Writing Planning Domains and Problems in PDDL[1]

What is PDDL?

PDDL (the "Planning Domain Definition Language") is a recent attempt to standardise planning domain and problem description languages. It was developed mainly to make the International Planning Competition (IPC) series possible. The planning competition compares the performance of planning systems on sets of benchmark problems, so a common language for specifying problems must be used.

PDDL contains STRIPS, ADL and much, much more. Most planners, however, do not support full PDDL. The majority support only the STRIPS subset, or some small extension of it. PDDL has evolved over time, with minor and major revisions published at each planning competition: generally recognised versions are 1.0 (used in IPC 1998), 2.0 (used in IPC 2000), 2.1 (used in IPC 2002), 2.2 (used in IPC 2004), 3.0 (used in IPC 2006) and 3.1 (used in IPC 2008).

Important note: As mentioned above, most planners do *not* support all elements of (any version of) PDDL. Moreover, many planners have their own little "excentricities", meaning that they may interpret certain PDDL constructs incorrectly, or require minor variations in syntax that is at odds with the official language specification. Some examples:

- Some planners have an implicit constraint that all arguments to an action are distinct.
- Some planners require action preconditions and/or effects to be written as conjunctions (*i.e.*, as `(and ...)`) even when the precondition/effect contains only one atomic condition, or even no condition at all.
- Most planners actually ignore the `:requirements` part of the domain definition. But, in spite of this, some planners may fail to parse a domain definition if this part is missing or if it contains a keyword they do not recognise.

A *useful rule of thumb* when writing PDDL is to always use the simplest constructs that are sufficient to express the problem. And, of course, *always read the documentation* for the planner you are trying to use.

Resources

- A list of (on-line) papers related to PDDL. This list includes the "official" specifications of the PDDL versions used in each of the past planning competitions.
- The PDDL plan validator. The plan validator is a tool that takes as input a planning problem, specified in PDDL, and a plan, and automatically checks if the plan is correct. (It can also computes the "value" of the plan, if the problem specification includes a plan metric.) There is also a tool for checking if a PDDL problem specification is syntactically correct.
- INVAL is another implementation of a plan validator (available here).
- Another PDDL tutorial (by Malte Helmert).

Examples

Many examples of PDDL domain and problem definitions can be found on the web. Here are a few pointers:

- Modelling the Wumpus World in PDDL. A commented example of several (bad and good) ways of modelling the Wumpus World in PDDL.
- The benchmark problems used in past planning competitions are all available from the competition webpages. To find them, go to http://ipc.icaps-conference.org/ and follow the links.
- The FF domain collection contains domain definitions a problem generators for many domains. (Note: Many of these are early versions of IPC domains.)

1 http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html

- Commented examples from an AI course at Linköping University:
  - The Logistics domain and example problem.
  - An encoding of the (n^2-1)-puzzle: domain and example problem.

Where to find planners?

A lot of planning systems have been developed by researchers, and many of them are available on the web, but can be a bit hard to find if you don't know what to look for. Here are some pointers:

- The web pages of the 2008 planning competition have downloadable source for all planners that participated in the competition. They should be fairly straightforward to compile (read the instructions).
- The earlier competition web pages don't have downloadable planners, but usually links to researchers web pages where you might find code.
- The Planet web pages contain a comprehensive but outdated list (i.e., these are mainly older planning systems).
- The Yochan group at ASU ran a planning course as recently as 2008, and also provide some links.

Parts of a PDDL Problem Definition

A PDDL definition consists of two parts: The *domain* and the *problem* definition.

Note: Although not required by the PDDL standard, most planners require that the two parts are in separate files.

Comments

Comments in a PDDL file start with a semicolon (";") and last to the end of the line.

Requirements

Because PDDL is a very general language and most planners support only a subset, domains may declare requirements. The most commonly used requirements are:

`:strips`
: The most basic subset of PDDL, consisting of STRIPS only.

`:equality`
: This requirement means that the domain uses the predicate `=`, interpreted as equality.

`:typing`
: This requirement means that the domain uses types (see **Typing** below).

`:adl`
: Means that the domain uses some or all of ADL (*i.e.* disjunctions and quantifiers in preconditions and goals, quantified and conditional effects).

The Domain Definition

The domain definition contains the domain predicates and operators (called *actions* in PDDL). It may also contain types (see **Typing**, below), constants, static facts and many other things, but, again, these are not supported by the majority of planners.

