AI Planning
Planning Representation.

Eva Onaindia

Universitat Politècnica de València

Acknowledgements

Most of the slides used in this course are taken or are modifications from Dana Nau's lecture slides for the textbook *Automated Planning,* licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License:

http://creativecommons.org/licenses/by-nc-sa/2.0/
http://creativecommons.org/licenses/by/3.0/es/

I would like to gratefully acknowledge Dana Nau's contributions and thank him for generously permitting me to use aspects of his presentation material.

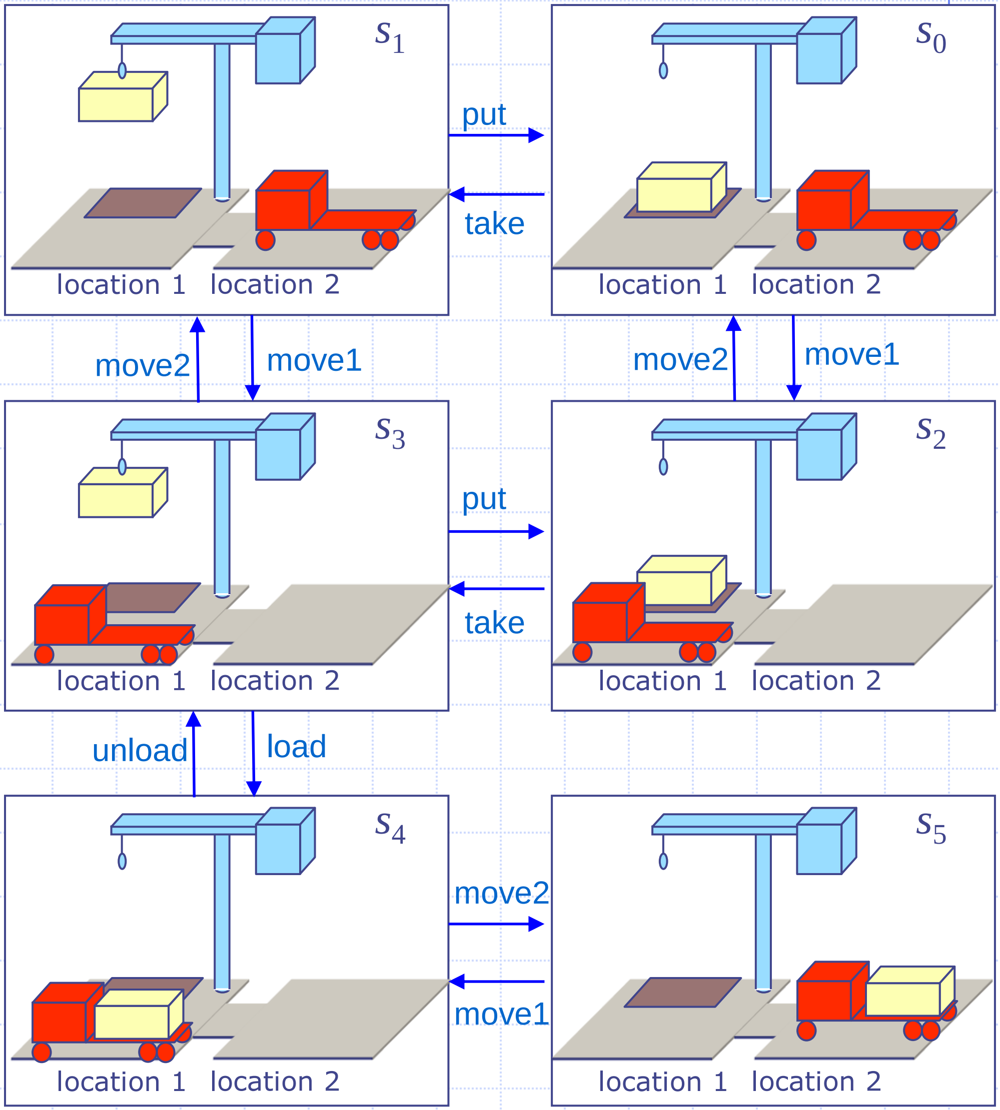
Outline.

- Planning representation
- Classical representation (ex. DWR and blocks world)
- State-variable representation (ex. DWR and blocks world)
- Comparisons
- PDDL: Planning Domain Description Language

Quick Review of Classical Planning

- Classical planning requires all eight of the restrictive assumptions:

  A0: Finite
  A1: Fully observable
  A2: Deterministic
  A3: Static
  A4: Attainment goals
  A5: Sequential plans
  A6: Implicit time
  A7: Offline planning

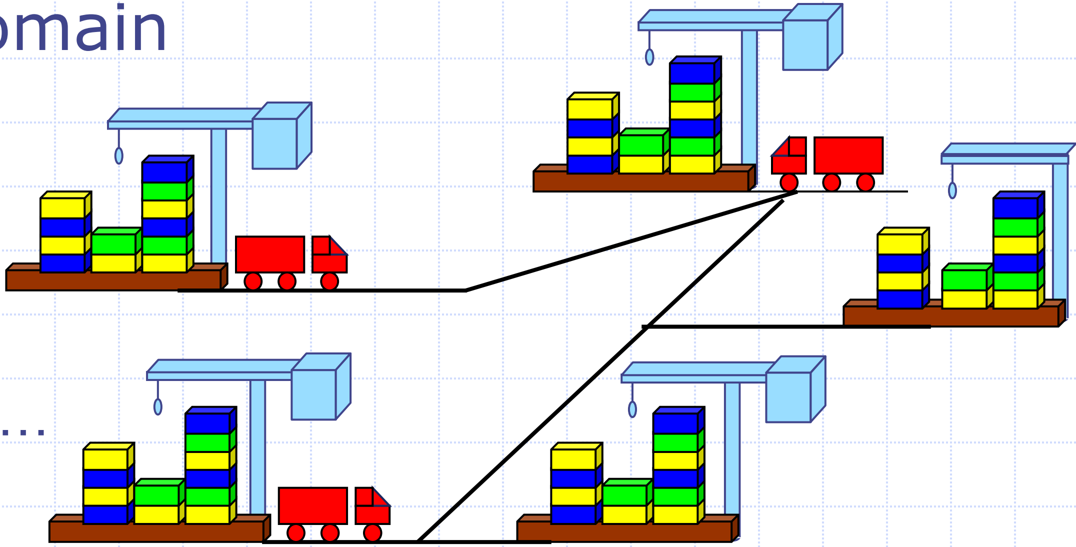


Planning representation. Motivation.

