IA Planning
Lecture 2: PDDL (Planning Domain Description Language)

Halim Djerroud

LISV
Laboratoire d'ingénierie
des systèmes de Versailles

UVSQ
université PARIS-SACLAY

revision: 0.1

Course Outline

1. Motivation and context of PDDL.
2. Components of a planning problem.
3. Structure of a PDDL file (domain / problem).
4. Practical example: Gripper.
5. Basic syntax (types, predicates, actions, goals).
6. Running a planner.
7. Hands-on practice: Blocks-World.

Introduction to PDDL

Why a standard language for planning?

Chapter objectives:

1. Understand the role and importance of PDDL in automated planning
2. Place PDDL historically: from STRIPS to IPC competitions
3. Master the structure of domain and problem files

Why PDDL? The stakes of standardization

Before PDDL: each planner used its own formalism

→ Difficulty comparing approaches

→ Limited reusability of models

Benefits of PDDL:

Standardization: Common language accepted by the international community

Interoperability: One problem, multiple compatible planners

Scalability: Successive versions (1.2, 2.1, 3.0, 3.1) enriching expressive capabilities

Real impact: Robotics, logistics, video games, space missions, etc.

PDDL: from STRIPS to IPC

STRIPS legacy (1971)

- States/actions representation
- Preconditions and effects
- Blocks world

Birth of PDDL (1998)

- Created for IPC
- Lisp-inspired syntax
- Progressive extensions

International Planning Competition

- Biennial competition since 1998
- Standardized benchmark
- Stimulates research

Version evolution

1.2 Base (types, predicates)

2.1 Time, numeric fluents

3.0 Preferences, constraints

3.1 Object functions

PDDL architecture: domain vs problem

Fundamental separation

PDDL distinguishes **generic knowledge** (domain) from **specific instance** (problem)

Domain File

Content:

- Object types
- Predicates
- Actions (generic schemas)
- Functions (optional)

Reusable for multiple problems

Problem File

Content:

- Concrete objects
- Initial state
- Goal to achieve
- Metric (optional)

Specific to one instance

Example: Gripper domain

Scenario: A robot with two arms must transport balls between rooms

```
(define (domain gripper)
  (:requirements :strips :typing :action-costs)

  (:types room ball arm)

  (:predicates
    (at-robby ?r - room)          ; the robot is in room ?r
    (at ?b - ball ?r - room)      ; ball ?b is in ?r
    (free ?a - arm)               ; arm ?a is free
    (carry ?b - ball ?a - arm)) ; arm ?a carries ball ?b

  ;; Actions will be defined later (pick, drop, move)
)
```

Note: Declaration of `:requirements`, hierarchical types, and parameterized predicates

Components of a planning problem

What elements need to be defined to model a problem?

Chapter objectives:

1. Identify the seven fundamental building blocks of a planning problem
2. Understand the role and interaction between each component
3. Master their syntactic representation in PDDL

Key principle

A planning problem = **world model** + **task to accomplish**

The 7 fundamental components

World modeling:

- **Objects** Manipulated entities (balls, robots, rooms...)
- **Predicates** Boolean relations and properties
  Ex: (at ball1 rooma)
- **Functions** Numeric values (distances, costs...)
  Ex: (distance rooma roomb) = 10
- **Actions** Possible operations to transform the world
  Ex: pick, drop, move

Task definition:

- **Initial state** Starting configuration
  Where are the objects?
- **Goal** Conditions to satisfy
  What do we want to achieve?
- **Metric** Optimization criterion
  Minimize cost?

Analogy: cooking recipe

Analogy: the problem as a recipe

PDDL Component	Cooking equivalent
Objects	Ingredients (eggs, flour, bowl...)
Predicates	Ingredient states (raw, mixed...)
Functions	Quantities (temperature, mass...)
Actions	Operations (mix, bake, pour...)
Initial state	Raw ingredients on the counter
Goal	Cake ready to serve
Metric	Minimal preparation time

Note

This analogy illustrates the domain/problem separation: the **domain** contains generic actions (how to mix), the **problem** specifies concrete ingredients and the target dish.