The format of a (simple) domain definition is:

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
               (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
               ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
```

```
   [:effect EFFECT_FORMULA]
  )

 (:action ACTION_2_NAME
   ...)

 ...)
```

Elements in []'s are optional, for those not familiar with formal grammars.

Names (domain, predicate, action, *et c.*) are usually made up of alphanumeric characters, hyphens ("`-`") and underscores ("_"), but there may be some planners that allow less.

Parameters of predicates and actions are distinguished by their beginning with a question mark ("?").

The parameters used in predicate declarations (the `:predicates` part) have no other function than to specify the number of arguments that the predicate should have, *i.e.* the parameter names do not matter (as long as they are distinct). Predicates can have zero parameters (but in this case, the predicate name still has to be written within parenthesises).

What do Predicates Mean?

One thing that is important to understand is that, apart from the special predicate `=`, predicates in a domain definition have *no intrinsic meaning*. The `:predicates` part of a domain definition specify only what are the predicate names used in the domain, and their number of arguments (and argument types, if the domain uses typing). The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition.

It is common to make a distinction between *static* and *dynamic* predicates: a static predicate is not changed by any action. Thus in a problem, the true and false instances of a static predicate will always be precisely those listed in the initial state specification of the problem definition. Note that there is no syntactic difference between static and dynamic predicates in PDDL: they look exactly the same in the `:predicates` declaration part of the domain. Nevertheless, some planners may support different constructs around static and dynamic predicates, for example allowing static predicates to be negated in action preconditions but not dynamic ones.

Action Definitions

All parts of an action definition except the name are, according to the PDDL specification, optional (although, of couse, an action without effects is pretty useless). However, for an action that has no preconditions some planners may require an "empty" precondition, on the form `:precondition ()` or `:precondition (and)`, and some planners may also require an empty `:parameter` list for actions without parameters).

Note: Some planners require that the arguments to an action are all different, *i.e.* the same object may not instantiate two parameters. This may cause some difficulties (*e.g.* problems becomming unsolvable) if one is not aware of it.

Precondition Formulas

In a STRIPS domain, a precondition formula may be:

- An atomic formula:
  `(PREDICATE_NAME ARG1 ... ARG_N)`
  The predicate arguments must be parameters of the action (or constants declared in the domain, if the domain has constants).
- A conjunction of atomic formulas:
  `(and ATOM1 ... ATOM_N)`

If the domain uses `:equality`, an atomic formula may also be of the form `(= ARG1 ARG2)`. Many planners that

support equality also allow negated equality, which is written `(not (= ARG1 ARG2))`, even if they do not allow negation in any other part of the definition.

In an ADL domain, a precondition may in addition be:

- A general negation, conjunction or disjunction:
  `(not CONDITION_FORMULA)`
  `(and CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
  `(or CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- A quantified formula:
  `(forall (?V1 ?V2 ...) CONDITION_FORMULA)`
  `(exists (?V1 ?V2 ...) CONDITION_FORMULA)`

Effect Formulas

In PDDL, the effects of an action are not explicitly divided into "adds" and "deletes". Instead, negative effects (deletes) are denoted by negation.

In a STRIPS domain, an effect formula may consist of:

- An added atom:
  `(PREDICATE_NAME ARG1 ... ARG_N)`
  The predicate arguments must be parameters of the action (or constants declared in the domain, if the domain has constants).
- A deleted atom:
  `(not (PREDICATE_NAME ARG1 ... ARG_N))`
- A conjunction of atomic effects:
  `(and ATOM1 ... ATOM_N)`

The equality predicate (`=`) can of course not occur in an effect formula: no action can make two identical things be not identical!

In an ADL domain, an effect formula may in addition contain:

- A conditional effect:
  `(when CONDITION_FORMULA EFFECT_FORMULA)`
  The interpretation is that the specified effect takes place only if the specified condition formula is true in the state where the action is executed. Conditional effects are usually placed within quantifiers.
- A universally quantified formula:
  `(forall (?V1 ?V2 ...) EFFECT_FORMULA)`

The Problem Definition

The problem definition contains the objects present in the problem instance, the initial state description and the goal.

The format of a (simple) problem definition is:

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
  )
```

Note: Some planners may require that the `:requirements` specification appears also in the problem definition (usually either directly before or directly after the `:domain` specification).