- In most problems, far too many states to try to represent all of them explicitly as s_0, s_1, s_2, …
- Represent each state as a set of features
  - e.g.,
    - a vector of values for a set of variables
    - a set of ground atoms in some first-order language *L*
- Define a set of *operators* that can be used to compute state-transitions
- Don’t give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed

Classical Representation

- Start with a first-order language
    - Language of first-order logic
  - Restrict it to be *function-free*
    - Finitely many predicate symbols and constant symbols, but *no* function symbols
- Example: the DWR domain
  - Locations: loc1, loc2, …
  - Containers: c1, c2, …
  - Pallet: p1, p2, …
  - Robot carts: r1, r2, …
  - Cranes: crane1, crane2, …



Classical Representation

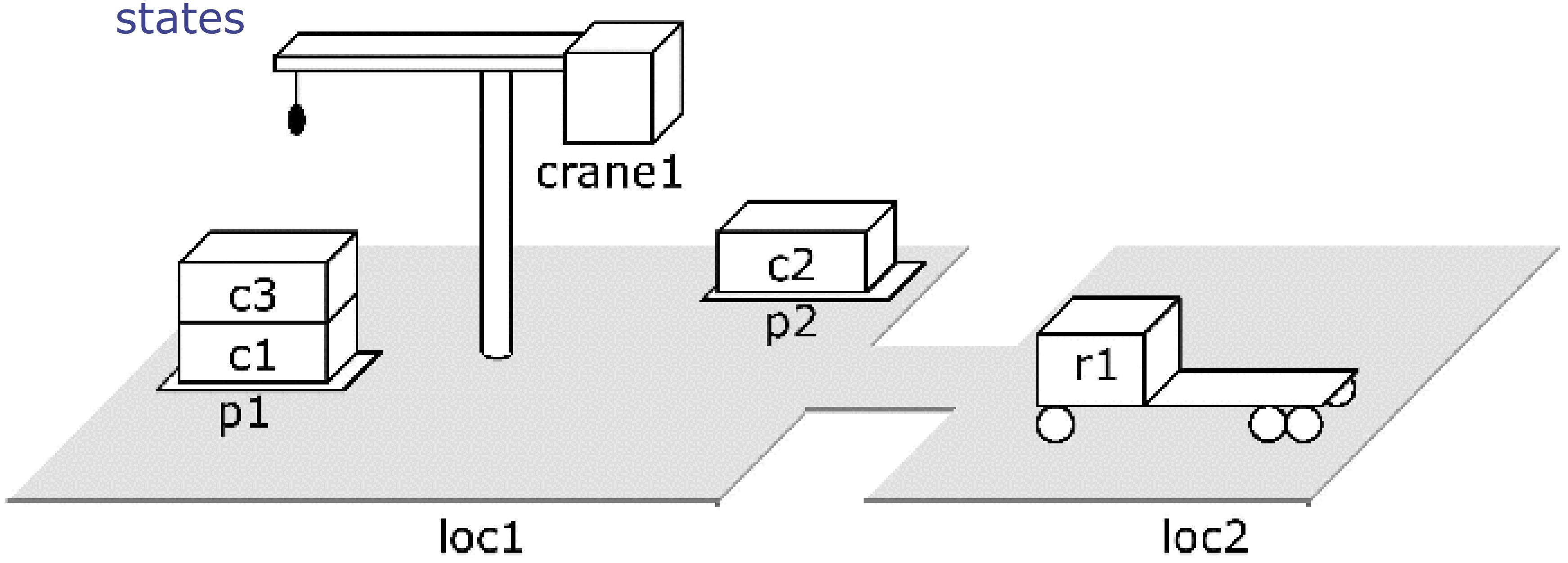
- *Atom*: predicate symbol and args
  - Use these to represent both fixed and dynamic relations

adjacent(l,l')	attached(p,l)
occupied(l)	at(r,l)
loaded(r,c)	unloaded(r)
holding(k,c)	empty(k)
in(c,p)	on(c,c')
top(c,p)	belong(k,l)

- *Ground* expression: contains no variable symbols - e.g., in(c1,p3)
- *Unground* expression: at least one variable symbol - e.g., in(c1,x)
- *Substitution*: $\theta = \{x_1 \leftarrow v_1,\ x_2 \leftarrow v_2,\ \ldots,\ x_n \leftarrow v_n\}$
  - Each x_i is a variable symbol; each v_i is a term
- *Instance* of e: result of applying a substitution θ to e
  - Replace variables of e simultaneously, not sequentially

States

- *State*: a set *s* of ground atoms
  - The atoms represent the things that are true in one of Σ's states
  - Only finitely many ground atoms, so only finitely many possible states



s_1={attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

States

- Literal = ground atom
- Two types of literals: positive literals, negative literals
  - in(c1,p1): positive literal representing a true statement
  - ¬occupied(loc1): negative literal representing a true statement
- State with only positive literals => negation by failure, what is not explicitly represented is false
- State with positive and negative literals => explicit representation of true positive and negative information
  - s1={attached(p1,loc1), in(c1,p1), …, ¬occupied(loc1), ¬in(c1,p2), ¬in(c3,p2),¬in(c2,p1), ¬at(r1,loc1) …}

Operators

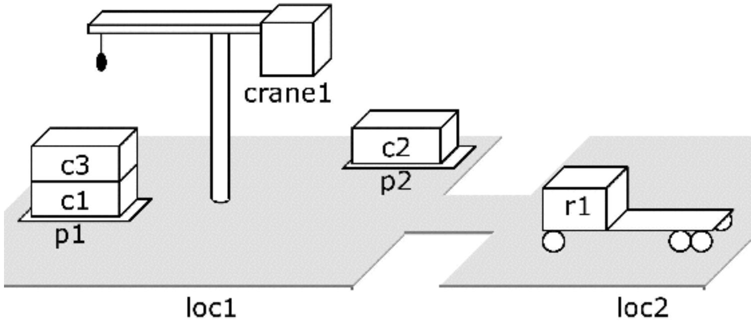
- *Operator*: a triple o=(name(o), precond(o), effects(o))
  - precond(o): *preconditions*
    - literals that must be true in order to use the operator
  - effects(o): *effects*
    - literals the operator will make true
  - name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
    - n is an *operator symbol* - must be unique for each operator
    - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o
- Purpose of name(o) is so we can refer unambiguously to instances of o
- Rather than writing each operator as a triple, we'll usually write it in the following format:

