

PDDLyte  
A Partial Implementation  
of  
The Planning Domain Definition Language

John Martin Jr.  
jdm2213@columbia.edu

May 14, 2014

# Contents

Introduction	iii
Language Tutorial: Pacman . . . . .	iii
Executing a program . . . . .	iv
Example program walk through . . . . .	v
Language Reference Manual	vi
Lexical Conventions . . . . .	vi
Comments . . . . .	vi
Whitespace . . . . .	vi
Symbolic Expressions . . . . .	vi
Atoms . . . . .	vii
Symbols . . . . .	vii
Atom Variants . . . . .	vii
S-expressions . . . . .	vii
Predicates . . . . .	viii
Conjunctions . . . . .	viii
Define . . . . .	viii
Domains . . . . .	viii
Operators . . . . .	viii
Problems . . . . .	ix
Scope . . . . .	x
Architecture	xi
Processing Sequence . . . . .	xi
Planner . . . . .	xi
Graph Traversal . . . . .	xii
Graph Generation . . . . .	xii
Bytecode Instructions . . . . .	xiv
Bytecode Interpreter . . . . .	xv
Limitations . . . . .	xv
Lessons Learned	xvi
Language Design . . . . .	xvi
Project Log . . . . .	xvi
Testing . . . . .	xxvii
Source Code	xxviii
Bibliography	lxvii

## Introduction

PDDL<sub>Y</sub>TE is a symbolic planning language used to specify and solve STRIPS planning problems[2]. The language adopts a minimal subset of syntax from its more expressive predecessor PDDL: the Planning Domain Definition Language[1]. When planning problems are formalized in PDDL<sub>Y</sub>TE, state-space graphs are generated and traversed for solutions. Provided a solution is available, the plan will be executed and returned.

## Language Tutorial: Pacman

A great application for PDDL<sub>Y</sub>TE comes from a video game. Consider Pacman and his persistent desire for food. He wishes to eat the bananas located in the third square  $g$  when he originates in the first square,  $s_0$ .

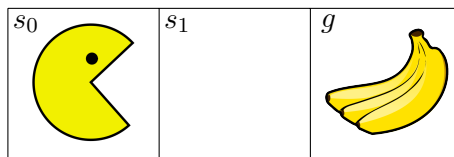


Figure 1: Pacman Domain

In the example program `pacman.pdly`, Pacman's intentions are made explicit with PDDL<sub>Y</sub>TE. His initial spot on the grid is labeled with `:init`, and his desired spot with `:goal`. The compiler's job will be to connect these two spots with valid instances of the move action.

Now this problem is so simple that we can visualize every possible move in our heads. That image should resemble something like this abstract graph: Clearly the best way to

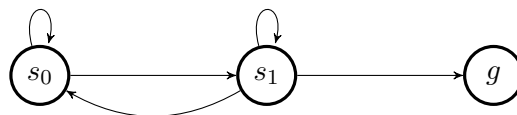


Figure 2: Pacman State-space Graph

reach the goal is to move to  $s_1$ , then move to  $g$ . Let's verify our intuition by executing the program.

## pacman.pdly

```

1 (define (domain pman)
2   (:predicates
3     (adj ?s1-square ?s2-square)
4     (at ?what-thing ?si-square)
5     (fix ?what-thing ?si-square)
6   )
7
8   (:action move
9     :parameters (?who-thing ?from-square ?to-square)
10    :precondition (and (adj ?from ?to) (at ?who ?from))
11    :effect (and (not (at ?who ?from)) (at ?who ?to))
12  )
13 )
14
15 (define (problem pman_prob)
16   (:objects
17     s0-square s1-square g-square
18     pacman-thing bananas-thing
19   )
20
21   (:init
22     (adj s0 s1) (adj s1 s0)
23     (adj s1 g) (adj g s1)
24     (fix bananas g)
25     (at pacman s0)
26   )
27
28   (:goal
29     (fix bananas g)
30     (at pacman g)
31   )
32 )

```

---

To solve and execute this example program run

```

$./pddlyte -e pacman.pdly
move( pacman s0 s1 )
move( pacman s1 g )

```

Our intuition is correct! Pacman can reach the goal state if he moves from  $s_0$  to  $s_1$ , then from  $s_1$  to  $g$ .

## Executing a program

To execute a program in general, use the `-e` flag when calling `pddlyte`. This will compile the input file and invoke the planner. The solution will be further compiled, executed, and output.

```
$./pddlyte -e [pddlyte file]
```

To output the bytecode instructions used to simulate plan execution, use the `-c` flag.

```
$. /pddlyte -c [pddlyte file]
```

To output the entire path from start to end use the `-p` flag.

```
$. /pddlyte -p [pddlyte file]
```

## Example program walk through

The Pacman program starts with a description of the grid domain. The grid is arranged with three adjacent squares that Pacman and the banana can occupy. The `:predicates` expression declares facts about the grid that can apply for any given state. A state where `(adj s0 s1)` applies is one where square  $s_0$  is adjacent to  $s_1$ . Similarly, a state where `(at pacman s0)` applies is one where Pacman is at  $s_0$ . And a state where `(fix bananas g)` is one where the bananas are fixed at the goal square  $g$ .

Pacman transitions between adjacent squares according to the move operator. Moves are only applicable when the `:precondition` has been satisfied. To move from one square to another, the start and end squares must be adjacent, and the thing being moved must occupy the starting square:

```
(and (adj ?from ?to) (at ?who ?from)).
```

The movement causes the thing to disappear from where it started and appear at its destination. This is described with a conjunction of predicates in the `:effect` expression:

```
(and (not (at ?who ?from)) (at ?who ?to)).
```

When formally written, the move operator uses three `:parameters` to describe how a thing can transition from one square to another – something like

```
move( ?who ?from ?to).
```

The remaining code details the problem specifics. Initially, Pacman occupies  $s_0$ ; the bananas occupy  $g$ , and the grid is arranged as illustrated in Figure 1. The equivalent set of predicates comprises the `:init` expression. For the goal state, all that matters is Pacman gets to the bananas. Therefore, the `:goal` expression just contains a minimum set of predicates needed to describe the terminal condition:

```
(fix bananas g) (at pacman g).
```

Further details on program syntax and semantics can be found in chapter 2: Language Reference Manual.

# Language Reference Manual

## Lexical Conventions

PDDL<sub>Y</sub>TE programs consist of a domain and problem. Domains contain all the declarations that shape the search space. Problems contain the minimal set of definitions needed to configure the planner with the search space.

The PDDL<sub>Y</sub>TE compiler can get away with using a minimal front end because all evaluation is done symbolically. There are only three primary token classes: symbols, comments, and parentheses. Parentheses only serve to separate symbolic expressions.

### 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.

### Whitespace

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

### 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*. This is the primary data type used to represent every PDDL<sub>Y</sub>TE expression. As a result, there's no precedence nor associativity to consider.

$$\langle s\text{-expression} \rangle ::= \langle atom \rangle \mid \langle list \rangle$$
$$\langle atom \rangle ::= \epsilon \mid \langle symbol \rangle$$
$$\langle list \rangle ::= (\epsilon) \mid \langle s\text{-expression} \rangle . \langle list \rangle$$

*Note: the empty list is a valid symbolic expression.*

S-expressions are comprised of atoms, symbolically-expressed dotted pairs<sup>1</sup>, and lists.

---

<sup>1</sup>A dotted pair can be thought of as a construction of two units. This is needed to represent mixed lists.

## Atoms

Atoms can be empty or symbolic. Symbolic characters take on elements from a subset of the ASCII set.

$$\langle symbol \rangle ::= \langle char \rangle (\langle char \rangle \mid \langle digit \rangle)^*$$

$$\langle char \rangle ::= 'A'-'Z' \mid 'a'-'z' \mid '+' \mid '-' \mid '*' \mid '/' \mid '_' \mid '<' \mid '>' \mid '!' \mid '?' \mid ':'$$

$$\langle digit \rangle ::= '0'-'9'$$

## Symbols

Symbolic atoms are alphanumeric strings used to represent every object of a program. *Symbol* is a colloquialism for *symbolic atom*.

### Atom Variants

There are two distinct contexts in which atoms are used. They're used as variables in the domain declarations and as concrete values in the problem specification.

$$\langle variable\ atom \rangle ::= '?' \langle symbol \rangle '-' \langle symbol \rangle \mid '?' \langle symbol \rangle$$

Variable atoms are distinguished from their grounded counterparts with the question mark prefix. Parameter declarations require variable atoms to append their types as a suffix. This is shown in the grammar separated with a dash. Explicitly identifying parameters with a type allows the compiler to check the predicates are being used consistently throughout the program. The types are not explicitly required in the bodies of action schemas.

$$\langle grounded\ atom \rangle ::= \langle symbol \rangle '-' \langle symbol \rangle \mid \langle symbol \rangle$$

Grounded atoms are used when symbols need to take on concrete values. This occurs exclusively in the problem statement. Type is explicitly required in the object declarations. All other instances of grounded atoms only specify values.

When types are specified everything past the first instance of a dash is considered the symbol type.

## S-expressions

S-expressions are used to represent the compound datatypes of PDDL<sub>YTE</sub>.

## Predicates

The most primitive s-expression is a predicate. A predicate defines a relationship for one or two symbol parameters. This can be a static relation that holds from state to state or a fluent relation. A predicate will remain static unless it appears in an applied operator's body. The leftmost symbol refers to the predicate's name.

$$\begin{aligned} \langle \textit{predicate} \rangle ::= & (\langle \textit{symbol} \rangle \langle \textit{variable atom} \rangle \langle \textit{variable atom} \rangle) \\ & | (\langle \textit{symbol} \rangle \langle \textit{grounded atom} \rangle \langle \textit{grounded atom} \rangle) \\ & | (\langle \textit{symbol} \rangle \langle \textit{variable atom} \rangle) \\ & | (\langle \textit{symbol} \rangle \langle \textit{grounded atom} \rangle) \end{aligned}$$

## Conjunctions

Conjunctions group a set of predicates as a single statement with the `and` prefix. A conjunction can be positive, negative, or mixed. The `not` prefix refers to a single negative predicate.

$$\begin{aligned} \langle \textit{conjunction} \rangle ::= & (\textit{'and'} (\langle \textit{positive conjunction} \rangle | \langle \textit{negative conjunction} \rangle))^+ \\ & | \langle \textit{negative conjunction} \rangle ::= (\textit{'not'} \langle \textit{predicate} \rangle) \\ & | \langle \textit{positive conjunction} \rangle ::= \langle \textit{predicate} \rangle \end{aligned}$$

## Define

The two s-expressions that represent PDDL<sub>YTE</sub> programs are defined with the `define` predicate:

$$\langle \textit{definition} \rangle ::= (\textit{'define'} \langle \textit{domain} \rangle | \langle \textit{problem} \rangle )$$

## Domains

The domain s-expression contains declarations of every predicate and operator.

$$\langle \textit{domain} \rangle ::= (\textit{'domain'} \langle \textit{symbol} \rangle) \langle \textit{predicates} \rangle \langle \textit{operator} \rangle^+$$

The `:predicates` prefix is used to declare every predicate in a program. Variable parameter types must be explicitly specified for all declared predicates. Predicate names must all be unique.

$$\langle \textit{predicates} \rangle ::= (\textit{'predicates'} \langle \textit{predicate} \rangle)^+$$

## Operators

Operators describe transitions in the planning space. The `:action` prefix is used to declare an operator. The term *action* refers to a grounded instance<sup>2</sup> of a variable *operator* declaration. All operator names must be unique.

---

<sup>2</sup>The term *grounded* indicates parameters are bound to concrete values.



$$\langle operator \rangle ::= (':action' \langle symbol \rangle \langle parameters \rangle \langle pre-condition \rangle \langle effect \rangle)$$

Operator parameters are declared with the `:parameters` s-expression. The list of variable atoms must specify their types explicitly. Parameter binding occurs from left to right. Parameter names are local to the operator body and must be unique.

$$\langle parameters \rangle ::= (':parameters' \langle variable atom \rangle^+)$$

The *precondition* is a conjunction that must be true before a transition takes place. The conjunction is mixed with positive and negative variable predicates. The predicate parameters reference the `:parameters` declaration; so explicit type specification is not appropriate. However, their names should match the declaration.

$$\langle pre-condition \rangle ::= (':precondition' \langle conjunction \rangle )$$

The *effect* is a conjunction that materialize after transitions take place. Effect declarations share syntax with `:precondition`, but with an additional balancing constraint. For every negative predicate there must be a matching positive predicate that maintains the total number of state predicates. This constraint is a product of using STRIPS states. Negative predicates are removed from a state. Then positive predicates are added.

$$\langle effect \rangle ::= (':effect' \langle conjunction \rangle)$$

## Problems

Problem s-expressions configure the planning graph with the initial and final state.

$$\langle problem \rangle ::= (':problem' \langle symbol \rangle) \langle objects \rangle \langle initial state \rangle \langle goal state \rangle$$

*Objects* are the concrete entities that exist in a problem. They provide symbolic values to ground variable predicates. The object set is declared with a list of grounded atoms. Each atom must explicitly specify its type. Atom names must be unique.

$$\langle objects \rangle ::= (':objects' \langle grounded atom \rangle^+)$$

The initial state defines the conditions that are true in the system's starting configuration. The set of grounded predicates must reference parameter values in the object set. The number of state predicates remain fixed throughout execution; so every predicate that describes the domain must be included in the initial state<sup>3</sup>.

$$\langle initial state \rangle ::= (':init' \langle grounded predicate \rangle^+)$$

The goal state has similar syntax to the initial state; its set of grounded predicates must reference parameter values in the object set. However, only the minimum set of predicates that distinguish the goal state are required.

$$\langle goal state \rangle ::= (':goal' \langle grounded predicate \rangle^+)$$


---

<sup>3</sup>This is *The Closed World Assumption*[], meaning that only the predicates specified as true in the initial state are considered true throughout planning.

## Scope

Scope is enforced with a symbol table hierarchy in `strips.ml`. One global symbol table contains all the top-level declarations and type specifications. Local declarations inside the operator and predicate bodies use their own symbol tables with a separate name space from the global.

name	attribute
Parent table	-
:domain	<domain name>
:problem	<problem name>
:predicate-<predicate name>	:action
:action-<action name>	:action
:type-<type name>	:type
:predicate-<predicate name>-p<i>-<type name>	:typespec

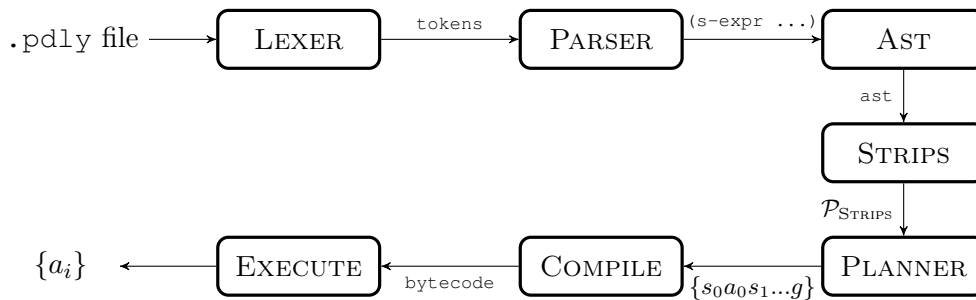
Table 1: Global Symbol Table Structure

Parameter type specifications are created from `:predicates` to ensure type consistency throughout the program. Each type specification contains a predicate name, parameter number, and parameter type name to provide descriptive error messages.

