PDDLyte

A Partial Implementation
of
The Planning Domain Definition Language

John Martin Jr.
jdm2213@columbia.edu

May 14, 2014

Contents

Introduction

PDDLYTE 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 PDDLYTE, 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 PDDLYTE 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.

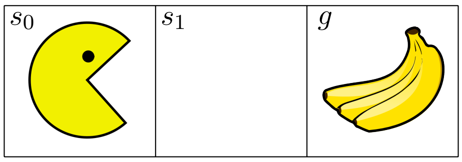


Figure 1: Pacman Domain

In the example program `pacman.pdly`, Pacman's intentions are made explicit with PDDLYTE. His initial spot on the grid is labled 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

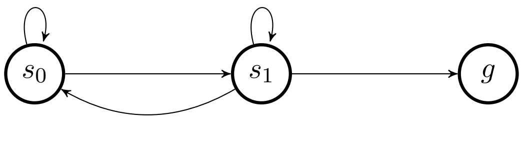


Figure 2: Pacman State-space Graph

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

pacman.pdly

```
(define (domain pman)
  (:predicates
   (adj ?s1-square ?s2-square)
   (at ?what-thing ?si-square)
   (fix ?what-thing ?si-square)
  )

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

(define (problem pman_prob)
  (:objects
   s0-square s1-square g-square
   pacman-thing bananas-thing
  )

  (:init
   (adj s0 s1) (adj s1 s0)
   (adj s1 g) (adj g s1)
   (fix bananas g)
   (at pacman s0)
  )

  (:goal
    (fix bananas g)
    (at pacman g)
  )
)
```

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

PDDLYTE 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 PDDLYTE 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 PDDLYTE expression. As a result, there's no precedence nor associativity to consider.

$\langle \textit{s-expression} \rangle ::= \langle \textit{atom} \rangle \mid \langle \textit{list} \rangle$

$\langle \textit{atom} \rangle ::= \epsilon \mid \langle \textit{symbol} \rangle$

$\langle \textit{list} \rangle ::= (\epsilon) \mid \langle \textit{s-expression} \rangle \cdot \langle \textit{list} \rangle$

Note: the empty list is a valid symbolic expression.

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

[1] 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.

⟨*symbol*⟩ ::= ⟨*char*⟩(⟨*char*⟩ | ⟨*digit*⟩)*

⟨*char*⟩ ::= ‘A’-‘Z’ | ‘a’-‘z’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘_’ | ‘<’ | ‘>’ | ‘!’ | ‘?’ | ‘:’

⟨*digit*⟩ ::= ‘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.

⟨*variable atom*⟩ ::= ‘?’⟨*symbol*⟩‘-’⟨*symbol*⟩ | ‘?’⟨*symbol*⟩

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.

⟨*grounded atom*⟩ ::= ⟨*symbol*⟩‘-’⟨*symbol*⟩ | ⟨*symbol*⟩

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 PDDLYTE.

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.

⟨*predicate*⟩ ::= (⟨*symbol*⟩ ⟨*variable atom*⟩ ⟨*variable atom*⟩)
 | (⟨*symbol*⟩ ⟨*grounded atom*⟩ ⟨*grounded atom*⟩)
 | (⟨*symbol*⟩ ⟨*variable atom*⟩)
 | (⟨*symbol*⟩ ⟨*grounded atom*⟩)

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.

⟨*conjunction*⟩ ::= ('`and`' (⟨*positive conjunction*⟩ | ⟨*negative conjunction*⟩)$^+$)
 | ⟨*negative conjunction*⟩ ::= ('`not`' ⟨*predicate*⟩)
 | ⟨*positive conjunction*⟩ ::= ⟨*predicate*⟩

Define

The two s-expressions that represent PDDLYTE programs are defined with the `define` predicate:

⟨*definition*⟩ ::= ('`define`' ⟨*domain*⟩ | ⟨*problem*⟩)

Domains

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

⟨*domain*⟩ ::= ('`domain`' ⟨*symbol*⟩)⟨*predicates*⟩⟨*operator*⟩$^+$

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.

⟨*predicates*⟩ ::= ('`:predicates`' ⟨*predicate*⟩$^+$)

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 [2] of a variable *operator* declaration. All operator names must be unique.

[2] The term *grounded* indicates parameters are bound to concrete values.

⟨*operator*⟩ ::= (‘`:action`’ ⟨*symbol*⟩ ⟨*parameters*⟩ ⟨*pre-condition*⟩ ⟨*effect*⟩)

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.

⟨*parameters*⟩ ::= (‘`:parameters`’ ⟨*variable atom*⟩$^+$)

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.

⟨*pre-condition*⟩ ::= (‘`:precondition`’ ⟨*conjunction*⟩)

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.

⟨*effect*⟩ ::= (‘`:effect`’ ⟨*conjunction*⟩)

Problems

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

⟨*problem*⟩ ::= (‘`problem`’ ⟨*symbol*⟩) ⟨*objects*⟩ ⟨*initial state*⟩ ⟨*goal state*⟩

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.

⟨*objects*⟩ ::= (‘`:objects`’ ⟨*grounded atom*⟩$^+$)

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[3].

⟨*initial state*⟩ ::= (‘`:init`’ ⟨*grounded predicate*⟩$^+$)

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.

⟨*goal state*⟩ ::= (‘`:goal`’ ⟨*grounded predicate*⟩$^+$)

[3] 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`

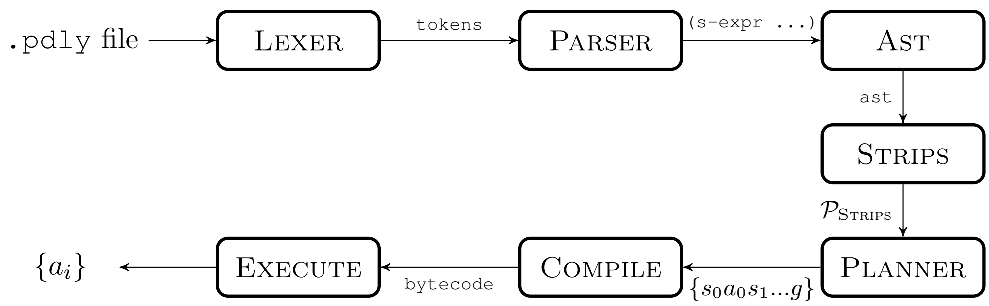
Table 1: Global Symbol Table Structure

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 PDDLYTE 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 $S = \{s_0a_0, s_1a_1, ...g\}$.

The PDDLYTE 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 $S = \{s_0 a_0, s_1 a_1, ...g\}$.

Algorithm 1 Prioritized, Depth-first Search

globals $\mathcal{P}_{\text{STRIPS}} = \{s_0, g, A\}$
procedure DFS($S = s_0, \nu = \{\}$)
    if $S = g$ **then return** S
    else
        $fringe \leftarrow$ successors(S, A, ν)//compute all successors in s
        if $fringe = \{\}$ **then**
            if state space exhausted **then return** error
            else
                DFS($S_{i-1}, S \cup \nu$)//backtrack
        else
            $priority\_queue \leftarrow$ prioritize $fringe$
            $S_{i+1} \leftarrow$ pop $priority\_queue$ // successor
            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

```
globals a, S
procedure UNIFY(P_a^+, P_a^-, σ = {})
    if P_a^+ = {} then
        if P_a^- = {} then return {a(σ')}
        else
            p_a^- ← pop P_a^-
            P_s ← predicates in S matching p_a^-
            for all p_s ∈ P_s do
                σ' ← extend( p_s, p_a^- ) //bind parameters in p_a^- based on values in p_s
                if σ' consistent with σ then
                    UNIFY({}, P_a^-, σ')
                else return {}
    else
        p_a^+ ← pop P_a^+
        P_s ← predicates in S matching p_a^+
        for all p_s ∈ P_s do
            σ' ← extend( p_s, p_a^+ ) //bind parameters in p_a^+ based on values in p_s
            if σ' consistent with σ then
                if P_a^+ \ p_a^+ = {} then//last state predicate
                    if complimentary effects (a , σ') then return {}
                    else UNIFY(P_a^+ \ p_a^+, P_a^-, σ')
                else UNIFY(P_a^+ \ p_a^+, P_a^-, σ')
            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 σ 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 σ_{p_s} for each state predicate $p_s \in P_s$ are extended such that the corresponding operator precondition σ_{p_a} is bound to their values. In the Pacman example, the state predicate

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

can be extended to the operator precondition

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

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

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 σ'. 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

PDDLYTE 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 it's 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 cliche': 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

```
commit a75f1e3aec5dcad23a2f5e425a419f8e88946c38
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 23:32:22 2014 -0400

    last commit

commit cbb9ee8f46635a681e813f9f8b59f21474599411
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 23:32:06 2014 -0400

    last commit

commit f17c73713072a99163b731a3e9eeda2de13642d7
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 23:31:35 2014 -0400

```

```
    removed ast prints

commit 4b6b148c758d8c8cad33c2723e41afd7f5ddc6db
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 22:07:13 2014 -0400

    fixed bug in unification algorithm: processing negative preconditions

commit a555266c0f09cf4c66e2a4c0bd9c3b65826ab56f
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 17:15:46 2014 -0400

    dependent upon util.ml

commit 6b6d8d10d6f0b6933c87eb57b77d75fd80d61ee2
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 17:15:03 2014 -0400

    fully supports scope and type checking

commit d9b485379d5270703b5f831b5515e9870fa62c1d
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed May 14 17:14:28 2014 -0400

    cleaned up test dependencies

commit 8a003af45a5d82c3276de87c0241418d72600cf5
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 21:11:14 2014 -0400

    updated symbol tokens to start with one character and contain zero or more alphanumeric characters therafter

commit 2f4d02cabf02c1637f2f209ba6a012d49e51b4e9
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:59:09 2014 -0400

    depends on util.ml for print

commit cbd83e89c4e5a0e080f6d1490a7c40bc62d54410
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:40:04 2014 -0400

    implemented explored list to backtrack more reliably

commit ed12e5ce622ba3a971360593eeff5ae21401c287
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:39:18 2014 -0400

    added print functions and list filtering functions

commit 15782fe8f83de91934d0eeb1150d8bd85e48ad71
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:38:06 2014 -0400

```