- **Planning domain**: language plus operators
  - Corresponds to a set of state-transition systems
  - Example: operators for the DWR domain

$\mathsf{move}(r, l, m)$
;; robot r moves from location l to location m
precond: $\mathsf{adjacent}(l, m), \mathsf{at}(r, l), \neg\,\mathsf{occupied}(m)$
effects: $\mathsf{at}(r, m), \mathsf{occupied}(m), \neg\,\mathsf{occupied}(l), \neg\,\mathsf{at}(r, l)$

$\mathsf{load}(k, l, c, r)$
;; crane k at location l loads container c onto robot r
precond: $\mathsf{belong}(k, l), \mathsf{holding}(k, c), \mathsf{at}(r, l), \mathsf{unloaded}(r)$
effects: $\mathsf{empty}(k), \neg\,\mathsf{holding}(k, c), \mathsf{loaded}(r, c), \neg\,\mathsf{unloaded}(r)$

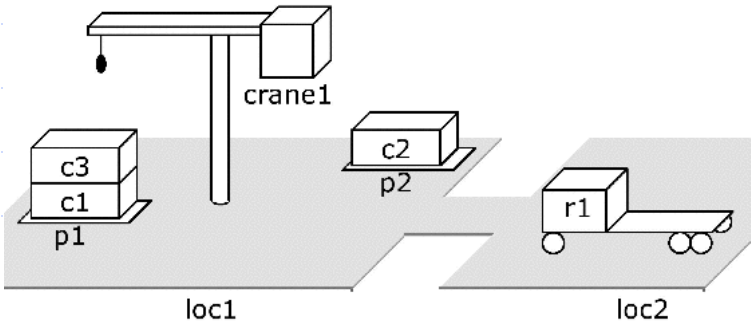
$\mathsf{unload}(k, l, c, r)$
;; crane k at location l takes container c from robot r
precond: $\mathsf{belong}(k, l), \mathsf{at}(r, l), \mathsf{loaded}(r, c), \mathsf{empty}(k)$
effects: $\neg\,\mathsf{empty}(k), \mathsf{holding}(k, c), \mathsf{unloaded}(r), \neg\,\mathsf{loaded}$



$\mathsf{put}(k, l, c, d, p)$
;; crane k at location l puts c onto d in pile p
precond: $\mathsf{belong}(k, l), \mathsf{attached}(p, l), \mathsf{holding}(k, c), \mathsf{top}(d, p)$
effects: $\neg\,\mathsf{holding}(k, c), \mathsf{empty}(k), \mathsf{in}(c, p), \mathsf{top}(c, p), \mathsf{on}(c, d), \neg\,\mathsf{top}(d, p)$

$\mathsf{take}(k, l, c, d, p)$
;; crane k at location l takes c off of d in pile p
precond: $\mathsf{belong}(k, l), \mathsf{attached}(p, l), \mathsf{empty}(k), \mathsf{top}(c, p), \mathsf{on}(c, d)$
effects: $\mathsf{holding}(k, c), \neg\,\mathsf{empty}(k), \neg\,\mathsf{in}(c, p), \neg\,\mathsf{top}(c, p), \neg\,\mathsf{on}(c, d), \mathsf{top}(d, p)$

Actions



take(k, l, c, d, p)

;; crane k at location l takes c off of d in pile p

precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)

effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

take(crane1,loc1,c3,c1,p1)

precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

- An *action* is a ground instance (via substitution) of an operator
- Note that an action's name identifies it unambiguously
  - take(crane1,loc1,c3,c1,p1)

Notation

- Let S be a set of literals. Then
  - S^+ = {atoms that appear positively in S}
  - S^- = {atoms that appear negatively in S} ;; implicitly or explicitly
- Let a be an operator or action. Then
  - precond$^+(a)$ = {atoms that appear positively in a's preconditions}
  - precond$^-(a)$ = {atoms that appear negatively in a's preconditions}
  - effects$^+(a)$ = {atoms that appear positively in a's effects}
  - effects$^-(a)$ = {atoms that appear negatively in a's effects}

take(k, l, c, d, p)

;; crane k at location l takes c off of d in pile p

precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)

effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

- effects$^+$(take(k,l,c,d,p)) = {holding(k,c), top(d,p)}
- effects$^-$(take(k,l,c,d,p)) = {empty(k), in(c,p), top(c,p), on(c,d)}

Applicability

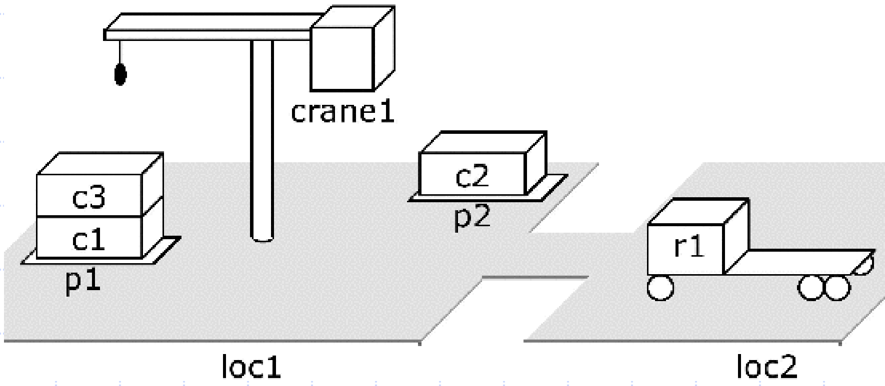
- An action *a* is *applicable* to a state *s* if *s* satisfies precond(*a*),
  - i.e., if precond$^+(a) \subseteq s$ and precond$^-(a) \cap s = \varnothing$

- An action:

take(crane1,loc1,c3,c1,p1)

precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)



- A state it's applicable to

s_1 = {**attached(p1,loc1)**, in(c1,p1), in(c3,p1), **top(c3,p1)**, **on(c3,c1)**, on(c1,p1), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,p2), **belong(crane1,loc1)**, **empty(crane1)**, adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}

