Check that the preconditions are satisfied.

   If either of these checks fails, backtrack to get another operator.
3. Apply the operator:
   1. Make changes to the world: delete from and add to the world state.
   2. Add operator to the list of operators already applied.
   3. Go to Step 1.

Finding a solution in Prolog

- The main work of generating a plan is done by the **`solve/4`** predicate.

```
     % First check if the Goal states are a subset of the current state.
solve(State, Goal, Plan, Plan):-
     is_subset(Goal, State)

solve(State, Goal, Sofar, Plan):-
     opn(Op, Precons, Delete, Add),      % get first operator
     \+ member(Op, Sofar),               % check for looping
     is_subset(Precons, State),          % check preconditions hold
     delete_list(Delete, State, Remainder),   % delete old states
     append(Add, Remainder, NewState),        % add new states
     solve(NewState, Goal, [Op|Sofar], Plan). % recurse
```

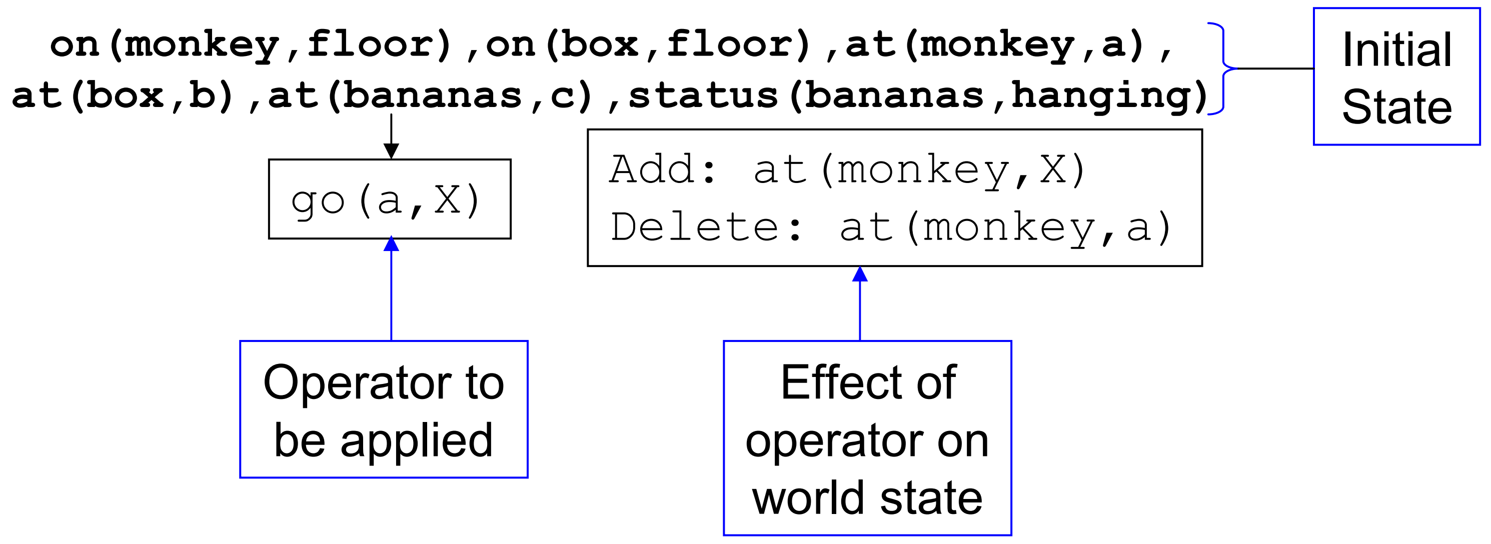
- On first glance this seems very similar to a normal depth-first search algorithm (unifies with first possible move and pursues it)

Why use operators?