# Architecture

## Processing Sequence

The PDDL<sub>YTE</sub> compiler processes a planning problem into a sequence of executed actions  $\{a_i\}$ . Tokenized input files are parsed into s-expressions then arranged into an abstract syntax tree. The abstract syntax tree is semantically analyzed throughout its translation to a STRIPS planning problem  $\mathcal{P}_{\text{STRIPS}}$ . This intermediate data structure proved to be more amenable with common planning algorithms than an abstract syntax tree. Well-posed problems with solutions are output from the planner then output as an action-state sequence. The compiler translates the plan into bytecode that can be executed on a virtual planning machine. The final output is printed as a sequence of actions.



## Planner

The planner's objective is to connect the initial state  $s_0$  to the goal  $g$  using a sequence of actions  $\{a_i\}$ . This is accomplished using a glorified guess-and-check method to find states  $s_{i+1}$  closer to the goal than  $s_i$ .

The planner starts in  $s_0$  with a set of available operators  $A$ . It looks in  $A$  for every applicable instance  $a_i \in A$  that transitions the system to  $s_1$ . At this point, it guesses which  $a_i$  to transition with. After transitioning to  $s_1$ , the planner checks if  $s_1 = g$ . More often than not, it guesses wrong,  $s_1 \neq g$ , and needs to make another guess. The planner can even sometimes take a wrong transition and need to turn around. In these cases it helps to leave a Hansel-and-Gretel trail back to  $s_0$ . The planner leaves breadcrumbs in the form of *partial plans*. The partial plan in  $s_1$  is  $S_1 = \{s_0a_0, s_1\}$ . If the problem is well posed, the planner will eventually reach the goal  $g$  and return  $S = \{s_0a_0, s_1a_1, \dots, g\}$ .

The PDDL<sub>YTE</sub> planner models the space of every possible  $S_i$  as an abstract graph; so that finding  $g$  amounts to generating and traversing through the graph. Figure 2 illustrates this concept. The planner is implemented in `planner.ml`.

## Graph Traversal

Algorithm 1. orchestrates the various ways to traverse the graph.

**Transitioning:** Every reachable successor of a node  $S$  defines the *search fringe*. The fringe includes backward transitions, but no self loops. When the fringe is non-empty it is prioritized with respect to its depth in the search space. This characterizes the depth-first nature of the algorithm and encourages the planner to make forward progress. Equally deep successors are chosen at random to decrease the chances of backtracking.

**Backtracking:** When the fringe is empty it implies there exists no unexplored successors in  $S$ . Provided that  $S \neq S_0$ , there's a fighting chance the goal can still be reached. Under these conditions, the planner removes the last state and action from the partial plan and re-evaluates the fringe from the prior state  $S_{i-1}$ . Only when every avenue leaving the initial state backtracks to the start does the planner give up and return an error.

**Goal Satisfaction:** Before the planner computes the fringe, it checks to see if the partial plan  $S$ , has reached the goal state,  $g$ . The goal criteria is satisfied when the entire set of predicates in  $g \subset s_g \in S$ . Once the goal state is reached, the planner returns the entire sequence of states and actions that lead to it  $S = \{s_0a_0, s_1a_1, \dots, g\}$ .

---

### Algorithm 1 Prioritized, Depth-first Search

---

```

1: globals  $\mathcal{P}_{\text{STRIPS}} = \{s_0, g, A\}$ 
2: procedure DFS( $S = s_0, \nu = \{\}$ )
3:   if  $S = g$  then return  $S$ 
4:   else
5:      $fringe \leftarrow \text{successors}(S, A, \nu)$  //compute all successors in  $s$ 
6:     if  $fringe = \{\}$  then
7:       if state space exhausted then return error
8:       else
9:         DFS( $S_{i-1}, S \cup \nu$ ) //backtrack
10:    else
11:       $priority\_queue \leftarrow \text{prioritize } fringe$ 
12:       $S_{i+1} \leftarrow \text{pop } priority\_queue$  // successor
13:      DFS( $S_{i+1}, S \cup \nu$ )

```

---

## Graph Generation

The search space grows at each call of the high-level successors routine. The routine initially computes every applicable successor from  $S$ , then filters those transitions that over complicate the global quest to reach the goal.

**Unification:** The planner uses a unification algorithm to compute each applicable successor. An *applicable successor* is the state resulting from a grounded instance of some operator  $a_i$  whose parameter bindings are consistent with  $S$ . In the simple Pacman example,

```
move( pacman , sq1 , sq2 )
```

is an grounded instance of the move operator  $a_i$  from  $S$ . The parameter bindings

$$?who = pacman, ?from = sq1, ?to = sq2$$

are consistent with  $S$  because the variables  $?who, ?from, ?to$  are bound to the same object values as they are in the operator arguments. Therefore the node  $S_{i+1}$  which results from applying that operator instance is an applicable successor.

---

**Algorithm 2** STRIPS Unification
 

---

```

1: globals  $a, S$ 
2: procedure UNIFY( $P_a^+, P_a^-, \sigma = \{\}$ )
3:   if  $P_a^+ = \{\}$  then
4:     if  $P_a^- = \{\}$  then return  $\{a(\sigma')\}$ 
5:     else
6:        $p_a^- \leftarrow \text{pop } P_a^-$ 
7:        $P_s \leftarrow \text{predicates in } S \text{ matching } p_a^-$ 
8:       for all  $p_s \in P_s$  do
9:          $\sigma' \leftarrow \text{extend}(p_s, p_a^-)$  //bind parameters in  $p_a^-$  based on values in  $p_s$ 
10:        if  $\sigma'$  consistent with  $\sigma$  then
11:          UNIFY( $\{\}, P_a^-, \sigma'$ )
12:        else return  $\{\}$ 
13:     else
14:        $p_a^+ \leftarrow \text{pop } P_a^+$ 
15:        $P_s \leftarrow \text{predicates in } S \text{ matching } p_a^+$ 
16:       for all  $p_s \in P_s$  do
17:          $\sigma' \leftarrow \text{extend}(p_s, p_a^+)$  //bind parameters in  $p_a^+$  based on values in  $p_s$ 
18:         if  $\sigma'$  consistent with  $\sigma$  then
19:           if  $P_a^+ \setminus p_a^+ = \{\}$  then //last state predicate
20:             if complementary effects ( $a, \sigma'$ ) then return  $\{\}$ 
21:             else UNIFY( $P_a^+ \setminus p_a^+, P_a^-, \sigma'$ )
22:           else UNIFY( $P_a^+ \setminus p_a^+, P_a^-, \sigma'$ )
23:         else continue

```

---

*I apologize to the reader for the unsightly complexity of this algorithm. After struggling to implement it from simpler pseudo code, I decided to spare future hackers the same frustration and provide more detail.*

For each operator  $a_i \in A$ , the unification algorithm attempts to find a substitution  $\sigma$  consistent with  $S$ . The substitution is incrementally accumulated in two main loops that unify operator preconditions with each state predicate. First the positive preconditions  $P_a^+$  of operator  $a$  are processed. Only state predicates  $P_s$  that match with a given  $p_a^+$  are chosen to process each iteration. This design reduces the search complexity, because

inapplicable predicates are ignored. The grounded parameters  $\sigma_{p_s}$  for each state predicate  $p_s \in P_s$  are extended such that the corresponding operator precondition  $\sigma_{p_a}$  is bound to their values. In the Pacman example, the state predicate

$$(\text{adj } \text{sq1 } , \text{ sq2 } ), \text{ with } \sigma_{p_s} = \{\text{sq1}, \text{sq2}\}$$

can be extended to the operator precondition

$$(\text{adj } ?\text{from } , ?\text{to } ), \text{ with } \sigma_{p_a} = \{?\text{from}, ?\text{to}\}$$

such that  $\sigma' : \sigma_{p_s} \mapsto \sigma_{p_a} = \{?\text{from} \leftarrow \text{sq1}, ?\text{to} \leftarrow \text{sq2}\}$ .

When this set of bindings is consistent with  $s$ , the predicate is removed from the set  $P_a^+$  and UNIFY is recursively called with the reduced set of positive preconditions and updated substitution  $\sigma'$ . Inconsistent substitutions are discarded and another state predicate is chosen to unify with. When the last positive precondition is validated – before the second main loop – the algorithm checks if the current substitution results in complimentary effects. *Complimentary effects* result in self loops because the preconditions are reversed with the effects. If the algorithm finds a self-looping substitution, it returns the empty set.

After all the positive preconditions are successfully unified, the negative preconditions  $P_a^-$  are checked for consistency with  $s$ . If no negative precondition falsifies the current substitution, then a grounded operator instance has been identified. The algorithm returns the singleton set of the grounded instance.

**Filtering:** Duplicate successors are filtered to avoid self loops. In addition, previously-visited nodes are filtered to avoid thrashing and infinite cycles that may occur in the graph.

## Bytecode Instructions

After plans are solved, the compiler translates the sequence of state predicates and actions into a custom bytecode representation. The bytecode instructions are defined in `compile.ml` and are specific to STRIPS planning problems.

- `Push_pred`: Push a predicate on the stack
- `Push_action`: Push an action on the stack
- `Pop_pred`: Pop a predicate from the stack
- `Goal`: Predicates defining the goal state have been satisfied: pop all predicates from the stack and halt execution.

## Bytecode Interpreter

PDDL<sub>YTE</sub> uses a rudimentary runtime interpreter to simulate plan execution. The stack-based bytecode runs on a virtual planning machine that processes predicates and actions. This idea is loosely based on the PROLOG instruction set [5] and reversed goal-stack planning.[3]

Instructions and the stack are stored in two different arrays. Execution is carried out with the recursive `exec` function, which uses a frame pointer, stack pointer, and program counter during operation. The program counter is used to index into the instruction list. State predicates are pushed to the stack until the goal conditions are satisfied, or the entire state is on the stack. When every state predicate is on the stack and the goal is unsatisfied, an action is applied to transition the system to the next state. State transition is emulated by popping the current state predicates from the stack and pushing the action whose preconditions are satisfied from the prior stack state. This process is repeated until the predicates defining the goal are on the stack, at which point they're popped off and the machine halts.

The program counter is simply an index into the instructions array for the current instruction. Each recursive call to `exec` changes the program counter one position by one to execute the next instruction, or decremented in the case of a predicate being popped off the stack. The stack grows from the index zero upwards to its maximum size of 1023.

The bytecode interpreter stack is arbitrarily limited to 1024 values. These limits are not enforced by the compiler, but the bytecode interpreter will crash when they are exceeded. The length of the plan is limited in this sense because the size of a state plus the maximum length of the plan must fit on the stack at once.

## Limitations

To simplify the compiler design, a number of constraints were placed on various data types and algorithms. The translation from abstract syntax tree to a STRIPS planning problem only supports flat and conjunctions. Predicates are limited to a maximum of two parameters, with a minimum of one.

## Lessons Learned

The most valuable advice I can give is also the most cliché: start implementing everything as soon as possible. Writing a compiler requires a highly idiosyncratic understanding of its objectives. Many of those idiosyncrasies will only appear during integration. The quicker integration problems are solved, the more time there is to test. Testing is the key to success on this project, because it's more impressive to do simple things perfectly than sophisticated things unreliably.

There's a idiom which says "when you're a hammer everything looks like a nail." It means that you approach problems the same way because that's all you know. Working on this project, and with Ocaml, made me feel like a pattern-matching hammer. I wish I had spent more time upfront understanding design patterns and gaining deeper insight into the language, instead of trying to pattern match my way to infinity.

## Language Design

Design requirements and interface definitions were critical for making progress. Taking time to think about each module's input and output helped to shape the high-level compiler design. Working to a schedule provided continuity to my progress; it helped me identify when to put in more effort and when to ask for help. The project git log is included.

## Project Log

```
git log
1 commit a75f1e3aec5dcad23a2f5e425a419f8e88946c38
2 Author: John Martin <jdmartin86@gmail.com>
3 Date: Wed May 14 23:32:22 2014 -0400
4
5     last commit
6
7 commit cbb9ee8f46635a681e813f9f8b59f21474599411
8 Author: John Martin <jdmartin86@gmail.com>
9 Date: Wed May 14 23:32:06 2014 -0400
10
11    last commit
12
13 commit f17c73713072a99163b731a3e9eeda2de13642d7
14 Author: John Martin <jdmartin86@gmail.com>
15 Date: Wed May 14 23:31:35 2014 -0400
16
```



```
17     removed ast prints
18
19 commit 4b6b148c758d8c8cad33c2723e41afd7f5ddc6db
20 Author: John Martin <jdmartin86@gmail.com>
21 Date:   Wed May 14 22:07:13 2014 -0400
22
23     fixed bug in unification algorithm: processing negative preconditions
24
25 commit a555266c0f09cf4c66e2a4c0bd9c3b65826ab56f
26 Author: John Martin <jdmartin86@gmail.com>
27 Date:   Wed May 14 17:15:46 2014 -0400
28
29     dependent upon util.ml
30
31 commit 6b6d8d10d6f0b6933c87eb57b77d75fd80d61ee2
32 Author: John Martin <jdmartin86@gmail.com>
33 Date:   Wed May 14 17:15:03 2014 -0400
34
35     fully supports scope and type checking
36
37 commit d9b485379d5270703b5f831b5515e9870fa62c1d
38 Author: John Martin <jdmartin86@gmail.com>
39 Date:   Wed May 14 17:14:28 2014 -0400
40
41     cleaned up test dependencies
42
43 commit 8a003af45a5d82c3276de87c0241418d72600cf5
44 Author: John Martin <jdmartin86@gmail.com>
45 Date:   Sun May 11 21:11:14 2014 -0400
46
47     updated symbol tokens to start with one character and contain zero or
48     more alphanumeric characters thereafter
49
50 commit 2f4d02cabf02c1637f2f209ba6a012d49e51b4e9
51 Author: John Martin <jdmartin86@gmail.com>
52 Date:   Sun May 11 12:59:09 2014 -0400
53
54     depends on util.ml for print
55
56 commit cbd83e89c4e5a0e080f6d1490a7c40bc62d54410
57 Author: John Martin <jdmartin86@gmail.com>
58 Date:   Sun May 11 12:40:04 2014 -0400
59
60     implemented explored list to backtrack more reliably
61
62 commit ed12e5ce622ba3a971360593eeff5ae21401c287
63 Author: John Martin <jdmartin86@gmail.com>
64 Date:   Sun May 11 12:39:18 2014 -0400
65
66     added print functions and list filtering functions
67
68 commit 15782fe8f83de91934d0eeb1150d8bd85e48ad71
69 Author: John Martin <jdmartin86@gmail.com>
70 Date:   Sun May 11 12:38:06 2014 -0400
71
```

xviii    Lessons Learned

```
71        updated comments
72
73        commit 4c80d4f40ac2b0874f1d87c101ddfd53e9cb2e48
74        Author: John Martin <jdmartin86@gmail.com>
75        Date:    Sun May 11 12:37:45 2014 -0400
76
77        added comments
78
79        commit edf51362bf27390c961a3117b8833a04f43f5b2d
80        Author: John Martin <jdmartin86@gmail.com>
81        Date:    Sun May 11 12:36:59 2014 -0400
82
83        updated print functions
84
85        commit 51c5b491a463938ed563dc9980c034187921dbfa
86        Author: John Martin <jdmartin86@gmail.com>
87        Date:    Sat May 10 02:20:05 2014 -0400
88
89        updated makefile
90
91        commit 58a006d57cd57db7b5fe8e776d83e0a121be105c
92        Author: John Martin <jdmartin86@gmail.com>
93        Date:    Sat May 10 02:18:55 2014 -0400
94
95        added comments and arranged according to dependency
96
97        commit 13ae253107db4d102c83bfb53fd0299e03b08a3a
98        Author: John Martin <jdmartin86@gmail.com>
99        Date:    Sat May 10 02:16:22 2014 -0400
100
101        updated error prints
102
103        commit 6e74c2dc2a14b47cff86756cceedf85cd291dc0a3
104        Author: John Martin <jdmartin86@gmail.com>
105        Date:    Sat May 10 02:14:32 2014 -0400
106
107        initial commit
108
109        commit f56ca4f4fc75elf94ac1c11165b23106330a9350
110        Author: John Martin <jdmartin86@gmail.com>
111        Date:    Sat May 10 02:09:18 2014 -0400
112
113        filtering duplicate successors, randomly permuting the priority queue,
              replace plan_to with visited
114
115        commit 1406f4245b7920bb1407b632f10a9d186e0042d9
116        Author: John Martin <jdmartin86@gmail.com>
117        Date:    Thu May 8 21:36:49 2014 -0400
118
119        removed test functions
120
121        commit 40d20bb7fa70a28c1d4bf90b36d89493ce1fb33f
122        Author: John Martin <jdmartin86@gmail.com>
123        Date:    Thu May 8 21:34:25 2014 -0400
124
```