Executing an Applicable Action

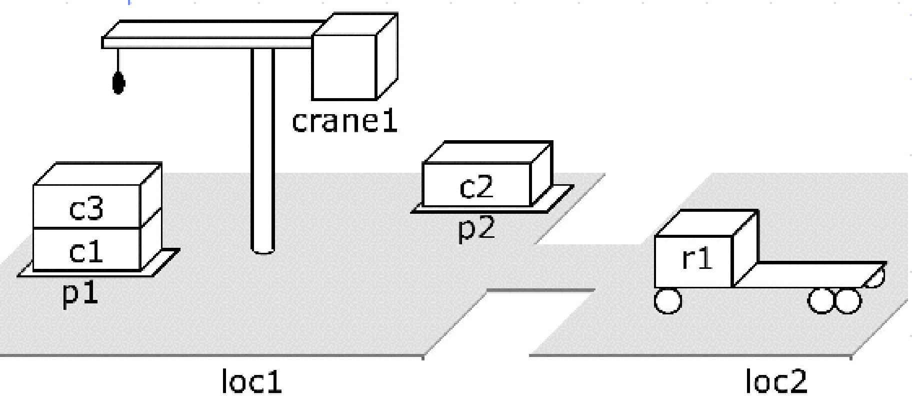
- Remove a's negative effects, and add a's positive effects

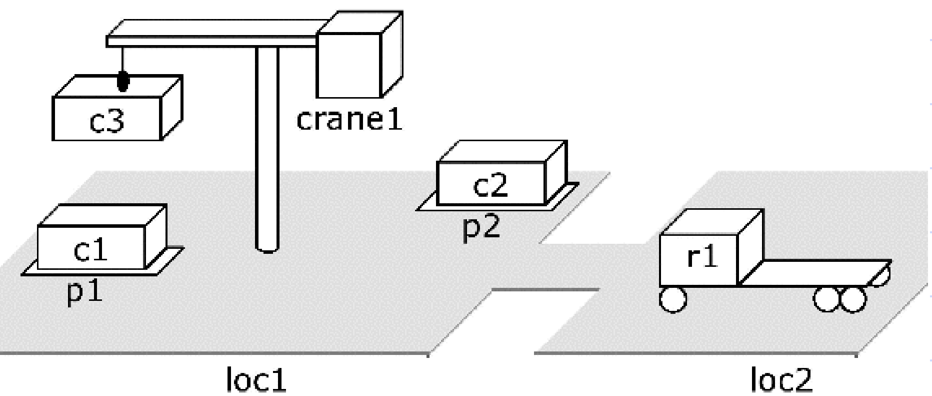
$\gamma(s,a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

take(crane1,loc1,c3,c1,p1)

precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)





s_2 = {attached(p1,loc1), in(c1,p1), *~~in(c3,p1)~~*, *~~top(c3,p1)~~*, *~~on(c3,c1)~~*, on(c1,p1), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,p2), belong(crane1,loc1), *~~empty(crane1)~~*, adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2, unloaded(r1), **holding(crane1,c3), top(c1,p1)**}

Planning Problems

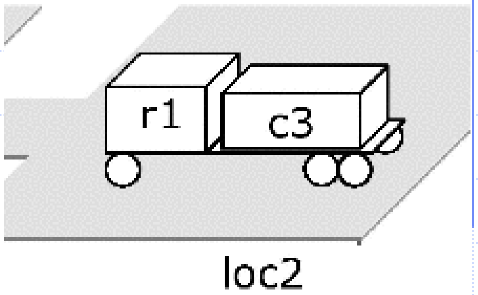
- Given a planning domain (language L, operators O)
  - *Statement* of a planning problem: a triple $P=(O,s_0,g)$
    - O is the collection of operators
    - s_0 is a state (the initial state)
    - g is a set of literals (the goal formula)
  - Planning problem: $\mathcal{P} = (\Sigma,s_0,S_g)$
    - s_0 = initial state
    - S_g = set of goal states
    - $\Sigma = (S,A,\gamma)$ is a state-transition system
    - S = {all sets of ground atoms in L}
    - A = {all ground instances of operators in O}
    - γ = the state-transition function determined by the operators
- I'll often say "planning problem" when I mean the statement of the problem

Plans and Solutions

- *Plan*: any sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is an instance of an operator in O
- The plan is a *solution* for $P=(O,s_0,g)$ if it is executable and achieves g
  - i.e., if there are states $s_0, s_1, \ldots, s_n$ such that
    - $\gamma(s_0,a_1) = s_1$
    - $\gamma(s_1,a_2) = s_2$
    - …
    - $\gamma(s_{n-1},a_n) = s_n$
    - s_n satisfies g

Example

$g_1 = \{$loaded(r1,c3), at(r1,loc2)$\}$



- Let $P_1 = (O, s_1, g_1)$, where
  - O = {the five DWR operators given earlier}

