PDDLyte Language Reference Manual

John Martin Jr.
jdm2213@columbia.edu

March 14, 2014

Contents

Introduction

The PDDLYTE language, whose name derives from the Planning Domain Definition Language (PDDL)[1], is a symbolic, specification language used to formulate and solve planning problems.

Let's quickly move into an example to motivate further explanation. Now the simplest, logically-coherent PDDLYTE program only translates input to output.

```
(define (domain square)
        (:predicates (at ?what ?where))
        (:action squat
         :parameters  ()
         :precondition()
         :effect      ()
        )
)

(define (problem simple)
        (:domain square)
        (:objects ant sq)
        (:init (at ant sq))
        (:goal (at ant sq))
)
```

From the program we can deduce there are two different definitions. One seems to describe a space of sorts. And the other definition relates to a problem. In essence, this is all a PDDLYTE program is: two definitions. The *domain definition* describes a planning environment. Whereas the *problem definition* describes a plan configuration. It turns out that a domain and problem description, as we've formulated it, are all anyone needs to formulate a general planning problem! Plans are formulated with an initial state, a goal description, and a domain on which to plan over. Provided the problem can be solved, the solution can be represented as a sequence of actions, providing a map that leads from an initial state to a goal state(s).

PDDLYTE can describe much more interesting problems than what has been shown. After understanding the contents of this document, the reader should be able to recognize the PDDLYTE's expressive power. Solving classical planning problems will then become simple.

Lexical Conventions

In the most basic form, programs are comprised of two symbolic expressions. One is a domain definition; the other is a problem definition. Beyond symbolism, three atomic datatypes, white space, and comments compose programs.

Tokens

There are three primary token classes: atoms, keywords, and separators.

Symbolic Expressions

Symbolic expressions are recursively-defined data types which can nest atoms and lists to an arbitrary depth – succinctly referred to as *s-expressions*.

$$\begin{aligned} \langle \text{s-expression} \rangle &\models \langle \text{atom} \rangle \mid \langle \text{list} \rangle \\ \langle \text{atom} \rangle &\models \lambda \mid \langle \text{number} \rangle \mid \langle \text{identifier} \rangle \\ \langle \text{list} \rangle &\models ( ) \mid (\langle \text{s-expression} \rangle \cdot \langle \text{list} \rangle) \end{aligned}$$

Note: the empty list is a valid symbolic expression.

S-expressions are comprised of atoms, symbolically-expressed dotted pairs[1], and lists.

Atoms

Atoms are empty, numeric, or symbolic.

$$\begin{aligned} \langle \text{atom} \rangle &\models \lambda \mid \langle \text{number} \rangle \mid \langle \text{symbol} \rangle \\ \langle \text{number} \rangle &\models (0 \ldots 9)^{+} \\ \langle \text{symbol} \rangle &\models (\text{a} \ldots \text{z} \mid + \mid - \mid * \mid / \mid \_ \mid < \mid > \mid ! \mid ? \mid :)^{+} \end{aligned}$$

Figure 1. decomposes symbolic expressions to show their connection to atoms and lists.

Comments

Comments begin with a semicolon (;) and terminate with the line it occupies. Furthermore, they do not nest and may not be composed within other comments. Words beyond the semi-colon, to the end of the line are invisible to evaluation.

$$\langle \text{comment} \rangle \models ;$$

[1] A dotted pair can be thought of as a construction of two units.

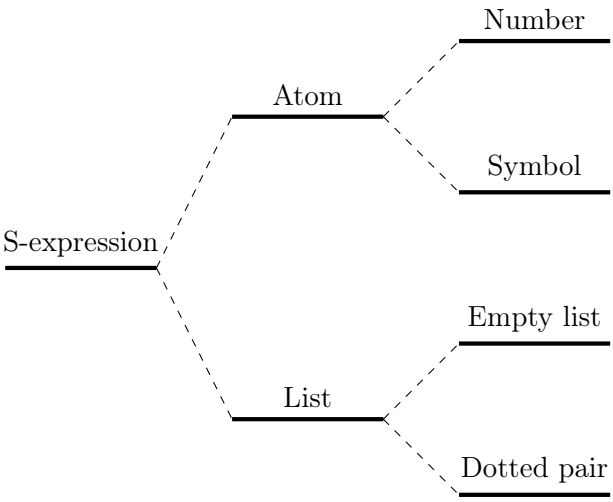

Figure 1 Decomposition of symbolic expressions[2]

Whitespace

Whitespace consists of any sequence of blank, tab, or newline characters.

Symbols

Symbolic atoms are strings of alphanumeric characters used to label the various objects in a planning problem. *Symbol* is a colloquialism for *symbolic atom*.

Variables

Variables label a symbol's spot in memory. These are immutable unless prefixed with question marks.

$$\langle \text{variable} \rangle \quad \models \quad ?\langle \text{symbol} \rangle$$

Keywords

Any symbol prefixed with a colon is a keyword.

$$\langle \text{keyword} \rangle \quad \models \quad :\langle \text{symbol} \rangle$$

Separators

Separators serve to distinguish different data types. Parentheses mark the boundaries of lists. While spaces separate data within a list.

$$\langle \text{separator} \rangle \quad \models \quad (\quad | \quad) \quad | \quad \text{␣}$$

Procedures

Symbols are used to label procedures and their parameters.

$$\langle \text{procedure} \rangle \quad \models \quad (\langle \text{symbol} \rangle \, \langle \text{list} \rangle \, \langle \text{list} \rangle^{+} \,)$$

Procedures are fully explained in the next section.