- In fact, **`solve/4` is** performing depth-first search through the space of possible actions but because actions are represented as operators instead of predicate definitions the result is significantly different:
  - A range of different actions can be selected using the same predicate `opn/4`.
  - The effect an action has on the world is made explicit. This allows actions to be chosen based on the preconditions of sub-goals: directing our search towards the goal rather than searching blindly.
  - Representing the state of the world as a list allows it to be dynamically modified without the need for asserting and retracting facts from the database.
    - solve/4 tries multiple operators when forced to backtrack due to failure. Database manipulation does not revert back to the original state during backtracking so we couldn't use it to generate a plan in this manner.

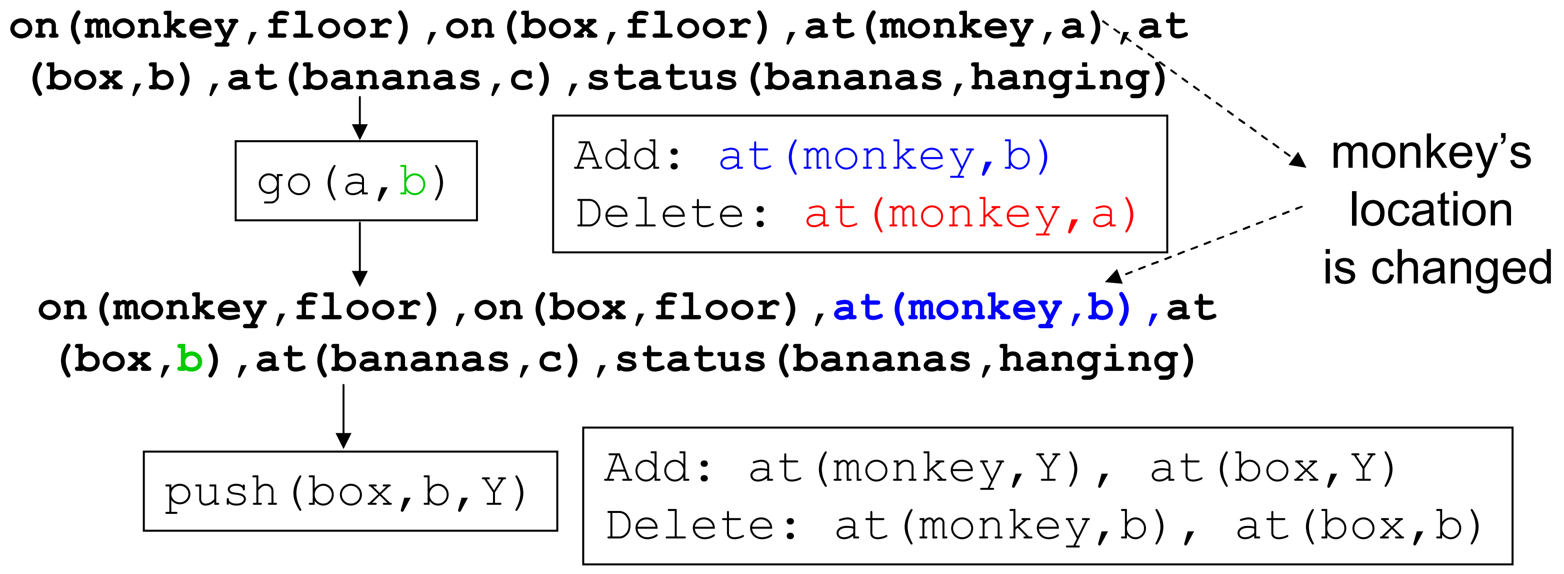
Representing the plan

- `solve/4` is a *linear planner*: it starts at the initial state and tries to find a series of operators that have the cumulative effect of adding the goal state to the world.
- We can represent its behaviour as a flow-chart.


on(monkey,floor),on(box,floor),at(monkey,a),
at(box,b),at(bananas,c),status(bananas,hanging)
Initial State
go(a,X)
Add: at(monkey,X)
Delete: at(monkey,a)
Operator to be applied
Effect of operator on world state

- When an operator is applied the information in its preconditions is used to instantiate as many of its variables as possible.
- Uninstantiated variables are carried forward to be filled in later.

