Planning and Reasoning Project: Syntax Checker for PDDL

Domenico Alfano

1 Introduction

PDDL is a standardized Planning Domain Definition Language which is widely used to describe planning domains as well as problem instances. PDDL in other terms, is an action-centred language, inspired by the STRIPS, Stanford Research Institute Problem Solver, formulations of planning problems. At its core is a simple standardisation of the syntax for expressing this familiar semantics of actions, using pre-conditions and post-conditions to describe the applicability and effects of actions.
The syntax is inspired by Lisp, so much of the structure of a domain description is a Lisp-like list of parenthesised expressions. An early design decision in the language was to separate the descriptions of parameterised actions that characterise domain behaviours from the description of specific objects, initial conditions and goals that characterise a problem instance. Thus, a planning problem is created by the pairing of a domain description with a problem description. The same domain description can be paired with many different problem descriptions to yield different planning problems in the same domain.
The parameterisation of actions depends on the use of variables that stand for terms of the problem instance. The pre-conditions and post-conditions of actions are expressed as logical propositions constructed from predicates and argument terms (objects from a problem instance) and logical connectives. Although the core of PDDL is a STRIPS formalism, the language extends beyond that.
The extended expressive power includes the ability to express a type structure for the objects in a domain, typing the parameters that appear in actions and constraining the types of arguments to predicates, actions with negative preconditions and conditional effects and the use of quantification in expressing both pre-conditions and post-conditions.
The PDDL language supports many different levels of expressivity. As you will see later, any domain definition must explicitly declare its expressivity requirements in the shape of a *:requirements* clause. This allows a PDDL parser to verify that you only use the expressivity you have explicitly requested, and allows a planner to easily tell you whether you are requesting features it does not support.
A later language called ADL, Action Description Language, added support for a number of different extensions to STRIPS: Negative preconditions, disjunctive preconditions, disjunctive goals, and a few other extensions.
Having made this introduction to the considered language, in the next sections we will present the developed Syntax Checker by focusing our attention on the Grammar structure of PDDL and on the Semantic check of actions making some preliminary simplifications such as removing unexecutable and useless actions and useless predicates.

2 PDDL Syntax

A PDDL definition consists of two parts: The domain definition and the problem (instance) definition. Although not required by the PDDL standard, many planners require that the two parts are in separate files.

2.1 PDDL Domain Definition

The domain definition gives each domain a name and specifies the predicates and operators that are available in the domain (though operators are called actions in PDDL). It may also specify types (see Typing, below), constants, static facts and many other things, some of which are not supported by the majority of planners. The format of a (simple) domain definition is as follows. Elements in [ ] are optional. You may want to keep one or two of the example domain definitions open while you read this.

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl] ...)
  [(:types T1 T2 T3 T4 ...)]
  (: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
    ...)
  ...)
```

Names (domain, predicate, action, etc.) may usually contain alphanumeric characters, hyphens () and underscores (_), though there may be some planners that allow less. Many of the reserved keywords start with a colon, so that newly invented keywords will not interfere with names you have already used for other purposes in your domain and problem specifications. Variables, such as parameters of predicates and actions, are distinguished by beginning with a question mark (?). Comments in a PDDL file start with a semicolon (;) and last to the end of the line.

2.1.1 Requirements

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

- *:strips*, The most basic subset of PDDL, consisting of STRIPS only.
- *:equality*, The domain uses the predicate =, interpreted as equality.
- *:typing*, The domain uses object types (see below).
- *:adl*, The domain uses some or all of ADL (i.e. disjunctions and quantifiers in preconditions and goals, and quantified and conditional effects).

2.1.2 Predicate Definitions

Under :predicates, you define all predicates (boolean state variables) to be used in the domain. The parameters variables used each predicate declarationhave 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 parentheses).

2.1.3 Action Definitions

As shown above, an action is generally specified as follows:

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

All parts of an action definition except the name are, according to the spec, 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 ()* (some planners may also require an empty *:parameter* list for actions without parameters). Parameter types are declared in the same way as in predicate specifications. This ensures that planners do not try to apply actions using unexpected object types.

2.1.4 Precondition Formulas

In a STRIPS domain, a precondition formula may be one of the following (see also the example above):

- 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 *:adl* or *:negated-precondition*, an atomic formula may also be of the form:

(**not** (PREDICATE_NAME ARG1 ... ARG_N))

specifying that the given fact must be false. 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, specified with *:adl*, a precondition may in addition be:

- A general negation, conjunction, disjunction, or implication:

  (**not** CONDITION_FORMULA)
  (**and** CONDITION_FORMULA1 ... CONDITION_FORMULA_N)
  (**or** CONDITION_FORMULA1 ... CONDITION_FORMULA_N)
  (imply CONDITION_FORMULA1 CONDITION_FORMULA_2)

- A quantified formula:

  (**forall** (?V1 ?V2 ...) CONDITION_FORMULA)
  (**exists** (?V1 ?V2 ...) CONDITION_FORMULA)

2.1.5 Effect Formulas

In PDDL, the effects of an action are not explicitly divided into adds and deletes, as in some other planning languages. Instead, negative effects (deletes) are specified using a negation operator. In a STRIPS domain, an effect formula may consist of the following:

- 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 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, or vice versa! 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 often but not always placed within quantifiers.

- A universally quantified formula:

  (**forall** (?V1 ?V2 ...) EFFECT_FORMULA)

2.2 PDDL Problem Definition

The problem definition first specifies its name and which domain it belongs to. It then specifies the objects present in the problem instance, the initial state of the world, 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)
)
```