The initial state description (the `:init` section) is simply a list of all the ground atoms that are true in the initial state. All other atoms are by definition false. The goal description is a formula of the same form as an action precondition. All predicates used in the initial state and goal description should naturally be declared in the corresponding domain.

In difference to action preconditions, however, the initial state and goal descriptions should be *ground*, meaning that all predicate arguments should be object or constant names rather than parameters. (An exception is quantified goals in ADL domains, where of course the quantified variables may be used within the scope of the quantifier. Note, however, the even some planners that claim to support ADL do not support quantifiers in goals.)

Typing

PDDL has a (very) special syntax for declaring parameter and object types. If types are to be used in a domain, the domain should declare the requirement `:typing`.

Type names have to be declared before they are used (which usually means before the `:predicates` declaration). This is done with the declaration

```
(:types NAME1 ... NAME_N)
```

To declare the type of a parameter of a predicate or action one writes `?X - TYPE_OF_X`. A list of parameters of the same type can be abbreviated to `?X ?Y ?Z - TYPE_OF_XYZ`. Note that the hyphen between parameter and type name has to be "free-standing", *i.e.* surrounded by whitespace.

The syntax is the same for declaring types of objects in the problem definition.

Plan Quality Criterion

Recent versions of PDDL have a mechanism for expressing the measure of plan quality that planners should try to optimise. However, most planners will only optimise one particular metric, and many do not try to optimise anything at all (just to find any plan as quickly as possibly).

Sum of Action Costs

To specify action costs, it is necessary to add a "fluent" that keeps track of the cost. A fluent is like a state variable/predicate, but its value is a number instead of true or false. Again, note that PDDLs fluent syntax is very expressive, and most planners will only accept a limited use.

To declare fluents, add the following section to the domain specification:

```
(:functions
  (total-cost)
 )
```

Specifying that the cost of an action is *C* is done by stating that the action has the effect of increasing the `total-cost` fluent.

```
:effect (and ... (increase (total-cost) C))
```

The fluent *must* be initialised. In the `:init` section of the problem definition, add

```
(= (total-cost) 0)
```

Finally, to specify that minimising the sum of action costs is the objective, a `:metric` section is added to the problem definition:

```
(define (problem ..)
  .
  .
  .
  (:metric minimize (total-cost))
```

```
 )
```

Note: The PDDL plan validator (VAL) requires that the keyword `:fluents` is present in the `:requirements` section of the domain whenever fluents are used in the domain. Some planners will instead require it to contain `:action-costs`, and some will not recognise either of those.

To make the cost of an action dependent on its arguments, it is possible to declare static functions, and use those in the `increase` effect. For example:

```
(define (domain travel)
  .
  .

  (:functions
    (distance ?from ?to)
    (total-cost)
   )

 (:action go
   :parameters (?from ?to)
   :precondition (and (in ?from) (road ?from ?to))
   :effect (and (not (in ?from)) (in ?to)
                (increase (total-cost) (distance ?from ?to)))
 )
)
```

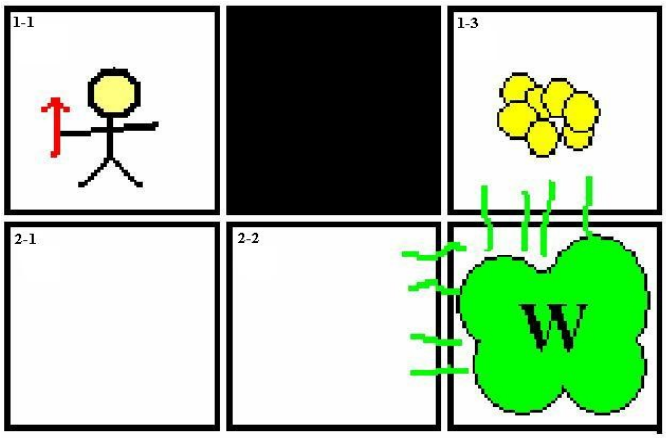
The values of static functions are listed in the initial state section of the problem specification (just like the values of static predicates):

```
(define (problem ..)
  .
  .

  (:init ...
         (= (distance A B) 10)
         (= (distance A C) 35)
         ...
   )

  .
  .
 )
```

In principle, PDDL allows any combination of functions, constants and arithmetic operations (`+`, `-`, `*`, `/`) on the left-hand side of an `increase` effect, and in the `:metric` section of the problem. However, many planners require that the metric fluent is called `total-cost`, and accept only constants, and perhaps static functions, in `increase` effects.