```
    updated comments

commit 4c80d4f40ac2b0874f1d87c101ddfd53e9cb2e48
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:37:45 2014 -0400

    added comments

commit edf51362bf27390c961a3117b8833a04f43f5b2d
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 11 12:36:59 2014 -0400

    updated print functions

commit 51c5b491a463938ed563dc9980c034187921dbfa
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 10 02:20:05 2014 -0400

    updated makefile

commit 58a006d57cd57db7b5fe8e776d83e0a121be105c
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 10 02:18:55 2014 -0400

    added comments and arranged according to dependency

commit 13ae253107db4d102c83bfb53fd0299e03b08a3a
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 10 02:16:22 2014 -0400

    updated error prints

commit 6e74c2dc2a14b47cff86756cceef85cd291dc0a3
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 10 02:14:32 2014 -0400

    initial commit

commit f56ca4f4fc75e1f94ac1c11165b23106330a9350
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 10 02:09:18 2014 -0400

    filtering duplicate successors, randomly permuting the priority queue, replace plan_to with visited

commit 1406f4245b7920bb1407b632f10a9d186e0042d9
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:36:49 2014 -0400

    removed test functions

commit 40d20bb7fa70a28c1d4bf90b36d89493ce1fb33f
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:34:25 2014 -0400

```

```
    tested with pacman.pdly

commit 651240204314dffb60925e0304dc838de754d844
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:33:23 2014 -0400

    removed test functions and updated interface

commit bf2aedc2c7d33812b674cb686474936c569d4d5d
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:32:45 2014 -0400

    removed test functions and updated interface

commit e443764f47b2dcd93ed24a9a1444f99781341e5a
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:31:58 2014 -0400

    tested with pacman.pdly

commit 1fc26f8d667e43dba2400849535d43b2996324b7
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:29:37 2014 -0400

    fixed print

commit df85e092646785b1037b5eea87571c1d749c55b8
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 21:23:23 2014 -0400

    cleaned up a bit and updated interface

commit e149413bc2d93339983ccde4e32dc1d33a1112b4
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 08:56:47 2014 -0400

    removed module tests

commit 956e257091a68a3b86bfd5511a88cf877fdcc41e
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 08:55:15 2014 -0400

    clean up

commit 703013f6b473e4c7b8d6c0d77a9466d625a39665
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 08:54:47 2014 -0400

    supports compiler flags

commit b7e83cb96a94ed6b2a048cdb7c7b491a818e7ccd
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 08:52:55 2014 -0400

    Initial commit of runtime environment
```

```

commit df40a16399d04a3002b6c55e4ee2f110cb7c6740
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu May 8 08:51:25 2014 -0400

    initial commit of compiler to IR

commit 9ff2ebe119ecbd3c1dec64363b7fe3eaca30b726
Author: John Martin <jdmartin86@gmail.com>
Date:   Tue May 6 07:54:55 2014 -0400

    Updated planner to use partial plans of state-action stripes

commit f96610d4649c773095a43b5f4e6fee9291d113e0
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun May 4 22:07:01 2014 -0400

    re-wrote the planner to represent nodes as partial plans ... everything seems to work -- no big deal

commit 75ab3a89a0f8f190f283888942a5c5109f9ad564
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat May 3 02:00:21 2014 -0400

    new planner structure

commit 9b86f48d01a32d09d705ae0fb2a4241f4a145b59
Author: John Martin <jdmartin86@gmail.com>
Date:   Fri May 2 22:55:20 2014 -0400

    intermediate save

commit bb247d43073e49430ff8ee2982277f7e74e79ad3
Author: John Martin <jdmartin86@gmail.com>
Date:   Tue Apr 29 08:02:47 2014 -0400

    pacman works!

commit aa131f563d4aad8efccc21a57a79df11575bf3fb
Author: John Martin <jdmartin86@gmail.com>
Date:   Mon Apr 28 07:23:49 2014 -0400

    applicative actions and successor function partially tested

commit 716e78aa20719cf7313155a6a8fe995d4c34e29d
Merge: 69693d9 71369cc
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 20:32:31 2014 -0400

    deleted sast

commit 71369cc9bce548e8f852a61588bc65fd4bb4aa16
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 20:50:27 2014 -0400

```

```
    Delete sast.mli

commit cc94b9f26cd471c5970a5b7da35c88e61ac39c3c
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 20:50:17 2014 -0400

    Delete sast.ml

commit 69693d9ffb4fc040644fdf9664670141f18a084c
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 20:27:15 2014 -0400

    buildable planner outline

commit 115b9208b35e7f972f429387f5001ee946471bea
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 19:59:34 2014 -0400

    supports strips test

commit dfee6bd0c5a9ee8f7b6e13c39c4dea486210914f
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 19:57:46 2014 -0400

    supports strips problem

commit 488b8b35a28975298dc16a888bb73639e496ad6b
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 19:57:00 2014 -0400

    re-structured and tested translation to strips problem

commit ab5d1ebb0b0025617dc7b03b9c3f6e289e284732
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 12:49:56 2014 -0400

    cleaned up warnings and removed expr_sym type

commit 1780319eed4080cad1ba31ecd04f5e8ec157517f
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 12:09:40 2014 -0400

    initial commit of top-level compiler

commit f77ec710c13f0038877286c54e97d9aff05f4e31
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 12:08:58 2014 -0400

    updated for strips and pddlyte modules

commit 26df45f4fce9886ec17a9f5912cc1ab2e8e5aae2
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 23 12:07:08 2014 -0400

```

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

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

    removed symt.ml and symt.mli

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

    full support added for symbol table translation

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

    partial support added for action translation

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

    full support for predicate translation

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

    partial symbol table support added

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

    added support for symbol table

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

    initial commit

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

    added code outlines for successor function and forward search

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

```

```
    format updates

commit 534053b5817916241d1f39fa31347400dd27357f
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Apr 19 22:32:13 2014 -0400

    symbols support uppercase letters and question marks

commit 767b9363291eb6a8c4d1f69d49d484a66cc0b69b
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Apr 19 22:30:15 2014 -0400

    small updates

commit 44d2b560335b06fa02a921f89dd16c4b8b57d360
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Apr 19 22:26:14 2014 -0400

    added support for the planner

commit 9ada41e897444e4aa605ced4da5a7e45e7aa5924
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Apr 19 22:25:34 2014 -0400

    Added nil types for predicates, conjunctions, and atoms -- to represent nil returns

commit 4c44604d9342660a85082d9d55f1828209ef12c7
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Apr 19 22:23:41 2014 -0400

    initial commit -- applicative actions only support positive preconditions

commit c8b377a86501d395513da7b818e9acfdf316cdf0
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 16 10:46:07 2014 -0400

    nearly full support added: actions, objects, inits, goals all parse

commit e1f7a747ba130b01397594b2a8723d3b89da8f39
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 16 08:41:00 2014 -0400

    support fror predicate lists, domains and problem definitions, and partial support for actions

commit 7188dfb1cfea9c26900e2ee97aaaf9aa47c47f08
Author: John Martin <jdmartin86@gmail.com>
Date:   Tue Apr 15 15:19:22 2014 -0400

    reformulated tree types -- still in progress

commit af7d5e66a0e80d4c46752a78d9c044e4c6c759a5
Author: John Martin <jdmartin86@gmail.com>
```

```
Date:   Sun Apr 6 14:50:44 2014 -0400

    supports sast

commit 62f0536cf766cbee83422a982d6a386ce2246dc9
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun Apr 6 12:37:25 2014 -0400

    Initial commit

commit 31d4fd5c2e6102639ff337c9988f1f4a052a4abe
Author: John Martin <jdmartin86@gmail.com>
Date:   Sun Apr 6 00:05:18 2014 -0400

    Partially tested

commit 94b6a45c7412baa75a45d1cb0543e353e2c1bb60
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu Apr 3 23:42:28 2014 -0400

    Added planner nodes and procedures

commit f5517abe43f207741f61798f07c637b6adefd0c3
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Apr 2 00:16:16 2014 -0400

    supports define expressions

commit ec16acd1b0277dfd64627703f02c001e6efc73eb
Author: John Martin <jdmartin86@gmail.com>
Date:   Mon Mar 31 19:58:54 2014 -0400

    Initial commit of abstract syntax tree

commit 838a5a4d23c00642b697c29404da0eeb19303aa5
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Mar 15 23:12:54 2014 -0400

    Initial commit of Makefile

commit 59839146eb0825079a2ce3d03062c5a907c701e1
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Mar 15 23:11:27 2014 -0400

    Initial commit of s-expressions

commit 4e3b4f51aa23396a05cf26ba2cef53abf7b019c2
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Mar 15 23:10:45 2014 -0400

    Initial commit of parser

commit b0e95764af360bf7b94a10473c26265b13c3f91d
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Mar 15 23:10:09 2014 -0400
```

```

    Initial commit of lexer

commit 54b1badff976980c3ead7b01c567c033b349a4a5
Author: John Martin <jdmartin86@gmail.com>
Date:   Sat Mar 15 23:08:57 2014 -0400

    Initial commit of abstract syntax tree

commit dbf275504d12d8f9e5e06394c52ffdcfee718b65
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Sat Mar 15 09:33:24 2014 -0400

    Initial commit -- finial draft

commit f59aada1fd8e1df3609d696c2494db5964c6a72e
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Thu Feb 13 13:29:52 2014 -0500

    final draft

commit a599292d30582e683680a78aa96524fb634b0087
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Mon Feb 10 23:13:26 2014 -0500

    near-final draft

commit 42da7b1138d950741c6bf17710733239ecc97d19
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Sun Feb 9 19:10:40 2014 -0500

    partially-completed draft including integer support

commit e1484e27d61fc10618df7678be02b2c2deb8ace6
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Sun Feb 9 01:50:38 2014 -0500

    included example, pacman program

commit 09135655994b94478a0268666e7e9ce46cabfd74
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu Feb 6 00:32:12 2014 -0500

    updated solution

commit 85bf75121dcabb5788cf5928e749b120f0a4c6ff
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu Feb 6 00:29:13 2014 -0500

    fixed errors in goal

commit dfb2d8425b7370ce30100786b9396672be8d9af0
Merge: 6f5d0f6 45a8b2d
Author: John Martin <jdmartin86@gmail.com>
Date:   Thu Feb 6 00:04:18 2014 -0500
```

```

    Merge remote-tracking branch 'upstream/master'

commit 45a8b2db8dff6e75786efcd0bd2d4a006a83b73e
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Sun Feb 9 01:16:04 2014 -0500

    partial draft

commit 6f5d0f68805368aab1723f771ed48c6f97c9e56d
Author: John Martin <jdmartin86@gmail.com>
Date:   Wed Feb 5 23:54:06 2014 -0500

    initial commit

commit 6b3155a9d2862256f8a09f2ada411941ccd872a2
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Tue Feb 4 21:23:28 2014 -0500

    Initial commit of rough drafts

commit 480529afa7b5dc2958cf15e5a53ebc19ed16fce7
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Mon Feb 3 18:14:48 2014 -0500

    commiting rough draft

commit 6f3606e75e0a16ab25acc3045e2a01602297016a
Author: jdmartin86 <jdmartin86@gmail.com>
Date:   Mon Feb 3 18:14:20 2014 -0500

    initial commit of reference database

commit 142bece7534face487a14704d5a161d26b5306a7
Author: John Martin <jdmartin86@gmail.com>
Date:   Mon Feb 3 18:06:01 2014 -0500

    initial commit of blank document files

commit 97872d3623def0dd0f4e9f27ac1f5dba917a421f
Author: John Martin <jdmartin86@gmail.com>
Date:   Mon Feb 3 17:34:09 2014 -0500

    initial commit of directory structure

commit a93a9995c37c92bb3c65e8534c3bf139ac4f2c5d
Author: John Martin <jdmartin86@gmail.com>
Date:   Mon Feb 3 17:30:58 2014 -0500

    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

```
(*
 * sexpr.mli
 *
 *      S-expressions.
 *
 *)


(* Type of atomic expressions. *)
type atom =
    | Atom_unit
    | Atom_int  of int
    | Atom_sym   of string


(* Type of all S-expressions. *)
type expr =
    | Expr_atom of atom
    | Expr_list of expr list


(* Convert an S-expression to a string.
   This version makes the structure of the S-expression explicit. *)
val string_of_expr : expr -> string


(* Convert an S-expression to a string.
   This version prints the S-expression like a Scheme expression.