The object list can be typed or untyped. The initial state description (the *:init* section) is simply a list of all the ground (variable-free) atoms that are true in the initial state. All other atoms are by definition false. The goal description is not a state or list of literals, but is instead expressed as a formula of the same form as an action precondition.

3 Survey of the work

The idea behind the implementation of the project is to build a finite-state machine. The program processes both the files separately: for each file it creates a state initially set to 0, which is increased or decreased depending on which token is found, until the final state is reached. If the token found is not correct, the program raises an exception based on the type of the error.
Using this approach, the syntax may be harder to check but allows an easier debugging, since we can keep trace of the line and the token number which raised the exception. On the other hand, many algorithms (such Fast Downward) process all the syntax in one step, so that the control is easier but they lose every information on the original structure of the file.
Having made the syntax check, the program performs a semantic check removing unexecutable and useless actions and useless predicates.
Finally, if both files don't contain errors, a message is printed with the predicates, actions, init, objects and goal.

The project has been written in Python 3. It contains the following files:

- checker.py: this program is the custom syntax checker;
- data/: this folder contains four files which indicate the state transitions:
  1. "d_states.txt" used to check a single elements in the domain;
  2. "dd_states.txt" used to check two elements in the domain;
  3. "p_states.txt" used to check a single elements in the problem;
  4. "dd_states.txt" used to check two elements in the problem.
- examples/: this folder contains some domain and problem files in order to test the program:
  1. "complete_domain.pddl" is a simple pddl file that can be used to test the exceptions;
  2. "unexecutable_action_domain.pddl" is a pddl file that can be used to check the removal of unexecutable actions;

3. “useless_action_domain.pddl” is a pddl file that can be used to check the removal of useless actions;
4. “useless_predicate_domain.pddl” is a pddl file that can be used to check the removal of useless predicates.

- src/: this folder contains some files of fast downward that have been modified for the integration:
  1. “translate/translate.py”;
  2. “translate/options.py”;
  3. “translate/pddl_parser/pddl_file.py”;
  4. “translate/pddl_parser/parsing_functions.py”.

The program can be run in two ways:

- independently: if all it is needed is just the syntax and semantic check, the program can be launched independently with the following command:

```
python3 checker.py [domain] [problem]
```

- integrated in fast-downward: since the program is completely integrated in fast-downward, it can be launched as usual to perform the custom check and find a solution of the plan. The steps to be done are:
  1. point at the folder of the project;
  2. execute the following commands:

```
hg clone http://hg.fast-downward.org DIRNAME
cp -r src DIRNAME
cd DIRNAME
./build.py
```

  3. run Fast Downward with the following command:

```
./fast-downward.py [domain] [problem] --search 'astar(blind())'
--translate-options --custom
```

4 Implementation

The program consists of a class PDDL_Checker, composed of several variables and 12 methods.