Bindings

Variable bindings are not supported outside plan specifications and procedures.

S-expressions

PDDLYTE derives enormous representational power from symbolic expressions. S-expressions represent both procedures and data. This implies that, in principle, PDDLYTE programs can be used as data and even generated as output!

Procedures Revisited

Procedures consist of a name, parameter list, and body. *Names* are symbols. *Parameters* are variables, bound and assigned upon procedure entry. The *body* of a procedure consists of the forms that are evaluated when the procedure is used.
Processing is identical to LISP:

- *Bind* parameters[2]
- *Assign* parameters upon entry to a procedure[3]
- *Evaluate* the body with the parameters and return the result[4]
- Discard the argument binding

All procedures are evaluated as symbolic expressions; so there's no precedence nor associativity to consider.

Basic Procedures

Several basic procedures are included with PDDLYTE.

Conjunction — Logical conjunctions are formed with the `and` predicate:

$$\langle \text{conjunction} \rangle \quad \models \quad (\text{and } \langle \text{list} \rangle\ )$$

Disjunction — Logical disjunctions are formed with the `or` predicate:

$$\langle \text{disjunction} \rangle \quad \models \quad (\text{or } \langle \text{list} \rangle\ )$$

Negation — A logical conjunction is inverted with the `not` predicate:

$$\langle \text{negation} \rangle \quad \models \quad (\text{not } \langle \text{list} \rangle\ )$$

The function returns `T` if its argument is nil, otherwise `nil`.

[2]Binding is considered the process of reserving memory for a symbol.
[3]Assignment is considered the process of storing a value, for a specified binding, in memory.
[4]Evaluation, a nebulous term, here, is considered a cascading process of assignment value recovery.

Planning Procedures

Other procedures are used to specify planning problems.

Procedure definition — planning procedures are created with the `define` predicate:

$$\langle \text{definition} \rangle \quad \models \quad (\text{define } (\langle \text{symbol} \rangle \ \langle \text{symbol} \rangle) \ \langle \text{list} \rangle \ \langle \text{list} \rangle^{+})$$

This shares semantics with Lisp, but is restricted to support the only two planning procedures of Pddlyte.

Domain — The domain procedure establishes variable bindings and transition operators of the planning environment. Only a single domain procedure is permitted per file.

$$\langle \text{domain} \rangle \quad \models \quad (\text{domain } \langle \text{symbol} \rangle) \ \langle \text{type} \rangle \ \langle \text{action} \rangle \ \langle \text{predicate} \rangle$$

The expansion of this grammar is deferred to the next section.

Problems — The problem procedure establishes variable bindings that configure the planning graph.

$$\begin{aligned} \langle \text{problem} \rangle &\quad \models \quad (\text{problem } \langle \text{symbol} \rangle) \ \langle \text{domain name} \rangle \ \langle \text{object} \rangle \ \langle \text{initial state} \rangle \ \langle \text{goal} \rangle \\ \langle \text{domain name} \rangle &\quad \models \quad (\text{:domain } \langle \text{symbol} \rangle) \end{aligned}$$

The remaining grammar has been deferred to the next section.

Planning S-expressions

Types — Types are symbols that specify objects of the domain. This attribute is an extension of PDDL, but will be inherently supported with PDDLyte.

$$\langle \text{type} \rangle \quad \models \quad (\text{:types } \langle \text{symbol} \rangle^{+})$$

Actions — Actions are conditions which define system transitions.

$$\begin{aligned} \langle \text{action} \rangle &\quad \models \quad (\text{:action } \langle \text{parameter} \rangle \ \langle \text{pre-condition} \rangle \ \langle \text{effect} \rangle) \\ \langle \text{parameter} \rangle &\quad \models \quad (\text{:parameters } \langle \text{symbol} \rangle^{+}) \\ \langle \text{pre-condition} \rangle &\quad \models \quad (\text{:precondition } \langle \text{list} \rangle^{+}) \\ \langle \text{effect} \rangle &\quad \models \quad (\text{:effect } \langle \text{list} \rangle^{+}) \end{aligned}$$

Pre-conditions are logical conjunctions that must be satisfied before transitions take place. *Effects* are logical conjunctions which must be satisfied and valid before the transition is completed. Actions with no pre-conditions are always valid. The parameters are those used in the conditions.

Predicates — Predicates define relationships between object variables. These can be static relations that hold from state to state or fluent relations. Each predicate is defined with a name symbol and one or more name-type pairs; where the object name is separated from the type with a dash:

$$\langle \text{predicate} \rangle \quad \models \quad (\text{:predicates } \langle \text{list} \rangle^{+})$$

An example predicate has been shown for clarity.

```
(:predicates (pred ?<name> - <type>))
```

Objects — Objects describe types that exist in a problem, but that are absent from the domain specification.

$$\langle \text{object} \rangle \quad \models \quad (\text{:objects } \langle \text{list} \rangle^{*})$$

Note: This s-expression is optional.

Initial State — The initial state defines the conditions that are true in the system's starting configuration.

$$\langle \text{initial state} \rangle \quad \models \quad (\text{:init } \langle \text{list} \rangle^{+})$$

Goal Description — The goal description, analogous to the initial state, describes the terminal conditions.

$$\langle \text{goal} \rangle \quad \models \quad (\text{:goal } \langle \text{list} \rangle^{+})$$

Bibliography

[1] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl — the planning domain definition language. Technical report, AIPS Planning Competition Committee.

[2] Patrick Henry Winston and Berthhold Klaus Paul Horn. *LISP*. Addison Wesley.