   THIS MAY NOT BE NECESSARY -- JM
   *)
val string_of_expr2 : expr -> string
```

ast.mli

```
(* the abstract syntax tree *)

(* symbols *)
type sym = string

type atom =
  | Atom_var of sym (* ?symbol *)
  | Atom_gnd of sym (* symbol *)
```

```
  | Atom_nil

type predicate =
  | Pred_var of sym * atom list (* (predname vatom v/gatom?) *)
  | Pred_gnd of sym * atom list (* (predname gatom gatom?) *)
  | Pred_nil

type conjunction =
  | Conj_and of conjunction list
  | Conj_neg of predicate
  | Conj_pos of predicate
  | Conj_nil

type action =
{
  name         : sym;
  parameters   : atom list;
  precondition : conjunction;
  effect       : conjunction
}

type expr =
  | Expr_domain     of sym * expr list (* (define ( domain pman ) ... )*)
  | Expr_problem    of sym * expr list (* (define ( problem prob ) ...)*)
  | Expr_predicates of predicate list  (* :predicates body *)
  | Expr_init       of predicate list  (* :init body *)
  | Expr_goal       of predicate list  (* :goal body *)
  | Expr_action     of action          (* :action ... *)
  | Expr_objects    of atom list       (* :objects body *)
  | Expr_unit                          (* () *)

val string_of_syms : string list -> string
val sym_of_atom : Sexpr.atom -> string
val astatom_of_atomsym : Sexpr.atom -> atom
val astatom_of_sexpr : Sexpr.expr -> atom
val pred_of_sexpr : Sexpr.expr -> predicate
val params_of_sexpr : Sexpr.expr list -> atom list
val conj_of_sexpr : Sexpr.expr -> conjunction
val action_of_sexpr : Sexpr.expr list -> action
val ast_of_sexpr : Sexpr.expr -> expr
val string_of_atom : atom -> sym
val string_of_pred : predicate -> string
val string_of_conj : conjunction -> string
val string_of_params : atom list -> string
val string_of_precond : conjunction -> string
val string_of_effect : conjunction -> string
val string_of_action : action -> string
val string_of_ast : expr -> string
```

util.mli

```
(* util.mli *)

```

```
val sprintf : ('a, unit, string) format -> 'a
val printf : ('a, out_channel, unit) format -> 'a
val spaces : int -> string
val string_of_syms : string list -> string
val permutation : 'a list -> 'a list
val filter_duplicates : 'a list -> 'a list
val heads : 'a list list -> 'a list
```

strips.mli

```
(* symbol table interface *)

type sym = string

(* problem table for planner *)
type strips_problem = (* TODO: make these names consistent with others *)
    {
      mutable init : Ast.predicate list;
      mutable goal : Ast.predicate list;
      mutable ops  : Ast.action list    ;
    }

(* environment *)
type env =
    {
      parent: env option;
      bindings: (sym, sym) Hashtbl.t;
      mutable problem: strips_problem
    }

val make : env option -> env

(* convert an ast expression into a strips problem *)
val strips_of_ast : env -> Ast.expr -> unit
```

planner.mli

```
(* planner interface *)

module Atomhash :
  sig
    type key = Ast.atom
    type 'a t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val reset : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
```

```
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val length : 'a t -> int
    val stats : 'a t -> Hashtbl.statistics
  end

val intersect : 'a list -> 'a list -> 'a list
val backtrack : 'a list -> 'a list
val prioritize : 'a list list -> 'a list list
val remove : 'a -> 'a list -> 'a list
val lookup_op : Ast.action list -> Ast.sym -> Ast.action
val param_name : Ast.atom -> Ast.atom
val partition_conjunction :
  Ast.conjunction -> Ast.conjunction list * Ast.conjunction list
val conj_pred : Ast.conjunction -> Ast.predicate
val dump_keys : 'a Atomhash.t -> Atomhash.key list
val dump_vals : 'a Atomhash.t -> 'a list
val pred_name : Ast.predicate -> Ast.sym
val pred_val : Ast.predicate -> Ast.atom list
val bind : 'a Atomhash.t -> Atomhash.key list -> 'a list -> 'a Atomhash.t
val visited : Ast.predicate list -> Ast.predicate list list ->
  Ast.predicate list list
val swap : 'a -> 'a -> 'a list -> 'a list
val ground_pred : Ast.atom Atomhash.t -> Ast.predicate -> Ast.predicate
val ground_op : Ast.atom Atomhash.t -> Ast.action -> Ast.predicate
val action_bindings : Ast.predicate -> Ast.action -> Ast.atom Atomhash.t
val extend :
  Ast.atom Atomhash.t ->
  Ast.predicate -> Ast.predicate -> Ast.atom Atomhash.t
val bindings_valid : 'a Atomhash.t -> 'a Atomhash.t -> bool
val matching_preds :
  Ast.predicate -> Ast.predicate list -> Ast.predicate list
val partition_to_predicates :
  Ast.conjunction -> Ast.predicate list * Ast.predicate list
val goal_test : 'a list list -> 'a list -> bool
val search_exhausted : 'a list list -> 'a list -> bool
val partition_to_grounded_effect :
  Ast.action -> Ast.predicate -> Ast.predicate list * Ast.predicate list
val complimentary_effects : Ast.predicate -> Ast.action -> bool
val successor :
  Ast.predicate list ->
  Ast.action list -> Ast.predicate -> Ast.predicate list
val unify :
  Ast.action ->
  Ast.predicate list * Ast.predicate list ->
  Ast.atom Atomhash.t -> Ast.predicate list -> Ast.predicate list
val applicable_instance :
  Ast.action -> Ast.predicate list -> Ast.predicate list
val applicable_instances :
  Ast.action -> Ast.predicate list -> Ast.predicate list
val applicable_actions :
  Ast.predicate list -> Ast.action list -> Ast.predicate list
val successors :
```

```
  Ast.predicate list list ->  Ast.action list ->
  Ast.predicate list list list -> Ast.predicate list list list
val fsearch : Strips.strips_problem -> Ast.predicate list list
val solve : Strips.strips_problem -> Ast.predicate list list
val string_of_plan : Ast.predicate list list -> string
```

compile.mli

```
(* compile.mli *)

type instruction =
  | Push_action of string
  | Push_pred of string
  | Pop_pred
  | Goal
(* TODO: clean up! *)
val string_of_instruction : instruction -> string
val print_bytecode : instruction list -> string
val known_action : Ast.sym -> Ast.action list -> bool
val translate_params : Ast.atom list -> string
val translate_action : Ast.predicate -> instruction
val translate_pred : Ast.predicate -> instruction
val pop_for : int -> instruction list
val translate :
  Ast.predicate list list -> Strips.strips_problem -> instruction list
```

execute.mli

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

lexer.mll

```
(* lexer.mll
 *
 * This file converts program text into tokens
 *)

{ open Parser }

let wspc = [' ' '\t' '\n']
let chrs = ['A'-'Z' 'a'-'z' '+' '-' '*' '/' '=' '<' '>' '!' ':' '_' '?']
let digt = ['0'-'9']
let symb = chrs(chrs | digt)*

rule token = parse
  | wspc                { token lexbuf }
  | ';'                 { comment lexbuf }
```

```
  | '('                  { TOK_LPAR }
  | ')'                  { TOK_RPAR }
  | "NIL"                { TOK_UNIT }
  | (digt)+ as nm        { TOK_INT(int_of_string nm)} (* unused *)
  | symb     as sm       { TOK_SYM(sm)  }
  | eof                  { TOK_EOF }
  | _ as char { failwith ("illegal␣character␣" ^ Char.escaped char) }
  and comment = parse
  | '\n'         { token lexbuf }         (* commentary ends with lines *)
  | _            { comment lexbuf }       (* ignore other characters *)
```

parser.mly

```
/* parser
 *
 * this translates a sequence of tokens into s-expressions.
 * each call to the parser returns an s-expression. at the
 * end of the file the parser will return None; hence the
 * program returns option.
 */

%{ open Sexpr %}

/* token declarations */
%token TOK_LPAR TOK_RPAR
%token          TOK_UNIT
%token <int>    TOK_INT
%token <string> TOK_SYM
%token          TOK_EOF

/* parse the five non-terminals of the syntax */
%start parse
%type <Sexpr.expr option> parse
%type <Sexpr.expr>        sexpr
%type <Sexpr.atom>        atom
%type <Sexpr.expr list>   slist
%type <Sexpr.expr list>   sexpr_list

%%

/* rules */

parse:
/* an s-expression, or none if eof is encountered*/
 | TOK_EOF       { None     }
 | sexpr         { Some $1 }

sexpr:
/* an s-expresion, an atom or list of s-expressions */
 | atom          { Expr_atom $1 }
 | slist         { Expr_list $1 }

atom:
```

```
/* an atom, which can be a unit, int or string. */
 | TOK_UNIT      { Atom_unit     }
 | TOK_INT       { Atom_int   $1 }
 | TOK_SYM       { Atom_sym   $1 }

slist:
/* a list of s-expressions, with parentheses */
 | TOK_LPAR sexpr_list TOK_RPAR { List.rev $2 }

sexpr_list:
/* the list contents of s-expressions, sans parentheses. */
 | /* nothing */     { [ ] }       /* empty list */
 | sexpr_list sexpr { $2 :: $1 } /* sexpr list */


%%
```

sexpr.ml

```
(*
 * sexpr.ml
 *
 *      S-expressions.
 *
 *)

type atom =
  | Atom_unit
  | Atom_int   of int
  | Atom_sym   of string

type expr =
  | Expr_atom of atom
  | Expr_list of expr list


(* Convert an atom into a string. *)
let string_of_atom a =
  match a with
    | Atom_unit    -> "NILL"
    | Atom_int   i -> string_of_int i
    | Atom_sym   s -> s


