

# SHOP3 Manual

## Table of Contents

<a href="#">1 Introduction</a>
<a href="#">2 Execution Environment</a>
<a href="#">3 Running SHOP3</a>
<a href="#">3.1 Loading the Planner</a>
<a href="#">3.2 Executing the Planner</a>
<a href="#">3.2.1 FIND-PLANS</a>
<a href="#">3.2.2 FIND-PLANS-STACK</a>
<a href="#">3.2.3 DO-PROBLEMS</a>
<a href="#">3.2.4 Common Keyword Arguments</a>
<a href="#">3.3 Tracing</a>
<a href="#">3.4 Other Debugging Features</a>
<a href="#">3.5 Syntax Checks</a>
<a href="#">3.6 Debugging Suggestions</a>
<a href="#">3.7 Hook Routines</a>
<a href="#">4 The SHOP3 Formalism</a>
<a href="#">4.1 Symbols</a>
<a href="#">4.2 General Lisp Expressions</a>
<a href="#">4.3 Terms</a>
<a href="#">4.3.1 List Terms</a>
<a href="#">4.3.2 Eval Terms</a>
<a href="#">4.3.3 Call-terms</a>
<a href="#">4.4 Logical Atoms</a>

## [4.5 Logical Expressions](#)

### [4.5.1 Conjuncts](#)

### [4.5.2 Disjuncts](#)

### [4.5.3 Negations](#)

### [4.5.4 Implications](#)

### [4.5.5 Universal Quantifications](#)

### [4.5.6 Assignments](#)

### [4.5.7 Eval expressions](#)

### [4.5.8 Call-expressions](#)

### [4.5.9 Enforce-expressions](#)

### [4.5.10 Setof-expressions](#)

### [4.5.11 Bagof-expressions](#)

## [4.6 Logical Precondition](#)

### [4.6.1 First Satisfiers Precondition](#)

### [4.6.2 Sorted Precondition](#)

## [4.7 Axioms](#)

## [4.8 Task Atoms](#)

## [4.9 Task Lists](#)

## [4.10 Operators](#)

### [4.10.1 Operators: Legacy Syntax](#)

## [4.11 Methods](#)

## [4.12 Planning Domain](#)

### [4.12.1 Simple Form](#)

### [4.12.2 Extended form](#)

### [4.12.3 Inclusion directives](#)

## [4.13 Planning Problem](#)

## [4.14 Planning Problem Set](#)

## [4.15 Plans](#)

## [5 PDDL Support](#)

### [5.1 PDDL Methods](#)

<a href="#">5.1.1 Enhanced Preconditions</a>
<a href="#">5.1.2 PDDL Domain Classes</a>
<a href="#">6 The SHOP Theorem Prover</a>
<a href="#">7 The SHOP Unifier</a>
<a href="#">8 Plan Grapher</a>
<a href="#">9 Internal Technical Information</a>
<a href="#">9.1 Internal Knowledge Structures</a>
<a href="#">9.1.1 Substitutions</a>
<a href="#">9.1.2 States and Satisfiers</a>
<a href="#">9.2 Formal Semantics</a>
<a href="#">9.2.1 Semantics of Operators</a>
<a href="#">9.2.2 Semantics of Methods</a>
<a href="#">9.2.3 Semantics of Plans</a>
<a href="#">9.3 Key Functions in SHOP3</a>
<a href="#">10 Differences between SHOP 1.x, SHOP2 and SHOP3</a>
<a href="#">10.1 SHOP 1.x Syntax Comparison</a>
<a href="#">10.2 SHOP 1.x Functionality Comparison</a>
<a href="#">11 General Notes on SHOP3</a>
<a href="#">12 Acknowledgments</a>
<a href="#">13 References</a>
<a href="#">Function and Macro Index</a>
<a href="#">Type and Class Index</a>
<a href="#">Variable Index</a>
<a href="#">Concept Index</a>
<a href="#">Colophon</a>

Next: [Introduction](#), Previous: [\(dir\)](#), Up: [\(dir\)](#) [[Contents](#)][[Index](#)]

# SHOP3 Manual



Robert P. Goldman<sup>1</sup>, Dana Nau<sup>2</sup>, and Contributors

<sup>1</sup> SIFT, LLC

319 North First Avenue, Suite 400

Minneapolis, MN 55401, USA

<sup>2</sup> Department of Computer Science

University of Maryland

College Park, MD 20742, USA

---

Next: [Execution Environment](#), Previous: [Top](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 1 Introduction

AI planning is the subfield of artificial intelligence (AI) that aims at automating processes of *means-ends reasoning*. In general, AI planning is the problem of finding a sequence of actions that, executed in a specified initial state, will reach a goal state. This is a problem with applications to diverse areas including manufacturing, autonomous space

and deep sea exploration, medical treatment, and military operations, to name just a few. This is the manual for SHOP3, the third major version of the Simple Hierarchical Ordered Planner.

AI planning systems takes as input a *domain* – a description of available actions, relations, etc.; a *problem* – a description of the initial state of the system, and an *objective*, a task to be performed or goal to be achieved. From these, it generates a *plan*: a sequence of actions that, if performed with the expected results, will attain the objective. This is the key function performed by SHOP3, although, as will be seen in this manual, many additional functions are offered.

SHOP3 is a domain-independent planning system based on **ordered task decomposition**, a modified version of Hierarchical Task Network (HTN) planning that involves planning for tasks in the same order that they will later be executed. An HTN, or *decomposition*, planner proceeds by decomposing *nonprimitive tasks* recursively into smaller and smaller subtasks, until *primitive tasks* are reached that can be performed directly using the planning operators.<sup>1</sup> This manual does not give an introduction to HTN planning or AI planning in general, for that we recommend the above-cited textbook by Ghallab, *et al.*, and/or the research papers describing SHOP3.

SHOP and SHOP 2 were originally developed at the Computer Science Department of the University of Maryland, College Park, by Prof. Dana Nau's research group. This manual draws heavily on material from the manual for SHOP 2, which was, in turn based, in part, on the JSHOP documentation written by F. Sun Yaman, with additional material from Yue Cao's December 2000 draft of the SHOP 2 documentation and pseudocode from [Nau *et al.*, 2001]. Some updates to the SHOP 2 manual were made by Robert P. Goldman and John Maraist, of SIFT.

SHOP3 contains two important subsystems that perform useful functions as part of SHOP3, but that can be used on their own. The first is the *unifier*, which computes the most general unifier of two logical formulas, encoded as lisp s-expressions. The second subsystem is the *theorem-prover*, which performs Prolog-style rule-based Horn clause deduction over state sequences.

Robert P. Goldman and Ugur Kuter have a paper in the European Lisp Symposium describing SHOP3: [Hierarchical Task Network Planning in Common Lisp: the case of SHOP3](#).

The planners in the SHOP family have the following distinctive characteristics:

- SHOP knows the current state-of-the-world at each step of the planning process.
- It has great expressive power, far beyond that of conventional PDDL planners. For example, in the preconditions of operators and methods it can do mixed symbolic/numeric computations and execute calls to external programs.
- It can be used to create very efficient domain-specific planning algorithms. The software distribution includes several examples of such domain-specific algorithms.
- SHOP3 incorporates many features from [PDDL](#), e.g., support for

quantifiers and conditional effects in methods and operators.

- SHOP2 and SHOP3 (unlike SHOP) allow a combination of partially ordered and fully ordered task networks through the use of the `:unordered` and `:ordered` keywords.
- SHOP3 task networks are *less* expressive than full HTN planners such as UMCP, which have labeled tasks in their task networks and allow arbitrary ordering constraints.
- SHOP3 allows branch-and-bound optimization of plan costs. For small problems, this capability can be used to find the absolute minimum cost plans. For larger problems, this capability can be used with time limits to get the lowest cost plan that is found within the given time limit.
- SHOP3 adds support for the Planner Domain Description Language (PDDL), and updates the SHOP language for easier domain engineering.

---

Next: [Running SHOP3](#), Previous: [Introduction](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 2 Execution Environment

SHOP3 is written in Common Lisp. To be able to run SHOP3, you will need to have Common Lisp installed on your computer. We have run SHOP3 successfully under the following implementations of Common Lisp, and we would be interested in hearing your reports about other implementations:

- Allegro Common Lisp v. 10.x (on x86 and x86\_64 Linux and Mac OSX);
- Steel Bank Common Lisp, v. 1.4+ on x86 and x86\_64, Linux and Mac OSX.
- Clozure Common Lisp, version 1.11 on Mac OS X and Linux.

We suspect that there may be some difficulties in running SHOP3 on Windows; most of these have to do with getting the ASDF system definitions to work on Windows, not with SHOP3 proper. Please contact us if you encounter difficulties. We welcome reports of experiences with other platforms and CL implementations, and will attempt to support users who wish to bring SHOP3 up on other combinations.

Note that SHOP3 is known **not** to work with Embeddable Common Lisp (ECL).

SHOP3 is distributed with a system definition written using the open-source ASDF system definition facility (for more information, see <http://common-lisp.net/projects/asdf/>). You should insure that all of the .asd files in the SHOP3 distribution can be found by ASDF, per the instructions given with ASDF, and then SHOP3 should load without any problems. See Section 5.1 for more details about how to load SHOP3. All of the CL implementations we know of ship with ASDF pre-installed. To enable it you may have to

```
(require :asdf)
```

---

Next: [The SHOP3 Formalism](#), Previous: [Execution Environment](#), Up: [Top](#)  
[\[Contents\]](#)[\[Index\]](#)

## 3 Running SHOP3

The latest version of SHOP3 is loaded by using the ASDF system definition facility. The first of the following subsections explains how to use ASDF to load SHOP3. Note that previous methods of starting SHOP3, by hand-coded load files, and `mk:deftsystem`, are no longer supported. There are two ways to execute the SHOP3 planning process: `find-plans`, which finds plans for a single planning problem, and `do-problems`, which finds plans for a planning problem set. Subsection 5.2 describes the use of these functions. Subsection 5.3 describes the functions `shop-trace` and `shop-untrace`, which are the primary mechanisms for debugging SHOP3 domain descriptions and problem specifications. Subsection 5.4 describes some additional features that may also be useful for debugging domain descriptions and problems for SHOP3. Finally, subsection 5.5 describes some hook routines that can be used to customize the behavior of SHOP3.

---

Next: [Executing the Planner](#), Previous: [Running SHOP3](#), Up: [Running SHOP3](#)  
[\[Contents\]](#)[\[Index\]](#)

### 3.1 Loading the Planner

The SHOP3 planner should be loaded into your lisp environment using ASDF. Assuming that ASDF is properly installed, and the `SHOP3.asd` system definition file can be found by ASDF, the following command should get the system loaded:

```
(asdf:load-system "shop3")
```

SHOP3 is defined in the `SHOP3` package (and uses the `SHOP3.theorem-prover` package). The easiest way to use the system for experimentation will be to change to the predefined `:SHOP-USER` package and work in there:

```
(in-package :SHOP-USER)
```

If you are working on a larger or more ambitious project, it will be more appropriate for you to work in a package of your own definition, which should, at least, use the `SHOP3` and `COMMON-LISP` packages.

---

Next: [Tracing](#), Previous: [Loading the Planner](#), Up: [Running SHOP3](#)  
[\[Contents\]](#)[\[Index\]](#)

### 3.2 Executing the Planner

---

Next: [find-plans-stack](#), Previous: [Executing the Planner](#), Up: [Executing the Planner](#) [[Contents](#)][[Index](#)]

### 3.2.1 FIND-PLANS

The find-plans function has one mandatory argument, the name of a planning problem, and a set of optional keyword arguments. It returns up to four values. Find-plans will always return two values: (1) a list of plans and (2) the total amount of CPU time used during planning (in seconds). If the `:plan-tree` argument (see below) is non-NIL, then two additional values will be returned: (3) a list of plan tree data structures and (4) a list of final state data structures. From the plan state data structures, the user can extract full state trajectories for the plans.

