PDDLyte
A Partial Implementation of The Planning Domain Definition Language

John Martin Jr.
jdm2213@columbia.edu

February 4, 2014

Contents

Introduction

The PDDLyte language, whose name derives from the Planning Domain Definition Language (PDDL)[?], is a specification language used to find solutions to planning problems. Similarly to PDDL, problems are specified with an initial state, a set of actions, and a set of goal states. From there, PPDLyte uses causal reasoning to deduce solutions, provided they exist, with a sequence of actions that map the initial state to the goal state(s). The PDDLyte language is limited in comparison to its predecessor, in that it only supports classical planning problems for a single agent: finite, observable, deterministic, fully-accessible environment descriptions. Another distinguishing characteristic of PDDLyte is in the way it's compiled.

Current PDDL implementations use a LISP interpreter to verify the solutions. This is where the life of PDDL ends. The PDDLyte implementation will take a .pdly file as input and will generate C code as an intermediate product; after which it is translated into X86 assembly code. With this design, the high-level reasoning of PDDLyte solutions can easily interface with systems-level C code. This language can be used to express high-level constraints on system. In essence, this will generate a library.

Can I manage to provide graphical output?

Language Tutorial

Language Reference Manual

The syntax of my language is chosen to be nearly identical to PDDL, sans all the parenthesis.

Datatypes

PDDLyte is a strongly-typed language. It inherits nearly all its syntax from PDDL – sans the annoying parenthesis – and only a subset of the datatypes. This is intended to provide the minimum set, required to solve classical-planning problems. The basic types in PDDLyte are boolean, integer, character, strings, and user-defined types.
Domains — Domains are the global problem descriptor. The requirements, predicates, actions, constants, etc. will be defined in this structure and used to. Only a single domain may be defined per file.
Problems — Problems include an initial state and one or more goal descriptions. All problems are associated with a domain, which provides the context for the solution to be in.
Initial State — define the predicates that are true in the system's starting configuration. This can be any valid state within the domain. Unless predicates are specified as true in this statement, they are set false.

Goal Description — The goal description defines the predicate(s) that are true in the system's final configuration(s)

Actions — Actions are the operators that transition the system between states. These are the basic elements which a solution is built[?]. There are pre-conditions and effects associated with each action. The pre-conditions must be satisfied for the transition to take place. Furthermore, the effects must be valid according to an active problem description. If no pre-conditions are specified, then an action is always valid. The action definition must know the type of parameter its preconditions are composed of.

Predicates — Predicates consist of a list of declarations that define a property of an object. These can be either true or false.

Effects — Effects describe changes that occur when an action is completed. Predicate states are not affected in anyway by Effects. When an action is complete, the effect is simply set. It is the responsibility of the user to negate the preconditions of an action upon completion. Effects may be conditional.

Types — PDDLyte allows for any object to be defined. As far as the language is concerned, these are just labels for memory.

Axioms — Axioms are logical formulas that enforce relationships among propositions in a static state. Each axiom is applied in a specified context (i.e. state) and has an implication.

Requirements — packages to be used. I'm not sure how to implement this

Constants — this is a symbol, representing a type, which has the same meaning for all problems of a given domain [?].

Comments — Comments begin with `/*` and end in `*/`. Furthermore, they do not nest and may not be composed within comments.

In addition to the planning specification datatypes, the PDDLyte language supports several primitive datatypes, which have identical counterparts in C.

Datatype	Description
`int`	Integer $\mathbb{Z}$
`char`	Character
`string`	String of characters
`either`	Union of multiple types
`fluent`	Static type with variable value

Scope

Name	Scope
Domain name	Global
Problem name	Global
Constant	Domain
Type	Domain
Action	Domain
Predicate	Domain
Effect	Domain

Control Statements

- not
- and
- or
- forall

- exists
- when — If a predicate is true before an action, then the effect occurs afterward.

Operators

Operators are listed below, in order of precedence.

Arithmetic operators

Operator	Description	Type
`%`	Modulo	$\mathbb{Z}$
`*`	Multiplication	all
`/`	Division	all
`+`	Addition	all
`-`	Subtraction	all

Conditional Operators

Operator	Description	Type
`==`	Equals	all
`!=`	not equal	all
`<`	Less than	all
`>`	Greater than	all
`<=`	Less than or equal to	all
`>=`	Greater than or equal to	all
`||`	Logical OR	boolean
`&&`	Logical AND	boolean
`|`	Bit-wise OR	all
`&`	Bit-wise AND	all

Architecture

Project Plan

The project schedule was inspired from previous students' plans. I have a total of fourteen weeks to organize, code, test, and finalize the code.
1.) Determine language objectives — ???
2.) Design the language syntax — ???
a.) Write the scanner and parser — ???
b.) Design the syntax tree types — ???
3.) Write code to output tree to graphviz
4.) Design the byte code — ???
5.) Implement the transition logic — ???
6.) Project report — ???
7.) Test — ???

Version control was accomplished with GitHub. Automating the build process is done with Make files and bash shell scripts. GraphViz will be used for visualization in the debug phase. And finally, LaTeXis utilized for document generation.

Test Plan

Lessons Learned

Functions

- exist() – checks the existence of a data type [?].
- at() – returns true if the system occupies the input
- require() – to allow for extensibility, this function links in other code.

test and set

Example programs

Monkeys and Bananas

A monkey is in a room tormented by a bundle of bananas hanging from the ceiling, just out of its reach. Inside the room, there is also a chair and a stick. Both items are proportioned just right to allow the monkey enough reach to retrieve the bananas – provided the monkey has the sense to use the items properly. Once the chair is positioned under the bananas, the monkey can step up and use the stick to wackk the bundle free.

Rule 90