(* Return a string of n spaces. *)
let spaces n = String.make n ' '

let string_of_expr sx =
  let rec iter_string_of_expr sx indent =
    begin
      match sx with
        | Expr_atom a ->
          let s = string_of_atom a in
```

```
          Printf.sprintf "Expr_atom[%s]" s
        | Expr_list slist ->
          "\n"
          ^ (spaces indent)
          ^ "Expr_list[␣"
          ^ (iter_string_of_expr_list slist (indent + 2))
          ^ "␣]"
    end
  and iter_string_of_expr_list slist indent =
    begin
      match slist with
        | [] -> ""
        | [s] -> iter_string_of_expr s indent
        | h :: t ->
          (iter_string_of_expr h indent)
          ^ "␣"
          ^ (iter_string_of_expr_list t indent)
    end
  in
  iter_string_of_expr sx 0


let string_of_expr2 sx =
  let rec iter_string_of_expr sx indent =
    begin
      match sx with
        | Expr_atom a ->
          let s = string_of_atom a in
          Printf.sprintf "%s" s
        | Expr_list slist ->
          "\n"
          ^ (spaces indent)
          ^ "("
          ^ (iter_string_of_expr_list slist (indent + 2))
          ^ ")"
    end
  and iter_string_of_expr_list slist indent =
    begin
      match slist with
        | [] -> ""
        | [s] -> iter_string_of_expr s indent
        | h :: t ->
          (iter_string_of_expr h indent)
          ^ "␣"
          ^ (iter_string_of_expr_list t indent)
    end
  in
  iter_string_of_expr sx 0
```

ast.ml

```
(* ast.ml *)
open Sexpr
```

```
open Util

type sym = string

type atom =
  | Atom_var of sym (* ?symbol *)
  | Atom_gnd of sym (* symbol *)
  | Atom_nil

type predicate =
  | Pred_var of sym * atom list (* (predname vatom v/gatom?) *)
  | Pred_gnd of sym * atom list (* (predname gatom gatom?) *)
  | Pred_nil

type conjunction =
  | Conj_and of conjunction list
  | Conj_neg of predicate
  | Conj_pos of predicate
  | Conj_nil

type action =
{
  name         : sym;
  parameters   : atom list;
  precondition : conjunction;
  effect       : conjunction
}

type expr =
  | Expr_domain     of sym * expr list (* (define ( domain pman ) ... )*)
  | Expr_problem    of sym * expr list (* (define ( problem prob ) ...)*)
  | Expr_predicates of predicate list  (* :predicates body *)
  | Expr_init       of predicate list  (* :init body *)
  | Expr_goal       of predicate list  (* :goal body *)
  | Expr_action     of action          (* :action ... *)
  | Expr_objects    of atom list       (* :objects body *)
  | Expr_unit                          (* () *)

let sprintf  = Printf.sprintf

let spaces n = String.make n ' '

(* string from string list *)
let rec string_of_syms sym_lst =
  (match sym_lst with
    | []   -> ""
    | [s] -> s
    | h::t -> h ^ " " ^ (string_of_syms t)
  )

(* sexpr.atom -> string *)
let sym_of_atom a =
  ( match a with
    | Atom_unit -> "" (* shouldn't be used *)
    | Atom_sym s -> s
```

```
  | Atom_int _ ->
   let error_msg = "improper␣symbol␣conversion" in
   failwith error_msg
 )

(* sexpr.atom -> ast.atom *)
let astatom_of_atomsym atom_sym =
 ( match atom_sym with
  | Atom_sym s ->
   let prefix = String.get s 0 in
   ( match prefix with
    | '?' -> Atom_var s
    | _ -> Atom_gnd s
   )
  | _ -> failwith "only␣accepts␣type␣atom␣sym"
 )

(* sexpr.expr -> ast.atom *)
let astatom_of_sexpr sx =
 ( match sx with
  | Expr_atom ( Atom_sym s ) ->
   let prefix = String.get s 0 in
   ( match prefix with
    | '?' -> Atom_var s
    | _ -> Atom_gnd s
   )
  | _ -> failwith "only␣accepts␣type␣atom␣sym"
 )

(* TODO: account for nil cases *)
(* this will need to be recursive for nested calls *)
(* sexpr.expr -> ast.predicate *)
let pred_of_sexpr sx =
( match sx with
 | Expr_list l ->
  ( match l with
   | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ; Expr_atom p2 ] ->
    let a1 = astatom_of_atomsym p1 and
      a2 = astatom_of_atomsym p2 in
    ( match a1 with
     | Atom_var _ ->
      Pred_var( name , [a1 ; a2] )
     | Atom_gnd _ ->
      ( match a2 with
       | Atom_var _ ->
        Pred_var( name , [a1 ; a2] )
       | Atom_gnd _ ->
        Pred_gnd( name , [a1 ; a2] )
       | Atom_nil ->
        let error_msg = "unrecognized␣predicate" in
        failwith error_msg
      )
     | Atom_nil ->
      let error_msg = "unrecognized␣predicate" in
      failwith error_msg
```

```
    )
   | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ] ->
    let a1 = astatom_of_atomsym p1 in
    ( match a1 with
     | Atom_var _ ->
      Pred_var( name , [a1] )
     | _ ->
      Pred_gnd( name , [a1] )
    )
   | _ ->
    let error_msg = "unrecognized␣predicate" in
    failwith error_msg
  )
 | _ -> failwith "unrecognized␣conjunction␣structure"
)

(* sexpr.expr list -> atom list *)
let params_of_sexpr sx =
 ( match sx with
  | Expr_atom _ :: _ ->
   List.map astatom_of_sexpr sx
  | _ -> failwith "syntax␣error:␣unrecognized␣parameter"
 )

(* sexpr.expr -> conjunction *)
let rec conj_of_sexpr sx =
 ( match sx with
  | Expr_list l ->
   ( match l with
    | [ Expr_atom( Atom_sym name ) ; Expr_atom p1 ; Expr_atom p2 ] ->
     let a1 = astatom_of_atomsym p1
     and a2 = astatom_of_atomsym p2 in
     ( match a1 with
      | Atom_var _ ->
       Conj_pos(Pred_var( name , [a1 ; a2] ))
      | Atom_gnd _ ->
       ( match a2 with
        | Atom_var _ ->
         Conj_pos(Pred_var( name , [a1 ; a2] ))
        | Atom_gnd _ ->
         Conj_pos(Pred_gnd( name , [a1 ; a2] ))
        | Atom_nil ->
         let error_msg = "unrecognized␣predicate" in
         failwith error_msg
       )
      | Atom_nil ->
       let error_msg = "unrecognized␣predicate" in
       failwith error_msg
     )
    | [ Expr_atom ( Atom_sym name ) ; Expr_atom p1 ] ->
     let a1 = astatom_of_atomsym p1 in
     ( match a1 with
      | Atom_var _ ->
       Conj_pos(Pred_var( name , [a1] ))
      | _ ->
```

```
              Conj_pos(Pred_gnd( name , [a1] ))
            )
          | Expr_atom ( Atom_sym "and" ) :: body -> (* body is list *)
            Conj_and ( List.map conj_of_sexpr body )
          | [ Expr_atom ( Atom_sym "not" ) ; p1 ] ->
            Conj_neg ( pred_of_sexpr p1 ) (*TODO: nesting ands *)
          | _ ->
            let error_msg = "invalid␣conjunction" in
            failwith error_msg
        )
      | _ ->
        let error_msg = "invalid␣conjunction" in
        failwith error_msg
    )

(* sexpr.expr list -> ast.action *)
let action_of_sexpr sx =
  ( match sx with
    | Expr_atom _ :: (* :action *)
        Expr_atom ( Atom_sym name ) ::
        Expr_atom ( Atom_sym ":parameters" ) :: Expr_list params ::
        Expr_atom ( Atom_sym ":precondition" ) :: Expr_list precond ::
        Expr_atom ( Atom_sym ":effect" ) :: Expr_list effect :: [] ->
      {
        name = name;
        parameters = params_of_sexpr params;
        precondition = conj_of_sexpr (Expr_list(precond));
        effect = conj_of_sexpr (Expr_list(effect))
      }
    | _ ->
      let error_msg = "unrecognized␣action␣structure" in
      failwith error_msg
  )

let rec ast_of_sexpr sx =
  (match sx with
    | Expr_atom a ->
      let error_msg = "unrecognized␣s-expression" in
      failwith error_msg
    | Expr_list l ->
      ( match l with
        | Expr_atom ( Atom_sym ":predicates" ) :: body ->
          Expr_predicates( List.map pred_of_sexpr body )
        | Expr_atom ( Atom_sym ":action" ) :: body ->
          Expr_action( action_of_sexpr l )
        | Expr_atom ( Atom_sym ":objects" ) :: body ->
          Expr_objects( List.map astatom_of_sexpr body )
        | Expr_atom ( Atom_sym ":init" ) :: body ->
          Expr_init( List.map pred_of_sexpr body )
        | Expr_atom ( Atom_sym ":goal" ) :: body ->
          Expr_goal( List.map pred_of_sexpr body )
        | Expr_atom ( Atom_sym "define" ) ::
            Expr_list l :: body -> (* body *)
          ( match l with
            | [Expr_atom ( Atom_sym "domain" ) ;
```

```
               Expr_atom ( Atom_sym name )] ->
              Expr_domain( name , List.map ast_of_sexpr body )
            | [Expr_atom ( Atom_sym "problem" ) ;
               Expr_atom ( Atom_sym name )] ->
              Expr_problem( name , List.map ast_of_sexpr body )
            | _ ->
              let error_msg = "unrecognized␣s-expression" in
              failwith error_msg
          )
        | _ ->
          let error_msg = "unrecognized␣s-expression" in
          failwith error_msg
      )
  )


let string_of_atom atom =
  ( match atom with
    | Atom_var a -> a
    | Atom_gnd a -> a
    | Atom_nil -> "NIL"
  )

let string_of_pred pred =
  ( match pred with
    | Pred_var( name , params ) ->
      sprintf "PRED_VAR(␣%s␣,␣%s␣)\n"
        name
        (string_of_syms(List.map string_of_atom params))
    | Pred_gnd( name , params ) ->
      sprintf "PRED_GND(␣%s␣,␣%s␣)\n"
        name
        (string_of_syms(List.map string_of_atom params))
    | Pred_nil ->
      "PRED_NIL()\n"
  )

let rec string_of_conj conj =
  let recurse = string_of_conj in
  ( match conj with
    | Conj_and c ->
      sprintf "CONJ_AND(%s)"
      (string_of_syms (List.map recurse c))
    | Conj_pos c ->
      sprintf "CONJ_POS(%s)"
        (string_of_pred c)
    | Conj_neg c ->
      sprintf "CONJ_NEG(%s)"
        (string_of_pred c)
    | Conj_nil -> ""
  )

let string_of_params params =
  sprintf "PARAMETERS(␣%s␣)"
  (string_of_syms (List.map string_of_atom params))
```