Function: **find-plans** *problem &key domain which verbose gc pp plan-tree optimize-cost collect-state time-limit explanation depth-cutoff state tasks state-type hand-steer leashed out-stream*

`find-plans` looks for solutions to the planning problem named `problem`. The keyword arguments are as follows: `:which` tells what kind of search to do. Its possible values are: `:first` – depth-first search, returning the first plan found. `:all` – depth-first search for \*all\* plans. `:shallowest` – depth-first search for the shallowest plan in the search space (this usually is also the shortest plan). If there's more than one such plan, return the first.

*:all-shallowest – depth-first search for all shallowest plans.  
:id-first – iterative deepening search, returning the first plan.  
:id-all – iterative deepening search for all shallowest plans.  
:random – Randomized search. Used by Monroe. Not for normal*

SHOP2 domains, since normal SHOP2 domains have order-dependent semantics.

*:verbose* says how much information to print about the plans  
**SHOP2**

`finds`. Its values can be any of the following: 0 or `nil` – print nothing 1 or `:stats` – print some statistics on SHOP2's operation 2 or `:plans` – print the stats and print all plans found, but omit operator costs and omit all operators whose names start with "!!" 3 or `:long-plans` – print the stats and plans, including all operator costs and all operators (even those whose names start with "!!")

*:gc* says whether to do a garbage collection before calling `seek-plans`  
*:plan-tree* indicates whether or not to return plan tree(s).  
*:collect-state* indicates whether or not to return final state(s). For backward-

compatibility, states are also returned whenever `:plan-tree` is true. This should probably eventually change.



*return values:*

plans found --- a list of plans. Each plan is a list that alternates a between instantiated operators and costs  
 run time --- floating point value in seconds /if/ the plan-tree keyword argument is supplied, there will be two additional return values:  
 plan-trees --- a list of plan trees, whose form is specified elsewhere.  
 final-states --- a list of final state structures, one per plan.

Next: [do-problems](#), Previous: [find-plans](#), Up: [Executing the Planner](#)  
[\[Contents\]](#)[\[Index\]](#)

### 3.2.2 FIND-PLANS-STACK

Function: **find-plans-stack** *problem &key domain verbose plan-tree gc no-dependencies repairable rationale out-stream which*

Top level search function for explicit-state search in SHOP3. Does not support the full range of options supported by SHOP3: only supports finding the first solution to problem. To comply with SHOP3, though, always returns a list of plans. If the plan-tree keyword argument is non-NIL, will return an enhanced plan tree, with causal links, unless no-dependencies is non-NIL.

Next: [Common Keyword Arguments](#), Previous: [find-plans-stack](#), Up: [Executing the Planner](#)  
[\[Contents\]](#)[\[Index\]](#)

### 3.2.3 DO-PROBLEMS

The do-problems function has one mandatory argument, which can either be the name of a planning problem set or a list of names of planning problems. It executes find-plans on each of the given planning problems and returns nil.

Function: **do-problems** *problems &rest keywords*

do-problems runs find-plans on each problem in problems, which may be either a problem-set name (a symbol) or a list of problems.

Returns nothing of interest: should only be run for what it displays on the console.

Previous: [do-problems](#), Up: [Executing the Planner](#) [\[Contents\]](#)[\[Index\]](#)

### 3.2.4 Common Keyword Arguments

The keyword arguments to find-plans and do-problems are as follows:

- *which* says what kind of search to do. Here are its possible values and what they mean. The default value of *which* is the value of the global variable \*which\* (whose default value is :first).

Value	Kind of search
-------	----------------

<code>:first</code>	Depth first search, stopping at the first plan found
<code>:all</code>	Depth-first search, but don't stop until all plans in <i>plans(S, T, M)</i> have been found
<code>:shallowest</code>	Depth-first search for the shallowest plan (or the first such plan if there is more than one of them). In many domains, this is also the least-cost plan
<code>:all-shallowest</code>	Depth-first search for all shallowest plans in the search space
<code>:id-first</code>	Iterative-deepening search, stopping after the first plan found
<code>:id-all</code>	Iterative-deepening search for all shallowest plans

The *iterative deepening* options, `:id-all` and `:id-first`, are equivalent to taking a modified version of `find-plans` that backtracks each time it reaches depth  $d$ , and calling it repeatedly with  $d = 1, 2, \dots$ , until a plan is found.

- `verbose` says what information to print out, as shown in the following table. The default value for `verbose` is 1.

**Value    What to print**

0 or <code>nil</code>	Nothing
1 or <code>:stats</code>	Some statistics about the search
2 or <code>:plans</code>	The statistics plus a succinct version of each plan found: internal operators (see <a href="#">internal operators</a> ), and operator costs are omitted).
3 or <code>:long-plans</code>	The statistics plus the complete version of each plan found

- If `gc` is non-`nil`, then `find-plans` calls the garbage collector just before starting its search, thus making it somewhat easier to get repeatable experimental results. The default value of `gc` is `T`.
- If `pp` is non-`nil`, then all printing done by SHOP3 is performed using the Common Lisp pretty-printing mechanism. This typically leads to more easily read output. The default value of `pp` is `T`.
- The `state` argument controls how states are represented internally. SHOP3 can have different performance characteristics depending on the value provided to this argument. If you are encountering out-of-memory errors in SHOP3 or you want to get the maximum speed possible from SHOP3 for a particular set of problems, you may wish to experiment with different values for this argument. The default value is `:mixed`, which represents states using a combination of lists and hash tables; this value has been shown to provide a reasonably good combination of speed and memory usage on a variety of test problems. The other values are `:list`, `:hash`, and `:bit`.
- The `optimize-cost` argument is used to perform planning with branch-and-bound optimization of the total plan cost. The default value for this argument is `nil`. If the value of this argument is `nil`, the optimization feature is disabled. If the value

of the argument is `t`, SHOP3 will search for plans with the minimum total cost. If the value of the argument is a number, SHOP3 will use the branch-and-bound technique to search for plans with cost less than or equal to the value of the argument. The optimization feature is written under the assumption that the costs of operators are always non-negative. If this assumption is invalid, SHOP3 will produce unreliable results (specifically it will prune out some valid plans). The interaction of `:optimize-cost` with the various options for `:which` can be subtle. Below are notes on each possible combination:

- `(:which :first :optimize-cost t)`

Under these arguments, SHOP3 returns the first plan found for which no other valid plan has a lower total cost. Note that this option may take much more time to run than using `(:which :first :optimize-cost nil)` since even after it finds the plan, it must keep searching to see if it can find a cheaper plan. However, this option may be significantly faster than `(:which :all :optimize-cost nil)` since the branch-and-bound mechanism will prune out non-optimal plans without having to consider them all the way to the end. In some cases, this will mean that `(:which :first :optimize-cost t)` terminates and `(:which :all :optimize-cost nil)` does not.

- `(:which :first :optimize-cost number)`

Under these arguments, SHOP3 returns the first plan found whose total cost is less than or equal to the number given. If there is no plan whose total cost is less than or equal to that number, SHOP3 will return no plans. Note that if the number given is large enough, these arguments can produce results much more quickly than with `(:which :first :optimize-cost t)`; specifically, as soon as SHOP3 finds a plan for which the cost is met, it can terminate and does not have to keep searching for cheaper plans.

- `(:which :all :optimize-cost t)`

Under these arguments, SHOP3 returns all plans for which no other valid plan has a lower total cost. Obviously, all plans returned under these options will have equal total cost.

- `(:which :all :optimize-cost number)`

Under these arguments, SHOP3 returns all plans with total cost less than or equal to the given number.

- `(:which :shallowest :optimize-cost t)`

Under these arguments, SHOP3 returns a plan that has the shallowest depth of all valid plans and for which there is no other shallowest depth valid plan which has a lower total cost. In other words, these arguments produce the cheapest of all shallowest plans (which, incidentally, is not necessarily the same thing as the shallowest of all cheapest plans).

- `(:which :shallowest :optimize-cost number)`

Under these arguments, SHOP3 returns a plan which has the shallowest depth of all valid plans and whose total cost is less than or equal to the given number. Note that if there is no plan whose cost is less than or equal to the number and whose depth is shallowest among all valid plans, then no plan will be returned (even if there are deeper plans which do have cost less than or equal to the number).

- `(:which :all-shallowest :optimize-cost t)`

Under these arguments, SHOP3 returns all plans which have the shallowest depth of all valid plans and for which there is no other shallowest depth valid plan which has a lower total cost.

- `(:which :all-shallowest :optimize-cost number)`

Under these arguments, SHOP3 returns all plans which have the shallowest depth and whose total cost is less than or equal to the given number.

- `(:which :id-first)` or `(:which :id-all)`

The `:id-first` and `:id-all` options produce the same results as the *shallowest* and *all-shallowest* arguments, respectively for each different combination with `:optimize-cost`. Note, however, that there are domains for which SHOP3 will terminate using *id-first* and *id-all* and will not terminate using other values for `:which`.

- The *time-limit* argument may either nil or a number. Its default is nil and if it is nil, no time limit is imposed on the planning process. If the *time-limit* argument is a number, SHOP3 will check the elapsed CPU time at the start of each step of the planning process, and if the number of seconds elapsed is greater than the argument value, SHOP3 will immediately terminate. The main use for this feature is in combination with `(:optimize-cost t)` argument, in order to return the optimal value found within the given time limit. For example, consider the call `(find-plans 'foo :verbose 1 :optimize-cost t :time-limit 120)`. This call addresses a problem named *foo*, and runs until it either finds the minimum cost plan or until 2 minutes have elapsed. It then returns the lowest cost plan that it found during that time. This functionality is inspired, in part, by Anytime Algorithms [Dean and Boddy, 1998].
- If *explanation* is non-nil, SHOP3 adds extra information at the end of each operator explaining how the preconditions for that operator were satisfied. Currently supports only logical atoms, and, and or; it doesn't work with forall, not, eval, etc. If this feature is used with the external-access-hook feature (see Section 5.4), any attribution information provided by the external-access-hook routine is included in the relevant explanation. The default value of *explanation* is nil.
- The *plan-tree* argument defaults to nil; if true, the planner will return two additional values: (1) a list of complete task decomposition trees for the plans and (2) a list of plan state data structures corresponding to the final states of each plan. Plan trees are encoded in a nested list format in which the

decomposition of an upper level task into lower level tasks is represented by the upper level task atom, followed by trees for each lower level task. The leaves of the tree, involving operators, are each lists of three elements: the cost of the operator, the task atom for the operator, and the numerical position of the operator in the plan (starting at 1). For example, a task (travel houston springfield) that was directly decomposed into operators, (!fly houston boston) with cost 200 and (!drive boston springfield) with cost 50, would have the following plan tree:

```
((travel houston springfield)
  (200 (!fly houston boston) 1)
  (50 (!drive boston springfield) 2))
```

---

Next: [Other Debugging Features](#), Previous: [Executing the Planner](#), Up: [Running SHOP3](#) [[Contents](#)][[Index](#)]

### 3.3 Tracing

There are two functions used for controlling the tracing mechanism in SHOP3: `shop-trace` and `shop-untrace`. These are similar to Lisp's `trace` and `untrace` functions. Once they have been invoked, subsequent calls to `find-plans` or `do-problems` will print out information about elements of the domain for which tracing is enabled whenever those elements are encountered. More specifically:

- `(shop-trace item)` will turn on tracing for *item*, which may be any of the following:
  - a method, axiom, operator, task, or goal;
  - one of the keywords `:methods`, `:axioms`, `:operators`, `:tasks`, `:goals`, or `:protections` in which case SHOP3 will trace *all* items of that type (`:goals` refers to predicates that are goals for the theorem-prover, and `:protections` refers to predicates used as arguments of `:protection` in operators);
  - the keyword `:states`, in which case SHOP3 will include the current state whenever it prints out a tracing message
  - the keyword `:plans` in which case SHOP3 will print diagnostic information whenever it has found a plan (and may be considering whether or not to keep the plan, depending on the `:which` and `:optimize` arguments of `seek-plans`).
  - The keyword `:all`, which will trace all available items, currently methods, axioms, operators, tasks, goals and protections.
- `(shop-trace item1 item2 ...)` will do the same for a list of items
- `(shop-trace)` will print a list of what's currently being traced
- `(shop-untrace item)` will turn off tracing for an item

- (shop-untrace *item1 item2 ...*) will turn off tracing for a list of items
- (shop-untrace) will turn off tracing for all items

Macro: **shop-trace** *&rest items*

– (shop-trace) with no arguments will return a list of what's currently being traced.