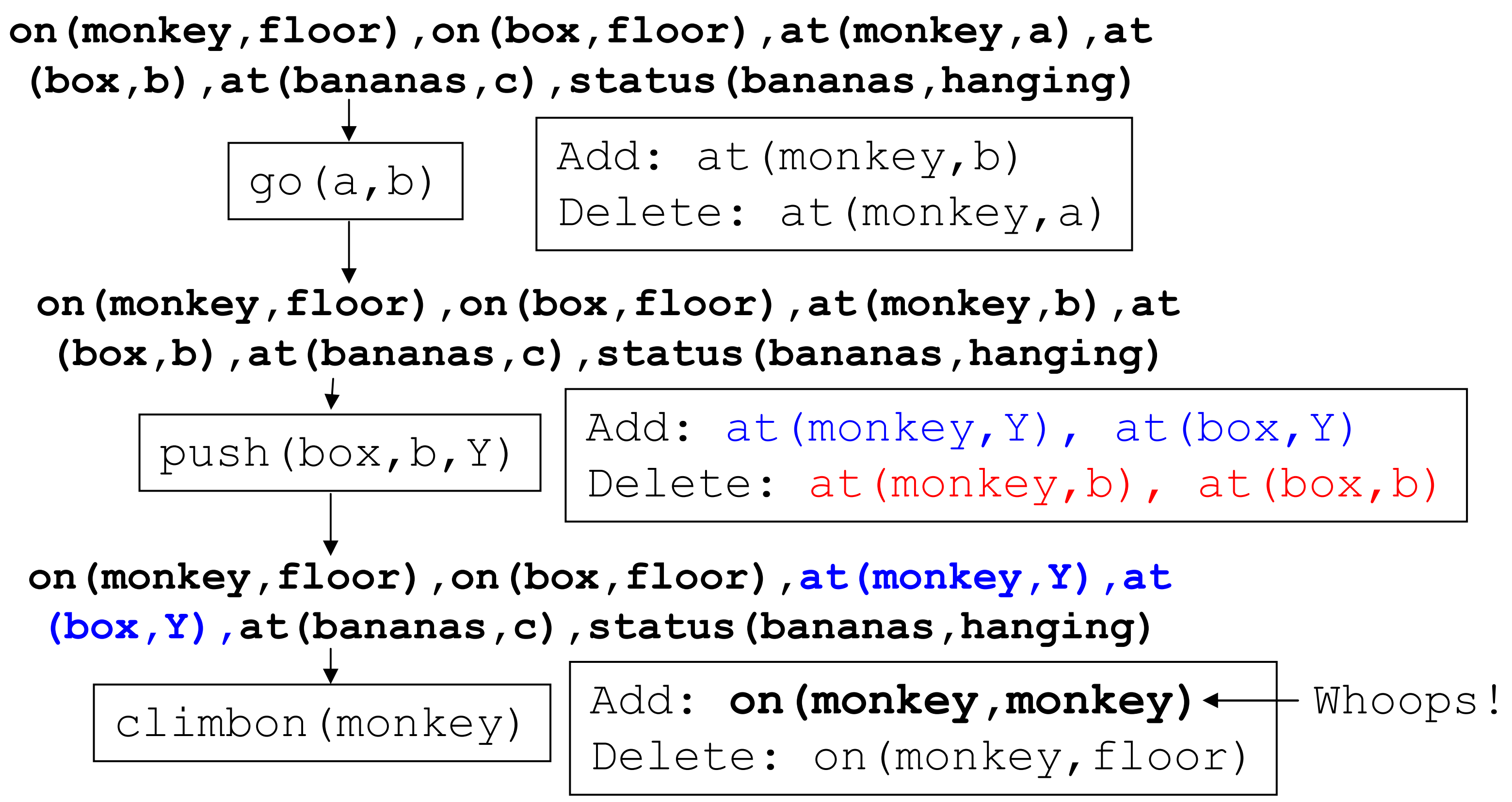
Representing the plan (2)



```
on(monkey,floor),on(box,floor),at(monkey,a),at
 (box,b),at(bananas,c),status(bananas,hanging)

        go(a,b)        Add:    at(monkey,b)
                       Delete: at(monkey,a)

on(monkey,floor),on(box,floor),at(monkey,b),at
 (box,b),at(bananas,c),status(bananas,hanging)

     push(box,b,Y)     Add:    at(monkey,Y), at(box,Y)
                       Delete: at(monkey,b), at(box,b)
```