```

let string_of_precond precond =
  sprintf "PRECONDITION(␣%s␣)"
  (string_of_conj precond)

let string_of_effect effect =
  sprintf "EFFECT(␣%s␣)"
  (string_of_conj effect)

let string_of_action act =
  act.name ^ "\n" ^
  (string_of_params act.parameters) ^ "\n" ^
  (string_of_precond act.precondition) ^ "\n" ^
  (string_of_effect act.effect)


let string_of_ast ast =
  let rec iter ast indent =
    let string_of_exprs e_list =
      (List.fold_left (^) ""
         (List.map (fun e -> "\n" ^ iter e (indent + 2)) e_list))
    in
    ( match ast with
      | Expr_unit -> sprintf "%sUNIT" (spaces indent)
      | Expr_domain ( name , body ) ->
        sprintf "%sDOMAIN[␣%s\n%s␣]"
          (spaces indent) name (string_of_exprs body)
      | Expr_problem ( name , body ) ->
        sprintf "%sPROBLEM[␣%s\n%s␣]"
          (spaces indent) name (string_of_exprs body)
      | Expr_predicates( preds ) ->
        sprintf "%sPREDICATES[␣%s␣]\n"
          ( spaces indent )
          (string_of_syms (List.map string_of_pred preds))
      | Expr_action ( act ) ->
        sprintf "%sACTION[␣%s␣]\n"
          (spaces indent)
          (string_of_action act)
      | Expr_objects ( objs ) ->
        sprintf "%sOBJECTS[␣%s␣]\n"
          (spaces indent)
          (string_of_syms (List.map string_of_atom objs))
      | Expr_init ( preds ) ->
        sprintf "%sINIT[␣%s␣]\n"
          (spaces indent)
          (string_of_syms (List.map string_of_pred preds))
      | Expr_goal ( preds ) ->
        sprintf "%sGOAL[␣%s␣]\n"
          (spaces indent)
          (string_of_syms (List.map string_of_pred preds))
    )
  in "\n" ^ iter ast 0 ^ "\n"
```

util.ml

```
(* util.ml *)
open Sexpr
open Ast
open Strips
open Planner

let sprintf  = Printf.sprintf

let printf = Printf.printf

let spaces n = String.make n ' '

(* string from string list *)
let rec string_of_syms = function
    | []   -> ""
    | [s] -> s
    | h::t -> h ^ "␣" ^ (string_of_syms t)

(* returns a random permutation of a list *)
let rec permutation list =
  let rec extract acc n = function
    | [] -> raise Not_found
    | h::t -> if n = 0 then (h, acc@t) else extract (h::acc) (n-1) t
  in
  let extract_rand list len =
    extract [] (Random.int len) list
  in
  let rec aux acc list len =
    if len = 0 then acc else
      let picked, rest = extract_rand list len in
      aux (picked::acc) rest (len-1)
  in aux [] list (List.length list)

(* deletes duplicates from a list *)
let rec filter_duplicates = function
    | []->[]
    | h::t-> h::(filter_duplicates(List.filter(fun x -> x<>h )t))

(* returns a list of heads from a list of lists *)
let heads list =
  let rec decapitate acc = function
    | [] -> List.rev acc
    | h::t -> decapitate ((List.hd h)::acc) t
  in decapitate [] list
```

strips.ml

```
(* symbol table generation *)
open Ast
open Util

type sym = string
```

```

(* problem table for planner *)
type strips_problem =
    {
      mutable init : Ast.predicate list;
      mutable goal : Ast.predicate list;
      mutable ops  : Ast.action list   ;
    }

(* environment *)
type env =
    {
      parent: env option;
      bindings: (sym, sym) Hashtbl.t;
      mutable problem: strips_problem (* TODO: test this *)
    }

(** level-one dependency **)

(* create a new strips problem and symbol table *)
let make parent =
  {
    parent = parent;
    bindings = Hashtbl.create 5;
    problem = { init = [] ; goal = [] ; ops = [] }
  }

(* lookup symbol table entry *)
let rec lookup env name =
   let { parent = p ; bindings = b } = env in
   try  Hashtbl.find b name
   with Not_found ->
        ( match p with
          | Some( parent ) -> lookup parent name
          | None ->
            let error_msg = "unknown␣symbol:␣" ^ name in
            failwith error_msg
        )

(* lookup symbol table entry with an atom *)
let lookup_atom env atom =
  ( match atom with
    | Atom_var a -> lookup env a
    | Atom_gnd a -> lookup env a
    | _ ->
      let error_msg = "unknown␣atom" in
      failwith error_msg
  )

(* insert entry into symbol table *)
let add env name value =
   let { parent = _ ; bindings = b } = env in
   Hashtbl.add b name value

(* insert initial state into strips problem *)
```

```
let add_init env state =
  env.problem.init <- state

(* insert goal state into strips problem *)
let add_goal env state =
  env.problem.goal <- state

(* insert operator into strips problem *)
let add_action env act =
  let acts = env.problem.ops in
  env.problem.ops <- act::acts

(* split the atom for its name, plus semantic checks *)
let atom_name a =
  if String.contains a '-' then
    let stop = (String.index a '-') in
    String.sub a 0 stop
  else
    let error_msg = "expected␣a␣type␣specification␣for␣'" ^ a ^"'"  in
    failwith error_msg

(* split the atom for its type, plus semantic checks *)
let atom_type a =
  if String.contains a '-' then
    let start = (String.index a '-') + 1 in
    let stop = (String.length a) - start in
    String.sub a start stop
  else
    let error_msg = "expected␣a␣type␣specification␣for␣'" ^ a ^ "'" in
    failwith error_msg

(* insert grounded atom into symbol table using object semantics *)
let translate_object env obj =
 let { parent = genv ; bindings = local } = env in
  ( match genv with
    | None ->
      let error_msg = "no␣global␣symbol␣table␣for␣parameters" in
      failwith error_msg
    | Some( global_env ) ->
      let { parent = _ ; bindings = global } = global_env in
      ( match obj with  (* seek type *)
        | Atom_gnd a ->
          ( try
              let _ = Hashtbl.find local (atom_name a) in
              let error_msg =
                "an␣object␣named␣'"
                ^ (atom_name a)
                ^ "'␣has␣already␣been␣declared␣in␣this␣scope" in
              failwith error_msg
            with Not_found -> (* object name is unique *)
              ( try
                  let _ = Hashtbl.find global (":type-"^(atom_type a)) in
                  add env (atom_name a) (atom_type a)
                with Not_found ->
                  let error_msg =
```

```
                    "expected␣type␣declaration␣for␣'"
                    ^ (atom_type a)
                    ^ "'"
                  in failwith error_msg
              )
          )
        | Atom_var a ->
          let error_msg =
            "expected␣'"
            ^ a
            ^ "'␣to␣be␣a␣grounded␣atom" in
          failwith error_msg
        | Atom_nil ->
          let error_msg = "expected␣object␣declaration␣to␣be␣non-empty" in
          failwith error_msg
      )
  )

(* prints the strips problem *)
let string_of_strips prob =
  let { init = s0 ; goal = g ; ops = acts } = prob in
  let title =
    "STRIPS␣PROBLEM\n" in
  let initial_state =
   "INITIAL␣STATE:\n"
    ^ (string_of_syms (List.map string_of_pred s0)) in
  let goal_state =
    "GOAL␣STATE:\n"
    ^ (string_of_syms (List.map string_of_pred g)) in
  let actions =
    "ACTIONS:\n"
    ^ (string_of_syms (List.map string_of_action acts)) in
  sprintf "%s\n%s\n%s\n%s\n"
    (title)
    (initial_state)
    (goal_state)
    (actions)

(** level-two dependency **)

(* insert a grounded atom into symbol table  *)
let translate_grounded_param env param =
  ( match param with  (* TODO: type? *)
    | Atom_gnd a ->
      add env ":static" (atom_name a)
    | _ -> failwith "parameter␣improperly␣parsed"
  )

(* *)
let translate_predicate_param env param = (* local parameter scope *)
  let { parent = genv ; bindings = local } = env in
  ( match genv with
    | None ->
      let error_msg = "no␣global␣symbol␣table␣for␣parameters" in
      failwith error_msg
```

```
  | Some( global_env ) ->
   let { parent = _ ; bindings = global } = global_env in
   ( match param with
    | Atom_var a ->
     ( try (* parameter name must be unique in local scope *)
       let _ = Hashtbl.find local (atom_name a) in
       let error_msg =
        "a␣predicate␣parameter␣named␣'"
        ^ (atom_name a)
        ^ "'␣has␣already␣been␣declared␣in␣its␣scope" in
       failwith error_msg
      with Not_found -> (* parameter name is unique *)
       let _ = add env (atom_name a) (atom_type a) in
       Hashtbl.replace global (":type-"^(atom_type a)) ":type"
     )
    | Atom_gnd a ->
     let error_msg =
      "expected␣'"
      ^ a
      ^ "'␣to␣be␣a␣variable␣parameter" in
     failwith error_msg
    | _ ->
     let error_msg =
      "expected␣␣parameter␣declaration␣to␣be␣non-empty" in
     failwith error_msg
   )
 )
(* insert variable atom into symbol table using parameter semantics *)
let translate_parameter_declaration env param =
 ( match param with
  | Atom_var a ->
(* check types of each atom exist and are used properly in declaration *)
   let { parent = p ; bindings = b } = env in
   ( try
     let _ = Hashtbl.find b (atom_name a) in
     let error_msg =
      "a␣parameter␣named␣'"
      ^ (atom_name a)
      ^ "'␣has␣already␣been␣declared␣in␣this␣scope" in
     failwith error_msg
    with Not_found -> (* parameter name is unique *)
     add env (atom_name a) (atom_type a)
   )
  | Atom_gnd a ->
   let error_msg =
    "expected␣'"
    ^ a
    ^ "'␣to␣be␣a␣variable␣parameter" in
   failwith error_msg
  | _ ->
   let error_msg =
    "expected␣parameter␣declaration␣to␣be␣non-empty" in
   failwith error_msg
 )
```

```
(* insert objects into symbol table *)
let translate_objects env params =
  let translate_obj = translate_object env in
  List.iter translate_obj params