Mapping to PDDL syntax

In the **domain** file

- `(:types ...)`
  Object type hierarchy
- `(:predicates ...)`
  Relation declarations
- `(:functions ...)`
  Numeric fluent declarations
- `(:action ...)`
  Parameterized action schemas

In the **problem** file

- `(:objects ...)`
  Concrete instances
- `(:init ...)`
  Predicates and functions initially true
- `(:goal ...)`
  Logical formula to satisfy
- `(:metric ...)`
  Function to optimize

$\Rightarrow$ Domain = reusable knowledge | Problem = specific instance

Concrete example: the Gripper world (1/3)

Scenario: Transport 4 balls from rooma to roomb with a two-armed robot

Concrete objects:

```
(:objects
  rooma roomb - room
  ball1 ball2
  ball3 ball4 - ball
  left right - arm
)
```

3 types, 8 instantiated objects

Domain predicates:

```
(at-robby ?r - room)
(at ?b - ball ?r - room)
(free ?a - arm)
(carry ?b - ball ?a - arm)
```

Describe the position and state of the system

Concrete example: the Gripper world (2/3)

Initial state: Configuration at time $t = 0$

```
(:init
  ; Robot position
  (at-robby rooma)

  ; Ball positions
  (at ball1 rooma) (at ball2 rooma)
  (at ball3 rooma) (at ball4 rooma)

  ; Arm states
  (free left) (free right)

  ; Numeric data (PDDL 2.1+)
  (= (length rooma roomb) 30)
  (= (total-cost) 0)
)
```

Interpretation: Robot and balls in rooma, arms free, distance between rooms = 30

Concrete example: the Gripper world (3/3)

Goal: Desired state (conditions to satisfy)

```
(:goal
  (and
    (at ball1 roomb)
    (at ball2 roomb)
    (at ball3 roomb)
    (at ball4 roomb)
  )
)
```

Optimization metric:

```
(:metric minimize (total-cost))
```

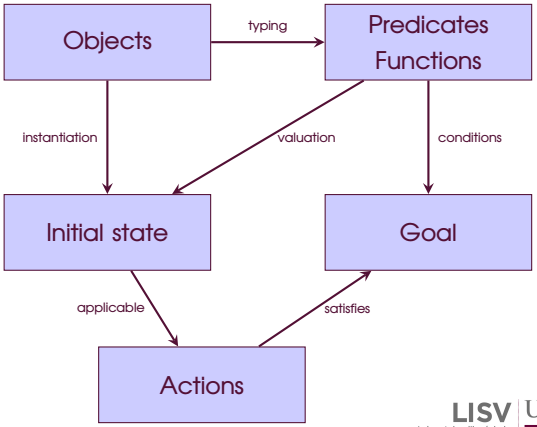
Important note

- The goal does **not specify how** to achieve it (no plan)
- The metric distinguishes satisficing vs. optimal planning
- Final robot position and arm states: unconstrained (flexibility)

Interactions between components

Reasoning chain:

1. **Objects** populate the world
2. **Predicates/functions** describe their state
3. The **initial state** sets truth values
4. **Actions** modify these values
5. The planner searches for a sequence achieving the **goal**



Summary: components at a glance

The 7 pillars of a planning problem

1. **Objects**: the universe of entities ($\approx$ variables)
2. **Predicates**: boolean relations between objects
3. **Functions**: numeric values associated with objects
4. **Initial state**: starting point (I)
5. **Goal**: success conditions (G)
6. **Actions**: allowed transitions ($I \rightarrow \ldots \rightarrow G$)
7. **Metrics**: plan quality criterion (optional)

Next step: Detailed structure of PDDL files and complete action syntax

Structure of PDDL files

Two complementary files: *domain* and *problem*

Chapter objectives:

1. Master the separation between **generic knowledge** / **specific instance**
2. Identify all sections of a PDDL file and their role
3. Be able to read, understand and write basic PDDL files

Fundamental principle

Domain (reusable) + **Problem** (specific) = Complete modeling

Separation principle: why two files?

Analogy with object-oriented programming:

- Domain = **class** (abstract definition)
- Problem = **instance** (concrete object)

Domain File

"How does the world work?"

Content:

- `:types` — Object categories
- `:predicates` — Possible relations
- `:functions` — Numeric values
- `:action` — Available operations

Reusable for N problems

Problem File

"What task to solve?"