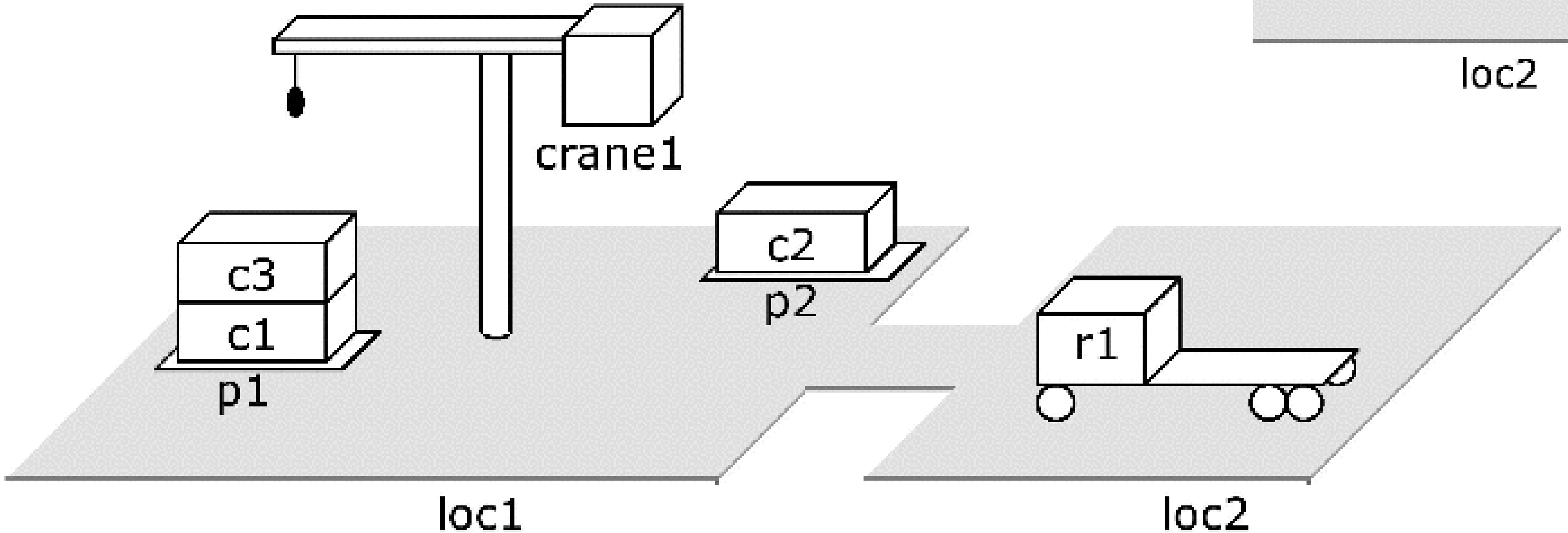
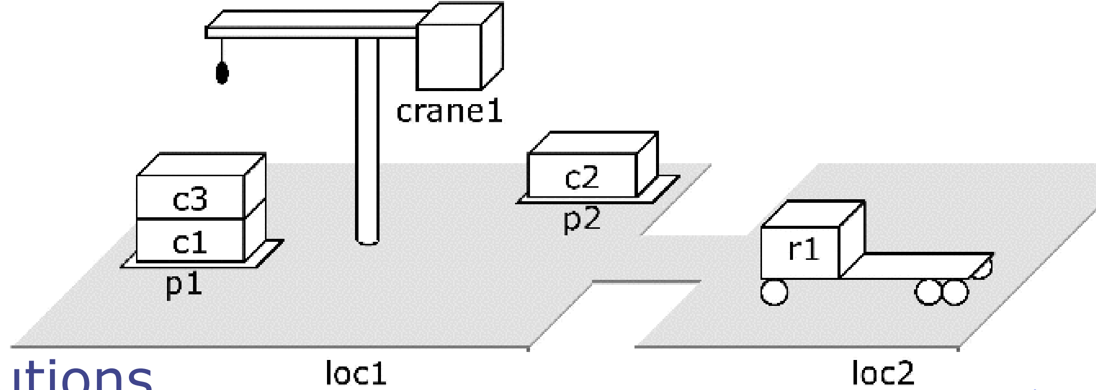


Figure 2.2: The DWR state s_1={attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

Example, continued

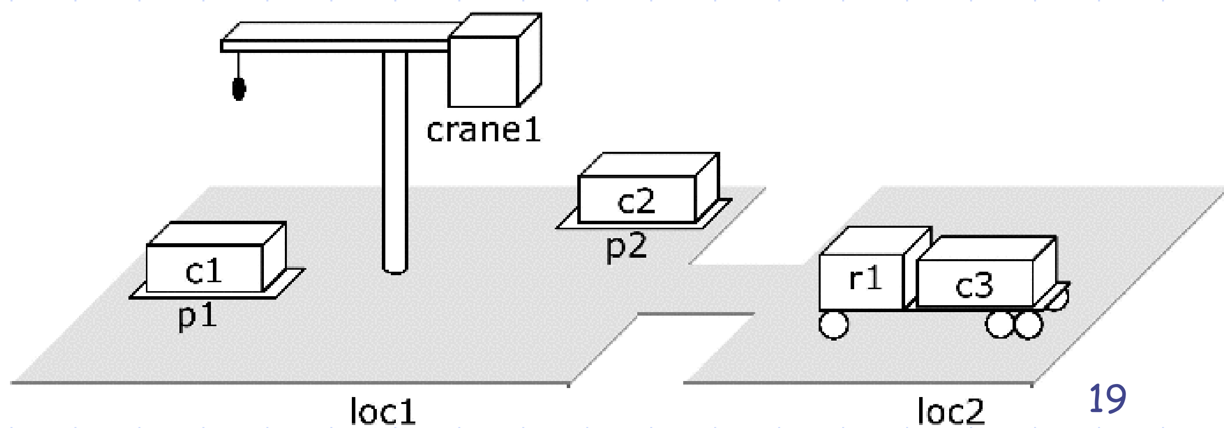


- P_1 has infinitely many solutions
- Here are three of them:

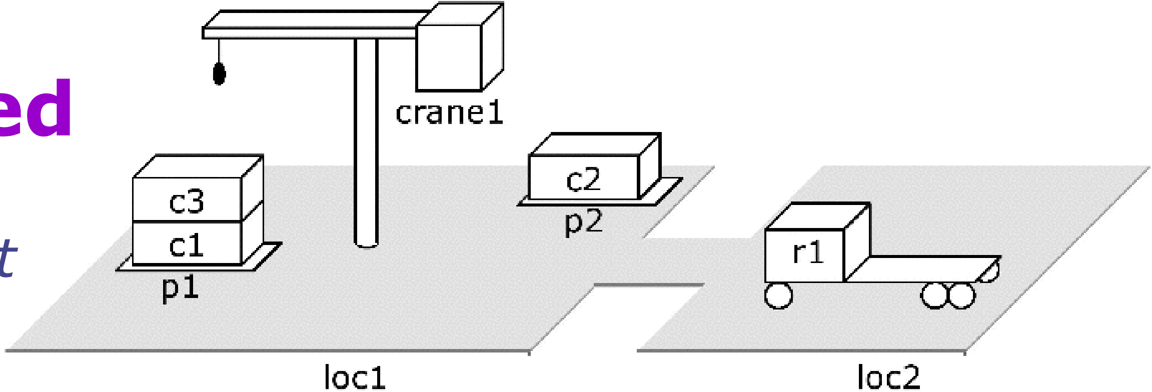
  ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

  ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

  ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
- They each produce this state:



Example, continued



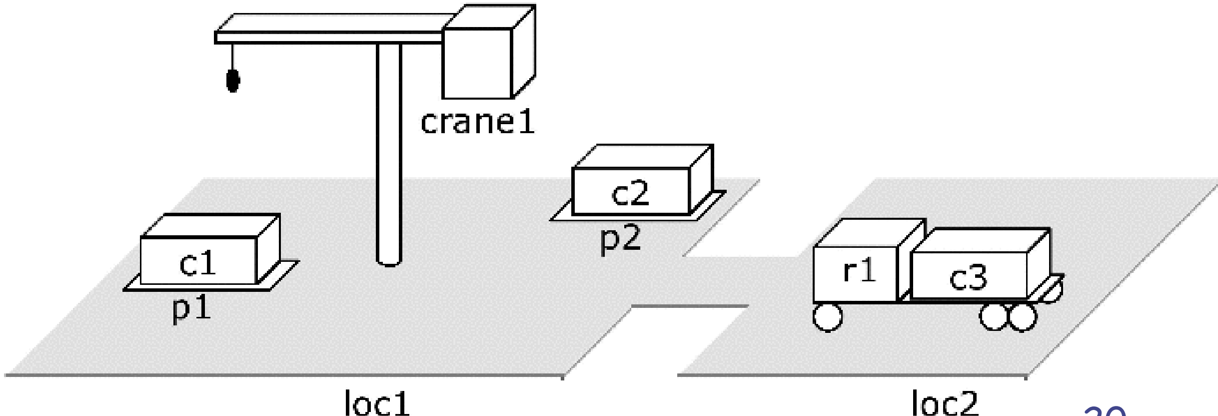
- The first one is *redundant*
  - Can remove actions and still have a solution

⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

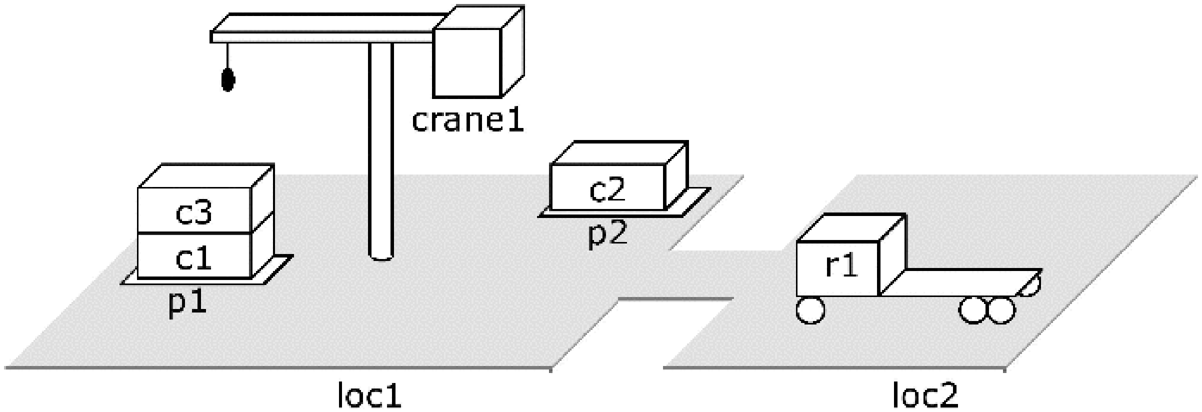
- The 2nd and 3rd are *irredundant*
- They also are *shortest*
  - No shorter solutions exist



State-Variable Representation

- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to *state variables*
  - Like fields in a record structure
- Classical and state-variable representations take similar amounts of space
  - Each can be translated into the other in low-order polynomial time

move(r, l, m)
;; robot r at location l moves to an adjacent location m
precond: $\mathrm{rloc}(r) = l$, $\mathrm{adjacent}(l, m)$
effects: $\mathrm{rloc}(r) \leftarrow m$



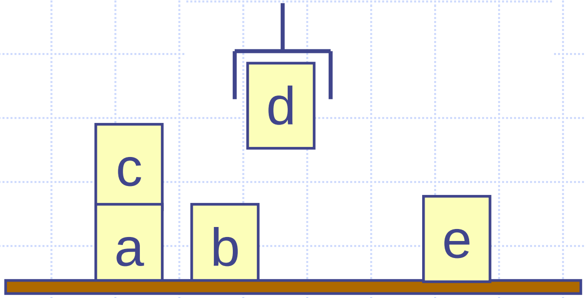
s_1 = {top(p1)=c3,
cpos(c3)=c1,
cpos(c1)=pallet,
holding(crane1)=nil,
rloc(r1)=loc2,
loaded(r1)=nil, }

Example: The Blocks World

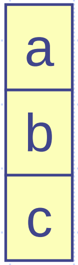
- Infinitely wide table, finite number of children's blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There's a robot gripper that can hold at most one block

- Want to move blocks from one configuration to another
  - e.g.,

initial state

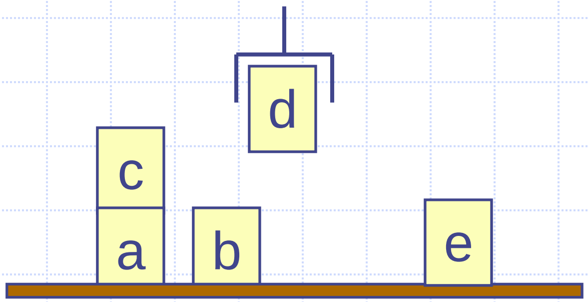


goal



- Like a special case of DWR with one location, one crane, some containers, and many more piles than you need
- I'll give classical and state-variable formulations
  - For the case where there are five blocks

Classical Representation: Symbols



- Constant symbols:
  - The blocks: a, b, c, d, e
- Predicates:
  - ontable(*x*) - block *x* is on the table
  - on(*x*,*y*) - block *x* is on block *y*
  - clear(*x*) - block *x* has nothing on it
  - holding(*x*) - the robot hand is holding block *x*
  - handempty - the robot hand isn't holding anything