– (shop-trace *item*) will turn on tracing for *item*.

*item* may be any of the following:

- the name of a method, axiom, operator, task, or predicate;
  - one of the keywords :methods, :axioms, :operators, :tasks, :goals, :effects, or :protections, in which case shop will trace all items of that type (:GOALS, :effects, and :protections refer to three different ways predicates can occur: as goals to be satisfied, and as effects or protections in operators);
  - a pair of the form (:task <taskname>), (:method <methodname>). shop will break when attempting to expand the task, or apply the method, respectively.
  - the keyword :states, in which case shop will include the current state whenever it prints out a tracing message
  - the keyword :all in which case shop will print out all the tracing information it knows how to.
- (shop-trace *item1 item2 ...*) will do the same for a list of items

---

Next: [Syntax Checks](#), Previous: [Tracing](#), Up: [Running SHOP3](#) [[Contents](#)] [[Index](#)]

### 3.4 Other Debugging Features

There are three variables, namely *\*current-state\**, *\*current-plan\**, and *\*current-tasks\**, in SHOP3. These variables can be used to monitor the current status of the state, current plan and the list of current tasks to be accomplished respectively. Since these are the internal variables

of the SHOP3 planning system, the following functions are defined to access the current contents of those variables: *print-current-state*, *print-current-plan*, and *print-current-tasks*, respectively. Note that these are Lisp functions that must be called by using the Lisp evaluator. The best way to use these functions is to define dedicated methods in the domain that invoke the functions using *eval* or *call* expressions in their predicates. Those methods can then be used in the problem definition where debugging output is needed. For example, the following methods can be included in any domain description for this purpose:

```
(:method (print-current-state)
  ((eval (print-current-state)))
  ())
```

```
(:method (print-current-tasks)
  ((eval (print-current-tasks)))
  ())

(:method (print-current-plan)
  ((eval (print-current-plan)))
  ())
```

And these special purpose methods can be used in the task decompositions of other methods for debugging purposes. For example,

```
(:method
  (do-both ?x ?y)
  nil
  (:ordered
    (:task !do ?y)
    (:task print-current-state)
    (:task !do ?x))))
```

There is now a new variable, *\*break-on-backtrack\**, that will cause the Lisp environment to throw into a break loop when SHOP3 backtracks.

---

Next: [Debugging Suggestions](#), Previous: [Other Debugging Features](#), Up: [Running SHOP3](#) [[Contents](#)][[Index](#)]

### 3.5 Syntax Checks

We have adopted for SHOP3 the ‘singleton variable’ check common in Prolog implementations. Logic variables are used to express unification constraints on expressions. In practice, a singleton logical variable in a SHOP3 expression (a method, operator, or axiom definition) is often a typographical error. Accordingly, SHOP3 will issue a warning when it encounters a logical variable used only once. If the single use is correct, the proper (and nicely self-documenting) way to disable this warning is to use an *anonymous variable* (see see [Symbols](#)).

---

Next: [Hook Routines](#), Previous: [Syntax Checks](#), Up: [Running SHOP3](#) [[Contents](#)][[Index](#)]

### 3.6 Debugging Suggestions

When you have a problem that does not solve as expected, the following general recipe may help you home in on bugs in your domain definition:

1. Start by doing (SHOP-TRACE :TASKS) and then try FIND-PLANS again.
2. In many cases, the domain will be written so that there will be little or no backtracking. In this case, examine the output of the traced call to FIND-PLANS and look for the first backtracking point.
3. The above process should help you identify a particular task, either a primitive or a complex task, as a likely problem spot. If it's a primitive task, the next step is to examine the operator definition. If it's a complex task, you should check the method definitions. If you have any trouble identifying which method

definition is relevant, you can use (`SHOP-TRACE :METHODS`) to further focus your attention.

4. If visual inspection of method and operator definitions does not reveal the problem, you most likely have problems with precondition expressions. In this case, try using (`SHOP-TRACE :GOALS`), rerunning `FIND-PLANS` and check to see what's happened when your problematic method or operator's preconditions are checked.

This recipe has proven effective for finding the vast majority of bugs in SHOP3 domains.

---

Previous: [Debugging Suggestions](#), Up: [Running SHOP3](#) [[Contents](#)] [[Index](#)]

### 3.7 Hook Routines

SHOP3 recognizes several different hook routines. These are Lisp routines that may be defined by the user; if they are defined, they are invoked under specific circumstances. Hook routines are typically used when embedding SHOP3 in an application; they allow such an application to obtain additional information from SHOP3 or to affect it's behavior. There are three hooks that are recognized by SHOP3:

- (`plan-found-hook` state which plan cost depth) If this routine is defined, SHOP3 invokes it whenever it finds a plan. It can be useful for displaying and/or recording details about the plan. The arguments are the current state, the value for the `:which` argument that was provided to the planner, the plan, the cost of the plan, and the search depth at which the plan was found.
- (`trace-query-hook` type item additional-information state-atoms) If this routine is defined, SHOP3 invokes it whenever it invokes the tracing mechanism (see [Tracing](#)). The arguments include the type of item being traced (e.g., `:task`, `:method`), the item, the list of Lisp values that are printed by the tracing mechanism, and a list of logical atoms in the current state.
- (`external-access-hook` query) This hook routine is intended to allow SHOP3 to use an external source (such as a database) to determine the applicability of methods and operators. To use this hook routine, a domain must include one or more logical expressions that have the keyword `:external` as the first symbol. Such expressions must only involve a single logical atom, or a single conjunction of logical atoms. When SHOP3 attempts to find a binding that satisfies such an expression, it will first invoke `external-access-hook` to satisfy the expression; if that routine is undefined or returns `nil`, SHOP3 will then try to satisfy the expression using its internal knowledge state. The argument to `external-access-hook` is a list of the form `'(and (pred val val) ...)`. It returns a list of responses, each of which is a list of two elements: an attribution and a list of bindings for the unbound variables in the query. The attribution is stored for use with the *explanation* option for the planning system (see [explanation](#)). For example, consider a method that has the following precondition:



```
(or (and (clear ?b1)
         (clear ?b2)
         (clear ?b3))
    (:external and (on ?b1 ?b2) (on ?b2 ?b3)))
```

When this precondition is encountered and external-access-hook is defined, SHOP3 invokes that routine with the argument '(and (on ?b1 ?b2) (on ?b2 ?b3)). The routine might (for example) return the list:

```
'((database-123 ((?b1 block10) (?b2 block 11) (?b3 block 12)))
  (database-223 ((?b1 block20) (?b2 block 21) (?b3 block 22))))
```

---

Next: [PDDL Support](#), Previous: [Running SHOP3](#), Up: [Top](#) [[Contents](#)] [[Index](#)]

## 4 The SHOP3 Formalism

The inputs to SHOP3 are a *planning domain* and either a single *planning problem* or a *planning problem set*. Planning domains are composed of *operators*, *methods*, and *axioms*. Planning problems are composed of *logical atoms* (an initial state) and *tasks lists* (high-level actions to perform). Planning problem sets are composed of planning problems.

The components of a planning domain (operators, methods, and axioms) all involve *logical expressions*. These logical expressions combine *logical atoms* through a variety of forms (e.g., conjunction, disjunction). Logical atoms involve a *predicate symbol* plus a list of *terms*. Task lists in planning problems are composed of *task atoms*. The components of domains and problems are all ultimately defined by various *symbols*.

This section describes each of the aforementioned structures. It is organized in a bottom-up manner because the specification of higher-level structures is dependent on the specification of lower-level structures. For example, methods are defined after logical expressions because methods contain logical expressions.

---

Next: [General Lisp Expressions](#), Previous: [The SHOP3 Formalism](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.1 Symbols

In the structures defined below, there are five kinds of symbols: **variable symbols**, **constant symbols**, **function symbols**, **primitive task symbols**, and **compound task symbols**. To distinguish among these symbols, SHOP and SHOP3 both use the following conventions:

- a **variable symbol** can be any Lisp symbol whose name begins with a question mark (such as ?x or ?hello-there)
- an **anonymous variable symbol** can be any variable symbol with an underscore immediately following the question mark in its name (such as ?\_ or ?\_airplane). These variables will unify with any value, and need not co-refer (i.e., two copies of ?\_ in a single term need not unify with each other). These variables will also not trip the singleton variable check.
- a **primitive task symbol** can be any Lisp symbol whose name begins with an exclamation point (such as !unstack or !putdown)
- a **constant symbol**, a **function symbol**, a **predicate symbol**, or a **compound task symbol** can be any Lisp symbol whose name does not begin with a question mark or exclamation point

Any of the structures defined in the remaining sections are said to be **ground** if they contain no variable symbols.

---

Next: [Terms](#), Previous: [Symbols](#), Up: [The SHOP3 Formalism](#) [[Contents](#)] [[Index](#)]

## 4.2 General Lisp Expressions

A number of SHOP3 domain structures described in this section use **general Lisp expressions**. These are arbitrary pieces of Lisp code which can include functions, macros, special macro symbols (e.g., backquote), etc. When SHOP3 needs to get the value of a general Lisp expression, it first substitutes values for any variable symbols in the expression that are bound. Then it sends the entire expression into the Lisp evaluator to get a final value.

**Note:** Counter-intuitive bugs may arise when symbols are passed to lisp for evaluation (either as constants or as the values of variables). Remember that the lisp evaluator will assume that these are variables! If you wish them to be treated as symbols, you will need to quote them. This leads to a slightly undesirable oddity — variables that will be bound to, for example, numbers, can appear normally. Variables that will be bound to symbols will have to be quoted. See the discussion of Eval terms, below (Section 4.3.2).

---

Next: [Logical Atoms](#), Previous: [General Lisp Expressions](#), Up: [The SHOP3 Formalism](#) [[Contents](#)] [[Index](#)]

## 4.3 Terms

A **term** is any one of the following:

- a variable symbol
- a constant symbol
- a number
- a **list-term**
- an **eval-term**
- a **call-term**

---

Next: [Eval Terms](#), Previous: [Terms](#), Up: [Terms](#) [[Contents](#)][[Index](#)]

### 4.3.1 List Terms

A **list-term** is a term having the form

$$([list] \ t_1 \ t_2 \ \dots \ t_n \ [. \ l])$$

where *list* is an optional reserved word and each  $t_i$  is a term. This specifies that  $t_1 \ t_2 \ \dots \ t_n$  are the items of a list. If the final, optional element is included, the item *l* should evaluate to a list; all items in *l* are included in the list after  $t_1$  through  $t_n$ .

---

Next: [Call-terms](#), Previous: [List Terms](#), Up: [Terms](#) [[Contents](#)][[Index](#)]

### 4.3.2 Eval Terms

An **eval-term** is an expression of the form

$$(\text{eval } \textit{general-lisp-expression})$$

The value associated with an eval-term is determined as follows. First, any variable symbols which appear in *lisp-expression* and are bound are replaced by the values that they are bound to. Then, the entire expression is evaluated in Lisp. For example, if the variable symbol ?foo is bound to the value 3 then the term:

$$(\text{eval } (\text{mapcar } \#'(\text{lambda } (x) (+ x \text{ ?foo})) \ `(1 \ 2 \ ,(* \text{ ?foo } \text{ ?foo}))))$$

will have as its value a list containing the numbers 4, 5, and 12. Note that variable substitutions in eval terms are handled before any evaluation of the expression, as in Lisp macros. One implication of this fact is that variables with symbolic values must be explicitly quoted if they are to be treated as Lisp symbols. For example, if the variable ?foo is bound to the symbol BAR, the following eval term has the value (BAR BAZ):

$$(\text{eval } (\text{list } '\text{?foo } '\text{BAZ}))$$

if this were written

```
(eval (list ?foo 'BAZ))
```

it would cause a Lisp error when lisp attempts to find the value of BAR, which it would believe to be a variable.