Modelling the Wumpus World in PDDL: 1st try...[2]

```
(define (domain wumpus-a)
  (:requirements :strips) ;; maybe not necessary

  (:predicates
   (adj ?square-1 ?square-2)
   (pit ?square)
   (at ?what ?square)
   (have ?who ?what)
   (dead ?who))

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (adj ?from ?to)
                       (not (pit ?to))
                       (at ?who ?from))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?arrow ?victim ?where-victim)
    :precondition (and (have ?who ?arrow)
                       (at ?who ?where)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (at ?victim ?where-victim))
                 (not (have ?who ?arrow)))
    )
)

(define (problem wumpus-a-1)
  (:domain wumpus-a)
  (:objects
   sq-1-1 sq-1-2 sq-1-3
   sq-2-1 sq-2-2 sq-2-3
   the-gold
   the-arrow
   agent
   wumpus)

  (:init
   (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
   (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
   (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
   (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
   (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
   (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
   (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)

   (pit sq-1-2)

   (at the-gold sq-1-3)
   (at agent sq-1-1)
   (have agent the-arrow)
   (at wumpus sq-2-3))

  (:goal (and (have agent the-gold) (at agent sq-1-1)))
  )
```



2 http://users.cecs.anu.edu.au/~patrik/pddlman/wumpus.html

Resulting plan:

```
(MOVE THE-GOLD SQ-1-3 SQ-2-3)
(MOVE THE-GOLD SQ-2-3 SQ-2-2)
(MOVE THE-GOLD SQ-2-2 SQ-2-1)
(MOVE THE-GOLD SQ-2-1 SQ-1-1)
(TAKE AGENT THE-GOLD SQ-1-1)
```

Modelling the Wumpus World in PDDL: 2nd try...

```
(define (domain wumpus-b)
  (:requirements :strips)
  (:predicates
   (adj ?square-1 ?square-2)
   (pit ?square)

   (at ?what ?square)
   (have ?who ?what)

   (takeable ?what)
   (is-gold ?what)
   (is-arrow ?what)

   (alive ?who)
   (dead ?who))

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (alive ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (alive ?who)
                       (takeable ?what)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?arrow ?victim ?where-victim)
    :precondition (and (alive ?who)
                       (have ?who ?arrow)
                       (is-arrow ?arrow)
                       (at ?who ?where)
                       (alive ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (alive ?victim))
                 (not (at ?victim ?where-victim))
                 (not (have ?who ?arrow)))
    )
)

(define (problem wumpus-b-1)
  (:domain wumpus-b)
  (:objects sq-1-1 sq-1-2 sq-1-3
            sq-2-1 sq-2-2 sq-2-3
            the-gold the-arrow
            agent wumpus)
  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
         (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
         (pit sq-1-2)
         (at the-gold sq-1-3)
         (is-gold the-gold)
         (takeable the-gold)
         (at agent sq-1-1)
         (alive agent)
         (have agent the-arrow)
         (is-arrow the-arrow)
         (takeable the-arrow)
         (at wumpus sq-2-3)
         (alive wumpus))
```

```
  (:goal (and (have agent the-gold)
              (at agent sq-1-1)
         ))
  )
```

Resulting plan:

```
(MOVE AGENT SQ-1-1 SQ-2-1)
(MOVE AGENT SQ-2-1 SQ-2-2)
(MOVE AGENT SQ-2-2 SQ-2-3)
(MOVE AGENT SQ-2-3 SQ-1-3)
(TAKE AGENT THE-GOLD SQ-1-3)
(MOVE AGENT SQ-1-3 SQ-2-3)
(MOVE AGENT SQ-2-3 SQ-2-2)
(MOVE AGENT SQ-2-2 SQ-2-1)
(MOVE AGENT SQ-2-1 SQ-1-1)
```