Content:

- `:objects` — Concrete entities
- `:init` — Initial configuration
- `:goal` — Success conditions
- `:metric` — Optimality criterion

Specific to ONE task

Advantages of this architecture

1. **Reusability**
   - A single "Gripper" domain for hundreds of different problems
   - Modeling economy and facilitated maintenance
2. **Modularity**
   - Domain modification (adding actions) without touching problems
   - Benchmark creation: fixed domain, varied problems
3. **Conceptual clarity**
   - Clear separation between *world physics* and *task to accomplish*
   - Facilitates understanding and validation

$\Rightarrow$ This separation is at the core of PDDL usage in IPC

Anatomy of a domain file

Hierarchical structure (Lisp syntax):

```
(define (domain <domain-name>)
  ;; 1. Declaration of capabilities used
  (:requirements :strips :typing :fluents ...)
  ;; 2. Type hierarchy (optional)
  (:types
    <type1> <type2> - <super-type>
    ...)
  ;; 3. Predicates (boolean relations)
  (:predicates
    (<predicate-name> ?param1 - type1 ?param2 - type2)
    ...)
  ;; 4. Numeric functions (optional, PDDL 2.1+)
  (:functions
    (<function-name> ?param - type)
    ...)
  ;; 5. Actions (one or more)
  (:action <action-name>
    :parameters (...)
    :precondition (...)
    :effect (...))
)
```

Complete example: Gripper domain (1/2)

Context: Robot with 2 arms transporting balls between rooms

```
(define (domain gripper)
  (:requirements :strips :typing :action-costs)
  ;; Type hierarchy
  (:types room ball arm)
  ;; World relations
  (:predicates
    (at-robby ?r - room)        ; robot position
    (at ?b - ball ?r - room)    ; ball position
    (free ?a - arm)             ; available arm
    (carry ?b - ball ?a - arm) ; arm carrying a ball
  )
  ;; Numeric fluents
  (:functions
    (total-cost)                ; accumulated cost
    (length ?from ?to - room)  ; distance between rooms
  )
  ;; ... (actions on next slide)
)
```

Complete example: Gripper domain (2/2)

Definition of three actions:

```
;; Action 1: robot movement
(:action move
  :parameters (?from ?to - room)
  :precondition (at-robby ?from)
  :effect (and
            (at-robby ?to)
            (not (at-robby ?from))
            (increase (total-cost) (length ?from ?to))))

;; Action 2: pick up a ball
(:action pick
  :parameters (?b - ball ?r - room ?a - arm)
  :precondition (and (at ?b ?r) (at-robby ?r) (free ?a))
  :effect (and
            (carry ?b ?a)
            (not (at ?b ?r))
            (not (free ?a))))

;; Action 3: drop a ball
(:action drop
  :parameters (?b - ball ?r - room ?a - arm)
  :precondition (and (carry ?b ?a) (at-robby ?r))
  :effect (and
            (at ?b ?r)
            (free ?a)
            (not (carry ?b ?a))))
```

Anatomy of a problem file

Simpler structure (no actions):

```
(define (problem <problem-name>)
  ;; 1. Domain reference (MANDATORY)
  (:domain <domain-name>)
  ;; 2. Concrete object declaration
  (:objects
    <obj1> <obj2> - <type1>
    <obj3> - <type2>
    ...)
  ;; 3. Initial configuration
  (:init
    (<predicate1> obj1 obj2)
    (= (<function> obj3) value)
    ...)
  ;; 4. Success conditions
  (:goal
    (and <logical formula>))
  ;; 5. Optimization metric (optional)
  (:metric minimize (<function>))
)
```

Complete example: Gripper problem

```
(define (problem gripper-4balls)

  ;; Must exactly match the domain name
  (:domain gripper)

  ;; Instantiation of world objects
  (:objects
    rooma roomb - room
    ball1 ball2 ball3 ball4 - ball
    left right - arm
  )

  ;; Configuration at time t=0
  (:init
    (at-robby rooma)                  ; robot in rooma
    (free left) (free right)          ; arms available
    (at ball1 rooma) (at ball2 rooma)
    (at ball3 rooma) (at ball4 rooma)
    (= (length rooma roomb) 30)       ; distance = 30
    (= (total-cost) 0)                ; initial cost = 0
  )

  ;; Desired final state
  (:goal
    (and
      (at ball1 roomb)
      (at ball2 roomb)
      (at ball3 roomb)
      (at ball4 roomb)
    )
```

Consistency between domain and problem

Consistency rules (mandatory)

1. The name in `(:domain ...)` of the problem must **exactly** match the domain name
2. All **types** used in `:objects` must be defined in the domain
3. All **predicates** and **functions** in `:init` and `:goal` must be declared in the domain
4. The **arities** (number of parameters) must match

Common errors:

- Typo in domain name ⇒ "domain not found" error
- Object of undeclared type ⇒ parsing error
- Misspelled predicate in `:init` ⇒ predicate ignored (silent false negative!)