125        tested **with** pacman.pdly  
126  
127 commit 651240204314dfffb60925e0304dc838de754d844  
128 Author: John Martin <jdmartin86@gmail.com>  
129 Date:    Thu May 8 21:33:23 2014 -0400  
130  
131        removed test functions **and** updated interface  
132  
133 commit bf2aedc2c7d33812b674cb686474936c569d4d5d  
134 Author: John Martin <jdmartin86@gmail.com>  
135 Date:    Thu May 8 21:32:45 2014 -0400  
136  
137        removed test functions **and** updated interface  
138  
139 commit e443764f47b2dcd93ed24a9a1444f99781341e5a  
140 Author: John Martin <jdmartin86@gmail.com>  
141 Date:    Thu May 8 21:31:58 2014 -0400  
142  
143        tested **with** pacman.pdly  
144  
145 commit 1fc26f8d667e43dba2400849535d43b2996324b7  
146 Author: John Martin <jdmartin86@gmail.com>  
147 Date:    Thu May 8 21:29:37 2014 -0400  
148  
149        fixed print  
150  
151 commit df85e092646785b1037b5eea87571c1d749c55b8  
152 Author: John Martin <jdmartin86@gmail.com>  
153 Date:    Thu May 8 21:23:23 2014 -0400  
154  
155        cleaned up a bit **and** updated interface  
156  
157 commit e149413bc2d93339983ccde4e32dc1d33a1112b4  
158 Author: John Martin <jdmartin86@gmail.com>  
159 Date:    Thu May 8 08:56:47 2014 -0400  
160  
161        removed **module** tests  
162  
163 commit 956e257091a68a3b86bfd5511a88cf877fdcc41e  
164 Author: John Martin <jdmartin86@gmail.com>  
165 Date:    Thu May 8 08:55:15 2014 -0400  
166  
167        clean up  
168  
169 commit 703013f6b473e4c7b8d6c0d77a9466d625a39665  
170 Author: John Martin <jdmartin86@gmail.com>  
171 Date:    Thu May 8 08:54:47 2014 -0400  
172  
173        supports compiler flags  
174  
175 commit b7e83cb96a94ed6b2a048cdb7c7b491a818e7ccd  
176 Author: John Martin <jdmartin86@gmail.com>  
177 Date:    Thu May 8 08:52:55 2014 -0400  
178  
179        Initial commit **of** runtime environment

xx Lessons Learned

```
180
181 commit df40a16399d04a3002b6c55e4ee2f110cb7c6740
182 Author: John Martin <jdmartin86@gmail.com>
183 Date: Thu May 8 08:51:25 2014 -0400
184
185     initial commit of compiler to IR
186
187 commit 9ff2ebell19ecbd3c1dec64363b7fe3eaca30b726
188 Author: John Martin <jdmartin86@gmail.com>
189 Date: Tue May 6 07:54:55 2014 -0400
190
191     Updated planner to use partial plans of state-action stripes
192
193 commit f96610d4649c773095a43b5f4e6fee9291d113e0
194 Author: John Martin <jdmartin86@gmail.com>
195 Date: Sun May 4 22:07:01 2014 -0400
196
197     re-wrote the planner to represent nodes as partial plans ...
198         everything seems to work -- no big deal
199
200 commit 75ab3a89a0f8f190f283888942a5c5109f9ad564
201 Author: John Martin <jdmartin86@gmail.com>
202 Date: Sat May 3 02:00:21 2014 -0400
203
204     new planner structure
205
206 commit 9b86f48d01a32d09d705ae0fb2a4241f4a145b59
207 Author: John Martin <jdmartin86@gmail.com>
208 Date: Fri May 2 22:55:20 2014 -0400
209
210     intermediate save
211
212 commit bb247d43073e49430ff8ee2982277f7e74e79ad3
213 Author: John Martin <jdmartin86@gmail.com>
214 Date: Tue Apr 29 08:02:47 2014 -0400
215
216     pacman works!
217
218 commit aa131f563d4aad8efccc21a57a79df11575bf3fb
219 Author: John Martin <jdmartin86@gmail.com>
220 Date: Mon Apr 28 07:23:49 2014 -0400
221
222     applicative actions and successor function partially tested
223
224 commit 716e78aa20719cf7313155a6a8fe995d4c34e29d
225 Merge: 69693d9 71369cc
226 Author: John Martin <jdmartin86@gmail.com>
227 Date: Wed Apr 23 20:32:31 2014 -0400
228
229     deleted sast
230
231 commit 71369cc9bce548e8f852a61588bc65fd4bb4aa16
232 Author: John Martin <jdmartin86@gmail.com>
233 Date: Wed Apr 23 20:50:27 2014 -0400
```

234 Delete sast.mli  
235  
236 commit cc94b9f26cd471c5970a5b7da35c88e61ac39c3c  
237 Author: John Martin <jdmartin86@gmail.com>  
238 Date: Wed Apr 23 20:50:17 2014 -0400  
239  
240 Delete sast.ml  
241  
242 commit 69693d9ffb4fc040644fdf9664670141f18a084c  
243 Author: John Martin <jdmartin86@gmail.com>  
244 Date: Wed Apr 23 20:27:15 2014 -0400  
245  
246 buildable planner outline  
247  
248 commit 115b9208b35e7f972f429387f5001ee946471bea  
249 Author: John Martin <jdmartin86@gmail.com>  
250 Date: Wed Apr 23 19:59:34 2014 -0400  
251  
252 supports strips test  
253  
254 commit df6e6bd0c5a9ee8f7b6e13c39c4dea486210914f  
255 Author: John Martin <jdmartin86@gmail.com>  
256 Date: Wed Apr 23 19:57:46 2014 -0400  
257  
258 supports strips problem  
259  
260 commit 488b8b35a28975298dc16a888bb73639e496ad6b  
261 Author: John Martin <jdmartin86@gmail.com>  
262 Date: Wed Apr 23 19:57:00 2014 -0400  
263  
264 re-structured and tested translation to strips problem  
265  
266 commit ab5d1ebb0b0025617dc7b03b9c3f6e289e284732  
267 Author: John Martin <jdmartin86@gmail.com>  
268 Date: Wed Apr 23 12:49:56 2014 -0400  
269  
270 cleaned up warnings and removed expr\_sym type  
271  
272 commit 1780319eed4080cad1ba31ecd04f5e8ec157517f  
273 Author: John Martin <jdmartin86@gmail.com>  
274 Date: Wed Apr 23 12:09:40 2014 -0400  
275  
276 initial commit of top-level compiler  
277  
278 commit f77ec710c13f0038877286c54e97d9aff05f4e31  
279 Author: John Martin <jdmartin86@gmail.com>  
280 Date: Wed Apr 23 12:08:58 2014 -0400  
281  
282 updated for strips and pddlyte modules  
283  
284 commit 26df45f4fce9886ec17a9f5912cc1ab2e8e5aae2  
285 Author: John Martin <jdmartin86@gmail.com>  
286 Date: Wed Apr 23 12:07:08 2014 -0400  
287

## xxii Lessons Learned

288        initial commit **of** symt rename. fully supports symbol table translation  
          **and** strips plan formulation.

289

290 commit 6ab2271712655337f8b5cb50956f000efaeb978d  
291 Author: John Martin <jdmartin86@gmail.com>  
292 Date:    Wed Apr 23 12:06:00 2014 -0400  
293

294        removed symt.ml **and** symt.mli

295

296 commit 2cd1492c37d69a995cb4f6a83f5d7a0ca6d55b5c  
297 Author: John Martin <jdmartin86@gmail.com>  
298 Date:    Tue Apr 22 23:36:47 2014 -0400  
299

300        full support added **for** symbol table translation

301

302 commit b7fc3c8b99a0116aa08e212c347d964d8aaabeac  
303 Author: John Martin <jdmartin86@gmail.com>  
304 Date:    Tue Apr 22 10:25:59 2014 -0400  
305

306        partial support added **for** action translation

307

308 commit 8662d8a371cbd6c77186e2d60b234a27b9925a66  
309 Author: John Martin <jdmartin86@gmail.com>  
310 Date:    Tue Apr 22 09:06:39 2014 -0400  
311

312        full support **for** predicate translation

313

314 commit 439f37acf49f4fdc58cc713c06de8ee25b5ccbbc  
315 Author: John Martin <jdmartin86@gmail.com>  
316 Date:    Tue Apr 22 00:26:53 2014 -0400  
317

318        partial symbol table support added

319

320 commit 6f242667176e10a7dbc2a0ec43ebe24bf1840fc4  
321 Author: John Martin <jdmartin86@gmail.com>  
322 Date:    Sun Apr 20 22:37:26 2014 -0400  
323

324        added support **for** symbol table

325

326 commit aa19c0e5f91ebee9b29b5e5994d64f934c478b23  
327 Author: John Martin <jdmartin86@gmail.com>  
328 Date:    Sun Apr 20 22:35:34 2014 -0400  
329

330        initial commit

331

332 commit 94c12820324d1999ebecddca825faf99c63d91d5  
333 Author: John Martin <jdmartin86@gmail.com>  
334 Date:    Sun Apr 20 16:06:38 2014 -0400  
335

336        added code outlines **for** successor **function** **and** forward search

337

338 commit f0df0152d2f65ef8ccf9262f917cca5f825271cf  
339 Author: John Martin <jdmartin86@gmail.com>  
340 Date:    Sat Apr 19 22:33:50 2014 -0400  
341

342 format updates  
343  
344 commit 534053b5817916241d1f39fa31347400dd27357f  
345 Author: John Martin <jdmartin86@gmail.com>  
346 Date: Sat Apr 19 22:32:13 2014 -0400  
347  
348 symbols support uppercase letters and question marks  
349  
350 commit 767b9363291eb6a8c4d1f69d49d484a66cc0b69b  
351 Author: John Martin <jdmartin86@gmail.com>  
352 Date: Sat Apr 19 22:30:15 2014 -0400  
353  
354 small updates  
355  
356 commit 44d2b560335b06fa02a921f89dd16c4b8b57d360  
357 Author: John Martin <jdmartin86@gmail.com>  
358 Date: Sat Apr 19 22:26:14 2014 -0400  
359  
360 added support for the planner  
361  
362 commit 9ada41e897444e4aa605ced4da5a7e45e7aa5924  
363 Author: John Martin <jdmartin86@gmail.com>  
364 Date: Sat Apr 19 22:25:34 2014 -0400  
365  
366 Added nil types for predicates, conjunctions, and atoms -- to  
represent nil returns  
367  
368 commit 4c44604d9342660a85082d9d55f1828209ef12c7  
369 Author: John Martin <jdmartin86@gmail.com>  
370 Date: Sat Apr 19 22:23:41 2014 -0400  
371  
372 initial commit -- applicative actions only support positive  
preconditions  
373  
374 commit c8b377a86501d395513da7b818e9acfd316cdf0  
375 Author: John Martin <jdmartin86@gmail.com>  
376 Date: Wed Apr 16 10:46:07 2014 -0400  
377  
378 nearly full support added: actions, objects, inits, goals all parse  
379  
380 commit e1f7a747ba130b01397594b2a8723d3b89da8f39  
381 Author: John Martin <jdmartin86@gmail.com>  
382 Date: Wed Apr 16 08:41:00 2014 -0400  
383  
384 support for predicate lists, domains and problem definitions, and  
partial support for actions  
385  
386 commit 7188dfb1cfea9c26900e2ee97aaaf9aa47c47f08  
387 Author: John Martin <jdmartin86@gmail.com>  
388 Date: Tue Apr 15 15:19:22 2014 -0400  
389  
390 reformulated tree types -- still in progress  
391  
392 commit af7d5e66a0e80d4c46752a78d9c044e4c6c759a5  
393 Author: John Martin <jdmartin86@gmail.com>

## xxiv Lessons Learned

```
394 Date: Sun Apr 6 14:50:44 2014 -0400
395
396 supports sast
397
398 commit 62f0536cf766cbee83422a982d6a386ce2246dc9
399 Author: John Martin <jdmartin86@gmail.com>
400 Date: Sun Apr 6 12:37:25 2014 -0400
401
402 Initial commit
403
404 commit 31d4fd5c2e6102639ff337c9988f1f4a052a4abe
405 Author: John Martin <jdmartin86@gmail.com>
406 Date: Sun Apr 6 00:05:18 2014 -0400
407
408 Partially tested
409
410 commit 94b6a45c7412baa75a45d1cb0543e353e2c1bb60
411 Author: John Martin <jdmartin86@gmail.com>
412 Date: Thu Apr 3 23:42:28 2014 -0400
413
414 Added planner nodes and procedures
415
416 commit f5517abe43f207741f61798f07c637b6adefd0c3
417 Author: John Martin <jdmartin86@gmail.com>
418 Date: Wed Apr 2 00:16:16 2014 -0400
419
420 supports define expressions
421
422 commit ec16acd1b0277dfd64627703f02c001e6efc73eb
423 Author: John Martin <jdmartin86@gmail.com>
424 Date: Mon Mar 31 19:58:54 2014 -0400
425
426 Initial commit of abstract syntax tree
427
428 commit 838a5a4d23c00642b697c29404da0eeb19303aa5
429 Author: John Martin <jdmartin86@gmail.com>
430 Date: Sat Mar 15 23:12:54 2014 -0400
431
432 Initial commit of Makefile
433
434 commit 59839146eb0825079a2ce3d03062c5a907c701e1
435 Author: John Martin <jdmartin86@gmail.com>
436 Date: Sat Mar 15 23:11:27 2014 -0400
437
438 Initial commit of s-expressions
439
440 commit 4e3b4f51aa23396a05cf26ba2cef53abf7b019c2
441 Author: John Martin <jdmartin86@gmail.com>
442 Date: Sat Mar 15 23:10:45 2014 -0400
443
444 Initial commit of parser
445
446 commit b0e95764af360bf7b94a10473c26265b13c3f91d
447 Author: John Martin <jdmartin86@gmail.com>
448 Date: Sat Mar 15 23:10:09 2014 -0400
```