Modelling the Wumpus World in PDDL: 3rd time's a charm...

```
(define (domain wumpus-c)
  (:requirements :strips)
  (:predicates
   (at ?what ?square)
   (adj ?square-1 ?square-2)
   (pit ?square)
   (wumpus-in ?square)
     ;; <-> (exists ?x (and (is-wumpus ?x) (at ?x ?square) (not (dead ?x))
   (have ?who ?what)
   (is-agent ?who)
   (is-wumpus ?who)
   (is-gold ?what)
   (is-arrow ?what)
   (dead ?who))

  (:action move-agent
    :parameters (?who ?from ?to)
    :precondition (and (is-agent ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to))
                       (not (wumpus-in ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (is-agent ?who)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?with-what ?victim ?where-victim)
    :precondition (and (is-agent ?who)
                       (have ?who ?with-what)
                       (is-arrow ?with-what)
                       (at ?who ?where)
                       (is-wumpus ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (wumpus-in ?where-victim))
                 (not (have ?who ?with-what)))
    )

  (:action move-wumpus
    :parameters (?who ?from ?to)
    :precondition (and (is-wumpus ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to))
                       (not (wumpus-in ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to)
                 (not (wumpus-in ?from))
                 (wumpus-in ?to))
    )

)

(define (problem wumpus-c-1)
  (:domain wumpus-c)
  (:objects sq-1-1 sq-1-2 sq-1-3
            sq-2-1 sq-2-2 sq-2-3
            the-gold the-arrow
            agent wumpus)
  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
```

```
        (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
        (pit sq-1-2)
        (is-gold the-gold)
        (at the-gold sq-1-3)
        (is-agent agent)
        (at agent sq-1-1)
        (is-arrow the-arrow)
        (have agent the-arrow)
        (is-wumpus wumpus)
        (at wumpus sq-2-3)
        (wumpus-in sq-2-3))
  (:goal (and (have agent the-gold) (at agent sq-1-1)))
  )
```

Resulting plan:

```
(MOVE-AGENT AGENT SQ-1-1 SQ-2-1)
(MOVE-AGENT AGENT SQ-2-1 SQ-2-2)
(SHOOT AGENT SQ-2-2 THE-ARROW WUMPUS SQ-2-3)
(MOVE-AGENT AGENT SQ-2-2 SQ-2-3)
(MOVE-AGENT AGENT SQ-2-3 SQ-1-3)
(TAKE AGENT THE-GOLD SQ-1-3)
(MOVE-AGENT AGENT SQ-1-3 SQ-2-3)
(MOVE-AGENT AGENT SQ-2-3 SQ-2-2)
(MOVE-AGENT AGENT SQ-2-2 SQ-2-1)
(MOVE-AGENT AGENT SQ-2-1 SQ-1-1)
```

Modelling the Wumpus World in PDDL: using ADL...

```
(define (domain wumpus-adl)
  (:requirements :adl :typing)

  ;; object types
  (:types agent wumpus gold arrow square)

  (:predicates
   (adj ?square-1 ?square-2 - square)
   (pit ?square - square)
   (at ?what ?square)
   (have ?who ?what)
   (alive ?who))

  (:action move
    :parameters (?who - agent ?from - square ?to - square)
    :precondition (and (alive ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                        )
    :effect (and (not (at ?who ?from))
                 (at ?who ?to)

                 (when (pit ?to)
                   (and (not (alive ?who))))

                 (when (exists (?w - wumpus) (and (at ?w ?to) (alive ?w)))
                   (and (not (alive ?who)))))
    )

  (:action take
    :parameters (?who - agent ?where - square ?what)
    :precondition (and (alive ?who)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who - agent ?where - square ?with-arrow - arrow
                 ?victim - wumpus ?where-victim - square)
    :precondition (and (alive ?who)
                       (have ?who ?with-arrow)
                       (at ?who ?where)
                       (alive ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (not (alive ?victim))
                 (not (have ?who ?with-arrow)))
    )
)

(define (problem wumpus-adl-1)
  (:domain wumpus-adl)

  (:objects
   sq-1-1 sq-1-2 sq-1-3 sq-2-1 sq-2-2 sq-2-3 - square
   the-gold - gold
   the-arrow - arrow
   agent-1 - agent
   wumpus-1 - wumpus)

  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
         (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
         (pit sq-1-2)
         (at the-gold sq-1-3)
         (at agent-1 sq-1-1)
         (alive agent-1)
         (have agent-1 the-arrow)
         (at wumpus-1 sq-2-3)
         (alive wumpus-1))
```

```
(:goal (and (have agent-1 the-gold) (at agent-1 sq-1-1) (alive agent-1)))
)
```