- `solve/4` chooses the push operator this time as it is the next operator after `go/2` stored in the database and `go(a,X)` is now stored in the SoFar list so `go(X,Y)` can't be applied again.
- The preconditions of `push/3` require the monkey to be in the same location as the box so the variable location, X, from the last move inherits the value **b**.

Representing the plan (3)



```
on(monkey,floor),on(box,floor),at(monkey,a),at
 (box,b),at(bananas,c),status(bananas,hanging)
          |
        go(a,b)        Add:    at(monkey,b)
          |            Delete: at(monkey,a)
          v
on(monkey,floor),on(box,floor),at(monkey,b),at
 (box,b),at(bananas,c),status(bananas,hanging)
          |
     push(box,b,Y)     Add:    at(monkey,Y), at(box,Y)
          |            Delete: at(monkey,b), at(box,b)
          v
on(monkey,floor),on(box,floor),at(monkey,Y),at
 (box,Y),at(bananas,c),status(bananas,hanging)
          |
   climbon(monkey)     Add:    on(monkey,monkey) <-- Whoops!
                       Delete: on(monkey,floor)
```

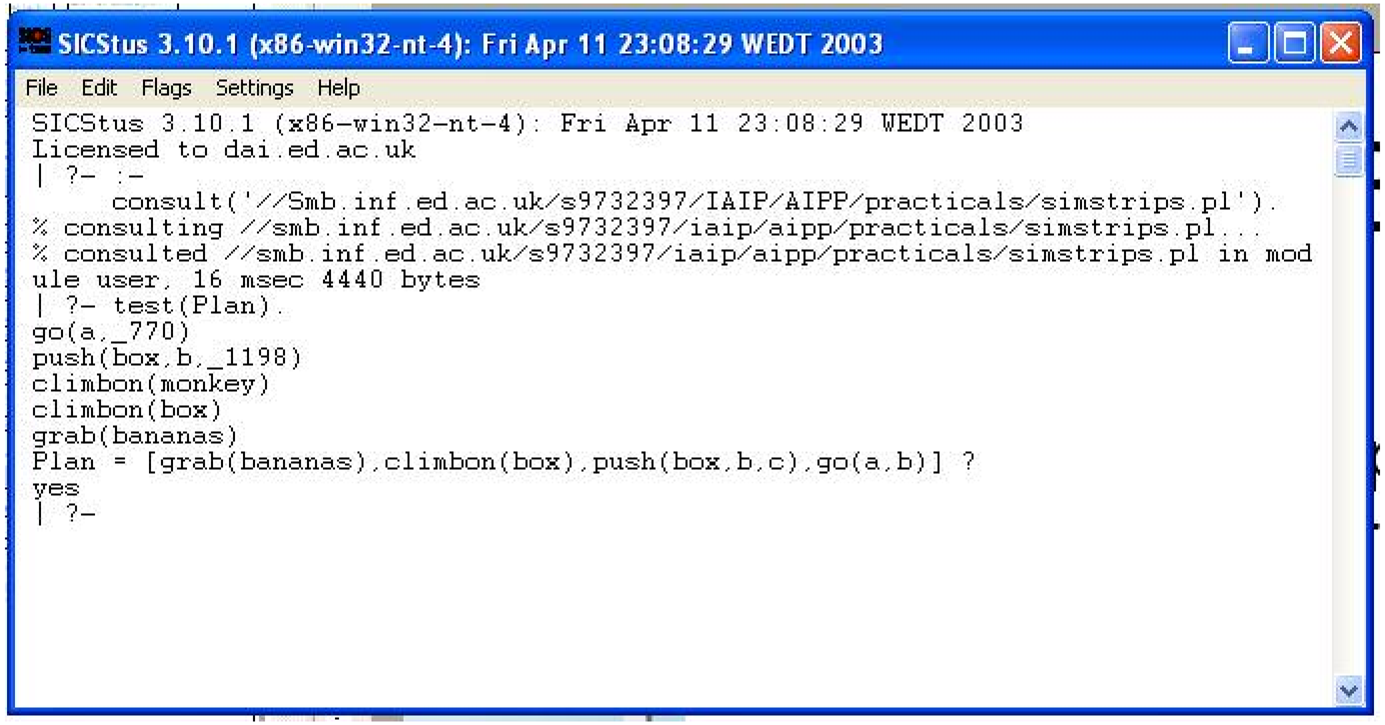
- The operator only specifies that the monkey must climb on something in the same location; not that it must be something other than itself!
- This instantiation fails once it tries to satisfy the preconditions for the **`grab/1`** operator. **`solve/4`** backtracks and matches **`climbon(box)`** instead.