4.1 Variables

- names: name of domain and problem;
- files: files of domain and problem split by line;
- actions_to_remove: set of actions to be removed;
- predicates_not_to_remove: set of predicates not to be removed;
- curr_pred: predicate that is currently processed;
- predicates: set of all predicates along with their starting and ending line and token;
- requirements: set of all requirements;
- curr_params: parameters of the function that is currently processed;
- curr_length: number of parameters of the predicate that is currently processed;
- curr_act: action that is currently processed;
- actions: set of all actions along with their parameters, preconditions, effects, starting and ending line and token;
- objects: set of all objects;
- init_: set of all initial states;
- init_precs: set of all initial predicates;
- goals: set of all goals;
- not_: variable used for managing the negative initial predicates;
- state: state of the file that is currently processed;
- d_states: transitions for single element in domain file;
- dd_states: transitions for 2 elements in domain file;
- p_states: transitions for single element in problem file;
- pp_states: transitions for 2 elements in problem file.

4.2 Methods

- set_files: this function stores the domain and problem files split by lines;
- parse_code: this function takes as input the domain of problem file and processes them token by token with the functions parse_token_domain or parse_token_problem. For each file, the state is initially set to 0, and then increased or decreased depending on which token is found, until the final state is reached;

- check_element: this function takes as input the token, the current state, the line number and the pattern to search: if the pattern matches the token, it returns the current state increased by 1, otherwise it raises an exception. The main patterns used are the variable term that matches any alphanumeric string, the variable param that matches any alphanumeric string which starts with a question mark and the variable req that matches any alphanumeric string which starts with a colon;

- check_elements: this function takes as input the token, the current state, the line number and a list containing two patterns to search and a number n: if the token matches the first pattern, it returns the current state increased by 1, if it matches the second patterns it returns the current state increased by n, otherwise it raises an exception;

- parse_token_domain: this function takes as input the token, the current state and the line number; the simplest transitions, which don't need additional control, are managed by the dictionaries d_states (used to check a single element), dd_states (used to check two elements). More complex transitions (such as setting the domain name or storing the predicates) are explicitly controlled by the function. The keys are the states and the values are the patterns to search (and the number of states to skip for the function check_elements);

- parse_token_problem: this function takes as input the token, the current state and the line number and works in the same way as parse_token_domain, although the structure of the file is completely different; the transitions are controlled by the dictionaries p_states (used to check a single element), pp_states (used to check two elements);

- check_logic: this function takes as input the token, the current state, the line number, the set where the action predicates are to be added; a variable v indicates if the parameters to be searched are in form of variable (with a starting question mark) or objects (as in the case of the goal). This function doesn't modify the main state: instead, it creates new states for every operator it encounters, and returns the same main state until there are no more additional states, so that the function parse_token checks the logic until the end of the block, after which it will continue with the main check. The new state is added at the end of the list, so that when the next token is processed, the new state has to be deleted in order to get back to the preceding state. The states 0, 1, 2 are for the predicates, the states 3, 4 for the and and or operators, the states 5, 6 for the not operator, the states 7, 8, 9 for the forall and exists operators, and the states 11, 12 for the when operator. For example, if we have (and(?x)(?y)), when the operator and is processed, the state will become 3 and at the next token a new state 0 is appended in order to check the predicates; the sequence of states is: [], [0], [3], [4,0], [4,1], [4], [4,0], [4,1], [4], [];

- check_parameter: this function takes as input the token, the line number and the pattern to search; it is used to check if the parameter has been used twice or if it is not declared in the action: if not, it is added to the current parameters set;

- create_state: this function is used to create a new state: it increments by one the last state, it checks if the current token is an open parenthesis, and it appends a new state starting from 0;

- remove_state: this function is used to remove the last state: it checks if the current token is a closed parenthesis, and it remove the last state;

- remove_actions: this function process every action and removes the useless and unexecutable actions. An action is useless when all its effects are neither in the preconditions

of the other actions, neither in the goals predicates, while it is unexecutable when all its preconditions are neither in the effects of the other actions, neither in the initial predicates. Also, it takes into account negative predicates. When a useless or unexecutable action is found, it removes the action, prints the new actions and the check stars over in order to check other actions that could depend on the one removed. Finally, the domain file is updated; it's important to note that an action is useless depending on the goal, so the resulting file can be different based on the current plan;

- update_domain: this function removes useless predicates and saves a new domain file called new_domain.pdll without the lines containing the useless or unexecutable actions or the useless predicates;

4.3 Exceptions

The exceptions are:

- Missing "%s" at line %d: this exception is raised if the program expects a specific character but there is another one;
- Missing "%s" or %s at line %d: this exception is raised if the program expects 2 characters but there is another one;
- Token "%s" not recognized at line %d: this exception is raised if there is some character at the end of the file;
- Missing character at line %d: this exception is raised if some character is missing at the end of the file;
- Different domain name at line %d: this exception is raised if the domain name and the one specified in the problem file are different;
- Predicate "%s" not declared at line %d: this exception is raised if the predicate used in the action or in the init is not declared in the predicates section;
- Object "%s" not declared at line %d: this exception is raised if the object used in the in the init or in the goal is not declared in the objects section;
- Different number of parameters in action "%s" at line %d: this exception is raised if the number of parameters of the predicates in the action is different than that specified in the predicates section;
- Parameter "%s" already declared in action "%s" at line %d: this exception is raised if the same parameter has been used twice in the action.
- Different number of parameters in predicate "%s" at line %d: this exception is raised if there are different number of parameters in at least one predicate.
- Domain file not found.
- Problem file not found.
- The first argument must be the domain file.
- The second argument must be the problem file.
- Too many arguments.
- Too less arguments.
- The command to launch it is: "python checker.py [domain_file] [problem_file]".

4.4 Integration in Fast Downward

The project has been completely integrated in Fast Downward, which can be run with the flags "–translate-options" and "–custom" in order to perform the custom syntax check. The first flag is necessary because the Fast Downward requires it if you want to add custom flags. The file modified are four:

- src/translate/translate.py:

  the lines 676-677

  ```
  task = pddl_parser.open(domain_filename=options.domain,
  task_filename=options.task)
  ```

  have been modified to

  ```
  task = pddl_parser.open(domain_filename=options.domain,
  task_filename=options.task, custom=options.custom)
  ```

- src/translate/options.py:

  the block in lines 54-57

  ```
  argparser.add_argument(
      "--custom", action="store_true",
      help="use custom parser")
  ```

  has been added

- src/translate/pddl_parser/pddl_file.py:

  the line 10

  ```
  sys.path.append(os.path.abspath(os.pardir))
  ```

  has been added and the lines 34-37

  ```
  domain_pddl = parse_pddl_file("domain", domain_filename)
  task_pddl = parse_pddl_file("task", task_filename)
  return parsing_functions.parse_task(domain_pddl, task_pddl)
  ```

  have been modified to

  ```
  if not custom:
      domain_pddl = parse_pddl_file("domain", domain_filename)
      task_pddl = parse_pddl_file("task", task_filename)
      return parsing_functions.parse_task(domain_pddl,
      task_pddl, False)

  else:
      parser = checker.PDDL_Checker()
      return parsing_functions.parse_task(domain_filename,
      task_filename, parser)
  ```

- src/translate/pddl_parser/parsing_functions.py:

  the lines 307-309 in the function parse_task

```
domain_name, domain_requirements, types, type_dict, constants,
predicates, predicate_dict, functions, actions,
axioms = parse_domain_pddl(domain_pddl)

task_name, task_domain_name, task_requirements, objects, init,
goal, use_metric = parse_task_pddl(task_pddl, type_dict, predicate_dict)
```