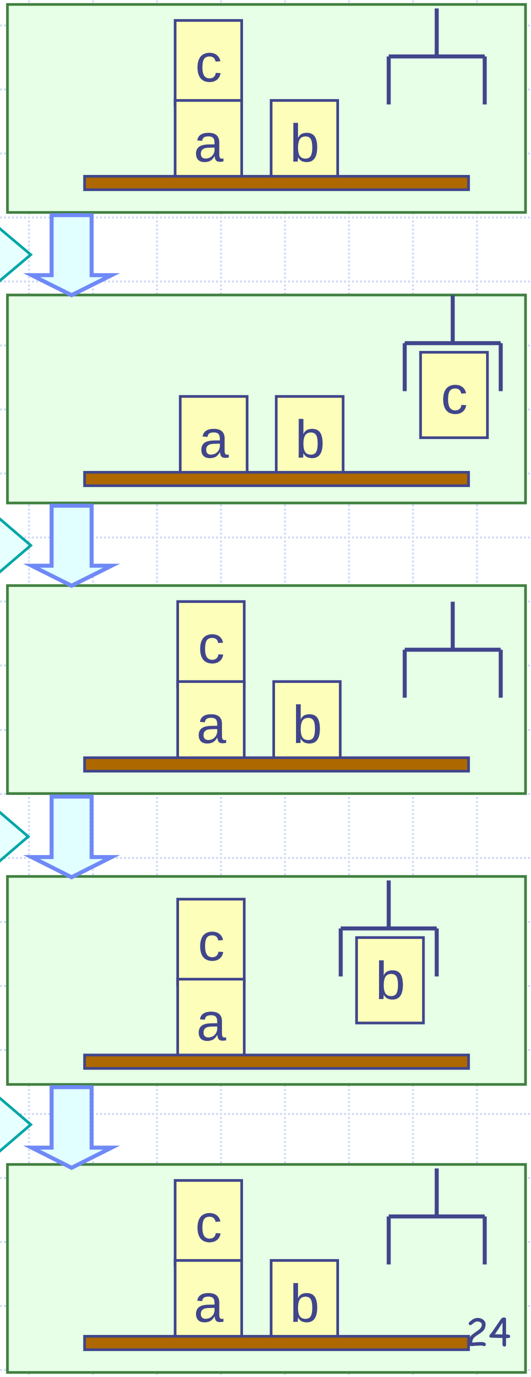
Classical Operators

unstack(x,y)
- Precond: on(x,y), clear(x), handempty
- Effects: $\neg$on(x,y), $\neg$clear(x), $\neg$handempty, holding(x), clear(y)

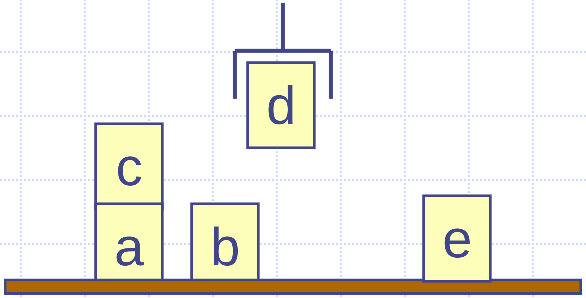
stack(x,y)
- Precond: holding(x), clear(y)
- Effects: $\neg$holding(x), $\neg$clear(y), on(x,y), clear(x), handempty

pickup(x)
- Precond: ontable(x), clear(x), handempty
- Effects: $\neg$ontable(x), $\neg$clear(x), $\neg$handempty, holding(x)

putdown(x)
- Precond: holding(x)
- Effects: $\neg$holding(x), ontable(x), clear(x), handempty



State-Variable Representation: Symbols



- Constant symbols:

  a, b, c, d, e — of type block

  0, 1, table, nil — of type other

- State variables:

pos(*x*) = y	if block *x* is on block *y*
pos(*x*) = table	if block *x* is on the table
pos(*x*) = nil	if block *x* is being held
clear(*x*) = 1	if block *x* has nothing on it
clear(*x*) = 0	if block *x* is being held or has another block on it
holding = *x*	if the robot hand is holding block *x*
holding = nil	if the robot hand is holding nothing

State-Variable Operators

unstack(x : block, y : block)

Precond: pos(x)=y, clear(y)=0, clear(x)=1, holding=nil

Effects: pos(x)=nil, clear(x)=0, holding=x, clear(y)=1

stack(x : block, y : block)

Precond: holding=x, clear(x)=0, clear(y)=1

Effects: holding=nil, clear(y)=0, pos(x)=y, clear(x)=1

pickup(x : block)

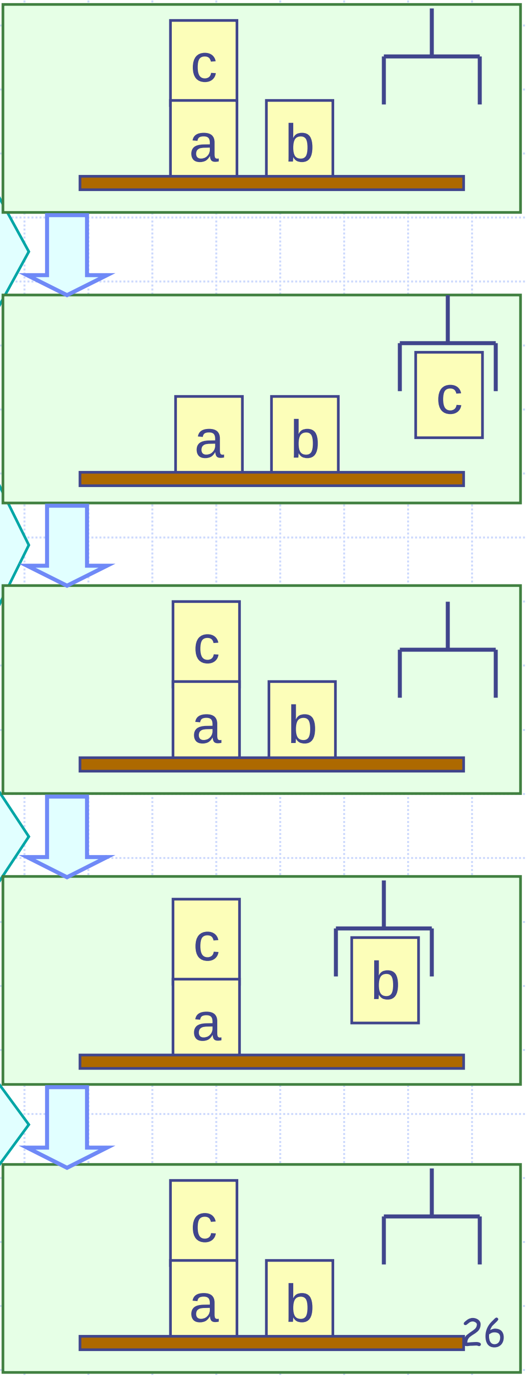
Precond: pos(x)=table, clear(x)=1, holding=nil

Effects: pos(x)=nil, clear(x)=0, holding=x

putdown(x : block)

Precond: holding=x

Effects: holding=nil, pos(x)=table, clear(x)=1



Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons
- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

PDDL.
Planning Domain Description Language.