Comparative table domain / problem

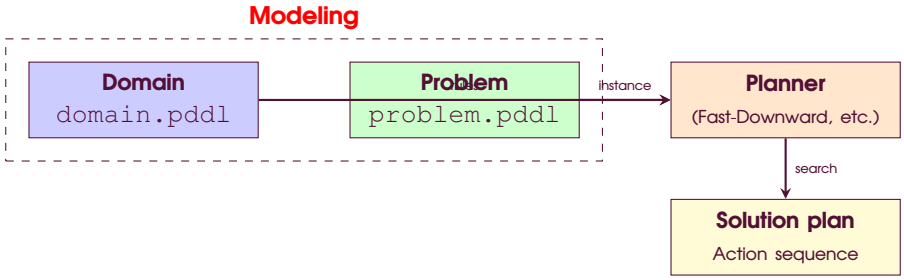
Aspect	Domain	Problem
Nature	World vocabulary and grammar	Concrete narrative instance
Content	Types, predicates, functions, actions	Objects, init, goal, metric
Scope	Applicable to a problem family	Unique, specific
Complexity	Defines *physics* (transition)	Defines *task* (bounds)
Reuse	Yes (e.g., 30 Gripper problems)	No (except variants)
Size	Proportional to possible actions	Proportional to objects
Typical file	`domain.pddl`	`problem01.pddl`

$\Rightarrow$ Think "class vs instance" or "model vs data"

Workflow for creating a PDDL model

1. **Analyze the application domain**
   - What are the entities? → Types
   - What relations describe them? → Predicates
   - What operations are possible? → Actions
2. **Write the domain file**
   - Structure types (hierarchy if relevant)
   - Declare predicates and functions
   - Model each action (preconditions, effects)
3. **Create one or more problems**
   - Instantiate concrete objects
   - Describe complete initial state
   - Specify goal (and metric if needed)
4. **Validate and test**
   - Check syntax (PDDL parser)
   - Test with a planner
   - Iterate if necessary

Visualization: from domain to plan



Interpretation:

- Domain and problem are the planner's **inputs**
- The planner performs a search in the state space
- The **plan** is the output: valid action sequence

Summary: PDDL file structure

Key takeaways

1. **Domain/problem separation**: distinction between physics vs. task
2. Domain = `:types`, `:predicates`, `:functions`, `:action`
3. Problem = `:domain`, `:objects`, `:init`, `:goal`, `:metric`
4. Both files must be **consistent** (names, types, predicates)
5. This architecture promotes reusability and modularity

Next step: Study of classic examples
(Blocks-World, navigation, logistics)

Basic syntax in PDDL

Fundamental elements: Types, Predicates, Actions, Goals

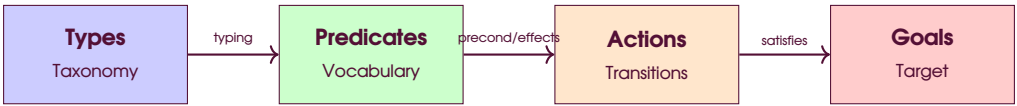
Chapter objectives:

1. Master the four essential syntactic building blocks of PDDL
2. Understand the semantics of each construction
3. Be able to write correct and expressive PDDL models

Key principle

PDDL syntax translates a planning problem into a **formal language** exploitable by an algorithm

The four pillars of PDDL syntax



Static role (description):

- Types: object categories
- Predicates: observable properties

Dynamic role (behavior):

- Actions: change operators
- Goals: termination condition

$\Rightarrow$ Together, they define a **state transition system**

1. Types: structuring the universe of discourse

Role: Hierarchically organize object categories

```
(:types
   room ball arm                  ; three independent types
   container vehicle - movable    ; simple inheritance
   robot human - agent            ; agent is the super-type
)
```

Syntax:

- `X - Y`: X inherits from Y
- Types without parent: independent
- Multi-level hierarchy possible

Advantages:

- Strong typing (error detection)
- Search space reduction
- Increased expressiveness

Gripper example

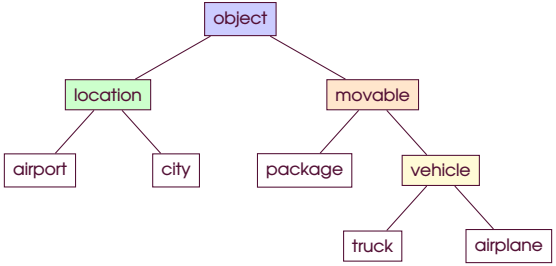
Types: `room`, `ball`, `arm`

Objects: `rooma`, `ball1`, `left` (type instances)