(* lookup parameters *)
let check_params env params = (* params = atom list *)
  let { parent = genv ; bindings = local } = env in
  ( match genv with
    | None ->
      let error_msg = "no␣global␣symbol␣table␣for␣predicate" in
      failwith error_msg
    | Some( global_env ) ->
      let rec check_parameter parameters =
        ( match parameters with
          | [] -> ()
          | Atom_var(a)::t ->
            ( try (* check that parameter name exists in the local scope *)
                let _ = Hashtbl.find local a in (* type not specified *)
                check_parameter t
              with Not_found ->
                let error_msg =
                  "expected␣parameter␣declaration␣for␣'"
                  ^ a
                  ^ "'"
                in failwith error_msg
            )
          | Atom_gnd(a)::t ->
            ( try (* check that parameter name exists in the local scope *)
                let _ = Hashtbl.find local a in (* type not specified *)
                check_parameter t
              with Not_found ->
                let error_msg =
                  "expected␣parameter␣declaration␣for␣'"
                  ^ a
                  ^ "'"
                in failwith error_msg
            )
          | _ ->
            let error_msg = "expected␣grounded␣or␣variable␣atom" in
            failwith error_msg
        )
      in check_parameter params
  )

(** level-three dependency **)

(* insert a list of variable atoms using parameter sematics *)
let translate_parameter_declarations env params =
  let translate_param = translate_parameter_declaration env in
  List.iter translate_param params

(* check and insert a list of predicate parameters into the symbol table
```

```
    *)
let translate_predicate_params env params =
  let translate_param = translate_predicate_param env in (* local parameter scope *)
  List.iter translate_param params

(* insert grounded atoms into symbol table using parameter semantics *)
let translate_gnd_params env params =
  let translate_gnd_param = translate_grounded_param env in
  List.iter translate_gnd_param params

(* lookup predicate declaration *)
let check_var_pred env pred = (* local action parameter scope *)
  let { parent = genv ; bindings = local } = env in
  ( match genv with
    | None ->
      let error_msg = "no␣global␣symbol␣table␣for␣predicate" in
      failwith error_msg
    | Some( global_env ) ->
      let { parent = _ ; bindings = global } = global_env in
      ( match pred with
        | Pred_var( name , params ) -> (* check pred is declared in global *)
          ( try
              let _ = Hashtbl.find global (":predicate-"^name) in
              check_params env params
            with Not_found ->
              let error_msg =
                "expected␣declaration␣for␣predicate␣'"
                ^ name in
              failwith error_msg
          )
        | Pred_gnd( name , params ) ->
          let error_msg =
            "expected␣'"
            ^ name
            ^ "'␣to␣be␣a␣variable␣predicate"
          in failwith error_msg
        | Pred_nil ->
          let error_msg =
            "expected␣predicate␣to␣be␣non-empty" in
          failwith error_msg
      )
  )

(* lookup a grounded predicate using state semantics *)
let check_gnd_pred env pred =
let { parent = genv ; bindings = local } = env in
  ( match genv with
    | None ->
      let error_msg = "no␣global␣symbol␣table␣for␣predicate" in
      failwith error_msg
    | Some( global_env ) ->
      let { parent = _ ; bindings = global } = global_env in
      ( match pred with
```

```
        | Pred_gnd( name , params ) -> (* check pred is declared in global *)
          ( try
              let _ = Hashtbl.find global (":predicate-"^name) in
              check_params env params
            with Not_found ->
              let error_msg =
                "expected␣declaration␣for␣predicate␣'"
                ^ name in
              failwith error_msg
          )
        | Pred_var( name , params ) ->
          let error_msg =
            "expected␣'"
            ^ name
            ^ "'␣to␣be␣a␣grounded␣predicate"
          in failwith error_msg
        | Pred_nil ->
          let error_msg =
            "expected␣predicate␣to␣be␣non-empty" in
          failwith error_msg
      )
  )

(** level-four dependency **)

let rec translate_precondition env precond = (* local action scope *)
  let recurse = translate_precondition env in
  ( match precond with
    | Conj_and conj ->
      List.iter recurse conj
    | Conj_neg pred ->
      check_var_pred env pred
    | Conj_pos pred ->
      check_var_pred env pred
    | _ ->
      let error_msg = "empty␣precondition" in
      failwith error_msg
  )

let translate_effect env effect =
  translate_precondition env effect

let translate_state env state =
  let check_state = check_gnd_pred env in
  List.iter check_state state

(* check and add a predicate declaration into the symbol table *)
let translate_predicate_declaration env pred =
    ( match pred with
      | Pred_var( name , params ) ->
        let { parent = p ; bindings = b } = env in
        ( try
          let _ = Hashtbl.find b (":predicate-"^name) in
          let error_msg =
```

```
              "a␣predicate␣named␣'"
              ^ name
              ^ "'␣has␣already␣been␣declared␣in␣this␣scope" in
            failwith error_msg
          with Not_found -> (* predicate name is unique *)
            let _ = add env (":predicate-"^name) ":predicate" in
            let parent = Some(env) in
            let param_env = make parent in
            translate_predicate_params param_env params
          )
        | Pred_gnd( name , params ) ->
          let error_msg =
            "expected␣'"
            ^ name
            ^ "'␣to␣be␣a␣variable␣predicate" in
          failwith error_msg
        | Pred_nil ->
          let error_msg =
            "predicate␣declaration␣should␣be␣non-empty" in
          failwith error_msg
    )

(* add parameter types to global scope *)
let translate_predicate_param_to_global env pred =
  ( match pred with
    | Pred_var( name , params ) ->
      let rec loop count p =
        ( match p with
          | [] -> ()
          | Atom_var(a)::t ->
            let c = sprintf "%d" count in
            let _ =
              add env
              ("predicate-"^name^"-p"^c^"-"^(atom_type a))
              ":typespec" in
            loop (count+1) t
          | _ ->
            let error_msg =
              "expected␣variable␣atom␣before␣this␣point␣..." in
            failwith error_msg
        )
      in loop 0 params
    | _ ->
      let error_msg = "expected␣variable␣predicate␣before␣this␣point..."
          in
      failwith error_msg
  )

let check_var_pred_types local_env pred =
( match pred with
  | Pred_var( name , params ) ->
    let rec loop count p =
      ( match p with
        | [] -> ()
        | Atom_var(a)::t ->
```

```
          let action_type = lookup local_env a in
          let c = sprintf "%d" count in
          let action_type_spec = "predicate-"^name^"-p"^c^"-"^action_type
              in
          let { parent = genv ; bindings = local } = local_env in
          ( match genv with
            | None ->
              let error_msg = "no␣global␣symbol␣table␣for␣predicate" in
              failwith error_msg
            | Some( global_env ) ->
              let { parent = _ ; bindings = global } = global_env in
              ( try
                  let _ = Hashtbl.find global action_type_spec in
                  loop (count+1) t
                with Not_found ->
                  let error_msg =
                    "expected␣a␣different␣type␣for␣parameter␣'"
                    ^ c
                    ^ "'␣of␣action␣predicate␣'"
                    ^ name
                    ^ "'"
                  in failwith error_msg
              )
          )
        | _ ->
          let error_msg =
            "expected␣variable␣atom␣before␣this␣point␣..." in
          failwith error_msg
      )
    in loop 0 params
  | _ ->
    let error_msg =
      "expected␣variable␣predicate␣before␣this␣point␣..." in
    failwith error_msg
)

let check_gnd_pred_types local_env pred =
( match pred with
  | Pred_gnd( name , params ) ->
    let rec loop count p =
      ( match p with
        | [] -> ()
        | Atom_gnd(a)::t ->
          let state_type = lookup local_env a in
          let c = sprintf "%d" count in
          let state_type_spec = "predicate-"^name^"-p"^c^"-"^state_type in
          let { parent = genv ; bindings = local } = local_env in
          ( match genv with
            | None ->
              let error_msg = "no␣global␣symbol␣table␣for␣predicate" in
              failwith error_msg
            | Some( global_env ) ->
              let { parent = _ ; bindings = global } = global_env in
              ( try
                  let _ = Hashtbl.find global state_type_spec in
```

```
                    loop (count+1) t
                  with Not_found ->
                    let error_msg =
                      "expected␣a␣different␣type␣for␣parameter␣'"
                      ^ c
                      ^ "'␣of␣state␣predicate␣'"
                      ^ name
                      ^ "'"
                    in failwith error_msg
                )
            )
          | _ ->
            let error_msg =
              "expected␣grounded␣atom␣before␣this␣point␣..." in
            failwith error_msg
        )
      in loop 0 params
    | _ ->
      let error_msg =
        "expected␣grounded␣predicate␣before␣this␣point␣..." in
      failwith error_msg
)

let rec check_types local_env conj =
  let recurse = check_types local_env in
  ( match conj with
    | Conj_and c ->
      List.iter recurse c
    | Conj_neg pred ->
      check_var_pred_types local_env pred
    | Conj_pos pred ->
      check_var_pred_types local_env pred
    | _ ->
      let error_msg = "empty␣conjugate" in
      failwith error_msg
  )


(** level-five dependency **)

(* inset predicates into symbol table *)
let translate_predicate_declaration env preds =
  let translate_pred = translate_predicate_declaration env in
  let _ = List.iter translate_pred preds in
  let translate_to_global = translate_predicate_param_to_global env in
  List.iter translate_to_global preds

(* insert action into symbol table and stips problem *)
let translate_action env act =
  let {
        name = n ;
        parameters = params ; (* enforce variable homogeneity *)
        precondition = precond ;
        effect = eff
      } = act in
```

```
  let { parent = p ; bindings = b } = env in
      ( try
          let _ = Hashtbl.find b (":action-"^n) in
          let error_msg =
            "an␣action␣named␣'"
            ^ n
            ^ "'␣has␣already␣been␣declared␣in␣this␣scope" in
          failwith error_msg
        with Not_found -> (* parameter name is unique *)
          let _ = add env (":action-"^n) ":action" in
          let parent = Some(env) in
          let param_env = make parent in (* make parameter table *)
          let _ = translate_parameter_declarations param_env params in
          let _ = translate_precondition param_env precond in
          let _ = check_types param_env precond in
          let _ = translate_effect param_env eff in
          check_types param_env eff
      )

(** level-six dependency **)

(* mapping from ast to strips problem -- semantic checks burried *)
let rec strips_of_ast env ast =
  let recurse = strips_of_ast env in
  ( match ast with
    | Expr_predicates( preds ) ->
      translate_predicate_declaration env preds
    | Expr_action( act ) ->
      let _ = translate_action env act in
      add_action env act
    | Expr_objects( objs ) ->
      translate_objects env objs
    | Expr_init( init ) ->
      let _ = translate_state env init in
      let check_problem_types = check_gnd_pred_types env in
      let _ = List.iter check_problem_types init in
      let p = env.parent in (* TODO: cleanup *)
      ( match p with
        | Some( parent ) ->
          add_init parent init
        | None ->
          let error_msg = ":init␣symbol␣table␣has␣no␣parent" in
          failwith error_msg
      )
    | Expr_goal( goal ) ->
      let _ = translate_state env goal in
      let check_problem_types = check_gnd_pred_types env in
      let _ = List.iter check_problem_types goal in
      let p = env.parent in
      ( match p with
         | Some( parent ) ->
           add_goal parent goal
         | None ->
           let error_msg = ":goal␣symbol␣table␣has␣no␣parent" in
           failwith error_msg
```