449  
450 Initial commit of lexer  
451  
452 commit 54blbadff976980c3ead7b01c567c033b349a4a5  
453 Author: John Martin <jdmartin86@gmail.com>  
454 Date: Sat Mar 15 23:08:57 2014 -0400  
455  
456 Initial commit of abstract syntax tree  
457  
458 commit dbf275504d12d8f9e5e06394c52ffdcfee718b65  
459 Author: jdmartin86 <jdmartin86@gmail.com>  
460 Date: Sat Mar 15 09:33:24 2014 -0400  
461  
462 Initial commit -- final draft  
463  
464 commit f59aada1fd8e1df3609d696c2494db5964c6a72e  
465 Author: jdmartin86 <jdmartin86@gmail.com>  
466 Date: Thu Feb 13 13:29:52 2014 -0500  
467  
468 final draft  
469  
470 commit a599292d30582e683680a78aa96524fb634b0087  
471 Author: jdmartin86 <jdmartin86@gmail.com>  
472 Date: Mon Feb 10 23:13:26 2014 -0500  
473  
474 near-final draft  
475  
476 commit 42da7b1138d950741c6bf17710733239ecc97d19  
477 Author: jdmartin86 <jdmartin86@gmail.com>  
478 Date: Sun Feb 9 19:10:40 2014 -0500  
479  
480 partially-completed draft including integer support  
481  
482 commit e1484e27d61fc10618df7678be02b2c2deb8ace6  
483 Author: jdmartin86 <jdmartin86@gmail.com>  
484 Date: Sun Feb 9 01:50:38 2014 -0500  
485  
486 included example, pacman program  
487  
488 commit 09135655994b94478a0268666e7e9ce46cabfd74  
489 Author: John Martin <jdmartin86@gmail.com>  
490 Date: Thu Feb 6 00:32:12 2014 -0500  
491  
492 updated solution  
493  
494 commit 85bf75121dcabb5788cf5928e749b120f0a4c6ff  
495 Author: John Martin <jdmartin86@gmail.com>  
496 Date: Thu Feb 6 00:29:13 2014 -0500  
497  
498 fixed errors in goal  
499  
500 commit dfb2d8425b7370ce30100786b9396672be8d9af0  
501 Merge: 6f5d0f6 45a8b2d  
502 Author: John Martin <jdmartin86@gmail.com>  
503 Date: Thu Feb 6 00:04:18 2014 -0500

xxvi    Lessons Learned

504  
505        Merge remote-tracking branch 'upstream/master'  
506  
507 commit 45a8b2db8dff6e75786efcd0bd2d4a006a83b73e  
508 Author: jdmartin86 <jdmartin86@gmail.com>  
509 Date:    Sun Feb 9 01:16:04 2014 -0500  
510  
511        partial draft  
512  
513 commit 6f5d0f68805368aab1723f771ed48c6f97c9e56d  
514 Author: John Martin <jdmartin86@gmail.com>  
515 Date:    Wed Feb 5 23:54:06 2014 -0500  
516  
517        initial commit  
518  
519 commit 6b3155a9d2862256f8a09f2ada411941ccd872a2  
520 Author: jdmartin86 <jdmartin86@gmail.com>  
521 Date:    Tue Feb 4 21:23:28 2014 -0500  
522  
523        Initial commit of rough drafts  
524  
525 commit 480529afa7b5dc2958cf15e5a53ebc19ed16fce7  
526 Author: jdmartin86 <jdmartin86@gmail.com>  
527 Date:    Mon Feb 3 18:14:48 2014 -0500  
528  
529        committing rough draft  
530  
531 commit 6f3606e75e0a16ab25acc3045e2a01602297016a  
532 Author: jdmartin86 <jdmartin86@gmail.com>  
533 Date:    Mon Feb 3 18:14:20 2014 -0500  
534  
535        initial commit of reference database  
536  
537 commit 142bece7534face487a14704d5a161d26b5306a7  
538 Author: John Martin <jdmartin86@gmail.com>  
539 Date:    Mon Feb 3 18:06:01 2014 -0500  
540  
541        initial commit of blank document files  
542  
543 commit 97872d3623def0dd0f4e9f27aclf5dba917a421f  
544 Author: John Martin <jdmartin86@gmail.com>  
545 Date:    Mon Feb 3 17:34:09 2014 -0500  
546  
547        initial commit of directory structure  
548  
549 commit a93a9995c37c92bb3c65e8534c3bf139ac4f2c5d  
550 Author: John Martin <jdmartin86@gmail.com>  
551 Date:    Mon Feb 3 17:30:58 2014 -0500  
552  
553        initial commit

---

## Testing

Each module was tested as a unit except the planner. Since the planner was the most critical module, every function was individually tested with the ocaml top-level. This proved to be immensely helpful when debugging higher-level logic.

## Source Code

### sexpr.mli

```
1 (*
2  * sexpr.mli
3  *
4  *   S-expressions.
5  *
6  *)
7
8
9 (* Type of atomic expressions. *)
10 type atom =
11   | Atom_unit
12   | Atom_int  of int
13   | Atom_sym  of string
14
15
16 (* Type of all S-expressions. *)
17 type expr =
18   | Expr_atom of atom
19   | Expr_list of expr list
20
21
22 (* Convert an S-expression to a string.
23    This version makes the structure of the S-expression explicit. *)
24 val string_of_expr : expr -> string
25
26
27 (* Convert an S-expression to a string.
28    This version prints the S-expression like a Scheme expression.
29
30    THIS MAY NOT BE NECESSARY -- JM
31    *)
32 val string_of_expr2 : expr -> string
```

### ast.mli

```
1 (* the abstract syntax tree *)
2
3 (* symbols *)
4 type sym = string
5
6 type atom =
7   | Atom_var of sym (* ?symbol *)
8   | Atom_gnd of sym (* symbol *)
```

```

9   | Atom_nil
10
11  type predicate =
12  | Pred_var of sym * atom list (* (predname vatom v/gatom?) *)
13  | Pred_gnd of sym * atom list (* (predname gatom gatom?) *)
14  | Pred_nil
15
16  type conjunction =
17  | Conj_and of conjunction list
18  | Conj_neg of predicate
19  | Conj_pos of predicate
20  | Conj_nil
21
22  type action =
23  {
24    name          : sym;
25    parameters    : atom list;
26    precondition  : conjunction;
27    effect        : conjunction
28  }
29
30  type expr =
31  | Expr_domain      of sym * expr list (* (define ( domain pman ) ... )*)
32  | Expr_problem     of sym * expr list (* (define ( problem prob ) ... )*)
33  | Expr_predicates  of predicate list (* :predicates body *)
34  | Expr_init        of predicate list (* :init body *)
35  | Expr_goal        of predicate list (* :goal body *)
36  | Expr_action      of action          (* :action ... *)
37  | Expr_objects    of atom list        (* :objects body *)
38  | Expr_unit        of ()              (* () *)
39
40  val string_of_syms : string list -> string
41  val sym_of_atom : Sexpr.atom -> string
42  val astatom_of_atomsym : Sexpr.atom -> atom
43  val astatom_of_sexpr : Sexpr.expr -> atom
44  val pred_of_sexpr : Sexpr.expr -> predicate
45  val params_of_sexpr : Sexpr.expr list -> atom list
46  val conj_of_sexpr : Sexpr.expr -> conjunction
47  val action_of_sexpr : Sexpr.expr list -> action
48  val ast_of_sexpr : Sexpr.expr -> expr
49  val string_of_atom : atom -> sym
50  val string_of_pred : predicate -> string
51  val string_of_conj : conjunction -> string
52  val string_of_params : atom list -> string
53  val string_of_precond : conjunction -> string
54  val string_of_effect : conjunction -> string
55  val string_of_action : action -> string
56  val string_of_ast : expr -> string

```

---

**util.mli**

```

1  (* util.mli *)
2

```

```

3 val sprintf : ('a, unit, string) format -> 'a
4 val printf : ('a, out_channel, unit) format -> 'a
5 val spaces : int -> string
6 val string_of_syms : string list -> string
7 val permutation : 'a list -> 'a list
8 val filter_duplicates : 'a list -> 'a list
9 val heads : 'a list list -> 'a list

```

---

## strips.mli

```

1 (* symbol table interface *)
2
3 type sym = string
4
5 (* problem table for planner *)
6 type strips_problem = (* TODO: make these names consistent with others *)
7   {
8     mutable init : Ast.predicate list;
9     mutable goal : Ast.predicate list;
10    mutable ops : Ast.action list ;
11  }
12
13 (* environment *)
14 type env =
15   {
16     parent: env option;
17     bindings: (sym, sym) Hashtbl.t;
18     mutable problem: strips_problem
19   }
20
21 val make : env option -> env
22
23 (* convert an ast expression into a strips problem *)
24 val strips_of_ast : env -> Ast.expr -> unit

```

---

## planner.mli

```

1 (* planner interface *)
2
3 module Atomhash :
4   sig
5     type key = Ast.atom
6     type 'a t
7     val create : int -> 'a t
8     val clear : 'a t -> unit
9     val reset : 'a t -> unit
10    val copy : 'a t -> 'a t
11    val add : 'a t -> key -> 'a -> unit
12    val remove : 'a t -> key -> unit
13    val find : 'a t -> key -> 'a
14    val find_all : 'a t -> key -> 'a list

```

```

15     val replace : 'a t -> key -> 'a -> unit
16     val mem : 'a t -> key -> bool
17     val iter : (key -> 'a -> unit) -> 'a t -> unit
18     val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
19     val length : 'a t -> int
20     val stats : 'a t -> Hashtbl.statistics
21 end
22
23 val intersect : 'a list -> 'a list -> 'a list
24 val backtrack : 'a list -> 'a list
25 val prioritize : 'a list list -> 'a list list
26 val remove : 'a -> 'a list -> 'a list
27 val lookup_op : Ast.action list -> Ast.sym -> Ast.action
28 val param_name : Ast.atom -> Ast.atom
29 val partition_conjunction :
30     Ast.conjunction -> Ast.conjunction list * Ast.conjunction list
31 val conj_pred : Ast.conjunction -> Ast.predicate
32 val dump_keys : 'a Atomhash.t -> Atomhash.key list
33 val dump_vals : 'a Atomhash.t -> 'a list
34 val pred_name : Ast.predicate -> Ast.sym
35 val pred_val : Ast.predicate -> Ast.atom list
36 val bind : 'a Atomhash.t -> Atomhash.key list -> 'a list -> 'a Atomhash.t
37 val visited : Ast.predicate list -> Ast.predicate list list ->
38     Ast.predicate list list
39 val swap : 'a -> 'a -> 'a list -> 'a list
40 val ground_pred : Ast.atom Atomhash.t -> Ast.predicate -> Ast.predicate
41 val ground_op : Ast.atom Atomhash.t -> Ast.action -> Ast.predicate
42 val action_bindings : Ast.predicate -> Ast.action -> Ast.atom Atomhash.t
43 val extend :
44     Ast.atom Atomhash.t ->
45     Ast.predicate -> Ast.predicate -> Ast.atom Atomhash.t
46 val bindings_valid : 'a Atomhash.t -> 'a Atomhash.t -> bool
47 val matching_preds :
48     Ast.predicate -> Ast.predicate list -> Ast.predicate list
49 val partition_to_predicates :
50     Ast.conjunction -> Ast.predicate list * Ast.predicate list
51 val goal_test : 'a list list -> 'a list -> bool
52 val search_exhausted : 'a list list -> 'a list -> bool
53 val partition_to_grounded_effect :
54     Ast.action -> Ast.predicate -> Ast.predicate list * Ast.predicate list
55 val complimentary_effects : Ast.predicate -> Ast.action -> bool
56 val successor :
57     Ast.predicate list ->
58     Ast.action list -> Ast.predicate -> Ast.predicate list
59 val unify :
60     Ast.action ->
61     Ast.predicate list * Ast.predicate list ->
62     Ast.atom Atomhash.t -> Ast.predicate list -> Ast.predicate list
63 val applicable_instance :
64     Ast.action -> Ast.predicate list -> Ast.predicate list
65 val applicable_instances :
66     Ast.action -> Ast.predicate list -> Ast.predicate list
67 val applicable_actions :
68     Ast.predicate list -> Ast.action list -> Ast.predicate list
69 val successors :

```

```

70 Ast.predicate list list -> Ast.action list ->
71 Ast.predicate list list list -> Ast.predicate list list list
72 val fsearch : Strips.strips_problem -> Ast.predicate list list
73 val solve : Strips.strips_problem -> Ast.predicate list list
74 val string_of_plan : Ast.predicate list list -> string

```

---

#### compile.mli

```

1 (* compile.mli *)
2
3 type instruction =
4   | Push_action of string
5   | Push_pred of string
6   | Pop_pred
7   | Goal
8 (* TODO: clean up! *)
9 val string_of_instruction : instruction -> string
10 val print_bytecode : instruction list -> string
11 val known_action : Ast.sym -> Ast.action list -> bool
12 val translate_params : Ast.atom list -> string
13 val translate_action : Ast.predicate -> instruction
14 val translate_pred : Ast.predicate -> instruction
15 val pop_for : int -> instruction list
16 val translate :
17   Ast.predicate list list -> Strips.strips_problem -> instruction list

```

---

#### execute.mli

```

1 (* execute.mli *)
2 val execute_instructions : Compile.instruction array -> unit
3 val execute_bytecode : Compile.instruction list -> unit

```

---

#### lexer.mll

```

1 (* lexer.mll
2   *
3   * This file converts program text into tokens
4   *)
5
6 { open Parser }
7
8 let wspc = [' ' '\t' '\n']
9 let chrs = ['A'-'Z' 'a'-'z' '+' '-' '*' '/' '=' '<' '>' '!' ':' '_' '?']
10 let digt = ['0'-'9']
11 let symb = chrs(chrs | digt)*
12
13 rule token = parse
14   | wspc           { token lexbuf }
15   | ';'           { comment lexbuf }

```



```

16 | '('          { TOK_LPAR }
17 | ')'          { TOK_RPAR }
18 | "NIL"        { TOK_UNIT }
19 | (digit)+ as nm      { TOK_INT(int_of_string nm) } (* unused *)
20 | symb   as sm      { TOK_SYM(sm) }
21 | eof          { TOK_EOF }
22 | _ as char { failwith ("illegal_character_" ^ Char.escaped char) }
23 and comment = parse
24 | '\n'        { token lexbuf }          (* commentary ends with lines *)
25 | _          { comment lexbuf }       (* ignore other characters *)