---

Previous: [Eval Terms](#), Up: [Terms](#) [[Contents](#)][[Index](#)]

### 4.3.3 Call-terms

A **call-term** is an expression of the form

```
(call f t1 t2 ... tn)
```

where  $f$  is a function symbol and each  $t_i$  is a term or a call-term. A call-term has a special meaning to SHOP3, because it tells SHOP3 that  $f$  is an attached procedure, i.e., that whenever SHOP3 needs to evaluate a precondition or task list that contains a call-term, SHOP3 should replace the call term with the result of applying the function  $f$  on the arguments  $t_1, t_2, \dots, t_n$ . (We later will define what preconditions and task lists are).

For example, the following call-term would have the value 6:

```
(call + (call + 1 2) 3)
```

Note that a call-term is not as expressive as an eval-term. In particular, it does not support the evaluation of Lisp macros (including macro characters such as backquote). Both call and eval are supported in SHOP3 because the former is compatible with JSHOP 1.0 and the latter is compatible with SHOP 1.x. SHOP3 users who are not interested in either form of compatibility may use either form.

---

Next: [Logical Expressions](#), Previous: [Terms](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.4 Logical Atoms

A **logical atom** has the form:

```
(p t1 t2 ... tn)
```

where  $p$  is a predicate symbol and each  $t_i$  is a term other than an eval- or call-term.

---

Next: [Logical Precondition](#), Previous: [Logical Atoms](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.5 Logical Expressions

A **logical expression** is a logical atom or any of the following complex expressions: **conjuncts**, **disjuncts**, **negations**, **implications**, **universal quantifications**, **assignments**, **eval expressions**, **call expressions**.

---

Next: [Disjuncts](#), Previous: [Logical Expressions](#), Up: [Logical Expressions](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.5.1 Conjuncts

A **conjunct** has the form

$$([\text{and}] \ l_1 \ l_2 \ \dots \ l_n)$$

where each  $l_i$  is a logical expression. Note that if there are 0 conjuncts (e.g., the expression is  $()$ ) then the form always evaluates to true.

---

Next: [Negations](#), Previous: [Conjuncts](#), Up: [Logical Expressions](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.5.2 Disjuncts

A **disjunct** is an expression of the form

$$([\text{or}] \ l_1 \ l_2 \ \dots \ l_n)$$

where  $l_1, l_2, \dots, l_n$  are logical expressions.

---

Next: [Implications](#), Previous: [Disjuncts](#), Up: [Logical Expressions](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.5.3 Negations

A **negation** is an expression of the form

$$([\text{not}] \ l)$$

where  $l$  is a logical expression.

---

Next: [Universal Quantifications](#), Previous: [Negations](#), Up: [Logical Expressions](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.5.4 Implications

An **implication** is an expression of the form

$$([\text{imply}] \ Y \ Z)$$

where  $Y$  and  $Z$  are logical expressions. The intent of an implication is to evaluate its logical counterpart; that is,  $(\hat{A} \rightarrow Y \hat{A} \sim Z)$ . Note that here,  $Y$  should be ground, or the semantic of the implication will be ambiguous.

---

Next: [Assignments](#), Previous: [Implications](#), Up: [Logical Expressions](#) [\[Contents\]](#)[\[Index\]](#)

#### 4.5.5 Universal Quantifications

A **universal quantification** expression is an expression of the form

$$(\text{forall } V E_1 E_2)$$

where  $E_1$  and  $E_2$  are logical expressions, and  $V$  is the list of variables in  $E_1$ . To satisfy a **universal quantification** expression, the following must hold: for each possible substitution  $u$  for variables in  $V$ , if  $E_1^u$  is satisfied then  $E_2^u$  must also be satisfied in the current state of the world. Note that this use of the keyword `forall` is distinct from its use in `add` and `delete` lists in operators (see [Operators](#)): the latter is used to express a set of effects rather than a logical expression and consequently has a different syntax.

---

Next: [Eval expressions](#), Previous: [Universal Quantifications](#), Up: [Logical Expressions](#) [\[Contents\]](#)[\[Index\]](#)

#### 4.5.6 Assignments

A simple **assignment** expression has the form

$$(\text{assign } v e)$$

where  $v$  is a variable symbol and  $e$  is general Lisp expression. The intent of an assignment expression is to bind the value of  $e$  to the variable symbol  $v$ . Variable substitutions in assignment expressions are done using literal substitutions, as with `eval` terms (see Section 4.3.2). For example, if `?foo` is bound to the symbol `IF` and `?bar` is bound to the number 0 then the following expression will bind the variable `?baz` to the list `(IF FISH)`:

$$(\text{assign } ?\text{baz } (?foo (< ?bar 3) (\text{list } '?foo 'fish) (/ 8 ?bar))))$$

Similarly, if `?foo` is bound to `LIST` and `?bar` is bound to 4 then the expression above will bind `?baz` to the list `(NIL (LIST FISH) 2)`.

SHOP3 also offers a **compound assignment** expression of this form:

$$(\text{assign* } v e)$$

As in the simple `assign`,  $v$  is a variable symbol and  $e$  is general Lisp expression. However, for `assign*`,  $e$  should evaluate to a *list* of possible values and through backtracking, SHOP3's theorem-prover will find all solutions with  $v$  bound to the various values of  $e$ , in turn.

---

Next: [Call-expressions](#), Previous: [Assignments](#), Up: [Logical Expressions](#) [\[Contents\]](#)[\[Index\]](#)

### 4.5.7 Eval expressions

An **eval-expression** has the same form as an [eval-term](#), q.v. Unlike an eval-term, however, an eval-expression is interpreted simply as either true or false rather than having some value which would be used as an argument to a predicate. Thus eval-expression typically invoke boolean Lisp functions such as `evenp` or `>=`.

---

Next: [Enforce-expressions](#), Previous: [Eval expressions](#), Up: [Logical Expressions](#) [[Contents](#)][[Index](#)]

### 4.5.8 Call-expressions

A **call-expression** has the same form as a [call-term](#), q.v. As with call-expressions, eval-expressions are interpreted as either true or false.

---

Next: [Setof-expressions](#), Previous: [Call-expressions](#), Up: [Logical Expressions](#) [[Contents](#)][[Index](#)]

### 4.5.9 Enforce-expressions

An **enforce-expression** has the form

```
(enforce  $t_1$  &rest error-args)
```

Enforce expressions are for goals that should *always* be satisfied. SHOP's theorem-prover will attempt to prove  $t$  and if it fails, will call error with *error-args*. For example

```
(enforce (x-position ?aircraft)
  "~A x-position undefined." (quote ?aircraft))
```

---

Next: [Bagof-expressions](#), Previous: [Enforce-expressions](#), Up: [Logical Expressions](#) [[Contents](#)][[Index](#)]

### 4.5.10 Setof-expressions

A **setof-expression** has the form

```
(setof var expr set-var)
```

Find all solutions to *expr*, and bind the *set of* values for *var* in *expr* to *set-var*. For example

```
(setof ?uav (uav ?uav) ?uavs)
```

will bind `?uavs` to the set of UAVs in the current state.

Note that the semantics of this operator are to fail if the *expr* is an unsatisfiable goal.

---

Previous: [Setof-expressions](#), Up: [Logical Expressions](#) [[Contents](#)][[Index](#)]

### 4.5.11 Bagof-expressions

The syntax for `bagof` is the same as for `setof` (Section 4.5.10), but binds *set-var* to a *bag* of results, which may contain duplicates, instead of a set.

---

Next: [Axioms](#), Previous: [Logical Expressions](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.6 Logical Precondition

A **logical precondition** is either a logical expression or one of the following special precondition forms: **first satisfier precondition**, **sorted precondition**.

---

Next: [Sorted Precondition](#), Previous: [Logical Precondition](#), Up: [Logical Precondition](#) [[Contents](#)][[Index](#)]

### 4.6.1 First Satisfiers Precondition

A **first satisfier precondition** has the form

$$(:\text{first } l_1 \ l_2 \ \dots \ l_n)$$

where each  $l_i$  is a logical expression. Such a precondition causes SHOP3 to consider only the first set of bindings that satisfies all of the given expressions. Alternative bindings will not be considered even if the first bindings found do not lead to a valid plan.

---

Previous: [First Satisfiers Precondition](#), Up: [Logical Precondition](#) [[Contents](#)][[Index](#)]

### 4.6.2 Sorted Precondition

A **sorted precondition** has the form

$$(:\text{sort-by } ?v \ [e] \ l)$$

where  $?v$  is a variable symbol,  $e$  is a general Lisp expression (which should evaluate to a comparison function), and  $l$  is a logical expression. Such a precondition causes SHOP3 to consider bindings for the precondition in a specific order. Specifically, bindings are sorted such that if the specified comparison function holds between values  $x$  and  $y$  then bindings that bind  $?v$  to  $x$  may not occur after bindings that bind  $?v$  to  $y$ . For example consider the precondition:

$$(:\text{sort-by } ?d \ #'> \ (\text{and } (\text{at } ?\text{here}) \ (\text{distance } ?\text{here } ?\text{there } ?d)))$$

This precondition will cause SHOP3 to consider bindings in decreasing (high to low) order of the value of  $?d$ . If the comparison function ( $e$ ) is omitted, it defaults to  $\#'$ , indicating increasing (low to high) order.

---

Next: [Task Atoms](#), Previous: [Logical Precondition](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]



## 4.7 Axioms

An **axiom** is an expression of the form

```
(:- a
  [name1] E1
  [name2] E2
  [name3] E3 ...
  [namen] En)
```

where the axiom's **head** is the symbol  $a$ , and its **tail** is the list  $([name_1] E_1 [name_2] E_2 [name_3] E_3 \dots [name_n] E_n)$  and each  $E_i$  is a logical expression and each  $name_i$  is a symbol called the *name* of  $E_i$ . The names of the expressions are optional. When a domain definition is loaded into SHOP3, a unique name will be generated for each conjunct if no name was given. These names have no semantic meaning to SHOP3, but are provided to help the user debug domain descriptions by looking at traces of SHOP3's behavior.

The intended meaning of an axiom is that  $a$  is true if  $E_1$  is true, or if  $E_1$  is false but  $E_2$  is true, or if all of  $E_1, E_2, \dots, E_{n-1}$  are false but  $E_n$  is true. For example, the following axiom says that a location is in walking distance if the weather is good and the location is within two miles of home, or if the weather is not good and the location is within one mile of home:

```
(:- (walking-distance ?x)
    good ((weather-is good) (distance home ?x ?d) (call <= ?d 2))
    bad ((distance home ?x ?d) (call <= ?d 1)))
```

---

Next: [Task Lists](#), Previous: [Axioms](#), Up: [The SHOP3 Formalism](#)  
[\[Contents\]](#)[\[Index\]](#)

## 4.8 Task Atoms

A **task atom** is an expression of any of the forms

```
(s t1 t2 ... tn)

(:task s t1 t2 ... tn)

(:task :immediate s t1 t2 ... tn)
```

where  $s$  is a task symbol and the arguments  $t_1, t_2, \dots, t_n$  are terms. The task atom is **primitive** if  $s$  is a primitive task symbol, and it is **compound** if  $s$  is a compound task symbol. The first and second forms are called an **ordinary task atom**; the third form is called an **immediate task atom**. The purpose of the `:immediate` keyword is to give a higher priority to the task, as described in the following subsection.

---

Next: [Operators](#), Previous: [Task Atoms](#), Up: [The SHOP3 Formalism](#)  
[\[Contents\]](#)[\[Index\]](#)

## 4.9 Task Lists

A **task list** is any of the following:

- an expression of the form  $(:unordered\ tasklist_1\ tasklist_2\ \dots\ tasklist_n)$ , where  $tasklist_1\ tasklist_2\ \dots\ tasklist_n$  are task lists;
- an expression of the form  $([:ordered]\ tasklist_1\ tasklist_2\ \dots\ tasklist_n)$ , where  $tasklist_1\ tasklist_2\ \dots\ tasklist_n$  are task lists.
- A task atom, see 4.8.