```
        )
    | Expr_domain( name , body ) ->
      add env name ":domain" ; List.iter recurse body
    | Expr_problem( name , body ) ->
      let parent = Some(env) in
      let new_env = make parent in
      add env name ":problem" ; List.iter (strips_of_ast new_env) body
    | _ ->
      let error_msg =
        "program␣may␣only␣contain␣a␣domain␣and␣problem␣declaration" in
      failwith error_msg
  )
```

planner.ml

```
open Ast
open Strips
open Util

(** level-one dependency **)

(* ( atom , atom ) Hashtbl *)
module Atomhash = Hashtbl.Make
  (struct
    type t = atom
    let equal x y = x = y
    let hash = Hashtbl.hash
   end)

(* returns string from state *)
let string_of_state state =
    (string_of_syms (List.map string_of_pred state))

(* returns string from plan *)
let string_of_plan plan =
  sprintf "\n%s"
    (string_of_syms (List.map string_of_state plan))

(* returns the intersection of two lists *)
let intersect l1 l2 = List.filter (fun x -> List.mem x l2) l1

(* returns the disjunction of two lists *)
let disjoin l1 l2 = List.filter (fun x -> not(List.mem x l2)) l1

(* removes the last state-action combination from a partial plan *)
let backtrack pplan =
  ( match pplan with
    | state::act::prior_pplan -> prior_pplan
    | _ ->
      let error_msg = "backtrack:␣improper␣input" in
      failwith error_msg
  )

```

```
(* sorts a list of partial plans with respect to their depth *)
let prioritize pplans =
  let randomized_pplans = permutation pplans in
  List.sort ( fun x y ->
    let l1 = List.length x and l2 = List.length y in
  if l1 < l2 then 1 else if l1 = l2 then 0 else -1 ) randomized_pplans

(* 'a -> 'a list -> 'a list *)
let remove pred state =
  List.filter (fun x -> if x = pred then false else true ) state

(* returns operator with matching name *)
let lookup_op opset name =
  let filter_name =
    (fun op -> if op.name = name then true else false ) in
  List.find filter_name opset

(* parses atom for parameter name *)
let param_name atom =
  try
  ( match atom with
    | Atom_gnd a ->
      let stop = (String.index a '-') in
      let name =  String.sub a 0 stop in
      Atom_gnd name
    | Atom_var a ->
      let stop = (String.index a '-') in
      let name =  String.sub a 0 stop in
      Atom_var name
    | Atom_nil -> Atom_nil
  )
  with Not_found ->
    let error_msg = "No␣parameter␣type␣specified␣in:␣" in
    failwith error_msg


(* partitions a conjunction into positive and negative lists *)
let partition_conjunction conj =
  ( match conj with
    | Conj_and c -> List.partition
      ( fun x -> match x with | Conj_pos _ -> true | _ -> false ) c
    | Conj_pos c -> ( [Conj_pos c] , []  )
    | Conj_neg c -> ( []  , [Conj_neg c] )
    | Conj_nil   -> ( []  , []  )
  )

(* ast.conjunction -> ast.predicate *)
let conj_pred conj =
( match conj with
    | Conj_neg nc -> nc
    | Conj_pos pc -> pc
    | _ -> Pred_nil
)

let dump_keys env =  Atomhash.fold (fun k _ acc -> k::acc) env []
```

```

let dump_vals env =  Atomhash.fold (fun _ v acc -> v::acc) env []

let pred_name pred =
  ( match pred with
    | Pred_var ( name , _ ) -> name
    | Pred_gnd ( name , _ ) -> name
    | _ -> ""
  )

let pred_val pred =
  ( match pred with
    | Pred_var ( _ , value ) -> value
    | Pred_gnd ( _ , value ) -> value
    | _ -> [Atom_nil]
  )

(* incarnation of iter2 with hash table replacement *)
let rec bind env names values =
( match ( names , values ) with
  | ( [] , [] ) -> env
  | n::t1 , v::t2 -> Atomhash.replace env n v; bind env t1 t2
  | ( _ , _ ) -> invalid_arg "bind"
)

(* filters states that are elements of the visited list *)
let filter_visited succs visited =
  let visited_states = heads visited in (* visited = list^3 *)
  List.filter(
    fun act_succ ->
      if ((intersect [(List.tl act_succ)] visited_states) = []) then true
      else false) succs

(* return the visited partial plan, if previously visted, else [] *)
let visited state pplan =
  let rec try_remembering acc pp =
    ( match pp with
      | [] -> acc
      | s::remaining_pp -> (* s = state ; t = act::state list *)
        if s = state then (* pplan head is a state *)
          try_remembering pp []
        else
          ( match remaining_pp with
            | [] ->
              try_remembering [] [] (* discovered new state *)
            | prior_act_state::t ->
              let prior_state = List.tl prior_act_state in
              let remaining_pplan = prior_state::t in
              try_remembering [] remaining_pplan
          )
    )
  in try_remembering [] pplan

(* swap an old element for a new element in some list *)
let swap old_pred new_pred state =
```

```
  let rec loop new_state old_state =
  ( match old_state with
    | [] -> List.rev new_state
    | state_pred::remaining_preds ->
      if state_pred = old_pred then
        loop (new_pred::new_state) remaining_preds
      else
        loop (state_pred::new_state) remaining_preds
  )
  in loop [] state

(** level-two dependency **)
(* grounds a variable predicate with a set of bindings *)
let ground_pred env pred =
  let keys = pred_val pred in
  let vals =
    List.map ( fun k -> Atomhash.find env k ) keys in
  ( match pred with
    | Pred_var( name , _ ) -> Pred_gnd( (pred_name pred) , vals )
    | _ ->
      let error_msg = "Failed␣to␣ground␣predicate" in
      failwith error_msg
  )

(* grounds an operator with a set of bindings *)
let ground_op env op =
  let names = List.map param_name op.parameters in
  let vals = List.map (fun n -> Atomhash.find env n) names in
  Pred_gnd( op.name , vals )

(* returns the bindings for an action *)
let action_bindings action op =
  let env = Atomhash.create (List.length op.parameters) in
  let names = List.map param_name op.parameters in
  let vals = pred_val action in
  let _ = List.iter2 ( fun n v -> Atomhash.add env n v ) names vals in
  env

(* extend bindings such that names = vals *)
let extend env name_pred val_pred =
  let test_env = Atomhash.copy env in
  bind test_env (pred_val name_pred) (pred_val val_pred)

(* checks if test bindings are consistent with reference bindings *)
let bindings_valid test_env ref_env =
  let test_names = dump_keys test_env in (*lookup every name in ref*)
  let test_vals = dump_vals test_env in
  let rec loop acc names values  =
    ( match ( names , values ) with
      | ( [] , [] ) -> acc
      | (test_name::t1 , test_val::t2 )->
        (try
          let ref_val = Atomhash.find ref_env test_name in
          if test_val = ref_val then loop (true::acc) t1 t2
          else [false] (* value mismatch *)
```

```
        with Not_found -> (* name not found *)
          loop acc t1 t2 )
      | ( _ , _ ) ->
        let error_msg = "bindings_valid:␣improper␣input" in
        failwith error_msg
    )
      in List.fold_left (fun x y -> x && y ) true (loop [] test_names test_vals)

(* returns a list of matching state predicates *)
let matching_preds pred state =
  let name = pred_name pred in
  let match_name =
    ( fun pred -> if name = (pred_name pred) then true else false) in
  List.filter match_name state

(* partitions a conjunction into positive and negative predicates *)
let partition_to_predicates conj =
  let ( pos_conj , neg_conj ) = partition_conjunction conj in
  let pred_of = List.map conj_pred in
  ( pred_of pos_conj , pred_of neg_conj )

(* checks if the goal is a subset of partial plan *)
let goal_test pplan goal =
  let state = List.hd pplan in
  let rec loop check_sum goal_preds =
    ( match goal_preds with
      | [] -> check_sum
      | gp::remaining_goal_preds ->
        let intersection = intersect [gp] state in
        ( match intersection with
          | [] -> [false]
          | _ -> loop (true::check_sum) remaining_goal_preds
        )
    )
in List.fold_left (fun x y -> x && y ) true (loop [] goal)

(** level-three dependency **)

(* checks if the partial plan is the intial state *)
let search_exhausted pplan init_state =
  goal_test pplan init_state (* TODO: is this valid for all cases?*)

(* outputs partitioned, grounded effects *)
let partition_to_grounded_effect op action =
  let env = action_bindings action op in (* create bindings *)
  let ( pos_preds , neg_preds ) = partition_to_predicates op.effect in
  let pos_effs = List.map (ground_pred env) pos_preds in
  let neg_effs = List.map (ground_pred env) neg_preds in
  ( pos_effs , neg_effs )

(** level-four dependency **)

(* check if positive effects are the same as the negative effects *)
let complimentary_effects action op =
```

```
  let ( pos_effs , neg_effs ) = partition_to_grounded_effect op action in
  let intersection = intersect pos_effs neg_effs in
  ( match intersection with
    | [] -> false
    | _ -> true
  )

(* apply an action and return the successor state *)
let successor state opset action = (* act is a grounded pred list *)
  let op = lookup_op opset (pred_name action) in
  let ( pos_effs , neg_effs ) = partition_to_grounded_effect op action in
  let rec remove_neg_effects s peffs neffs =
    ( match neffs with
      | [] -> action::peffs@s
      | neg_eff::t -> (* check for repeating effects in strips *)
        let matching_effects = matching_preds neg_eff pos_effs in
        ( match matching_effects with
          | [] ->
            remove_neg_effects (remove neg_eff s) t peffs
          | [pe] ->
            remove_neg_effects (swap neg_eff pe s) t (remove pe peffs)
          |  _ ->
            let error_msg =
              "Predicates␣may␣only␣appear␣once␣in␣effect␣declaration" in
            failwith error_msg
        )
    )
  in remove_neg_effects state pos_effs neg_effs

(** level-five dependency **)

(* attempts to unify state predicates with an operator's preconditions *)
let rec unify op ( pos_preds , neg_preds ) env state  =
  ( match pos_preds with
    | [] ->
      ( match neg_preds with
        | [] -> [ground_op env op]
        | np::remaining_np ->
          let state_preds = matching_preds np state in
          let rec unify_npreds all_sp =
            ( match all_sp with
              | [] ->
                let error_msg = "no␣matching␣predicates␣with␣state" in
                failwith error_msg
              | sp::remaining_sp ->
                let tenv = extend env np sp in
                if ( bindings_valid tenv env ) then
                  unify op ( [] , remaining_np ) tenv state
                else
                  []
            ) in unify_npreds state_preds
      )
    | pp::remaning_pp ->
      (* look at common state predictes *)
      let state_preds = matching_preds pp state in
```