```

---

### parser.mly

```

1  /* parser
2  *
3  * this translates a sequence of tokens into s-expressions.
4  * each call to the parser returns an s-expression. at the
5  * end of the file the parser will return None; hence the
6  * program returns option.
7  */
8
9  %{ open Sexpr %}
10
11 /* token declarations */
12 %token TOK_LPAR TOK_RPAR
13 %token      TOK_UNIT
14 %token <int> TOK_INT
15 %token <string> TOK_SYM
16 %token      TOK_EOF
17
18 /* parse the five non-terminals of the syntax */
19 %start parse
20 %type <Sexpr.expr option> parse
21 %type <Sexpr.expr>      sexpr
22 %type <Sexpr.atom>      atom
23 %type <Sexpr.expr list> slist
24 %type <Sexpr.expr list> sexpr_list
25
26 %%
27
28 /* rules */
29
30 parse:
31 /* an s-expression, or none if eof is encountered*/
32 | TOK_EOF      { None }
33 | sexpr       { Some $1 }
34
35 sexpr:
36 /* an s-expression, an atom or list of s-expressions */
37 | atom        { Expr_atom $1 }
38 | slist       { Expr_list $1 }
39
40 atom:

```

```

41 /* an atom, which can be a unit, int or string. */
42 | TOK_UNIT      { Atom_unit   }
43 | TOK_INT       { Atom_int    $1 }
44 | TOK_SYM       { Atom_sym    $1 }
45
46 slist:
47 /* a list of s-expressions, with parentheses */
48 | TOK_LPAR sexpr_list TOK_RPAR { List.rev $2 }
49
50 sexpr_list:
51 /* the list contents of s-expressions, sans parentheses. */
52 | /* nothing */ { [ ] } /* empty list */
53 | sexpr_list sexpr { $2 :: $1 } /* sexpr list */
54
55
56 %%

```

---

## sexpr.ml

```

1  (*
2  * sexpr.ml
3  *
4  *   S-expressions.
5  *
6  *)
7
8  type atom =
9    | Atom_unit
10   | Atom_int  of int
11   | Atom_sym  of string
12
13  type expr =
14    | Expr_atom  of atom
15    | Expr_list  of expr list
16
17
18  (* Convert an atom into a string. *)
19  let string_of_atom a =
20    match a with
21    | Atom_unit   -> "NIL"
22    | Atom_int    i -> string_of_int i
23    | Atom_sym    s -> s
24
25
26  (* Return a string of n spaces. *)
27  let spaces n = String.make n ' '
28
29  let string_of_expr sx =
30    let rec iter_string_of_expr sx indent =
31      begin
32        match sx with
33        | Expr_atom a ->
34          let s = string_of_atom a in

```

```

35     Printf.sprintf "Expr_atom[%s]" s
36 | Expr_list slist ->
37     "\n"
38     ^ (spaces indent)
39     ^ "Expr_list[_"
40     ^ (iter_string_of_expr_list slist (indent + 2))
41     ^ " _]"
42 end
43 and iter_string_of_expr_list slist indent =
44 begin
45     match slist with
46     | [] -> ""
47     | [s] -> iter_string_of_expr s indent
48     | h :: t ->
49         (iter_string_of_expr h indent)
50         ^ " _"
51         ^ (iter_string_of_expr_list t indent)
52 end
53 in
54 iter_string_of_expr sx 0
55
56 let string_of_expr2 sx =
57 let rec iter_string_of_expr sx indent =
58 begin
59     match sx with
60     | Expr_atom a ->
61         let s = string_of_atom a in
62         Printf.sprintf "%s" s
63     | Expr_list slist ->
64         "\n"
65         ^ (spaces indent)
66         ^ "("
67         ^ (iter_string_of_expr_list slist (indent + 2))
68         ^ ")"
69     end
70 and iter_string_of_expr_list slist indent =
71 begin
72     match slist with
73     | [] -> ""
74     | [s] -> iter_string_of_expr s indent
75     | h :: t ->
76         (iter_string_of_expr h indent)
77         ^ " _"
78         ^ (iter_string_of_expr_list t indent)
79     end
80 in
81 iter_string_of_expr sx 0

```

---

ast.ml

```

1 (* ast.ml *)
2 open Sexpr

```

```

3  open Util
4
5  type sym = string
6
7  type atom =
8    | Atom_var of sym (* ?symbol *)
9    | Atom_gnd of sym (* symbol *)
10   | Atom_nil
11
12 type predicate =
13   | Pred_var of sym * atom list (* (predname vatom v/gatom?) *)
14   | Pred_gnd of sym * atom list (* (predname gatom gatom?) *)
15   | Pred_nil
16
17 type conjunction =
18   | Conj_and of conjunction list
19   | Conj_neg of predicate
20   | Conj_pos of predicate
21   | Conj_nil
22
23 type action =
24 {
25   name          : sym;
26   parameters    : atom list;
27   precondition  : conjunction;
28   effect        : conjunction
29 }
30
31 type expr =
32   | Expr_domain    of sym * expr list (* (define ( domain pman ) ... )*)
33   | Expr_problem   of sym * expr list (* (define ( problem prob ) ... )*)
34   | Expr_predicates of predicate list (* :predicates body *)
35   | Expr_init      of predicate list (* :init body *)
36   | Expr_goal      of predicate list (* :goal body *)
37   | Expr_action    of action          (* :action ... *)
38   | Expr_objects   of atom list       (* :objects body *)
39   | Expr_unit      of ()              (* () *)
40
41 let sprintf = Printf.sprintf
42
43 let spaces n = String.make n ' '
44
45 (* string from string list *)
46 let rec string_of_syms sym_lst =
47   (match sym_lst with
48    | []   -> ""
49    | [s] -> s
50    | h::t -> h ^ "_" ^ (string_of_syms t)
51   )
52
53 (* sexpr.atom -> string *)
54 let sym_of_atom a =
55   ( match a with
56    | Atom_unit -> "" (* shouldn't be used *)
57    | Atom_sym s -> s

```

```

58     | Atom_int _ ->
59       let error_msg = "improper_symbol_conversion" in
60       failwith error_msg
61   )
62
63   (* sexpr.atom -> ast.atom *)
64   let astatom_of_atomsym atom_sym =
65     ( match atom_sym with
66       | Atom_sym s ->
67         let prefix = String.get s 0 in
68         ( match prefix with
69           | '?' -> Atom_var s
70           | _ -> Atom_gnd s
71         )
72       | _ -> failwith "only_accepts_type_atom_sym"
73     )
74
75   (* sexpr.expr -> ast.atom *)
76   let astatom_of_sexpr sx =
77     ( match sx with
78       | Expr_atom ( Atom_sym s ) ->
79         let prefix = String.get s 0 in
80         ( match prefix with
81           | '?' -> Atom_var s
82           | _ -> Atom_gnd s
83         )
84       | _ -> failwith "only_accepts_type_atom_sym"
85     )
86
87   (* TODO: account for nil cases *)
88   (* this will need to be recursive for nested calls *)
89   (* sexpr.expr -> ast.predicate *)
90   let pred_of_sexpr sx =
91     ( match sx with
92       | Expr_list l ->
93         ( match l with
94           | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ; Expr_atom p2 ] ->
95             let a1 = astatom_of_atomsym p1 and
96                 a2 = astatom_of_atomsym p2 in
97             ( match a1 with
98               | Atom_var _ ->
99                 Pred_var( name , [a1 ; a2] )
100             | Atom_gnd _ ->
101               ( match a2 with
102                 | Atom_var _ ->
103                   Pred_var( name , [a1 ; a2] )
104                 | Atom_gnd _ ->
105                   Pred_gnd( name , [a1 ; a2] )
106                 | Atom_nil ->
107                   let error_msg = "unrecognized_predicate" in
108                   failwith error_msg
109               )
110             | Atom_nil ->
111               let error_msg = "unrecognized_predicate" in
112               failwith error_msg

```

```

113     )
114   | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ] ->
115     let a1 = astatom_of_atomsym p1 in
116     ( match a1 with
117     | Atom_var _ ->
118       Pred_var( name , [a1] )
119     | _ ->
120       Pred_gnd( name , [a1] )
121     )
122   | _ ->
123     let error_msg = "unrecognized_predicate" in
124     failwith error_msg
125   )
126 | _ -> failwith "unrecognized_conjunction_structure"
127 )
128
129 (* sexpr.expr list -> atom list *)
130 let params_of_sexpr sx =
131   ( match sx with
132   | Expr_atom _ :: _ ->
133     List.map astatom_of_sexpr sx
134   | _ -> failwith "syntax_error:_unrecognized_parameter"
135   )
136
137 (* sexpr.expr -> conjunction *)
138 let rec conj_of_sexpr sx =
139   ( match sx with
140   | Expr_list l ->
141     ( match l with
142     | [ Expr_atom( Atom_sym name ) ; Expr_atom p1 ; Expr_atom p2 ] ->
143       let a1 = astatom_of_atomsym p1
144         and a2 = astatom_of_atomsym p2 in
145       ( match a1 with
146       | Atom_var _ ->
147         Conj_pos( Pred_var( name , [a1 ; a2] ) )
148       | Atom_gnd _ ->
149         ( match a2 with
150         | Atom_var _ ->
151           Conj_pos( Pred_var( name , [a1 ; a2] ) )
152         | Atom_gnd _ ->
153           Conj_pos( Pred_gnd( name , [a1 ; a2] ) )
154         | Atom_nil ->
155           let error_msg = "unrecognized_predicate" in
156           failwith error_msg
157         )
158       | Atom_nil ->
159         let error_msg = "unrecognized_predicate" in
160         failwith error_msg
161     )
162   | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ] ->
163     let a1 = astatom_of_atomsym p1 in
164     ( match a1 with
165     | Atom_var _ ->
166       Conj_pos( Pred_var( name , [a1] ) )
167     | _ ->

```

```

168         Conj_pos(Pred_gnd( name , [a1] ))
169     )
170 | Expr_atom ( Atom_sym "and" ) :: body -> (* body is list *)
171     Conj_and ( List.map conj_of_sexpr body )
172 | [ Expr_atom ( Atom_sym "not" ) ; p1 ] ->
173     Conj_neg ( pred_of_sexpr p1 ) (*TODO: nesting ands *)
174 | _ ->
175     let error_msg = "invalid_conjunction" in
176     failwith error_msg
177 )
178 | _ ->
179     let error_msg = "invalid_conjunction" in
180     failwith error_msg
181 )
182
183 (* sexpr.expr list -> ast.action *)
184 let action_of_sexpr sx =
185     ( match sx with
186     | Expr_atom _ :: (* :action *)
187         Expr_atom ( Atom_sym name ) ::
188         Expr_atom ( Atom_sym ":parameters" ) :: Expr_list params ::
189         Expr_atom ( Atom_sym ":precondition" ) :: Expr_list precondition ::
190         Expr_atom ( Atom_sym ":effect" ) :: Expr_list effect :: [] ->
191         {
192         name = name;
193         parameters = params_of_sexpr params;
194         precondition = conj_of_sexpr (Expr_list(precond));
195         effect = conj_of_sexpr (Expr_list(effect))
196         }
197     | _ ->
198         let error_msg = "unrecognized_action_structure" in
199         failwith error_msg
200     )
201
202 let rec ast_of_sexpr sx =
203     (match sx with
204     | Expr_atom a ->
205         let error_msg = "unrecognized_s-expression" in
206         failwith error_msg
207     | Expr_list l ->
208         ( match l with
209         | Expr_atom ( Atom_sym ":predicates" ) :: body ->
210             Expr_predicates( List.map pred_of_sexpr body )
211         | Expr_atom ( Atom_sym ":action" ) :: body ->
212             Expr_action( action_of_sexpr l )
213         | Expr_atom ( Atom_sym ":objects" ) :: body ->
214             Expr_objects( List.map astatom_of_sexpr body )
215         | Expr_atom ( Atom_sym ":init" ) :: body ->
216             Expr_init( List.map pred_of_sexpr body )
217         | Expr_atom ( Atom_sym ":goal" ) :: body ->
218             Expr_goal( List.map pred_of_sexpr body )
219         | Expr_atom ( Atom_sym "define" ) ::
220             Expr_list l :: body -> (* body *)
221             ( match l with
222             | [Expr_atom ( Atom_sym "domain" ) ;

```

```

223         Expr_atom ( Atom_sym name )] ->
224         Expr_domain( name , List.map ast_of_sexpr body )
225     | [Expr_atom ( Atom_sym "problem" ) ;
226        Expr_atom ( Atom_sym name )] ->
227         Expr_problem( name , List.map ast_of_sexpr body )
228     | _ ->
229         let error_msg = "unrecognized_s-expression" in
230             failwith error_msg
231     )
232 | _ ->
233     let error_msg = "unrecognized_s-expression" in
234         failwith error_msg
235 )
236 )
237
238
239 let string_of_atom atom =
240   ( match atom with
241     | Atom_var a -> a
242     | Atom_gnd a -> a
243     | Atom_nil -> "NIL"
244   )
245
246 let string_of_pred pred =
247   ( match pred with
248     | Pred_var( name , params ) ->
249       sprintf "PRED_VAR(␣%s␣,␣%s␣)\n"
250         name
251         (string_of_syms(List.map string_of_atom params))
252     | Pred_gnd( name , params ) ->
253       sprintf "PRED_GND(␣%s␣,␣%s␣)\n"
254         name
255         (string_of_syms(List.map string_of_atom params))
256     | Pred_nil ->
257       "PRED_NIL()\n"
258   )
259
260 let rec string_of_conj conj =
261   let recurse = string_of_conj in
262   ( match conj with
263     | Conj_and c ->
264       sprintf "CONJ_AND(%s)"
265         (string_of_syms (List.map recurse c))
266     | Conj_pos c ->
267       sprintf "CONJ_POS(%s)"
268         (string_of_pred c)
269     | Conj_neg c ->
270       sprintf "CONJ_NEG(%s)"
271         (string_of_pred c)
272     | Conj_nil -> ""
273   )
274
275 let string_of_params params =
276   sprintf "PARAMETERS(␣%s␣)"
277     (string_of_syms (List.map string_of_atom params))