Type hierarchy: elaborate example

Logistics domain with inheritance:

```
(:types
   movable location - object       ; all places are objects
   package vehicle - movable       ; packages and vehicles are movable
   truck airplane - vehicle        ; vehicle specialization
   airport city - location         ; location types
)
```



Benefit: A predicate on `vehicle` automatically applies to `truck` and `airplane`

2. Predicates: the vocabulary of states

Role: Define observable boolean relations and properties

```
(:predicates
    (at-robby ?r - room)              ; unary predicate (1 argument)
    (at ?b - ball ?r - room)          ; binary predicate (2 arguments)
    (free ?a - arm)                   ; property of an object
    (carry ?b - ball ?a - arm)        ; relation between two objects
    (connected ?r1 ?r2 - room)        ; possibly symmetric relation
)
```

Conventions and best practices

- Variables prefixed with `?` (mandatory)
- Descriptive names in English (readability)
- Avoid redundancy (`at` vs `not-at`)
- Prefer positive predicates (closed world assumption)

Predicates vs Functions (PDDL 2.1+)

Predicates (boolean)

Values: true / false

Usage:

- Qualitative relations
- Discrete properties

Example:

State:

- Set of true predicates
- Closed world assumption

Functions (numeric)

Values: real numbers

Usage:

- Continuous quantities
- Metrics, costs, durations

Example:

State:

- Value assignments
- Enables optimization

$\Rightarrow$ Predicates for **qualitative**, functions for **quantitative**

3. Actions: transition operators

Canonical structure of an action:

```
(:action <name>
   :parameters (<list of typed variables>)
   :precondition (<logical formula>)
   :effect (<logical formula>)
)
```

Name	Unique action identifier (e.g., `move`, `pick`)
Parameters	Free variables instantiated during application
Precondition	What must be true **before** the action
Effect	What changes **after** the action

Important semantics

An action is a **schema**: each valid instantiation of parameters creates a **ground** (concrete) action applicable in a given state.

Simple example: `move` action

```
(:action move
   :parameters (?from ?to - room)
   :precondition (at-robby ?from)
   :effect (and
               (at-robby ?to)
               (not (at-robby ?from)))
)
```

Interpretation:

- **Abstract schema**: 2 variables (`?from`, `?to`)
- **Precondition**: Robot must be in `?from`
- **Positive effects**: `(at-robby ?to)` becomes true
- **Negative effects**: `(at-robby ?from)` becomes false

Concrete instantiation

If `?from = rooma` and `?to = roomb`:
Ground action: `(move rooma roomb)`

Complex action: `pick` (Gripper)

```
(:action pick
   :parameters (?b - ball ?r - room ?a - arm)
   :precondition (and
                    (at ?b ?r)          ; ball in the room
                    (at-robby ?r)       ; robot in the same room
                    (free ?a))          ; arm available
   :effect (and
              (carry ?b ?a)             ; + arm carries the ball
              (not (at ?b ?r))          ; - ball no longer on floor
              (not (free ?a)))          ; - arm no longer free
)
```

Important remarks:

- Multiple preconditions combined with `(and ...)`
- Mixed effects: adding and removing facts
- Consistency: `(at ?b ?r)` and `(not (at ?b ?r))` cannot coexist
- Completeness: all relevant changes are specified

Logical operators in actions

In preconditions:

```
; Conjunction (AND)
(and (at-robby ?r) (free ?a))

; Disjunction (OR)
(or (in-room ?r1) (in-room ?r2))

; Negation (requires :negative-preconditions)
(not (locked ?door))

; Quantification (PDDL 2.1+)
(forall (?b - ball) (in-box ?b))
(exists (?r - room) (empty ?r))
```

In effects:

```
; Conjunction of effects
(and (at ?b ?r) (not (free ?a)))

; Conditional effects (ADL)
(when (full ?container)
      (heavy ?container))

; Numeric effects (PDDL 2.1)
(increase (total-cost) 5)
(decrease (fuel ?v) 10)
(assign (speed ?v) 0)
```

Warning

Not all these operators are available in basic PDDL (:strips). Check the domain's `:requirements`!