The `:ordered` keyword, which is optional, specifies that SHOP3 must perform the task lists in the order that they are given. The `:unordered` keyword specifies that there is no particular ordering specified between  $tasklist_1, tasklist_2 \dots tasklist_n$ . With the use of the `:unordered` keyword, SHOP3 may interleave tasks between different task lists. Suppose we have two task lists as the following:

$$T = (:ordered\ t_1\ t_2\ \dots\ t_m);$$

$$U = (:ordered\ u_1\ u_2\ \dots\ u_n);$$

and that we have the main task list

$$M = (:unordered\ T\ U).$$

If none of the tasks have the `:immediate` keyword, then the tasks in  $T$  should be performed in the order given, and the tasks in  $U$  should also be performed in the order given—but it is permissible for SHOP3 to interleave the tasks of  $T$  and the tasks of  $U$ . However, if some of the tasks are immediate, then each time SHOP3 chooses the next task to accomplish, it needs to give a higher priority to the immediate tasks. For example, if  $t_1$  is immediate and  $u_1$  is not immediate, then SHOP3 should perform  $t_1$  before both  $t_2$  and  $u_1$ .

Note: A task with the `:immediate` keyword specifies that this task must be performed immediately when it has no predecessors. Therefore, we can allow only one task with the `:immediate` keyword in the list of tasks that have no predecessors. Otherwise, SHOP3's behavior on those tasks is undefined. In other words, it is not allowed to have two tasks in an `:unordered` list and both have the `:immediate` keyword. For instance, on the example above,  $t_1$  and  $u_1$  cannot both have the `:immediate` keyword.

---

Next: [Methods](#), Previous: [Task Lists](#), Up: [The SHOP3 Formalism](#)  
[\[Contents\]](#)[\[Index\]](#)

## 4.10 Operators

The arguments marked with keywords are all optional, and may appear in any order. The add-list, and delete-list default to being empty. The preconditions default to being vacuously true, and the cost defaults to 1.0.

*Note:* We still support SHOP2's original syntax for operators, described below (see [Operators Legacy Syntax](#)).

An **operator** is description of how to perform a *primitive* task, which cannot be decomposed further. An operator definition has the following form:

```
(:op head
  [:add add-list]
  [:delete delete-list]
  [:precond precondition]
  [:cost cost-fn])
```

where

- *head* (the operator's **head**) is a primitive task atom (i.e., a task atom with a task symbol that begins with an exclamation point)
- The operator's *precondition* is a logical expression.
- The operator's *delete-list* is a list for which each of the element may be any of following:
  - a logical atom
  - a protection condition (see below)
  - an expression of the form (forall *V E L*), where *V* is a list of variables in *E*, *E* is a logical expression, and *L* is a list of logical atoms
- The operator's *add-list* is a list of logical atoms that has the same form as the delete-list.
- *cost-fn* (the operator's **cost**) is a general Lisp expression. If *c* is omitted, the cost defaults to 1.

In the above definition, a **protection condition** is an expression of the form

```
(:protection a)
```

where *a* is a logical atom. The purpose of a protection condition in the add list is to tell SHOP3 that it should not execute any operator that deletes *a*. The purpose of a protection condition in the delete list is to cancel a previously added protection condition. For example, if we drive a delivery truck to a certain location in order to pick up a package, then we might not want to allow the truck to be moved away from that location until after we have picked up the package. To represent this, we might use the following operators:

```
(:op (!drive-to ?truck ?old-loc ?location)
  :delete ((at ?truck ?old-loc))
  :add ((at ?truck ?location)
        (:protection (at ?truck ?location))))
(:op (!pick-up ?truck ?package ?location)
  :delete
    ((at ?package ?location)
     (:protection (at ?truck ?location)))
  :add ((in ?package ?truck)))
```

As noted above, the head of the operator is a primitive task atom, so it must begin with a primitive task symbol, i.e., a symbol that begins with an exclamation point. Note that operator names which begin with *two* exclamation points have a special meaning in SHOP3; operators of this sort are known as **internal operators**. Internal operators are ones which are used for purposes internal to the planning process and are not intended to correspond to actions performed in the plan (e.g., to do some computation which will later be useful in deciding what actions to perform). Other than

requiring two exclamation points at the start of the name, the syntax for internal operators is identical to the syntax for other operators. SHOP3 handles internal operators exactly the same way as ordinary operators during planning. SHOP3 includes these operators in any plans that it returns at the end of execution. It may, however, omit them from the printout of the final plan (depending on the value of the `:verbose` argument (see [verbose](#))).

The primary reason that the internal operator syntax exists in SHOP3 is so that automated systems which use SHOP3 plans as an input can easily distinguish between those operators which involve action and those which were merely internal to the planning process.

When designing an operator, it is important to ensure that each variable symbol in the add list, delete list, and cost always be bound to a single value when the operator is invoked. Variable symbols can be bound in the head of the operator (by the method that invokes the associated primitive task) or in the precondition of the operator. An operator should be written such that for any variable appearing after the precondition, no two unifiers of the precondition have different bindings for that variable. SHOP3 does not check this requirement; if conflicting unifiers are available when applying an operator, it will apply one arbitrarily. This can lead to unpredictable behavior and plans with ambiguous semantics. In general, we recommend that operator preconditions be designed such that only one unifier is possible. However, SHOP3 will be able to correctly process operators that have multiple unifiers for preconditions as long as no two unifiers can provide different values for a variable that appears in the add list, delete list, or cost.

---

Previous: [Operators](#), Up: [Operators](#) [[Contents](#)][[Index](#)]

#### 4.10.1 Operators: Legacy Syntax

In our work with SHOP2, we found that operator definitions in its original syntax were prone to hard-to-detect syntax errors and typos that can give rise to difficult to identify garbage in/garbage out bugs. Particularly prevalent are hard-to-identify bugs arising when a programmer inadvertently reverses the order of add and delete lists in a SHOP3 operator. These problems are exacerbated by the extreme permissiveness of SHOP3's parser. This led us to the new syntax, described above, which relies on keywords to make operator definitions more readable, and less error-prone. Our new syntax also supports arbitrary order and the omission of empty components, without the "DWIM" parsing in SHOP2. Because of its many advantages, so we encourage you to adopt the new syntax, instead of continuing to use the classical form described here, although it is still supported.

The original operator definition syntax was as follows:

```
(:operator head precondition delete-list add-list [c])
```

The two operators described above are written in the old syntax as follows:

```
(:operator (!drive-to ?truck ?old-loc ?location)
  ())
```

```

      ((at ?truck ?old-loc))
      ((at ?truck ?location)
       (:protection (at ?truck ?location))))
  (:operator (!pick-up ?truck ?package ?location)
   ()
   ((at ?package ?location)
    (:protection (at ?truck ?location)))
   ((in ?package ?truck)))

```

For backwards compatibility with SHOP 1.x, SHOP3 will also accept operators where the precondition  $P$  is missing. In this case the domain definition pre-processing code puts a null precondition into the operator, which is always satisfied. *SHOP3's ability to recognize operators without preconditions is deprecated and is likely to disappear in the future.*

---

Next: [Planning Domain](#), Previous: [Operators](#), Up: [The SHOP3 Formalism](#)  
[\[Contents\]](#)[\[Index\]](#)

## 4.11 Methods

A **method** is a list of the form

```
(:method  $h$  [ $n_1$ ]  $C_1$   $T_1$  [ $n_2$ ]  $C_2$   $T_2$  ... [ $n_k$ ]  $C_k$   $T_k$ )
```

where

- $h$  (which is called the method's **head**) is a task atom in which no *call-* or *eval-terms* can appear;
- Each  $C_i$  (which is called a **precondition** for the method) is a logical precondition.
- Each  $T_i$  (which is called a **tail** of the method) is a task list. The task atoms in the list can contain call-terms.
- Each  $n_i$  is the *name* for the succeeding  $C_i$   $T_i$  pair. These names are optional and if omitted a unique name will be assigned for each pair. These names have no semantic meaning to SHOP3, but are provided in order to help the user debug domain descriptions by looking at traces of SHOP3's behavior.

A method indicates that the task specified in the method's head can be performed by performing all of the tasks in one of the methods tails when one that tail's precondition is satisfied. Note that the preconditions are considered in the given order, and a later precondition is considered *only* if all of the earlier preconditions are not satisfied. If there are multiple methods for a given task available at some point in time, all of these methods can be considered. Consequently, the following code:

```

(:method (eat ?food)
  (have-fork ?fork)
  ((!eat-with-fork ?food ?fork))
  (have-spoon ?spoon)
  ((!eat-with-spoon ?food ?spoon))

```

is semantically equivalent to the following code with multiple methods and explicitly exclusive preconditions:

```
(:method (eat ?food)
```

```
(have-fork ?fork)
((!eat-with-fork ?food ?fork)))

(:method (eat ?food)
  (and (not (have-fork ?fork)) (have-spoon ?spoon))
  (!eat-with-spoon ?food ?spoon))
```

In both of the above examples, the `!eat-with-spoon` operator may be performed only if the `(have-spoon ?spoon)` is satisfied *and* `(have-fork ?fork)` is not satisfied.

---

Next: [Planning Problem](#), Previous: [Methods](#), Up: [The SHOP3 Formalism](#)  
[\[Contents\]](#)[\[Index\]](#)

## 4.12 Planning Domain

A **planning domain** is an object that contains all of the information for solving a class of planning problems<sup>2</sup>. At a minimum, it will include definitions of the operators (or actions) and methods available in the domain. A planning domain definition may also contain axioms, or other items that are accepted by specific SHOP domain extensions. Finally, a domain definition can *include* other domains by reference (see , below).

---

Next: [Extended form](#), Previous: [Planning Domain](#), Up: [Planning Domain](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.12.1 Simple Form

A planning domain definition in the simple form looks like this:

```
(defdomain domain-name (i1 i2 ... in))
```

where *domain-name* is a symbol (which does not need to be quoted). Beginning users of SHOP should simply use the simple domain-name form of this argument.

Each item *i<sub>j</sub>* is one of the following: an operator, a method, or an axiom.

---

Next: [Inclusion directives](#), Previous: [Simple Form](#), Up: [Planning Domain](#)  
[\[Contents\]](#)[\[Index\]](#)

### 4.12.2 Extended form

The extended form of the SHOP3 domain definition looks like this:

```
(defdomain (domain-name &rest args) (i1
i2 ... in))
```

*args* includes the following keyword arguments:

- a `:type` keyword argument, allowing the domain modeler to

indicate a specific subclass of the SHOP3 domain class. E.g., one might have (*my-domain* :type *pddl-domain*).

- A `:redefine-ok` argument. If this is NIL (the default), `defdomain` will warn when the domain *domain-name* is already defined.
- A `:noset` argument. Currently this defaults to NIL, to provide for backward compatibility, but I would like to see this move to defaulting to T.

This is actually a bit of a kludge. The existing `defdomain` form, as a side-effect, sets the global variable *\*domain\**. If this were only a default domain name, that would be fine, but it is used everywhere as a special variable to mean “the domain within which we are planning.” So if there’s concurrent action, or there are multiple copies of SHOP (or its component libraries) running in a single Lisp image, bad things can happen. I would like to stamp out the use of *\*domain\** as a default domain.

The question of which additional arguments are accepted in *args* is a matter for the implementer of the specialized domain type being used. Any additional arguments will be passed to the `make-instance` method for the domain class.<sup>3</sup> SHOP3 extenders can create new subclasses of domain that accept initialization arguments. A first example of the use of this is the built-in `pddl-domain` class.

If you are using the extended form of `defdomain`, you should have in hand a new SHOP3 domain subclass, with a description of its arguments. If you do not, you should ignore the extended form.

---

Previous: [Extended form](#), Up: [Planning Domain](#) [[Contents](#)][[Index](#)]

### 4.12.3 Inclusion directives

A domain definition can include the items of another domain by reference using the `include` directive:

```
(:include domain-name file-name)
```

for example

```
(:include flight-operators
  #.(asdf:system-relative-pathname "core-domains" "domains.lisp"))
```

would take the text of the `flight-operators` domain, which should be found in the `domains.lisp` file. Note the use of the reader evaluation form – `#.` – to force evaluation of the expression that yields the pathname.