```



```

278
279 let string_of_precond precondition =
280   sprintf "PRECONDITION[_%s_]"
281   (string_of_conj precondition)
282
283 let string_of_effect effect =
284   sprintf "EFFECT[_%s_]"
285   (string_of_conj effect)
286
287 let string_of_action act =
288   act.name ^ "\n" ^
289   (string_of_params act.parameters) ^ "\n" ^
290   (string_of_precond act.precondition) ^ "\n" ^
291   (string_of_effect act.effect)
292
293
294 let string_of_ast ast =
295   let rec iter ast indent =
296     let string_of_exprs e_list =
297       (List.fold_left (^) ""
298        (List.map (fun e -> "\n" ^ iter e (indent + 2)) e_list))
299     in
300     ( match ast with
301     | Expr_unit -> sprintf "%sUNIT" (spaces indent)
302     | Expr_domain ( name , body ) ->
303       sprintf "%sDOMAIN[_%s\n%s_]"
304         (spaces indent) name (string_of_exprs body)
305     | Expr_problem ( name , body ) ->
306       sprintf "%sPROBLEM[_%s\n%s_]"
307         (spaces indent) name (string_of_exprs body)
308     | Expr_predicates( preds ) ->
309       sprintf "%sPREDICATES[_%s_]\n"
310         ( spaces indent )
311         (string_of_syms (List.map string_of_pred preds))
312     | Expr_action ( act ) ->
313       sprintf "%sACTION[_%s_]\n"
314         (spaces indent)
315         (string_of_action act)
316     | Expr_objects ( objs ) ->
317       sprintf "%sOBJECTS[_%s_]\n"
318         (spaces indent)
319         (string_of_syms (List.map string_of_atom objs))
320     | Expr_init ( preds ) ->
321       sprintf "%sINIT[_%s_]\n"
322         (spaces indent)
323         (string_of_syms (List.map string_of_pred preds))
324     | Expr_goal ( preds ) ->
325       sprintf "%sGOAL[_%s_]\n"
326         (spaces indent)
327         (string_of_syms (List.map string_of_pred preds))
328     )
329   in "\n" ^ iter ast 0 ^ "\n"

```

---

## util.ml

```

1  (* util.ml *)
2  open Sexpr
3  open Ast
4  open Strips
5  open Planner
6
7  let sprintf = Printf.sprintf
8
9  let printf = Printf.printf
10
11 let spaces n = String.make n ' '
12
13 (* string from string list *)
14 let rec string_of_syms = function
15   | [] -> ""
16   | [s] -> s
17   | h::t -> h ^ "_" ^ (string_of_syms t)
18
19 (* returns a random permutation of a list *)
20 let rec permutation list =
21   let rec extract acc n = function
22     | [] -> raise Not_found
23     | h::t -> if n = 0 then (h, acc@t) else extract (h::acc) (n-1) t
24   in
25   let extract_rand list len =
26     extract [] (Random.int len) list
27   in
28   let rec aux acc list len =
29     if len = 0 then acc else
30       let picked, rest = extract_rand list len in
31       aux (picked::acc) rest (len-1)
32   in aux [] list (List.length list)
33
34 (* deletes duplicates from a list *)
35 let rec filter_duplicates = function
36   | []->[]
37   | h::t-> h::(filter_duplicates(List.filter(fun x -> x<>h )t))
38
39 (* returns a list of heads from a list of lists *)
40 let heads list =
41   let rec decapitate acc = function
42     | [] -> List.rev acc
43     | h::t -> decapitate ((List.hd h)::acc) t
44   in decapitate [] list

```

## strips.ml

```

1  (* symbol table generation *)
2  open Ast
3  open Util
4
5  type sym = string

```

```

6
7 (* problem table for planner *)
8 type strips_problem =
9   {
10     mutable init : Ast.predicate list;
11     mutable goal : Ast.predicate list;
12     mutable ops  : Ast.action list  ;
13   }
14
15 (* environment *)
16 type env =
17   {
18     parent: env option;
19     bindings: (sym, sym) Hashtbl.t;
20     mutable problem: strips_problem (* TODO: test this *)
21   }
22
23 (** level-one dependency **)
24
25 (* create a new strips problem and symbol table *)
26 let make parent =
27   {
28     parent = parent;
29     bindings = Hashtbl.create 5;
30     problem = { init = [] ; goal = [] ; ops = [] }
31   }
32
33 (* lookup symbol table entry *)
34 let rec lookup env name =
35   let { parent = p ; bindings = b } = env in
36   try Hashtbl.find b name
37   with Not_found ->
38     ( match p with
39       | Some( parent ) -> lookup parent name
40       | None ->
41         let error_msg = "unknown_symbol:_" ^ name in
42         failwith error_msg
43     )
44
45 (* lookup symbol table entry with an atom *)
46 let lookup_atom env atom =
47   ( match atom with
48     | Atom_var a -> lookup env a
49     | Atom_gnd a -> lookup env a
50     | _ ->
51       let error_msg = "unknown_atom" in
52       failwith error_msg
53   )
54
55 (* insert entry into symbol table *)
56 let add env name value =
57   let { parent = _ ; bindings = b } = env in
58   Hashtbl.add b name value
59
60 (* insert initial state into strips problem *)

```

```

61 let add_init env state =
62   env.problem.init <- state
63
64 (* insert goal state into strips problem *)
65 let add_goal env state =
66   env.problem.goal <- state
67
68 (* insert operator into strips problem *)
69 let add_action env act =
70   let acts = env.problem.ops in
71   env.problem.ops <- act::acts
72
73 (* split the atom for its name, plus semantic checks *)
74 let atom_name a =
75   if String.contains a '-' then
76     let stop = (String.index a '-') in
77     String.sub a 0 stop
78   else
79     let error_msg = "expected_a_type_specification_for_" ^ a ^ "'" in
80     failwith error_msg
81
82 (* split the atom for its type, plus semantic checks *)
83 let atom_type a =
84   if String.contains a '-' then
85     let start = (String.index a '-') + 1 in
86     let stop = (String.length a) - start in
87     String.sub a start stop
88   else
89     let error_msg = "expected_a_type_specification_for_" ^ a ^ "'" in
90     failwith error_msg
91
92 (* insert grounded atom into symbol table using object semantics *)
93 let translate_object env obj =
94   let { parent = genv ; bindings = local } = env in
95   ( match genv with
96     | None ->
97       let error_msg = "no_global_symbol_table_for_parameters" in
98       failwith error_msg
99     | Some( global_env ) ->
100      let { parent = _ ; bindings = global } = global_env in
101      ( match obj with (* seek type *)
102        | Atom_gnd a ->
103          ( try
104            let _ = Hashtbl.find local (atom_name a) in
105            let error_msg =
106              "an_object_named_"
107              ^ (atom_name a)
108              ^ "'_has_already_been_declared_in_this_scope" in
109            failwith error_msg
110          with Not_found -> (* object name is unique *)
111            ( try
112              let _ = Hashtbl.find global (":type-^(atom_type a)) in
113              add env (atom_name a) (atom_type a)
114            with Not_found ->
115              let error_msg =

```

```

116             "expected_type_declaration_for_"
117             ^ (atom_type a)
118             ^ "/"
119             in failwith error_msg
120         )
121     )
122 | Atom_var a ->
123     let error_msg =
124         "expected_"
125         ^ a
126         ^ "_to_be_a_grounded_atom" in
127     failwith error_msg
128 | Atom_nil ->
129     let error_msg = "expected_object_declaration_to_be_non-empty" in
130     failwith error_msg
131 )
132 )
133
134 (* prints the strips problem *)
135 let string_of_strips prob =
136     let { init = s0 ; goal = g ; ops = acts } = prob in
137     let title =
138         "STRIPS_PROBLEM\n" in
139     let initial_state =
140         "INITIAL_STATE:\n"
141         ^ (string_of_syms (List.map string_of_pred s0)) in
142     let goal_state =
143         "GOAL_STATE:\n"
144         ^ (string_of_syms (List.map string_of_pred g)) in
145     let actions =
146         "ACTIONS:\n"
147         ^ (string_of_syms (List.map string_of_action acts)) in
148     sprintf "%s\n%s\n%s\n%s\n"
149         (title)
150         (initial_state)
151         (goal_state)
152         (actions)
153
154 (** level-two dependency **)
155
156 (* insert a grounded atom into symbol table *)
157 let translate_grounded_param env param =
158     ( match param with (* TODO: type? *)
159     | Atom_gnd a ->
160         add env ":static" (atom_name a)
161     | _ -> failwith "parameter_improperly_parsed"
162     )
163
164 (* *)
165 let translate_predicate_param env param = (* local parameter scope *)
166     let { parent = genv ; bindings = local } = env in
167     ( match genv with
168     | None ->
169         let error_msg = "no_global_symbol_table_for_parameters" in
170         failwith error_msg

```

```

171 | Some( global_env ) ->
172 | let { parent = _ ; bindings = global } = global_env in
173 | ( match param with
174 | | Atom_var a ->
175 | | ( try (* parameter name must be unique in local scope *)
176 | | let _ = Hashtbl.find local (atom_name a) in
177 | | let error_msg =
178 | | "a_predicate_parameter_named_"
179 | | ^ (atom_name a)
180 | | ^ "'_has_already_been_declared_in_its_scope" in
181 | | failwith error_msg
182 | | with Not_found -> (* parameter name is unique *)
183 | | let _ = add env (atom_name a) (atom_type a) in
184 | | Hashtbl.replace global (":type-"^(atom_type a)) ":type"
185 | | )
186 | | Atom_gnd a ->
187 | | let error_msg =
188 | | "expected_"
189 | | ^ a
190 | | ^ "'_to_be_a_variable_parameter" in
191 | | failwith error_msg
192 | | _ ->
193 | | let error_msg =
194 | | "expected_parameter_declaration_to_be_non-empty" in
195 | | failwith error_msg
196 | | )
197 | )
198 (* insert variable atom into symbol table using parameter semantics *)
199 let translate_parameter_declaration env param =
200 | ( match param with
201 | | Atom_var a ->
202 (* check types of each atom exist and are used properly in declaration *)
203 | | let { parent = p ; bindings = b } = env in
204 | | ( try
205 | | let _ = Hashtbl.find b (atom_name a) in
206 | | let error_msg =
207 | | "a_parameter_named_"
208 | | ^ (atom_name a)
209 | | ^ "'_has_already_been_declared_in_this_scope" in
210 | | failwith error_msg
211 | | with Not_found -> (* parameter name is unique *)
212 | | add env (atom_name a) (atom_type a)
213 | | )
214 | | Atom_gnd a ->
215 | | let error_msg =
216 | | "expected_"
217 | | ^ a
218 | | ^ "'_to_be_a_variable_parameter" in
219 | | failwith error_msg
220 | | _ ->
221 | | let error_msg =
222 | | "expected_parameter_declaration_to_be_non-empty" in
223 | | failwith error_msg
224 | | )
225 | )

```

```

226 (* insert objects into symbol table *)
227 let translate_objects env params =
228   let translate_obj = translate_object env in
229   List.iter translate_obj params
230
231 (* lookup parameters *)
232 let check_params env params = (* params = atom list *)
233   let { parent = genv ; bindings = local } = env in
234   ( match genv with
235     | None ->
236       let error_msg = "no_global_symbol_table_for_predicate" in
237       failwith error_msg
238     | Some( global_env ) ->
239       let rec check_parameter parameters =
240         ( match parameters with
241           | [] -> ()
242           | Atom_var(a)::t ->
243             ( try (* check that parameter name exists in the local scope
244                  *)
245               let _ = Hashtbl.find local a in (* type not specified *)
246               check_parameter t
247             with Not_found ->
248               let error_msg =
249                 "expected_parameter_declaration_for_"
250                 ^ a
251                 ^ "/"
252               in failwith error_msg
253             )
254           | Atom_gnd(a)::t ->
255             ( try (* check that parameter name exists in the local scope
256                  *)
257               let _ = Hashtbl.find local a in (* type not specified *)
258               check_parameter t
259             with Not_found ->
260               let error_msg =
261                 "expected_parameter_declaration_for_"
262                 ^ a
263                 ^ "/"
264               in failwith error_msg
265             )
266           | _ ->
267             let error_msg = "expected_grounded_or_variable_atom" in
268             failwith error_msg
269         )
270       in check_parameter params
271   )
272
273 (** level-three dependency **)
274
275 (* insert a list of variable atoms using parameter semantics *)
276 let translate_parameter_declarations env params =
277   let translate_param = translate_parameter_declaration env in
278   List.iter translate_param params

```

```

*)
279 let translate_predicate_params env params =
280   let translate_param = translate_predicate_param env in (* local
      parameter scope *)
281   List.iter translate_param params
282
283 (* insert grounded atoms into symbol table using parameter semantics *)
284 let translate_gnd_params env params =
285   let translate_gnd_param = translate_grounded_param env in
286   List.iter translate_gnd_param params
287
288 (* lookup predicate declaration *)
289 let check_var_pred env pred = (* local action parameter scope *)
290   let { parent = genv ; bindings = local } = env in
291   ( match genv with
292     | None ->
293       let error_msg = "no_global_symbol_table_for_predicate" in
294       failwith error_msg
295     | Some( global_env ) ->
296       let { parent = _ ; bindings = global } = global_env in
297       ( match pred with
298         | Pred_var( name , params ) -> (* check pred is declared in global
          *)
299         ( try
300           let _ = Hashtbl.find global (":predicate-"^name) in
301           check_params env params
302         with Not_found ->
303           let error_msg =
304             "expected_declaration_for_predicate_"
305             ^ name in
306           failwith error_msg
307         )
308         | Pred_gnd( name , params ) ->
309           let error_msg =
310             "expected_"
311             ^ name
312             ^ "'_to_be_a_variable_predicate"
313           in failwith error_msg
314         | Pred_nil ->
315           let error_msg =
316             "expected_predicate_to_be_non-empty" in
317           failwith error_msg
318       )
319   )
320
321 (* lookup a grounded predicate using state semantics *)
322 let check_gnd_pred env pred =
323 let { parent = genv ; bindings = local } = env in
324 ( match genv with
325   | None ->
326     let error_msg = "no_global_symbol_table_for_predicate" in
327     failwith error_msg
328   | Some( global_env ) ->
329     let { parent = _ ; bindings = global } = global_env in
330     ( match pred with

```



```

331 | Pred_gnd( name , params ) -> (* check pred is declared in global
332 *)
333 ( try
334   let _ = Hashtbl.find global (":predicate-"^name) in
335   check_params env params
336   with Not_found ->
337     let error_msg =
338       "expected_declaration_for_predicate_"
339       ^ name in
340     failwith error_msg
341 )
342 | Pred_var( name , params ) ->
343   let error_msg =
344     "expected_"
345     ^ name
346     ^ "'_to_be_a_ grounded_predicate"
347   in failwith error_msg
348 | Pred_nil ->
349   let error_msg =
350     "expected_predicate_to_be_non-empty" in
351   failwith error_msg
352 )
353
354 (** level-four dependency **)
355
356 let rec translate_precondition env precondition = (* local action scope *)
357   let recurse = translate_precondition env in
358   ( match precondition with
359   | Conj_and conj ->
360     List.iter recurse conj
361   | Conj_neg pred ->
362     check_var_pred env pred
363   | Conj_pos pred ->
364     check_var_pred env pred
365   | _ ->
366     let error_msg = "empty_precondition" in
367     failwith error_msg
368   )
369
370 let translate_effect env effect =
371   translate_precondition env effect
372
373 let translate_state env state =
374   let check_state = check_gnd_pred env in
375   List.iter check_state state
376
377 (* check and add a predicate declaration into the symbol table *)
378 let translate_predicate_declaration env pred =
379   ( match pred with
380   | Pred_var( name , params ) ->
381     let { parent = p ; bindings = b } = env in
382     ( try
383       let _ = Hashtbl.find b (":predicate-"^name) in
384       let error_msg =