  have beeen modified to

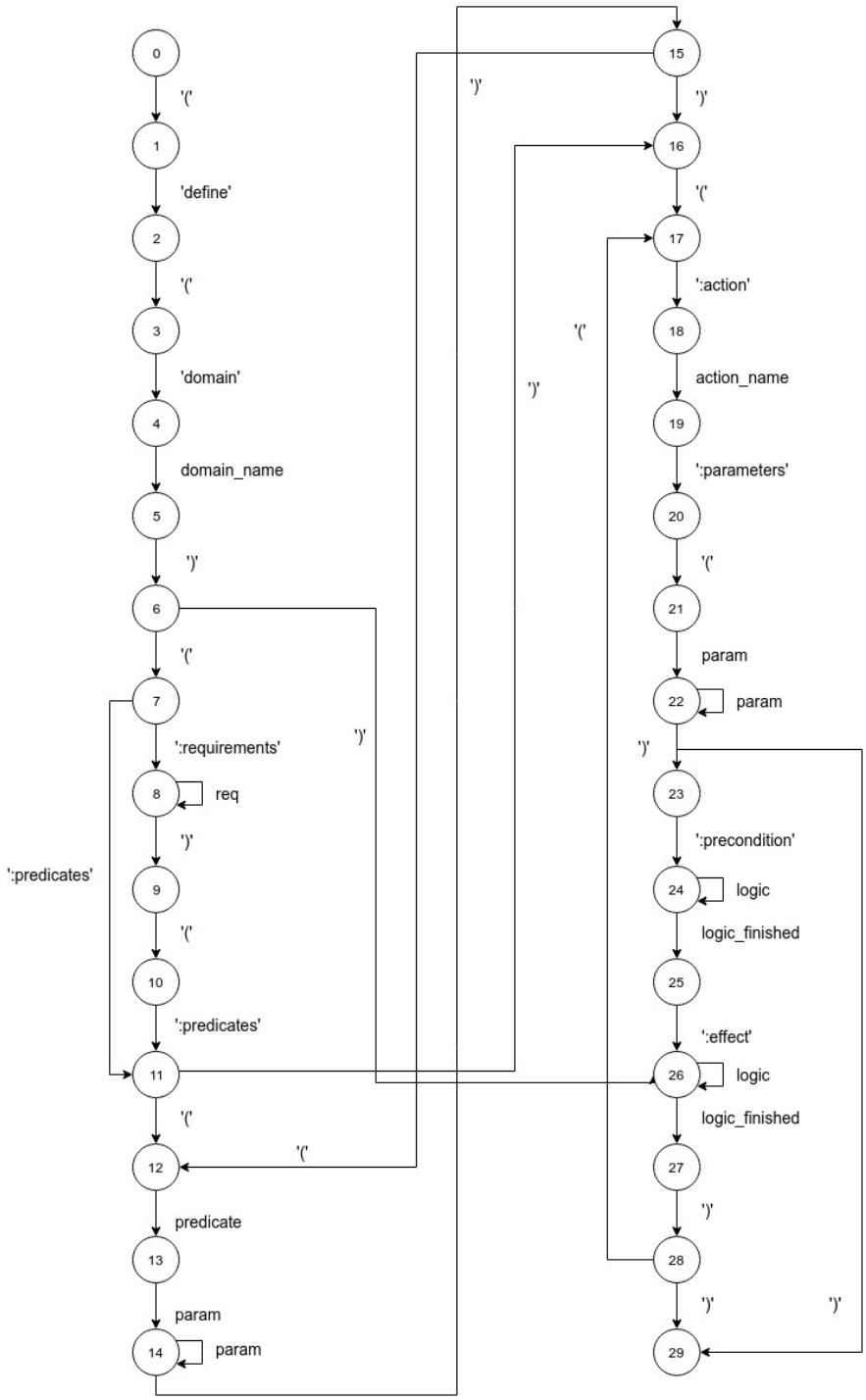
```
if not parser:
    domain_name, domain_requirements, types, type_dict, constants,
    predicates, predicate_dict, functions, actions,
    axioms = parse_domain_pddl(domain_pddl)

    task_name, task_domain_name, task_requirements, objects, init,
    goal, use_metric = parse_task_pddl(task_pddl, type_dict,
    predicate_dict)

else:
    # set files
    parser.set_files(domain_pddl, task_pddl)
    domain_name, domain_requirements, types, type_dict, constants,
    predicates, predicate_dict, functions, actions,
    axioms = parse_domain_custom(parser)

    task_name, task_domain_name, task_requirements, objects, init,
    goal, use_metric = parse_task_custom(parser, type_dict,
    predicate_dict)

    # remove actions
    parser.remove_actions()

    # update domain
    parser.update_domain()
```

  and the functions parse_domain_custom and parse_task_custom have been added in order to change the outputs of the custom syntax checker to the types that Fast Downward wants to find the solution for the plan.