- We will only use Classical Representation
- Examples: Blocks-world, Hanoi towers

- Two files: domain file and problem file
- Domain file: predicates, operators
- Problem file: problem objects, initial state
- PDDL BNF syntax provided

PDDL. Blocks-world. Domain file (I)

- Objects in the domain: blocks, table, robot-arm
- Properties of the objects

```
(define (domain blockword)
   (:predicates
           (clear ?x)
           (on-table ?x)
           (arm-empty)
           (holding ?x)
           (on ?x ?y))
```

PDDL. Blocks-world. Domain file (II)

```
(:action pickup
  :parameters (?ob)
  :precondition
      (and (clear ?ob)(on-table ?ob)(arm-empty))
  :effect
      (and (holding ?ob) (not (clear ?ob))
           (not (on-table ?ob))(not (arm-empty))))
```

```
(:action putdown
  :parameters (?ob)
  :precondition (and (holding ?ob))
  :effect
      (and (clear ?ob) (arm-empty) (on-table ?ob)
           (not (holding ?ob))))
```

PDDL. Blocks-world. Domain file (III)

```
(:action stack
  :parameters (?ob ?underob)
  :precondition
      (and (clear ?underob)(holding ?ob))
  :effect
      (and (clear ?ob) (arm-empty) (on ?ob ?underob)
           (not (holding ?ob))(not (clear ?underob))))
```

```
(:action unstack
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob)
                     (arm-empty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob))
               (not (arm-empty))))
)
```

PDDL. Blocks-world. Problem file

```
(define (problem tower6)
   (:domain blocksworld)
   (:objects a b c d e f)

   (:init (on-table a) (on-table b) (on-table c)
          (on-table d) (on-table e)(on-table f)
          (clear a) (clear b) (clear c) (clear d)
          (clear e)(clear f)(arm-empty))

   (:goal (and (on a b) (on b c) (on c d) (on d e)
               (on e f))))
```

PDDL. Blocks-world (typing-I)

- Using 'typing': define types of objects, an object hierarchy

```
(define (domain blocksworld)
  (:requirements :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block)
             )
```

PDDL. Blocks-world (typing-II)

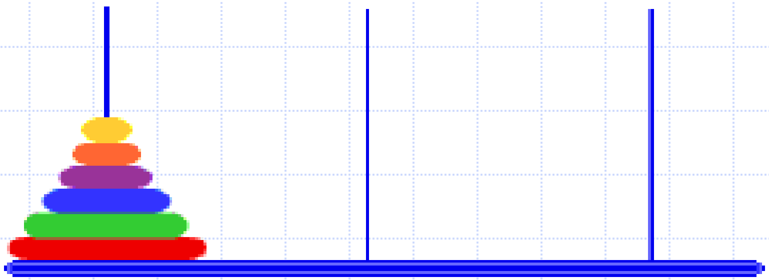
```
(:action stack
  :parameters (?ob - block ?underob - block)
  :precondition
      (and (clear ?underob)(holding ?ob))
  :effect
      (and (clear ?ob) (arm-empty) (on ?ob ?underob)
           (not (holding ?ob))(not (clear ?underob))))
```

```
(define (problem tower6)
   (:domain blocksworld)
   (:objects a b c d e f - block)

   (:init . . .)

   (:goal . . .)
```

PDDL. Hanoi towers.



- Three disks: large (L), medium (M), small (S)
- Three pegs: peg1 (P1), peg2 (P2), peg3 (P3)
- Two predicates: (at <disk> <disk|peg>)
  (clear <disk|peg>)

PDDL. Hanoi towers (domain I).

```
(define (domain hanoi)
  (:requirements :strips :typing :equality)
  (:types disk peg)
  (:predicates (at ?x - disk ?y - (either disk peg))
               (clear ?x - (either disk peg)))


  (:action move-large
       :parameters (?x - peg ?y - peg)
       :precondition (and (at L ?x) (clear L)(clear ?y))
       :effect
               (and (not (at L ?x))(at L ?y)
                    (not (clear ?y))(clear ?x)))
```

PDDL. Hanoi towers (domain II).

```
(:action move-medium
   :parameters (?x - (either peg disk) ?y - (either disk peg))
   :precondition (and (at M ?x)(clear M)
                      (clear ?y)(not (= ?y S)))
   :effect (and (not (at M ?x))(at M ?y)(not (clear ?y))
                (clear ?x)))


 (:action move-small
    :parameters (?x - (either peg disk) ?y - (either disk peg))
    :precondition (and (at S ?x)(clear S)(clear ?y))
    :effect
            (and (not (at S ?x))(at S ?y)
                 (not (clear ?y))(clear ?x)))
```

PDDL. Hanoi towers (problem).

```
(define (problem probhanoi1)
(:domain hanoi)
(:objects L M S - disk
          P1 P2 P3 - peg)

(:init (at S M)(at M L)(at L P1)
       (clear S)(clear P2)(clear P3))

(:goal (and (at S M)(at M L)(at L P3)))
)
```

PDDL. Hanoi towers (a different encoding)

```
(define (domain hanoi)
  (:requirements :strips :typing :equality
                 :negative-preconditions)
  (:types disk peg)
  (:predicates (at ?x - disk ?y - (either disk peg))
               (clear ?x - (either disk peg)))


  (:action move-large
      :parameters (?x - peg ?y - peg)
      :precondition (and (at L ?x) (clear L)
                         (not (at M ?y))(not (at S ?y)))
      :effect (and (not (at L ?x))(at L ?y)
                   (not (clear ?y))(clear ?x)))
```

PDDL. Hanoi towers (a different encoding with only one operator)

```
(define (domain hanoi)
  (:requirements :strips :typing)
  (:types disk peg)
  (:predicates (at ?x - disk ?y - (either disk peg))
               (clear ?x - (either disk peg))
               (smaller ?x - disk ?y - (either disk peg)))


  (:action move-disk
      :parameters (?disk - disk ?from - (either disk peg)
                   ?new-below - (either disk peg))
      :precondition (and (at ?disk ?from)
                         (clear ?disk)(clear ?new-below)
                         (smaller ?disk ?new-below))
      :effect (and (at ?disk ?new-below) (clear ?from)
                   (not (clear ?new-below))
                   (not (at ?disk ?from))))
)
```

PDDL. Hanoi towers (a different encoding with only one operator)

```
(define (problem probhanoi1)
(:domain hanoi)
(:objects L M S - disk
          P1 P2 P3 - peg)

(:init (at S M)(at M L)(at L P1)(clear S)(clear P2)
       (clear P3)(smaller S M)(smaller S L)(smaller M L)
       (smaller S P1)(smaller S P2)(smaller S P3)
       (smaller M P1)(smaller M P2)(smaller M P3)
       (smaller L P1)(smaller L P2)(smaller L P3))

(:goal (and (at S M)(at M L)(at L P3)))
)
```