```

## 1 Source Code

```
385         "a_predicate_named,'"
386         ^ name
387         ^ "'_has_already_been_declared_in_this_scope" in
388         failwith error_msg
389     with Not_found -> (* predicate name is unique *)
390         let _ = add env (":predicate-"^name) ":predicate" in
391         let parent = Some(env) in
392         let param_env = make parent in
393         translate_predicate_params param_env params
394     )
395 | Pred_gnd( name , params ) ->
396     let error_msg =
397         "expected,'"
398         ^ name
399         ^ "'_to_be_a_variable_predicate" in
400     failwith error_msg
401 | Pred_nil ->
402     let error_msg =
403         "predicate_declaration_should_be_non-empty" in
404     failwith error_msg
405 )
406
407 (* add parameter types to global scope *)
408 let translate_predicate_param_to_global env pred =
409     ( match pred with
410     | Pred_var( name , params ) ->
411         let rec loop count p =
412             ( match p with
413             | [] -> ()
414             | Atom_var(a)::t ->
415                 let c = sprintf "%d" count in
416                 let _ =
417                     add env
418                     ("predicate-"^name^-p"^c^-("^ (atom_type a)
419                     ":typespec" in
420                     loop (count+1) t
421             | _ ->
422                 let error_msg =
423                     "expected_variable_atom_before_this_point..." in
424                 failwith error_msg
425             )
426         in loop 0 params
427     | _ ->
428         let error_msg = "expected_variable_predicate_before_this_point..."
429         in
430         failwith error_msg
431     )
432 let check_var_pred_types local_env pred =
433     ( match pred with
434     | Pred_var( name , params ) ->
435         let rec loop count p =
436             ( match p with
437             | [] -> ()
438             | Atom_var(a)::t ->
```

```

439     let action_type = lookup local_env a in
440     let c = sprintf "%d" count in
441     let action_type_spec = "predicate-'^name^-p'^c^-'^action_type
      in
442     let { parent = genv ; bindings = local } = local_env in
443     ( match genv with
444     | None ->
445         let error_msg = "no_global_symbol_table_for_predicate" in
446         failwith error_msg
447     | Some( global_env ) ->
448         let { parent = _ ; bindings = global } = global_env in
449         ( try
450             let _ = Hashtbl.find global action_type_spec in
451             loop (count+1) t
452         with Not_found ->
453             let error_msg =
454                 "expected_a_different_type_for_parameter_'
455                 ^ c
456                 ^ "'_of_action_predicate_'
457                 ^ name
458                 ^ "'
459             in failwith error_msg
460         )
461     )
462     | _ ->
463         let error_msg =
464             "expected_variable_atom_before_this_point_..." in
465         failwith error_msg
466     )
467     in loop 0 params
468 | _ ->
469     let error_msg =
470         "expected_variable_predicate_before_this_point_..." in
471     failwith error_msg
472 )
473
474 let check_gnd_pred_types local_env pred =
475 ( match pred with
476 | Pred_gnd( name , params ) ->
477     let rec loop count p =
478         ( match p with
479         | [] -> ()
480         | Atom_gnd(a)::t ->
481             let state_type = lookup local_env a in
482             let c = sprintf "%d" count in
483             let state_type_spec = "predicate-'^name^-p'^c^-'^state_type in
484             let { parent = genv ; bindings = local } = local_env in
485             ( match genv with
486             | None ->
487                 let error_msg = "no_global_symbol_table_for_predicate" in
488                 failwith error_msg
489             | Some( global_env ) ->
490                 let { parent = _ ; bindings = global } = global_env in
491                 ( try
492                     let _ = Hashtbl.find global state_type_spec in

```

```

493         loop (count+1) t
494     with Not_found ->
495         let error_msg =
496             "expected_a_different_type_for_parameter_' "
497             ^ c
498             ^ "'_of_state_predicate_' "
499             ^ name
500             ^ "' "
501         in failwith error_msg
502     )
503 )
504 | _ ->
505     let error_msg =
506         "expected_grounded_atom_before_this_point_..." in
507     failwith error_msg
508 )
509 in loop 0 params
510 | _ ->
511     let error_msg =
512         "expected_grounded_predicate_before_this_point_..." in
513     failwith error_msg
514 )
515
516 let rec check_types local_env conj =
517     let recurse = check_types local_env in
518     ( match conj with
519     | Conj_and c ->
520         List.iter recurse c
521     | Conj_neg pred ->
522         check_var_pred_types local_env pred
523     | Conj_pos pred ->
524         check_var_pred_types local_env pred
525     | _ ->
526         let error_msg = "empty_conjugate" in
527         failwith error_msg
528     )
529
530
531 (** level-five dependency **)
532
533 (* inset predicates into symbol table *)
534 let translate_predicate_declaration env preds =
535     let translate_pred = translate_predicate_declaration env in
536     let _ = List.iter translate_pred preds in
537     let translate_to_global = translate_predicate_param_to_global env in
538     List.iter translate_to_global preds
539
540 (* insert action into symbol table and stips problem *)
541 let translate_action env act =
542     let {
543         name = n ;
544         parameters = params ; (* enforce variable homogeneity *)
545         precondition = precond ;
546         effect = eff
547     } = act in

```

```

548 let { parent = p ; bindings = b } = env in
549   ( try
550     let _ = Hashtbl.find b (":action-"^n) in
551     let error_msg =
552       "an_action_named_"
553       ^ n
554       ^ "'_has_already_been_declared_in_this_scope" in
555     failwith error_msg
556   with Not_found -> (* parameter name is unique *)
557     let _ = add env (":action-"^n) ":action" in
558     let parent = Some(env) in
559     let param_env = make parent in (* make parameter table *)
560     let _ = translate_parameter_declarations param_env params in
561     let _ = translate_precondition param_env precondition in
562     let _ = check_types param_env precondition in
563     let _ = translate_effect param_env eff in
564     check_types param_env eff
565   )
566
567 (** level-six dependency **)
568
569 (* mapping from ast to strips problem -- semantic checks burried *)
570 let rec strips_of_ast env ast =
571   let recurse = strips_of_ast env in
572   ( match ast with
573     | Expr_predicates( preds ) ->
574       translate_predicate_declaration env preds
575     | Expr_action( act ) ->
576       let _ = translate_action env act in
577       add_action env act
578     | Expr_objects( objs ) ->
579       translate_objects env objs
580     | Expr_init( init ) ->
581       let _ = translate_state env init in
582       let check_problem_types = check_gnd_pred_types env in
583       let _ = List.iter check_problem_types init in
584       let p = env.parent in (* TODO: cleanup *)
585       ( match p with
586         | Some( parent ) ->
587           add_init parent init
588         | None ->
589           let error_msg = ":init_symbol_table_has_no_parent" in
590           failwith error_msg
591       )
592     | Expr_goal( goal ) ->
593       let _ = translate_state env goal in
594       let check_problem_types = check_gnd_pred_types env in
595       let _ = List.iter check_problem_types goal in
596       let p = env.parent in
597       ( match p with
598         | Some( parent ) ->
599           add_goal parent goal
600         | None ->
601           let error_msg = ":goal_symbol_table_has_no_parent" in
602           failwith error_msg

```

## liv Source Code

```
603     )
604     | Expr_domain( name , body ) ->
605       add env name ":domain" ; List.iter recurse body
606     | Expr_problem( name , body ) ->
607       let parent = Some(env) in
608       let new_env = make parent in
609       add env name ":problem" ; List.iter (strips_of_ast new_env) body
610     | _ ->
611       let error_msg =
612         "program_may_only_contain_a_domain_and_problem_declaration" in
613       failwith error_msg
614   )
```

---

### planner.ml

```
1  open Ast
2  open Strips
3  open Util
4
5  (** level-one dependency **)
6
7  (* ( atom , atom ) Hashtbl *)
8  module Atomhash = Hashtbl.Make
9    (struct
10     type t = atom
11     let equal x y = x = y
12     let hash = Hashtbl.hash
13   end)
14
15  (* returns string from state *)
16  let string_of_state state =
17    (string_of_syms (List.map string_of_pred state))
18
19  (* returns string from plan *)
20  let string_of_plan plan =
21    sprintf "\n%s"
22      (string_of_syms (List.map string_of_state plan))
23
24  (* returns the intersection of two lists *)
25  let intersect l1 l2 = List.filter (fun x -> List.mem x l2) l1
26
27  (* returns the disjunction of two lists *)
28  let disjoin l1 l2 = List.filter (fun x -> not(List.mem x l2)) l1
29
30  (* removes the last state-action combination from a partial plan *)
31  let backtrack pplan =
32    ( match pplan with
33      | state::act::prior_pplan -> prior_pplan
34      | _ ->
35        let error_msg = "backtrack:_improper_input" in
36        failwith error_msg
37    )
38
```

```

39 (* sorts a list of partial plans with respect to their depth *)
40 let prioritize pplans =
41   let randomized_pplans = permutation pplans in
42   List.sort ( fun x y ->
43     let l1 = List.length x and l2 = List.length y in
44     if l1 < l2 then 1 else if l1 = l2 then 0 else -1 ) randomized_pplans
45
46 (* 'a -> 'a list -> 'a list *)
47 let remove pred state =
48   List.filter (fun x -> if x = pred then false else true ) state
49
50 (* returns operator with matching name *)
51 let lookup_op opset name =
52   let filter_name =
53     (fun op -> if op.name = name then true else false ) in
54   List.find filter_name opset
55
56 (* parses atom for parameter name *)
57 let param_name atom =
58   try
59     ( match atom with
60     | Atom_gnd a ->
61       let stop = (String.index a '-') in
62       let name = String.sub a 0 stop in
63       Atom_gnd name
64     | Atom_var a ->
65       let stop = (String.index a '-') in
66       let name = String.sub a 0 stop in
67       Atom_var name
68     | Atom_nil -> Atom_nil
69     )
70   with Not_found ->
71     let error_msg = "No_parameter_type_specified_in:_" in
72     failwith error_msg
73
74
75 (* partitions a conjunction into positive and negative lists *)
76 let partition_conjunction conj =
77   ( match conj with
78   | Conj_and c -> List.partition
79     ( fun x -> match x with | Conj_pos _ -> true | _ -> false ) c
80   | Conj_pos c -> ( [Conj_pos c] , [] )
81   | Conj_neg c -> ( [] , [Conj_neg c] )
82   | Conj_nil -> ( [] , [] )
83   )
84
85 (* ast.conjunction -> ast.predicate *)
86 let conj_pred conj =
87   ( match conj with
88   | Conj_neg nc -> nc
89   | Conj_pos pc -> pc
90   | _ -> Pred_nil
91   )
92
93 let dump_keys env = Atomhash.fold (fun k _ acc -> k::acc) env []

```

```

94
95 let dump_vals env = Atomhash.fold (fun _ v acc -> v::acc) env []
96
97 let pred_name pred =
98   ( match pred with
99     | Pred_var ( name , _ ) -> name
100    | Pred_gnd ( name , _ ) -> name
101    | _ -> ""
102   )
103
104 let pred_val pred =
105   ( match pred with
106     | Pred_var ( _ , value ) -> value
107     | Pred_gnd ( _ , value ) -> value
108     | _ -> [Atom_nil]
109   )
110
111 (* incarnation of iter2 with hash table replacement *)
112 let rec bind env names values =
113   ( match ( names , values ) with
114     | ( [] , [] ) -> env
115     | n::t1 , v::t2 -> Atomhash.replace env n v; bind env t1 t2
116     | ( _ , _ ) -> invalid_arg "bind"
117   )
118
119 (* filters states that are elements of the visited list *)
120 let filter_visited succs visited =
121   let visited_states = heads visited in (* visited = list^3 *)
122   List.filter(
123     fun act_succ ->
124       if ((intersect [(List.tl act_succ)] visited_states) = []) then true
125       else false) succs
126
127 (* return the visited partial plan, if previously visted, else [] *)
128 let visited state pplan =
129   let rec try_remembering acc pp =
130     ( match pp with
131     | [] -> acc
132     | s::remaining_pp -> (* s = state ; t = act::state list *)
133       if s = state then (* pplan head is a state *)
134         try_remembering pp []
135       else
136         ( match remaining_pp with
137         | [] ->
138           try_remembering [] [] (* discovered new state *)
139         | prior_act_state::t ->
140           let prior_state = List.tl prior_act_state in
141           let remaining_pplan = prior_state::t in
142           try_remembering [] remaining_pplan
143         )
144     )
145   in try_remembering [] pplan
146
147 (* swap an old element for a new element in some list *)
148 let swap old_pred new_pred state =

```



```

149 let rec loop new_state old_state =
150   ( match old_state with
151     | [] -> List.rev new_state
152     | state_pred::remaining_preds ->
153       if state_pred = old_pred then
154         loop (new_pred::new_state) remaining_preds
155       else
156         loop (state_pred::new_state) remaining_preds
157   )
158 in loop [] state
159
160 (** level-two dependency **)
161 (* grounds a variable predicate with a set of bindings *)
162 let ground_pred env pred =
163   let keys = pred_val pred in
164   let vals =
165     List.map ( fun k -> Atomhash.find env k ) keys in
166   ( match pred with
167     | Pred_var( name , _ ) -> Pred_gnd( (pred_name pred) , vals )
168     | _ ->
169       let error_msg = "Failed_to_ground_predicate" in
170       failwith error_msg
171   )
172
173 (* grounds an operator with a set of bindings *)
174 let ground_op env op =
175   let names = List.map param_name op.parameters in
176   let vals = List.map ( fun n -> Atomhash.find env n ) names in
177   Pred_gnd( op.name , vals )
178
179 (* returns the bindings for an action *)
180 let action_bindings action op =
181   let env = Atomhash.create (List.length op.parameters) in
182   let names = List.map param_name op.parameters in
183   let vals = pred_val action in
184   let _ = List.iter2 ( fun n v -> Atomhash.add env n v ) names vals in
185   env
186
187 (* extend bindings such that names = vals *)
188 let extend env name_pred val_pred =
189   let test_env = Atomhash.copy env in
190   bind test_env (pred_val name_pred) (pred_val val_pred)
191
192 (* checks if test bindings are consistent with reference bindings *)
193 let bindings_valid test_env ref_env =
194   let test_names = dump_keys test_env in (*lookup every name in ref*)
195   let test_vals = dump_vals test_env in
196   let rec loop acc names values =
197     ( match ( names , values ) with
198     | ( [] , [] ) -> acc
199     | ( test_name::t1 , test_val::t2 ) ->
200       (try
201         let ref_val = Atomhash.find ref_env test_name in
202         if test_val = ref_val then loop (true::acc) t1 t2
203         else [false] (* value mismatch *)