---

Next: [Planning Problem Set](#), Previous: [Planning Domain](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.13 Planning Problem

A **planning problem** has the form

```
(defproblem problem-name domain-name (a1 a2 ... an) T)
```

where *problem-name* is a symbol, *domain-name* is a symbol, each  $a_i$  is a ground logical atom, and  $T$  is a task list. This form defines a problem which may be solved by addressing the tasks in  $T$ , using the operators, methods and axioms in *domain-name*, starting in an initial state defined by the atoms  $a_1$  through  $a_n$ .

Macro: **defproblem** *problem-name* &rest *args*

(defproblem {<name>|<name-and-options>} <state> <tasks>) For backward compatibility, will support also (defproblem <name> <domain-name> <state> <tasks>).

The corresponding functional interface is:

Function: **make-problem** *problem-name-etc* *state* *tasks* &rest *extras*

make-problem creates a planning problem named *problem-name* by putting *state* and *task* onto PROBLEM-NAME's property list under the indicators :state and :tasks. As a side effect, sets the global variable \*problem\*.

---

Next: [Plans](#), Previous: [Planning Problem](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.14 Planning Problem Set

A **planning problem set** has the form

(def-problem-set *set-name* ( $p_1$   $p_2$  ...  $p_n$ ))

where *set-name* is a symbol and each  $p_i$  is the name of a planning problem.

---

Previous: [Planning Problem Set](#), Up: [The SHOP3 Formalism](#) [[Contents](#)][[Index](#)]

## 4.15 Plans

The previous subsections describe the inputs to SHOP3. This subsection describes the result that SHOP3 produces. A **plan** is a list of the form

( $h_1$   $c_1$   $h_2$   $c_2$  ...  $h_n$   $c_n$ )

where each  $h_i$  and  $c_i$ , respectively, are the head and the cost of a ground operator instance  $o_i$ . If  $p = (h_1 c_1 h_2 c_2 \dots h_n c_n)$  is a plan and  $S$  is a state, then  $p(S)$  is the state produced by starting with  $S$  and executing  $o_1, o_2, \dots, o_n$  in the order given. The **cost** of the plan  $p$  is  $c_1 + c_2 + \dots + c_n$  (thus, the cost of the empty plan is 0).

Often what is wanted is a truncated version of the plan sequence, with the *internal operators* (see [internal operators](#)) and costs removed. For this, one may use

Function: **shorter-plan** *plan*



Removes the internal operators and costs from a plan sequence, and returns the resulting new sequence. Non-destructive.

---

Next: [The SHOP Theorem Prover](#), Previous: [The SHOP3 Formalism](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 5 PDDL Support

The current release of SHOP3 provides a preliminary capability to incorporate PDDL domain definitions into a SHOP3 domain. You should be able to incorporate components of a PDDL domain definition into a SHOP3 domain definition of :type pddl-domain or simple-pddl-domain. A pddl-domain corresponds to a PDDL domain of the ADL type. The pddl-domain uses conditional-effects, existential-preconditions, universal-preconditions, and equality (note that these are *PDDL* conditional effects, existential preconditions and universal preconditions; these are *not* SHOP-syntax condition effects, etc.). A simple-pddl-domain will not have conditional-effects, existential-preconditions, universal-preconditions, or equality.

Currently the PDDL integration is a little bumpy, and details are in flux.

PDDL action names are translated into names that SHOP will recognize as primitives (e.g., move would become !move, and would have to be referenced that way in SHOP method definitions that use it).

You must splice the PDDL domain components into the SHOP domain definition. Typically, the best way to do this is to use an :include form (see ).

Fully ground STRIPS-style domains – i.e., those where a set of STRIPS operators has been exploded into a large set of ground operators by some automated transformation – will work very poorly.

Note that the parsing of PDDL domains in SHOP is not strict. This is intentional, because we don't want to make it impossible to include SHOP constructs together with PDDL constructs. However, there should probably be a *strict mode* that checks for true conformance with PDDL syntax.

---

Previous: [PDDL Support](#), Up: [PDDL Support](#) [[Contents](#)][[Index](#)]

### 5.1 PDDL Methods

In order to support replanning, we have developed *PDDL methods* that have more limited expressive power than normal SHOP3 methods, and that have clearer semantics. This feature is *experimental*: the syntax, semantics, and implementation are all likely to change.

PDDL method proposed syntax:

```
(:pddl-method head[i] precond[i] body[i])
```

As in normal SHOP3 methods (see Section 4.11), a PDDL Method's head is

a task. We notate it here as  $head_{ij}$  in order to indicate that some set of variables,  $\bar{I}$ , may appear free in the head. Similarly, we have  $precond_{ij}$  and  $body_{ij}$  to indicate that  $\bar{I}$  may appear free in the preconditions and the body. A PDDL method's preconditions must be an enhanced PDDL goal expression (see below). A PDDL method's body must be an ordered task network. In the interests of clean syntax, task network keywords (:ordered and :unordered) are *not* optional, and must be provided. For the moment, only ordered task networks are supported; unordered task networks may be supported later.

Note that provision for free variables does not relax the constraint that all operators must be ground when inserted into the plan. The intention is to allow the variables in  $\bar{I}$  to be bound by the preconditions. As in PDDL operators, only the variables in the head of a PDDL method are scoped over the preconditions and the body (in standard SHOP3 methods we have Prolog-style scoping where any free variable is implicitly scoped over the entire method).

Note also that PDDL methods do *not* permit multiple precondition sets and task networks in a single PDDL method form, and so do *not* support the if-then-else semantics of SHOP3 method forms. To get this kind of semantics in PDDL methods, the programmer must supply a set of methods with mutually-exclusive and exhaustive preconditions.

PDDL methods may only be used in domains that are a subclass of SIMPLE-PDDL-DOMAIN (see Section 8.1.2).

---

Next: [PDDL Domain Classes](#), Previous: [PDDL Methods](#), Up: [PDDL Methods](#) [[Contents](#)][[Index](#)]

### 5.1.1 Enhanced Preconditions

PDDL method preconditions are based on PDDL *goals*, but with the following extensions:

- A PDDL method's preconditions may use the :sort-by keyword. More precisely:

$P$	$::=$ PDDL goal	
$v$	$::=$ PDDL/SHOP variable	
$P_v$	$::=$ PDDL goal with $v$ appearing free	
precond	$::= P \mid$	$P_v$ with $v$ a method parameter $\mid$ (:sort-by $v$ [<rel>] $P_v$ )

- Variables that appear free in the head of the PDDL method may appear free in the precondition, hence the use of  $P$  and  $P_v$ .

PDDL goals are as described in the PDDL syntax (we have used the PDDL 2.1 paper as primary source). Note that PDDL goals with free variables are permitted, as long as the free variables appear in the parameters for the task.

Currently we do not support typing in PDDL method task heads, but this could change.

Previous: [Enhanced Preconditions](#), Up: [PDDL Methods](#) [[Contents](#)][[Index](#)]

### 5.1.2 PDDL Domain Classes

In order to implement the requirements of PDDL, SHOP3 contains a number of classes, which can be used when defining domains with `defdomain` (see [Planning Domain](#)).

- `SIMPLE-PDDL-DOMAIN` – this is the base class. All PDDL domain classes should include this class.
- `PDDL-DOMAIN` – for historical reasons, this class corresponds roughly to PDDL domains with the ADL requirements. Use of this class is *deprecated*; it was added without sufficient thought.
- `ADL-DOMAIN` – `SIMPLE-PDDL-DOMAIN` with `ADL-MIXIN` (see below).
- Mixin classes for different requirements:
  - `NEGATIVE-PRECONDITIONS-MIXIN`
  - `DISJUNCTIVE-PRECONDITIONS-MIXIN`
  - `UNIVERSAL-PRECONDITIONS-MIXIN`
  - `EXISTENTIAL-PRECONDITIONS-MIXIN`
  - `QUANTIFIED-PRECONDITIONS-MIXIN` – this simply inherits universal and existential preconditions, since they are typically used together.
  - `CONDITIONAL-EFFECTS-MIXIN`
  - `EQUALITY-MIXIN`
  - `PDDL-TYPING-MIXIN`
  - `COSTS-MIXIN`
  - `ADL-MIXIN`

---

Next: [The SHOP Unifier](#), Previous: [PDDL Support](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 6 The SHOP Theorem Prover

One of the main extensions that SIFT has made to SHOP3 is to make its theorem prover usable separately. The SHOP3 theorem-prover is very similar to Prolog, but operates over a state that changes over time. Note, however, that the prover does not (at least not yet) support temporal modalities such as “previously,” “next,” or “always.”

The theorem-prover is supplied only as a toolbox unlike the unifier (see [The SHOP Unifier](#)), which provides functionality that is ready to use “out-of-the box.” Specifically, it does not contain any built-in functionality for advancing the world state; those need to be developed by the

programmer, based on facilities supplied by SHOP3's "common" subsystem. The theorem-prover makes relies critically on the unifier subsystem, and re-exports from the theorem-prover package most of the key functions from the unifier package.

The top-level function of the theorem-prover is *query*, which is built on top of the more difficult-to-use *find-satisfiers*:

Generic Function: **query [shopthpr] goals state &key just-one domain record-dependencies**

More convenient top-level alternative to *find-satisfiers*. Manages optional arguments and ensures that the variable property is properly set in goals.

Function: **find-satisfiers [shopthpr] goals state &optional just-one level &key domain**

Find and return a list of binding lists that represents the answer to goals (a list representation of a deductive query), where state provides the database. level is a non-negative integer indicating the current depth of inference, used in debugging prints, and just-one is a boolean indicating whether you want just one answer (non-nil) or all answers (nil).

The theorem prover has its own notion of a *domain* object, but unlike the full SHOP3 domain, this one supports only specification of axioms (although it can be extended by the programmer).

Class: **thpr-domain [shopthpr]**

Class precedence list: thpr-domain, has-axioms-mixin, standard-object, t

An object representing a SHOP3 theorem prover domain.

---

Next: [Plan Grapher](#), Previous: [The SHOP Theorem Prover](#), Up: [Top](#)  
[\[Contents\]](#)[\[Index\]](#)

## 7 The SHOP Unifier

The SHOP3 unifier may be loaded stand-alone with

```
(asdf:load-system "shop3/unifier")
```

It provides a library that may be useful for general symbolic computation that requires unification of first order (or, indeed, higher order) logical expressions represented as lisp s-expressions.

Function: **unify [shopunif] e1 e2**

Checks to see whether or not E1 and E2 unify, returning a substitution (a binding list) if they do, or fail (\*not\* nil) if they don't.

The unify function is also exported from the SHOP3 theorem-prover package and the top level SHOP3 package.

To check the results of unification, SHOP3 offers these predicates:

Macro: **unify-p** [**shopunif**] *e1 e2*

It's painful (and bug-inducing) to have to remember to compare the result of unify with 'fail. This checks to see whether E1 and E2 unify, and returns t or nil, accordingly.

Macro: **unify-fail** [**shopunif**] *e1 e2*

It's painful (and bug-inducing) to have to remember to compare the result of unify with 'fail.

Macro: **unify-fail-p** [**shopunif**] *e1 e2*

Better-named alias for unify-fail.

A *binding-list* or *substitution* is a list of *bindings*:

Structure: **binding** [**shopunif**]

Class precedence list: binding, structure-object, t

A binding is a structure with a variable and a value.

Accessors:

Function: **binding-val** [**shopunif**] *instance*

Binding structure accessor

Function: **binding-var** [**shopunif**] *instance*

Binding structure accessor

Utility functions:

Function: **make-binding-list** [**shopunif**] *variables values*

Takes a list of variables and values and returns a binding-list.

Function: **binding-list-value** [**shopunif**] *var binding-list &optional if-not-found*

Return the value of var in binding-list. When there is no such value, raises an error if if-not-found is :error, else returns the value of if-not-found.