Conditional effects and ADL

Example: dropping a fragile ball

```
(:action drop-careful
   :parameters (?b - ball ?r - room ?a - arm)
   :precondition (and (carry ?b ?a) (at-robby ?r))
   :effect (and
             (at ?b ?r)
             (free ?a)
             (not (carry ?b ?a))
             ; Conditional effect: ball breaks if floor is hard
             (when (hard-floor ?r)
                   (broken ?b)))
)
```

Semantics:

- Unconditional effects: always applied
- `(when <condition> <effect>)`: effect applied **iff** condition true in resulting state
- Requires `:conditional-effects` in `:requirements`

4. Goals: specifying the objective

Role: Define the success condition (target state(s))

```
(:goal
   (and
     (at ball1 roomb)
     (at ball2 roomb)
     (at ball3 roomb)))
```

Important characteristics

- Logical formula (conjunction, disjunction...)
- Does **not specify how** to achieve the goal (no plan)
- Can be **partial**: does not constrain the entire state
- Satisfaction test: is the goal true in the final state?

Partial goal

`(:goal (at ball1 roomb))`

⇒ No matter where other balls or the robot are!

Complex goals and expressiveness

Goal with disjunction:

```
(:goal
   (or (at ball1 rooma)       ; either in rooma
       (at ball1 roomb))      ; or in roomb
)
```

Goal with quantification (PDDL 2.1):

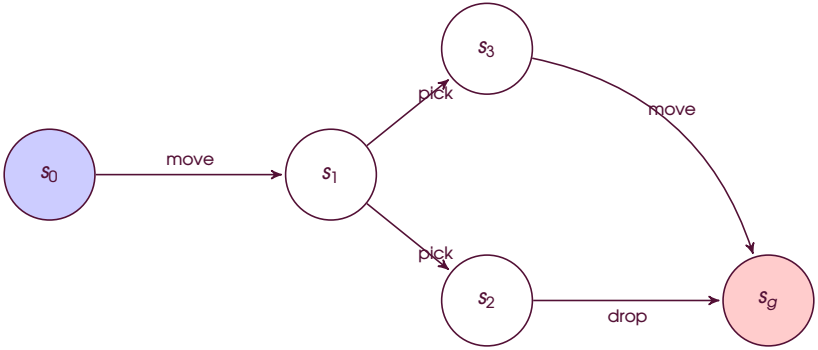
```
(:goal
   (forall (?b - ball)        ; for every ball
           (at ?b roomb))     ; it must be in roomb
)
```

Numeric goal:

```
(:goal
   (and (at package1 destination)
        (>= (fuel truck1) 20))    ; numeric constraint
)
```

⇒ Goal expressiveness depends on declared `:requirements`

Link with state space



PDDL modeling:

- **States** = predicate valuations
- **Arcs** = applicable actions
- **Labels** = action names

Planning = Search:

- Initial state s_0 given
- Goal state s_g satisfies : s_g
- Plan = path $s_0 \rightarrow \ldots \rightarrow s_g$

From syntax to semantics

Element	PDDL Syntax	Formal semantics
Types	`(:types room ball)`	Disjoint sets of objects
Predicates	`(at ?b - ball ?r - room)`	Relations $R \subseteq Ball \times Room$
State	`(:init (at ball1 rooma))`	Set of true ground predicates
Action	`:precondition / :effect`	Transition function $\delta(s, a) = s'$
Goal	`(:goal (at ball1 roomb))`	Set of satisfying states G
Plan	Action sequence	Path $s_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} s_g$

Important point

PDDL is a practical **notation** for expressing a state transition system. Planning remains fundamentally a **graph search problem**.

Complete minimal example: movement domain

```
(define (domain simple-move)
  (:requirements :strips :typing)

  (:types location agent)

  (:predicates
    (at ?a - agent ?l - location)
    (connected ?l1 ?l2 - location))

  (:action move
    :parameters (?a - agent ?from ?to - location)
    :precondition (and (at ?a ?from)
                       (connected ?from ?to))
    :effect (and (at ?a ?to)
                 (not (at ?a ?from))))
)
```

```
(define (problem move-prob1)
  (:domain simple-move)
  (:objects
    robot1 - agent
    locA locB locC - location)
  (:init
    (at robot1 locA)
    (connected locA locB)
    (connected locB locC))
  (:goal (at robot1 locC))
)
```

Summary: basic syntax in PDDL

The four syntactic pillars

1. **Types** (`:types`): object taxonomy, strong typing
2. **Predicates** (`:predicates`): boolean vocabulary of states
3. **Actions** (`:action`): transition schemas (precond/effects)
4. **Goals** (`:goal`): logical formula defining success

Key principles

- Lisp-inspired syntax (parentheses, prefixes)
- Separation of **declarative** (what) vs **procedural** (how)
- Expressiveness controlled by `:requirements`
- Formal semantics = state transition system