```

```

204         with Not_found -> (* name not found *)
205         loop acc t1 t2 )
206     | ( _ , _ ) ->
207         let error_msg = "bindings_valid:_improper_input" in
208         failwith error_msg
209     )
210     in List.fold_left (fun x y -> x && y ) true (loop [] test_names
211         test_vals)
212
213 (* returns a list of matching state predicates *)
214 let matching_preds pred state =
215     let name = pred_name pred in
216     let match_name =
217         ( fun pred -> if name = (pred_name pred) then true else false) in
218     List.filter match_name state
219
220 (* partitions a conjunction into positive and negative predicates *)
221 let partition_to_predicates conj =
222     let ( pos_conj , neg_conj ) = partition_conjunction conj in
223     let pred_of = List.map conj_pred in
224     ( pred_of pos_conj , pred_of neg_conj )
225
226 (* checks if the goal is a subset of partial plan *)
227 let goal_test pplan goal =
228     let state = List.hd pplan in
229     let rec loop check_sum goal_preds =
230         ( match goal_preds with
231         | [] -> check_sum
232         | gp::remaining_goal_preds ->
233             let intersection = intersect [gp] state in
234             ( match intersection with
235             | [] -> [false]
236             | _ -> loop (true::check_sum) remaining_goal_preds
237             )
238     )
239     in List.fold_left (fun x y -> x && y ) true (loop [] goal)
240
241 (** level-three dependency **)
242
243 (* checks if the partial plan is the initial state *)
244 let search_exhausted pplan init_state =
245     goal_test pplan init_state (* TODO: is this valid for all cases?*)
246
247 (* outputs partitioned, grounded effects *)
248 let partition_to_grounded_effect op action =
249     let env = action_bindings action op in (* create bindings *)
250     let ( pos_preds , neg_preds ) = partition_to_predicates op.effect in
251     let pos_effs = List.map (ground_pred env) pos_preds in
252     let neg_effs = List.map (ground_pred env) neg_preds in
253     ( pos_effs , neg_effs )
254
255 (** level-four dependency **)
256
257 (* check if positive effects are the same as the negative effects *)
258 let complimentary_effects action op =

```

```

258 let ( pos_effs , neg_effs ) = partition_to_grounded_effect op action in
259 let intersection = intersect pos_effs neg_effs in
260 ( match intersection with
261   | [] -> false
262   | _ -> true
263 )
264
265 (* apply an action and return the successor state *)
266 let successor state opset action = (* act is a grounded pred list *)
267   let op = lookup_op opset (pred_name action) in
268   let ( pos_effs , neg_effs ) = partition_to_grounded_effect op action in
269   let rec remove_neg_effects s peffs neffs =
270     ( match neffs with
271     | [] -> action::peffs@s
272     | neg_eff::t -> (* check for repeating effects in strips *)
273       let matching_effects = matching_preds neg_eff pos_effs in
274       ( match matching_effects with
275       | [] ->
276         remove_neg_effects (remove neg_eff s) t peffs
277       | [pe] ->
278         remove_neg_effects (swap neg_eff pe s) t (remove pe peffs)
279       | _ ->
280         let error_msg =
281           "Predicates_may_only_appear_once_in_effect_declaration" in
282         failwith error_msg
283       )
284     )
285   in remove_neg_effects state pos_effs neg_effs
286
287 (** level-five dependency **)
288
289 (* attempts to unify state predicates with an operator's preconditions *)
290 let rec unify op ( pos_preds , neg_preds ) env state =
291   ( match pos_preds with
292   | [] ->
293     ( match neg_preds with
294     | [] -> [ground_op env op]
295     | np::remaining_np ->
296       let state_preds = matching_preds np state in
297       let rec unify_npreds all_sp =
298         ( match all_sp with
299         | [] ->
300           let error_msg = "no_matching_predicates_with_state" in
301           failwith error_msg
302         | sp::remaining_sp ->
303           let tenv = extend env np sp in
304           if ( bindings_valid tenv env ) then
305             unify op ( [] , remaining_np ) tenv state
306           else
307             []
308         ) in unify_npreds state_preds
309     )
310   | pp::remaning_pp ->
311     (* look at common state predicates *)
312     let state_preds = matching_preds pp state in

```

```

313     let rec unify_ppreds all_sp =
314         ( match all_sp with
315         | [] -> []
316         | sp::remaining_sp ->
317             let tenv = extend env pp sp in (* temp sub.*)
318             if ( bindings_valid tenv env ) then
319                 ( match remaning_pp with
320                 | [] -> (* check for complimentary effects *)
321                     let action = ground_op tenv op in
322                     if ( complimentary_effects action op ) then []
323                     else unify op ( remaning_pp , neg_preds ) tenv state
324                 | _ -> unify op ( remaning_pp , neg_preds ) tenv state
325                 )
326             else (* try unifying pp with another state pred *)
327                 unify_ppreds remaining_sp
328         ) in unify_ppreds state_preds
329     )
330
331     (** level-six dependency **)
332     (* return the first valid action in a state *)
333     let applicable_instance op state =
334         let ( pos_preds , neg_preds ) =
335             partition_to_predicates op.precondition in
336         let env = Atomhash.create 10 in
337         unify op ( pos_preds , neg_preds ) env state
338
339     (* accumulate action instances from every possible state permutation *)
340     let applicable_instances op state =
341         let max_permutations = List.length state in
342         let rec find_instances app_actions permutations s =
343             if permutations = max_permutations then
344                 app_actions
345             else
346                 let app_action = applicable_instance op s in
347                 let s1 = (List.tl s)@[List.hd s] in
348                 find_instances (app_action::app_actions) (permutations+1) s1
349         in List.flatten ( find_instances [] 0 state )
350
351     (* all applicable actions, includes backward moves, but no self loops *)
352     let applicable_actions state opset =
353         let rec next_op actions ops =
354             ( match ops with
355             | [] -> List.flatten actions
356             | op::remaining_ops ->
357                 (* list of all op's instances in the state *)
358                 let action_instances = applicable_instances op state in
359                 next_op (action_instances::actions) remaining_ops
360             )
361         in next_op [] opset
362
363     (* returns all available nodes not previously visited *)
364     let successors pplan opset explored =
365         let state = List.hd pplan in
366         let actions = applicable_actions state opset in
367         ( match actions with

```

```

368 | [] ->
369   let error_msg =
370     "no_applicable_actions_found_for_state" in
371   failwith error_msg
372 | _ ->
373   let all_succs =
374     filter_visited(
375       filter_duplicates(
376         List.map (successor state opset) actions)) explored in
377   let rec filter_applicable_pplans succs =
378     ( match succs with
379     | [] -> applicable_pplans
380     | act_succ::remaining_succs ->
381       let act = List.hd act_succ and succ = List.tl act_succ in
382       if succ = state then(* already filtered *)
383         filter_applicable_pplans remaining_succs
384       else (* back move *)
385         let prior_pplan = visited succ pplan in
386         ( match prior_pplan with
387         | [] ->
388           let new_pplan = succ::[act]::pplan in
389           filter (new_pplan::applicable_pplans) remaining_succs
390         | p::pp ->
391           filter (prior_pplan::applicable_pplans) remaining_succs
392         )
393     )
394   in filter [] all_succs
395 )
396
397 (* states are partial plans *)
398 let fsearch problem =
399   let { init = s0 ; goal = g ; ops = opset } = problem in
400   let rec dfs state explored =
401     if ( goal_test state g ) then List.rev state
402     else
403       let fringe = successors state opset explored in
404       ( match fringe with
405       | [] ->
406         if search_exhausted state s0 then
407           let error_msg = "Problem_has_no_solution" in
408             failwith error_msg
409         else
410           dfs (backtrack state) (state::explored)
411       | _ ->
412         let priority_queue = prioritize fringe in
413         let succ = List.hd priority_queue in
414         dfs succ (state::explored)
415       )
416   in dfs [s0] []
417
418 let solve problem =
419   fsearch problem

```

---

## compile.ml

```

1  (* compile.ml *)
2  open Ast
3  open Strips
4  open Util
5  (*
6   - push state preds on stack until goal is satisfied
7   - if all preds are on stack, pop all off and push action
8   - push state preds on stack until goal is satisfied
9   - if all preds are on stack, pop all off and push next action
10 *)
11
12 type instruction =
13   | Push_action of string
14   | Push_pred of string
15   | Pop_pred
16   | Goal
17
18 let string_of_instruction instruction =
19   ( match instruction with
20     | Push_action action -> sprintf "\nPush_action_%s" action
21     | Push_pred pred -> sprintf "\nPush_pred_%s" pred
22     | Pop_pred -> "\nPop_pred"
23     | Goal -> "\nGoal\n"
24   )
25
26 let print_bytecode instructions =
27   (string_of_syms (List.map string_of_instruction instructions))
28
29 let string_of_atom atom =
30   ( match atom with
31     | Atom_var a | Atom_gnd a -> a
32     | Atom_nil -> ""
33   )
34
35 (* check if name is a known action *)
36 let rec known_action name opset =
37   ( match opset with
38     | [] -> false
39     | op::remaining_ops ->
40       if op.name = name then true
41       else known_action name remaining_ops
42   )
43
44 let rec translate_params params =
45   sprintf "%s"
46     (string_of_syms (List.map string_of_atom params))
47
48 let translate_action pred =
49   ( match pred with
50     | Pred_gnd( name , params ) ->
51       let translated_params = translate_params params in
52       let action = sprintf "%s(%s)" name translated_params in
53       Push_action( action )

```

```

54     | _ ->
55         let error_msg = "error_translating_predicate" in
56         failwith error_msg
57     )
58
59 let translate_pred pred =
60     ( match pred with
61     | Pred_gnd( name , params ) ->
62         let predicate =
63             sprintf "%s(%s)"
64                 name
65                 (string_of_syms(List.map string_of_atom params))
66         in
67         Push_pred predicate
68
69     | _ ->
70         let error_msg = "error_translating_predicate" in
71         failwith error_msg
72     )
73
74 let pop_for num =
75     let rec loop acc count =
76         if 0 < count then
77             loop (Pop_pred::acc) (count - 1)
78         else
79             acc
80     in loop [] num
81
82 (* translate plan into bytecode *)
83 let translate_plan problem =
84     let { init = s0 ; goal = g ; ops = opset } = problem in
85     let plan_preds = List.flatten plan in
86     let num_preds = List.length s0 in
87     let rec recurse instructions preds =
88         ( match preds with
89         | [] -> List.rev instructions
90         | pred::remaining_preds ->
91             ( match pred with
92             | Pred_gnd( name , params ) ->
93                 if known_action name opset then
94                     let pred_pops = pop_for num_preds in
95                     let translated_action = translate_action pred in
96                     let transition = translated_action::pred_pops in
97                     recurse (transition@instructions) remaining_preds
98                 else
99                     ( match remaining_preds with
100                    | [] -> recurse (Goal::instructions) remaining_preds
101                    | _ ->
102                        let translated_pred = translate_pred pred in
103                        recurse (translated_pred::instructions) remaining_preds
104                    )
105             | _ ->
106                 let error_msg = "error_translating_predicate" in
107                 failwith error_msg
108         )

```

## lxiv Source Code

```
109     )
110   in recurse [] plan_preds
```

---

### execute.ml

```
1  open Compile
2
3  let rec execute_instructions instructions =
4    let stack = Array.make 1024 "" in
5    let rec exec fp sp pc =
6      ( match instructions.(pc) with
7        | Push_action action ->
8          stack.(sp) <- action;
9          let _ = Printf.printf "%s\n" (stack.(sp)) in
10         exec fp (sp + 1) (pc + 1)
11        | Push_pred predicate ->
12         stack.(sp) <- predicate;
13         exec fp (sp + 1) (pc + 1)
14        | Pop_pred ->
15         exec fp (sp - 1) (pc + 1)
16        | Goal -> ()
17      )
18    in exec 0 0 0
19
20 let execute_bytecode instructions =
21   execute_instructions (Array.of_list instructions)
```

---

### pddlyte.ml

```
1  open Strips
2  (*
3   compiler flags
4   -a: print ast
5   -p: print plan (action sequence)
6   -c: compile and print instructions
7   -e: execute instructions
8  *)
9
10 (* compiler flags *)
11 type flag = Path | Compile | Execute
12
13 (* decode the compiler flag *)
14 let decode_flag argv =
15   if Array.length argv > 1 then
16     List.assoc argv.(1)
17       [ ("-p", Path);
18         ("-c", Compile);
19         ("-e", Execute) ]
20   else Execute
21
22 (* open infile *)
```



```

23 let decode_infile argv =
24   if Array.length argv > 1 then
25     open_in argv.(2)
26   else
27     open_in argv.(1)
28
29 (* compile to path *)
30 let print_plan infile =
31   let lexbuf = Lexing.from_channel infile in
32   let env = Strips.make None in
33   let rec loop () =
34     let sexpr = Parser.parse Lexer.token lexbuf in
35     (match sexpr with
36      | None ->
37        let problem = env.problem in
38        let plan = Planner.solve problem in
39        Printf.printf "%s" (Planner.string_of_plan plan);
40        flush stdout; ()
41      | Some s ->
42        let ast = Ast.ast_of_sexpr s in
43        let _ = Strips.strips_of_ast env ast in
44        flush stdout;
45        loop ()
46     )
47   in loop ()
48
49 (* compile to bytecode *)
50 let compile_program infile =
51   let lexbuf = Lexing.from_channel infile in
52   let env = Strips.make None in
53   let rec loop () =
54     let sexpr = Parser.parse Lexer.token lexbuf in
55     ( match sexpr with
56      | None ->
57        let problem = env.problem in
58        let plan = Planner.solve problem in
59        let instructions = Compile.translate plan problem in
60        Printf.printf "%s" (Compile.print_bytecode instructions);
61        flush stdout; ()
62      | Some s ->
63        let ast = Ast.ast_of_sexpr s in
64        let _ = Strips.strips_of_ast env ast in
65        flush stdout;
66        loop ()
67     )
68   in loop ()
69
70 (* compile and execute bytecode *)
71 let execute_program infile =
72   let lexbuf = Lexing.from_channel infile in
73   let env = Strips.make None in
74   let rec loop () =
75     let sexpr = Parser.parse Lexer.token lexbuf in
76     ( match sexpr with
77      | None ->

```

```

78     let problem = env.problem in
79     let plan = Planner.solve problem in
80     let instructions = Compile.translate plan problem in
81     Execute.execute_bytecode instructions; ()
82   | Some s ->
83     let ast = Ast.ast_of_sexpr s in
84     let _ = Strips.strips_of_ast env ast in
85     flush stdout;
86     loop ()
87   )
88 in loop ()
89
90 (* process input *)
91 let run_program flag infile =
92   ( match flag with
93     | Path ->
94       ( try
95         print_plan infile
96         with (Failure f) ->
97           Printf.fprintf stderr "ERROR:_%s\n" f;
98           close_in infile
99       )
100    | Compile ->
101      ( try
102        compile_program infile
103        with (Failure f) ->
104          Printf.fprintf stderr "ERROR:_%s\n" f;
105          close_in infile
106      )
107    | Execute ->
108      ( try
109        execute_program infile
110        with (Failure f) ->
111          Printf.fprintf stderr "ERROR:_%s\n" f;
112          close_in infile
113      )
114    )
115
116 (* pddlyte top-level *)
117 let _ =
118   if (Array.length Sys.argv > 3 || Array.length Sys.argv < 2) then
119     Printf.fprintf stderr "USAGE:_%s_[flag]_[input_filename]\n" Sys.argv
120     .(0)
121   else
122     let flag = decode_flag Sys.argv in
123     let infile = decode_infile Sys.argv in
124     run_program flag infile

```

---

## 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] Nils J. Nilsson and Richard E. Fikes. Strips: A new approach to the application of theorem proving to problem solving. Technical report, Artificial Intelligence Group, Stanford Research Institute.
- [3] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw Hill.
- [4] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall.
- [5] David H.D Warren. An abstract prolog instruction set. Technical report, Artificial Intelligence Center, Stanford Research Institute.