To apply a unifier in a target expression, use apply-substitution:

Macro: **apply-substitution** [**shopunif**] *target substitution*

apply-substitution searches through target, replacing each variable symbol with the corresponding value (if any) in substitution

The following are some predicates that may be useful to a programmer using the unifier at a low level.

Macro: **variable-p** [**shopunif**] *x*

An alias for `variablep`, for more consistent naming.

Function: **variablep** [shopunif] *x*

Is *x* a symbol representing a logical variable for SHOP's theorem-prover?

Function: **anonymous-var-p** [shopunif] *x*

Does *x* name an anonymous variable?

Function: **groundp** [shopunif] *literal*

Is *literal* a ground literal? Traverse the tree looking for a variable.

Finally, before unifying, it may be necessary to *standardize* the expressions – renaming variables to be unique.

Function: **standardize** [shopunif] *expr* &*optional subs*

Replace all variables in *expr* with newly-generated variables, with new names.

Next: [Internal Technical Information](#), Previous: [The SHOP Unifier](#), Up: [Top \[Contents\]\[Index\]](#)

## 8 Plan Grapher

In order to make the plan trees that SHOP3 can produce more useful, we have developed the SHOP3 plan grapher.

Function: **graph-plan-tree** [spg] *plan-forest* &*key tree-processor graph-object*

Takes a shop plan forest (*plan-tree*) as input, and returns a *cl-dot* graph object. Note that the *plan-tree* name is a misnomer that reflects shop *find-plans* misnomer. If you ask for plan trees from shop, you really get plan *\*forests\**.

Function: **graph-enhanced-plan-tree** [spg] *plan-tree* &*key graph-object label-dependencies show-dependencies*

Takes an enhanced shop plan tree (*plan-tree*) as input, and returns a *cl-dot* graph object.

Class: **enhanced-plan-tree-graph** [spg]

Class precedence list: *enhanced-plan-tree-graph*, *standard-object*, *t*

A null class that the user may subclass to tailor display of shop enhanced plan trees.

Class: **plan-tree-graph** [spg]

Class precedence list: *plan-tree-graph*, *standard-object*, *t*

A null class that the user may subclass to tailor display of shop plan

trees.

---

Next: [Version Differences](#), Previous: [Plan Grapher](#), Up: [Top](#) [[Contents](#)]  
[Index](#)]

## 9 Internal Technical Information

**This section has not been updated since the transition from SHOP2 to SHOP3, and its accuracy is likely compromised. *Caveat lector.***

This section presents information about the internal workings of the SHOP3 planning process. **Important Note:** This section is primarily of interest to planning researchers and planning system developers. Most SHOP3 users (especially beginning users) are advised to skip this section.

The first subsection presents some key internal knowledge structures that must be defined in order to completely specify the behavior of SHOP3. The second subsection presents the formal semantics of operators and plans. The third subsection describes an assortment of functions within SHOP3 that are used to accomplish those semantics.

---

Next: [Formal Semantics](#), Previous: [Internal Technical Information](#), Up: [Internal Technical Information](#) [[Contents](#)][[Index](#)]

### 9.1 Internal Knowledge Structures

The following SHOP3 internal knowledge structures must be defined in order to fully specify the semantics of plan generation in SHOP3.

---

Next: [States and Satisfiers](#), Previous: [Internal Knowledge Structures](#), Up: [Internal Knowledge Structures](#) [[Contents](#)][[Index](#)]

#### 9.1.1 Substitutions

**Update note:** *The variable bindings in a substitution are no longer represented by dotted pairs. Instead, they are represented by common lisp **structures** that have ‘binding-var’ and ‘binding-value’ slots. So, until there has been time for a rewrite, please read “binding structure” for “dotted pair” in the following. [rpg 12 June 2018]*

A **substitution** is a list of dotted pairs of the form

$$((x_1 . t_1) (x_2 . t_2) \dots (x_k . t_k))$$

where every  $x_i$  is a variable symbol and every  $t_i$  is a term. If  $e$  is an expression and  $u$  is the above substitution, then the **substitution**

**instance**  $e^u$  is the expression produced by starting with  $e$  and replacing each occurrence of each variable symbol  $x_i$  with the corresponding term  $t_i$ .

If  $d$  and  $e$  are two expressions, then:

- $d$  is a **generalization** of  $e$  if  $e$  is a substitution instance of  $d$ ;
- $d$  is a **strict generalization** of  $e$  if  $d$  is a generalization of  $e$  but  $e$  is not a generalization of  $d$ ;
- $d$  and  $e$  are **equivalent** if each is a generalization of the other.

If  $u$  and  $v$  are two substitutions, then:

- $u$  is a **generalization** of  $v$  if for every expression  $e$ ,  $e^u$  is a generalization of  $e^v$ ;
- $u$  is a **strict generalization** of  $v$  if for every expression  $e$ ,  $e^u$  is a strict generalization of  $e^v$ ;
- $u$  and  $v$  are **equivalent** if for every expression  $e$ ,  $e^u$  and  $e^v$  are equivalent.

If  $e$  is an expression and  $x_1, x_2, \dots, x_k$  are the variable symbols in  $e$ , then a **standardizer** for  $e$  is a substitution of the form

$$((x_1 \cdot y_1) (x_2 \cdot y_2) \dots (x_k \cdot y_k))$$

where each  $y_i$  is a new variable symbol that is not used anywhere else.

Note that if  $u$  is a standardizer for  $e$ , then  $e$  and  $e^u$  are equivalent expressions.

If  $d$  and  $e$  are expressions and there is a substitution  $u$  such that  $d^u = e^u$ , then  $d$  and  $e$  are **unifiable** and  $u$  is a **unifier** for them. A unifier of  $d$  and  $e$  is a **most general unifier** (or **mgu**) of  $d$  and  $e$  if it is a generalization of every unifier of  $d$  and  $e$ . Note that all mgu's for  $d$  and  $e$  are equivalent.

---

Previous: [Substitutions](#), Up: [Internal Knowledge Structures](#) [[Contents](#)] [[Index](#)]

### 9.1.2 States and Satisfiers

A **state** is a list of ground atoms intended to represent some "state of the world". A conjunct  $C$  is a **consequent** of a state  $S$  and an axiom list  $X$  if every logical expression  $I$  in  $C$  is a consequent of  $S$  and  $X$ . A logical expression  $I$  is a consequent of  $S$  and  $X$  if one of the following is true:

- $I$  is an atom in  $S$ ;
- $I$  is a ground expression of the form  $(\text{eval } p \ t_1 \ t_2 \ \dots \ t_n)$ , and the evaluation of  $p$  with arguments  $t_1, t_2, \dots, t_n$  returns a non-nil value;
- $I$  is a ground expression of the form  $(\text{call } p \ t_1 \ t_2 \ \dots \ t_n)$ , and the evaluation of  $p$  with arguments  $t_1, t_2, \dots, t_n$  returns a non-nil



value;

- $I$  is an expression of the form (not  $a$ ), and the atom  $a$  is not a consequent of  $S$  and  $X$ ;
- $I$  is an expression of the form (assign  $v$   $t$ ), where  $v$  is a variable symbol and  $t$  is any Lisp expression. The value of  $t$ , which was evaluated via a call to the Lisp evaluator, is a substitution of  $v$ , i.e. ( $v$  .  $t$ ). This term is always a consequent of  $S$  and  $X$ ;
- $I$  is an expression of the form (or  $I_1$   $I_2$  ...  $I_n$ ), where  $I_1, I_2, I_3, \dots, I_n$  are logical expressions, and at least one expression in this list is a consequent of  $S$  and  $X$ ;
- $I$  is an expression of the form (forall  $V$   $Y$   $Z$ ), where  $Y$  and  $Z$  are logical expressions and  $V$  is the list of variables in  $Y$  such that for every satisfier  $u$  that satisfies  $Y$  in  $S$  and  $X$ ,  $u$  also satisfies  $Z$  in  $S$  and  $X$ ;
- $I$  is an expression of the form (imply  $Y$   $Z$ ), where  $Y$  and  $Z$  are logical expressions such that a satisfier  $u$  satisfies  $Y$  in  $S$  and  $X$  also satisfies  $Z$  in  $S$  and  $X$ ;
- there exists a substitution  $v$  and an axiom( $\text{:- } a$   $n_1$   $C_1$   $n_2$   $C_2$  ...  $n_n$   $C_n$ ) in  $X$  such that  $I = a^v$  and one of the following holds:
  - $C_1^v$  is a consequent of  $S$  and  $X$ ;
  - $C_1^v$  is not a consequent of  $S$  and  $X$ , but  $C_2^v$  is a consequent of  $S$  and  $X$ ;
  - neither  $C_1^v$  nor  $C_2^v$  is a consequent of  $S$  and  $X$ , but  $C_3^v$  is a consequent of  $S$  and  $X$ ;
  - ...;
  - none of  $C_1^v, C_2^v, C_3^v, \dots, C_{n-1}^v$  is a consequent of  $S$  in  $X$ , but  $C_n^v$  is a consequent of  $S$  and  $X$ .

If  $C$  is a consequent of  $S$  and  $X$ , then it is a **most general consequent** of  $S$  and  $X$  if there is no strict generalization of  $C$  that is also a consequent of  $S$  and  $X$ .

Let  $S$  be a state,  $X$  be an axiom list, and  $C$  be an ordinary conjunct. If there is a substitution  $u$  such that  $C^u$  is a consequent of  $S$  and  $X$ , then we say that  $S$  and  $X$  **satisfy**  $C$  and that  $u$  is the **satisfier**. The satisfier  $u$  is a **most general satisfier** (or **mgs**) if there is no other satisfier that is a strict generalization of  $u$ . Note that  $C$  can have multiple non-equivalent mgs's. For example, suppose  $X$  contains the "walking distance" axiom given earlier, and  $S$  is the state

```
((weather-is good)
 (distance home convenience-store 1)
 (distance home supermarket 2))
```

Then for the conjunct ((walking-distance ?y)), there are two mgs's from  $S$  and  $X$ : ((?y . convenience-store)) and ((?y . supermarket)).

Let  $S$  be a state,  $X$  be an axiom list, and  $C = (:first\ C')$  be a tagged conjunct. If  $S$  and  $X$  satisfy  $C'$ , then the **most general satisfier** for  $C$  from  $S$  and  $X$  is the *first* mgs for  $C'$  that would be found by a left-to-right depth-first search. For example, if  $S$  and  $X$  are as in the previous example, then for the tagged conjunct  $(:first\ (walking-distance\ ?y))$ , the mgs from  $S$  and  $X$  is  $((?y\ .\ convenience-store))$ .

---

Next: [Key Functions in SHOP3](#), Previous: [Internal Knowledge Structures](#), Up: [Internal Technical Information](#) [[Contents](#)][[Index](#)]

## 9.2 Formal Semantics

Recall that a plan is a list of operator invocations with costs and that an operator has an add list and a delete list. Informally, the meaning of the plan is that the specified operators are performed in sequence, incurring the specified costs. Similarly, the meaning of the operator is that the assertions in the add list are added to the state and the assertions in the delete list are removed from the state. The meaning of a method is that when the method's precondition is satisfied, the task specified in the method's head can be performed by performing each of the tasks specified in the method's tail.

This subsection elaborates these informal notions, presenting detailed formal semantics of operators and plans. It is of particular use to anyone who has a SHOP3 domain and wishes to prove theorems (e.g., correctness, completeness, etc.) regarding plans generated in that domain.