Representing the plan (4)

```
on(monkey,floor),on(box,floor),at(monkey,a),at
(box,b),at(bananas,c),status(bananas,hanging)

        go(a,b)

on(monkey,floor),on(box,floor),at(monkey,b),at
(box,b),at(bananas,c),status(bananas,hanging)

        push(box,b,Y)        Y = c

on(monkey,floor),on(box,floor),at(monkey,Y),at
(box,Y),at(bananas,c),status(bananas,hanging)

        climbon(box)

on(monkey,box),on(box,floor),at(monkey,Y),at(b
ox,Y),at(bananas,c),status(bananas,hanging)

        grab(bananas)        Y = c

on(monkey,box),on(box,floor),at(monkey,c),at(b
ox,c),at(bananas,c),status(bananas,grabbed)   ←  GOAL
```

For the monkey to grab the bananas it must be in the same location, so the variable location **Y** inherits **c**. This creates a complete plan.

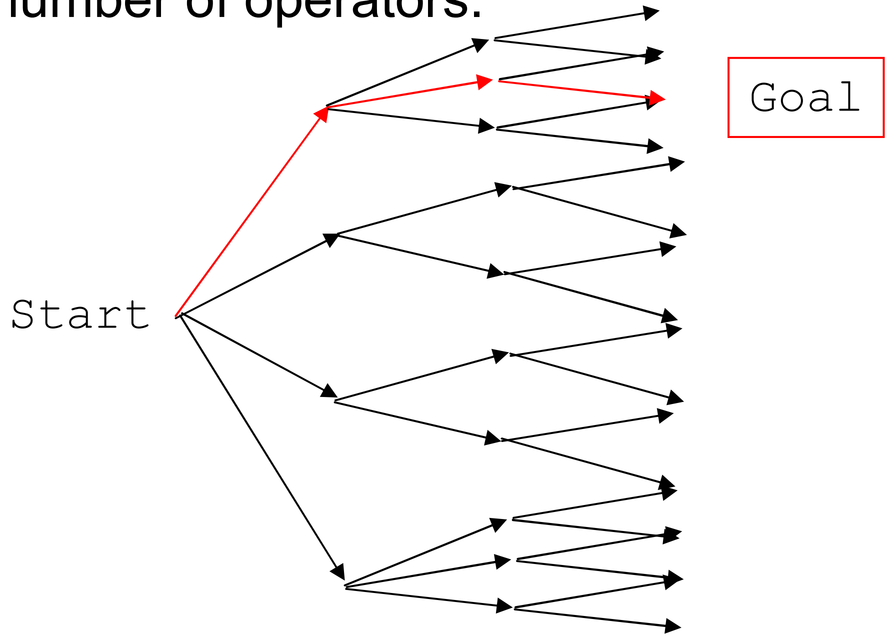
Monkey & Bananas Program

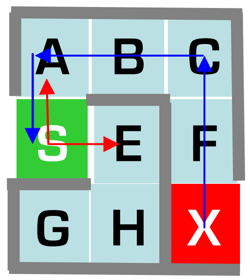

SICStus 3.10.1 (x86-win32-nt-4): Fri Apr 11 23:08:29 WEDT 2003
File
Edit
Flags
Settings
Help

```
SICStus 3.10.1 (x86-win32-nt-4): Fri Apr 11 23:08:29 WEDT 2003
Licensed to dai.ed.ac.uk
| ?- :-
     consult('//Smb.inf.ed.ac.uk/s9732397/IAIP/AIPP/practicals/simstrips.pl').
% consulting //smb.inf.ed.ac.uk/s9732397/iaip/aipp/practicals/simstrips.pl...
% consulted //smb.inf.ed.ac.uk/s9732397/iaip/aipp/practicals/simstrips.pl in mod
ule user, 16 msec 4440 bytes
| ?- test(Plan).
go(a,_770)
push(box,b,_1198)
climbon(monkey)
climbon(box)
grab(bananas)
Plan = [grab(bananas),climbon(box),push(box,b,c),go(a,b)] ?
yes
| ?-
```

Inefficiency of forwards planning

- Linear planners like this, that progress from the initial state to the goal state can be unsuitable for problems with a large number of operators.





- Searching backwards from the Goal state usually eliminates spurious paths.
  - This is called **Means Ends Analysis.**

Means Ends Analysis

- The *Means* are the available *actions*.
- The *Ends* are the *goals* to be achieved.
- To solve a list of *Goals* in state *State*, leading to state *FinalState*, do:
  - If all the *Goals* are true in *State* then *FinalState* = *State*. *Otherwise* do the following:
    1. Select a still unsolved *Goal* from *Goals*.
    2. Find an *Action* that adds *Goal* to the current state.
    3. Enable *Action* by solving the preconditions of *Action*, giving *MidState*.
    4. *MidState* is then added as a new *Goal* to *Goals* and the program recurses to step 1.
  - i.e. we search backwards from the Goal state, generating new states from the preconditions of actions, and checking to see if these are facts in our initial state.

PROLOG

Means Ends Analysis (2)

- Means Ends Analysis will usually lead straight from the Goal State to the Initial State as the branching factor of the search space is usually larger going forwards compared to backwards.
- However, more complex problems can contain operators with overlapping Add Lists so the MEA would be required to choose between them.
  - It would require *heuristics*.
- Also, linear planners like these will blindly pursue sub-goals without considering whether the changes they are making undermine future goals.
  - they need someway of *protecting their goals*.
- Both of these issues will be discussed in the next lecture.

Summary

- A Plan is a sequence of actions that changes the state of the world from an Initial state to a Goal state.
- Planning can be considered as a *logical inference problem*.
- *STRIPS* is a classic planning language.
  - It represents the state of the world as a list of facts.
  - *Operators* (actions) can be applied to the world if their preconditions hold.
    - The effect of applying an operator is to *add* and *delete* states from the world.
- A linear planner can be easily implemented in Prolog by:
  - representing operators as `opn(Name,[PreCons],[Add],[Delete]).`
  - choosing operators and applying them in a depth-first manner,
  - using backtracking-through-failure to try multiple operators.
- *Means End Analysis* performs backwards planning with is more efficient.