```
      let rec unify_ppreds all_sp =
        ( match all_sp with
          | [] -> []
          | sp::remaining_sp ->
            let tenv = extend env pp sp in (* temp sub.*)
            if ( bindings_valid tenv env ) then
              ( match remaning_pp with
                | [] -> (* check for complimentary effects *)
                  let action = ground_op tenv op in
                  if ( complimentary_effects action op ) then []
                  else unify op ( remaning_pp , neg_preds ) tenv state
                | _ -> unify op ( remaning_pp , neg_preds ) tenv state
              )
            else (* try unifying pp with another state pred *)
              unify_ppreds remaining_sp
        ) in unify_ppreds state_preds
  )

(** level-six dependency **)
(* return the first valid action in a state *)
let applicable_instance op state =
  let ( pos_preds , neg_preds ) =
    partition_to_predicates op.precondition in
  let env = Atomhash.create 10 in
  unify op ( pos_preds , neg_preds ) env state

(* accumulate action instances from every possible state permutation *)
let applicable_instances op state =
  let max_permutations = List.length state in
  let rec find_instances app_actions permutations s =
    if permutations = max_permutations then
      app_actions
    else
      let app_action = applicable_instance op s in
      let s1 = (List.tl s)@[(List.hd s)] in
      find_instances (app_action::app_actions) (permutations+1) s1
  in List.flatten ( find_instances [] 0 state )

(* all applicable actions, includes backward moves, but no self loops *)
let applicable_actions state opset =
  let rec next_op actions ops =
    ( match ops with
      | [] -> List.flatten actions
      | op::remaining_ops ->
        (* list of all op's instances in the state  *)
        let action_instances = applicable_instances op state in
        next_op (action_instances::actions) remaining_ops
    )
  in next_op [] opset

(* returns all available nodes not previously visited *)
let successors pplan opset explored =
  let state = List.hd pplan in
  let actions = applicable_actions state opset in
  ( match actions with
```

```
  | [] ->
   let error_msg =
    "no␣applicable␣actions␣found␣for␣state" in
   failwith error_msg
  | _ ->
   let all_succs =
    filter_visited(
     filter_duplicates(
      List.map (successor state opset) actions)) explored in
   let rec filter applicable_pplans succs =
    ( match succs with
     | [] -> applicable_pplans
     | act_succ::remaining_succs ->
      let act = List.hd act_succ and succ = List.tl act_succ in
       if succ = state then(* already filtered *)
       filter applicable_pplans remaining_succs
      else (* back move *)
        let prior_pplan = visited succ pplan in
        ( match prior_pplan with
         | [] ->
          let new_pplan = succ::[act]::pplan in
          filter (new_pplan::applicable_pplans) remaining_succs
         | p::pp ->
          filter (prior_pplan::applicable_pplans) remaining_succs
        )
    )
   in filter [] all_succs
 )

(* states are partial plans *)
let fsearch problem =
 let { init = s0 ; goal = g ; ops = opset } = problem in
 let rec dfs state explored =
  if ( goal_test state g ) then List.rev state
  else
   let fringe = successors state opset explored in
   ( match fringe with
    | [] ->
     if search_exhausted state s0 then
      let error_msg = "Problem␣has␣no␣solution" in
      failwith error_msg
     else
      dfs (backtrack state) (state::explored)
    | _ ->
     let priority_queue = prioritize fringe in
     let succ = List.hd priority_queue in
     dfs succ (state::explored)
   )
 in dfs [s0] []

let solve problem =
 fsearch problem
```

compile.ml

```
(* compile.ml *)
open Ast
open Strips
open Util
(*
   - push state preds on stack until goal is satisfied
   - if all preds are on stack, pop all off and push action
   - push state preds on stack until goal is satisfied
   - if all preds are on stack, pop all off and push next action
*)

type instruction =
  | Push_action of string
  | Push_pred of string
  | Pop_pred
  | Goal

let string_of_instruction instruction =
( match instruction with
  | Push_action action -> sprintf "\nPush_action␣%s" action
  | Push_pred pred -> sprintf "\nPush_pred␣%s" pred
  | Pop_pred -> "\nPop_pred"
  | Goal -> "\nGoal\n"
)

let print_bytecode instructions =
  (string_of_syms (List.map string_of_instruction instructions ))

let string_of_atom atom =
  ( match atom with
    | Atom_var a | Atom_gnd a -> a
    | Atom_nil -> ""
  )

(* check if name is a known action *)
let rec known_action name opset =
  ( match opset with
    | [] -> false
    | op::remaining_ops ->
      if op.name = name then true
      else known_action name remaining_ops
  )

let rec translate_params params =
    sprintf "%s"
      (string_of_syms (List.map string_of_atom params))

let translate_action pred =
  ( match pred with
    | Pred_gnd( name , params ) ->
      let translated_params = translate_params params in
      let action = sprintf "%s(␣%s␣)" name translated_params in
      Push_action( action )
```

```
    | _ ->
      let error_msg = "error␣translating␣predicate" in
      failwith error_msg
  )

let translate_pred pred =
   ( match pred with
    | Pred_gnd( name , params ) ->
      let predicate =
        sprintf "␣%s(␣%s␣)"
          name
          (string_of_syms(List.map string_of_atom params))
      in
      Push_pred predicate

    | _ ->
      let error_msg = "error␣translating␣predicate" in
      failwith error_msg
  )

let pop_for num =
  let rec loop acc count =
    if 0 < count then
      loop (Pop_pred::acc) (count - 1)
    else
      acc
  in loop [] num

(* translate plan into bytecode *)
let translate plan problem =
  let { init = s0 ; goal = g ; ops = opset } = problem in
  let plan_preds = List.flatten plan in
  let num_preds = List.length s0 in
  let rec recurse instructions preds =
    ( match preds with
      | [] -> List.rev instructions
      | pred::remaining_preds ->
        ( match pred with
          | Pred_gnd( name , params ) ->
            if known_action name opset then
              let pred_pops = pop_for num_preds in
              let translated_action = translate_action pred in
              let transition = translated_action::pred_pops in
              recurse (transition@instructions) remaining_preds
            else
              ( match remaining_preds with
                | [] -> recurse (Goal::instructions) remaining_preds
                | _ ->
                  let translated_pred = translate_pred pred in
                  recurse (translated_pred::instructions) remaining_preds
              )
          | _ ->
            let error_msg = "error␣translating␣predicate" in
            failwith error_msg
        )
```

```
    )
  in recurse [] plan_preds
```

execute.ml

```
open Compile

let rec execute_instructions instructions =
  let stack = Array.make 1024 "" in
  let rec exec fp sp pc =
    ( match instructions.(pc) with
      | Push_action action ->
        stack.(sp) <- action;
        let _ = Printf.printf "%s\n" (stack.(sp)) in
        exec fp (sp + 1) (pc + 1)
      | Push_pred predicate ->
        stack.(sp) <- predicate;
        exec fp (sp + 1) (pc + 1)
      | Pop_pred ->
        exec fp (sp - 1) (pc + 1)
      | Goal -> ()
    )
  in exec 0 0 0

let execute_bytecode instructions =
  execute_instructions (Array.of_list instructions)
```

pddlyte.ml

```
open Strips
(*
   compiler flags
   -a: print ast
   -p: print plan (action sequence)
   -c: compile and print instructions
   -e: execute instructions
*)

(* compiler flags *)
type flag =  Path | Compile | Execute

(* decode the compiler flag *)
let decode_flag argv =
  if Array.length argv > 1 then
    List.assoc argv.(1)
      [ ("-p", Path);
        ("-c", Compile);
        ("-e", Execute)]
  else Execute

(* open infile *)
```

```
let decode_infile argv =
  if Array.length argv > 1 then
    open_in argv.(2)
  else
    open_in argv.(1)

(* compile to path *)
let print_plan infile =
  let lexbuf = Lexing.from_channel infile in
  let env = Strips.make None in
  let rec loop () =
    let sexpr  = Parser.parse Lexer.token lexbuf in
    (match sexpr with
      | None ->
        let problem = env.problem in
        let plan = Planner.solve problem in
        Printf.printf "%s" (Planner.string_of_plan plan);
        flush stdout;()
      | Some s ->
        let ast = Ast.ast_of_sexpr s in
        let _ = Strips.strips_of_ast env ast in
        flush stdout;
        loop ()
    )
  in loop ()

(* compile to bytecode *)
let compile_program infile =
  let lexbuf = Lexing.from_channel infile in
  let env = Strips.make None in
  let rec loop () =
    let sexpr  = Parser.parse Lexer.token lexbuf in
    ( match sexpr with
      | None ->
        let problem = env.problem in
        let plan = Planner.solve problem in
        let instructions = Compile.translate plan problem in
        Printf.printf "%s" (Compile.print_bytecode instructions);
        flush stdout;()
      | Some s ->
        let ast = Ast.ast_of_sexpr s in
        let _ = Strips.strips_of_ast env ast in
        flush stdout;
        loop ()
    )
  in loop ()

(* compile and execute bytecode *)
let execute_program infile =
  let lexbuf = Lexing.from_channel infile in
  let env = Strips.make None in
  let rec loop () =
    let sexpr  = Parser.parse Lexer.token lexbuf in
   ( match sexpr with
      | None ->
```

```
        let problem = env.problem in
        let plan = Planner.solve problem in
        let instructions = Compile.translate plan problem in
        Execute.execute_bytecode instructions; ()
      | Some s ->
        let ast = Ast.ast_of_sexpr s in
        let _ = Strips.strips_of_ast env ast in
        flush stdout;
        loop ()
    )
  in loop ()

(* process input *)
let run_program flag infile =
    ( match flag with
        | Path ->
          ( try
              print_plan infile
            with (Failure f) ->
              Printf.fprintf stderr "ERROR:␣%s\n" f;
              close_in infile
          )
        | Compile ->
          ( try
              compile_program infile
            with (Failure f) ->
              Printf.fprintf stderr "ERROR:␣%s\n" f;
              close_in infile
          )
        | Execute ->
          ( try
              execute_program infile
            with (Failure f) ->
              Printf.fprintf stderr "ERROR:␣%s\n" f;
              close_in infile
          )
     )

(* pddlyte top-level *)
let _ =
   if (Array.length Sys.argv > 3 || Array.length Sys.argv < 2) then
      Printf.fprintf stderr "USAGE:␣%s␣[flag]␣[input_filename]\n" Sys.argv
          .(0)
   else

     let flag = decode_flag Sys.argv in
     let infile = decode_infile Sys.argv in
     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 theorm 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.