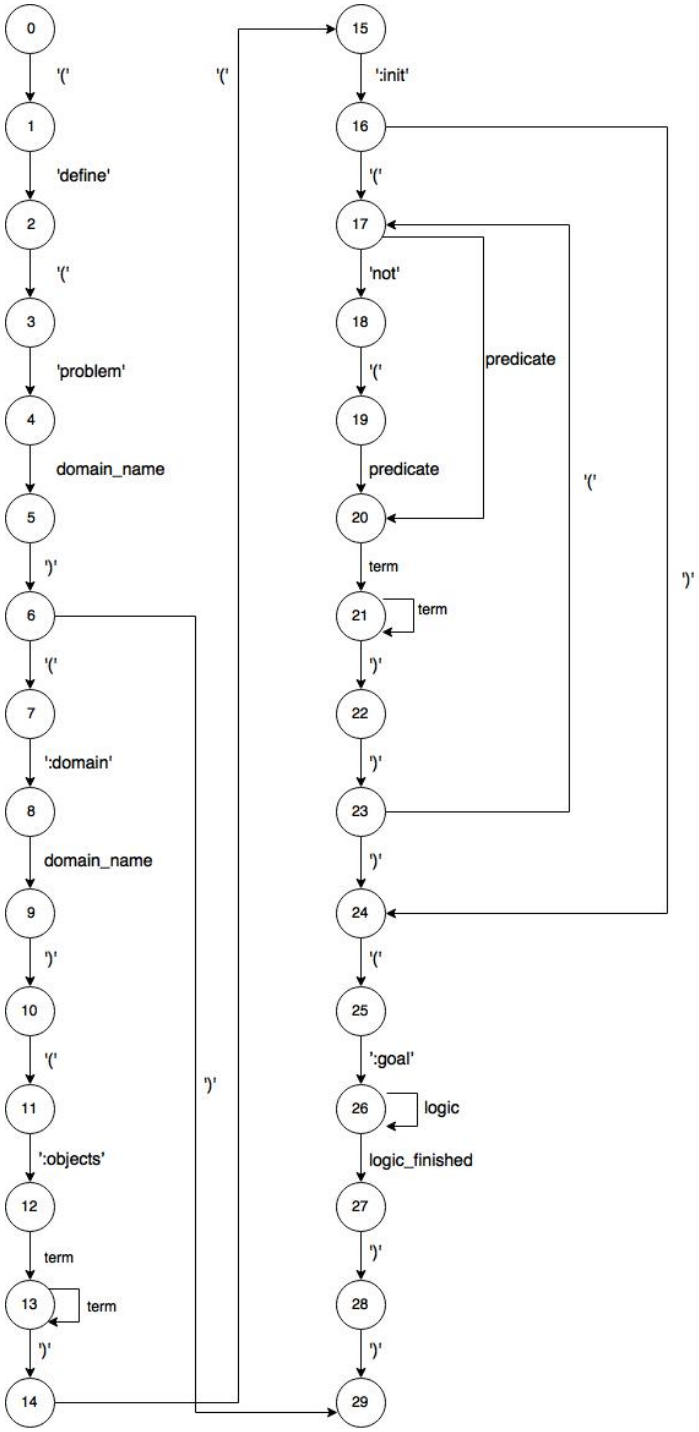
4.5 State Transitions

The following figures indicate the transitions between the states along with the element to check.

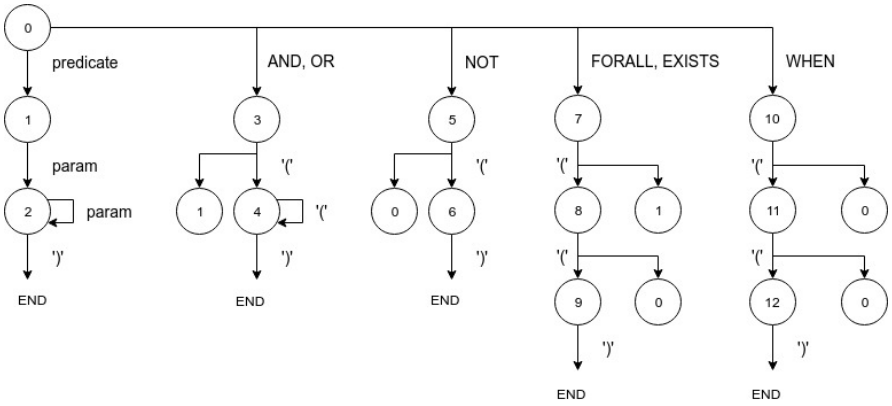
4.5.1 Domain



4.5.2 Problem


0
'('
1
'define'
2
'('
3
'problem'
4
domain_name
5
')'
6
'('
7
':domain'
8
domain_name
9
')'
10
'('
11
':objects'
12
term
13
term
')'
14
'('
15
':init'
16
'('
17
'not'
18
'('
19
predicate
20
term
21
term
')'
22
')'
23
')'
24
'('
25
':goal'
26
logic
logic_finished
27
')'
28
')'
29
')'
')'
predicate
'('
')'

4.5.3 Action


0
predicate
AND, OR
NOT
FORALL, EXISTS
WHEN
1
param
2
param
')'
END
3
'('
1
4
'('
')'
END
5
'('
0
6
')'
END
7
'('
8
1
'('
9
0
')'
END
10
'('
11
0
'('
12
0
')'
END