---

Next: [Semantics of Methods](#), Previous: [Formal Semantics](#), Up: [Formal Semantics](#) [[Contents](#)][[Index](#)]

### 9.2.1 Semantics of Operators

**Update note:** *SHOP3 now supports PDDL operators, which have cleaner (but more restricted) semantics. Their semantics is discussed in many treatments of the PDDL specification. We hope to add an explanation later. See Chapter 7 for discussion of PDDL support in SHOP3. [12 June 2018 – rpg]*

The intent of an operator is to specify that the task  $h$  can be accomplished at a cost of  $c$ , by modifying the current state of the world to remove every logical atom in  $D$  and add every logical atom in  $A$  if  $P$  is satisfied in the current state. In order to prevent plans from being ambiguous, there should be at most one operator for each primitive task symbol. *Furthermore, whenever an action is inserted into a plan, it must be **ground** – there must be no unbound parameters in its task head. [Inserted 12 June 2018 based on discussions with Ugur – rpg]*

Let  $S$  be a state,  $X$  be the list of axioms,  $L$  be the list of protected conditions,  $t$  be a primitive task atom, and  $o$  be a planning operator whose head, precondition, delete list, add list, and cost are  $h$ ,  $P$ ,  $D$ ,  $A$ , and  $c$ ,

respectively. Suppose that there is an mgu  $u$  for  $t$  and  $h$ , such that  $h^u$  is ground, that none of the ground atoms in  $D^u$  are in the list of protected conditions, and  $P^u$  is satisfied in  $S$ . Then we say that  $o^u$  is **applicable** to  $t$ , and that  $h^u$  is a **simple plan** for  $t$ . If  $S$  is a state, then the state and the protection list produced by executing  $o^u$  (or equivalently,  $h^u$ ) in  $S$  and  $L$  is the new state:

$$(S', L') = \text{result}(S, L, h^u) = \text{result}(S, L, o^u) = (S - D^u) \cup A^u.$$

where  $S'$  and  $L'$  are obtained by modifying the current state of the world and the list of protected conditions as follows:

- remove every logical atom in  $D^u$  from the current state;
- remove every protection condition in  $D^u$  from the list of protected conditions;
- for every expression (forall  $V Y Z$ ) in  $D^u$  and every satisfier  $v$  such that  $S$  and  $X$  satisfy  $Y^v$ , remove every logical atom in  $Z^u$  from the current state;
- for every expression (forall  $V Y Z$ ) in  $D^u$  and every satisfier  $v$  such that  $S$  and  $X$  satisfy  $Y^v$ , remove every protection condition in  $Z^u$  from the list of protected conditions;
- add every logical atom in  $A^u$  to the current state;
- add every protection condition in  $A^u$  to the list of protected conditions;
- for every expression (forall  $V Y Z$ ) in  $A^u$  and every satisfier  $v$  such that  $S$  and  $X$  satisfy  $Y^v$ , add every logical atom in  $Z^u$  to the current state;
- for every expression (forall  $V Y Z$ ) in  $A^u$  and every satisfier  $v$  such that  $S$  and  $X$  satisfy  $Y^v$ , add every protection condition in  $Z^u$  to the list of protected conditions.

Here is an example:

```

S      ((has-money john 40) (has-money mary 30))
T      (!set-money john 40 35)
      (:operator (!set-money ?person ?old ?new)
O      ((has-money ?person ?old))
      ((has-money ?person ?old))
      ((has-money ?person ?new)))
U      ((?person . john) (?old . 40) (?new . 35))
      (:operator (!set-money john 40 35)
o^u    ((has-money john 40))
      ((has-money john 40))
      ((has-money john 35)))
h^u    (!set-money john 40 35)
Result(S, h^u) ((has-money john 35) (has-money mary 30) )

```

```
result( $S, o^u$ ) ((has-money john 35) (has-money mary 30) )
```

Here is an example using the forall keyword

```

S      ((location l1) (location l2) (location l3) (truck-at
T      (!clear-locations)
        (:operator (!clear-locations)
O      ((forall (?l)
        ((location ?l)
        (not (truck-at ?t ?l)))
        ((location ?l))))
        ()))
U      ()
        (:operator (!clear-locations)
ou      (forall (?l)
        ((location ?l)
        (not (truck-at ?t ?l))))
        ((location ?l))
        ()))
hu =    (!clear-locations)
Result( $S, h^u$ ) ((location l1) (truck-at truck1 l1))
result( $S, o^u$ ) ((location l1) (truck-at truck1 l1))

```

---

Next: [Semantics of Plans](#), Previous: [Semantics of Operators](#), Up: [Formal Semantics](#) [[Contents](#)][[Index](#)]

### 9.2.2 Semantics of Methods

The purpose of a method is to specify the following:

- If the current state of the world satisfies  $C_1$ , then  $h$  can be accomplished by performing the tasks in  $T_1$  in the order given;
- otherwise, if the current state of the world satisfies  $C_2$ , then  $h$  can be accomplished by performing the tasks in  $T_2$  in the order given;
- ...;
- otherwise, if the current state of the world satisfies  $C_k$ , then  $h$  can be accomplished by performing the tasks in  $T_k$  in the order given.

Let  $S$  be a state,  $X$  be an axiom list,  $t$  be a task atom (which may or may not be ground), and  $m$  be the method  $(\text{:method } h \ C_1 \ T_1 \ C_2 \ T_2 \ \dots \ C_k \ T_k)$ . Suppose there is an mgu  $u$  that unifies  $t$  with  $h$ ; and suppose that  $m$  has a precondition  $C_i$  such that  $S$  and  $X$  satisfy  $C_i^u$  (if there is more than one such precondition, then let  $C_i$  be the first such precondition). Then we say that  $m$  is **applicable** to  $t$  in  $S$  and  $X$ , with the **active precondition**  $C_i$  and the **active tail**  $T_i$ . Then the result of applying  $m$  to  $t$  is the following set of task lists:

$$R = \{ \text{Call}((T_i^u)^v) : v \text{ is an mgs for } C_i^u \text{ from } S \text{ and } X \}$$

where *Call* is SHOP3's evaluation function (the function that evaluates the values of the call-terms in the form (call f  $t_1$   $t_2$  ..  $t_n$ )). Each task list  $r$  in  $R$  is called a **simple reduction** of  $t$  by  $m$  in  $S$  and  $X$ . Here is an example:

```

S      ((has-money john 40) (has-money mary 30))
X      nil
t      (transfer-money john mary 5)
      (:method (transfer-money ?p1 ?p2 ?amount)
      ((has-money ?p1 ?m1)
      (has-money ?p2 ?m2)
      (call >= ?m1 ?amount))
      (:ordered (:task !set-money ?p1 ?m1 (call - ?m1 ?amount))
      (:task !set-money ?p2 ?m2 (call + ?m2 ?amount)))))
M
u      ((?p1 . john) (?p2 . mary) (?amount . 5))
hu    (transfer-money john mary 5)
      ((has-money john ?m1)
      (has-money mary ?m2)
      (call >= ?m1 5))
C1u   (:ordered
T1u    (:task !set-money john ?m1 (call - ?m1 5))
      (:task !set-money mary ?m2 (call + ?m2 5)))
v      ((?m1 . 40) (?m2 . 30))
      ((has-money john 40)
      (has-money mary 30)
      (call >= 40 30))
(C1u)v  (:ordered
      (:task !set-money john 40 (call - 40 5))
      (:task !set-money mary 30 (call + 30 5)))
(Tu)v  (:ordered
call((Tu)v)  (:task !set-money john 40 35)
      (:task !set-money mary 30 35))

```

---

Previous: [Semantics of Methods](#), Up: [Formal Semantics](#) [[Contents](#)] [[Index](#)]

### 9.2.3 Semantics of Plans

Recall that a planning domain contains axioms, operators, and methods, and that a planning problem is a 4-tuple  $(S, M, L, D)$ , where  $S$  is a state,  $M$  is a task list,  $L$  is a protection list, and  $D$  is a domain representation. Let  $T$  be the list of tasks in  $M$  that have no predecessor (i.e., those tasks can be performed at this time if they are applicable). If  $t$  is a task in  $T$ , and  $S$  is a state, then a **reduction** of  $t$  in  $S$  and  $D$  with respect to  $M$  and  $L$  that results in a new planning problem  $(S', M', L', D)$  is defined as follows:

**if**  $t$  is a primitive task, **then**

$(S', L') = \text{result}(S, L, t)$ ;

$M' =$  the task list produced by removing  $t$  from  $M$

**else**  $t$  is a compound task, **then**

$S' = S$ ;

$L' = L$ ;

Suppose  $m$  is an applicable method to  $t$  in  $S$ , with unifier  $u$ , the

active precondition  $C_i$  and the active tail  $T_i$ .

$M'$  = the task list produced by replace  $t$  with  $T_i^u$  in  $M$

**endif**

If  $P = (p_1 p_2 \dots p_n)$  is a plan, then we say that  $P$  **solves**  $(S, M, L, D)$ , or equivalently, that  $P$  achieves  $M$  from  $S$  in  $D$  (we will omit the phrase "in  $D$ " if the identity of  $D$  is obvious) in any of the following cases:

**Case 1.**

both  $M$  and  $P$  are empty.

**Case 2.**

$T = (t_1 t_2 \dots t_k)$  is a list of tasks in  $M$  that have no predecessor for which there is a task  $t_i$  that has the `:immediate` keyword and is applicable to the current state  $S$ . Let  $(S', M', L') = \text{reduction}(t_i, S, M, L)$ . We say  $P$  **solves**  $(S, M, L, D)$  if either of the following is true.

- $t_i$  is primitive and  $p_1 = t_i$  and  $(p_2 p_3 \dots p_n)$  solves  $(S', M', L', D)$
- $t_i$  is not primitive, and  $P$  solves  $(S', M', L', D)$

**Case 3.**

$T = (t_1 t_2 \dots t_k)$  is a list of tasks in  $M$  that have no predecessor, where  $t_i$  is a task in  $T$  that's applicable to the current state  $S$ . Let  $(S', M', L') = \text{reduction}(t_i, S, M, L)$ . We say  $P$  **solves**  $(S, M, L, D)$  if either of the following is true.

- $t_i$  is primitive and  $p_1 = t_i$  and  $(p_2 p_3 \dots p_n)$  solves  $(S', M', L', D)$
- $t_i$  is not primitive and  $P$  solves  $(S', M', L', D)$

The planning problem  $(S, M, L, D)$  is **solvable** if there is a plan that solves it. For example, suppose that

```
S nil
M ((:ordered (:task do-both op1 op2)))
T ((:task do-both op1 op2))
L nil
(
  (:operator (!do ?operation) nil ((did ?operation)))
  (:method (do-both ?x ?y)
    nil
    (:ordered
      (:task !do ?x)
      (:task !do ?y)))
  (:method (do-both ?x ?y)
    nil
    (:ordered (:task !do ?y)
      (:task !do ?x))))
D
P1 ((do op1) 1 (do op2) 1)
P2 ((do op2) 1 (do op1) 1)
```

Then  $P_1$  and  $P_2$  are all of the plans that solve  $(S, M, L, D)$ .

---

Previous: [Formal Semantics](#), Up: [Internal Technical Information](#)  
[\[Contents\]](#)[\[Index\]](#)

### 9.3 Key Functions in SHOP3

Below are some important functions in the Lisp implementation of SHOP3. They should be of interest to anyone who wishes to modify SHOP3 or to directly access internal capabilities of SHOP3. Pseudocode algorithms for the main planning functions of SHOP3 are also presented.

`(apply-substitution e u)`

$e$  is an expression and  $u$  is a substitution. The function returns  $e^u$ .

`(compose-substitutions u v)`

If  $u$  and  $v$  are substitutions, then this function returns a substitution  $w$  such that for every expression  $e$ ,  $e^w = (e^u)^v$ .

`(standardizer e)`

This function returns a standardizer for  $e$ .

`(standardize e)`

This function is equivalent to `(apply-substitution e (standardizer e))`.

`(unify d e)`

This procedure returns an mgu for the expressions  $d$  and  $e$  if they are unifiable, and returns fail otherwise.

`(find-satisfiers C S &optional just-one)`

If  $C$  is a conjunct and  $S$  is a state, then this function returns a list of mgs's, one for every most general instance of  $C$  that is satisfied by  $S$ . If the optional argument *just-one* is not nil, then the function returns the first mgs it finds, rather than all of them. Calling `(find-satisfiers C S)` is roughly equivalent to calling the following (simplified) pseudocode:

**procedure** find-satisfiers( $C$ ,  $S$ )

**if**  $C$  is empty **then return** {nil}

$I$  = the first logical atom in  $C$ ;  $B$  = the remaining logical atoms in  $C$

answers = nil

**if**  $I$  is an expression of the form (not  $e$ ) **then**

**if** find-satisfiers( $e$ ,  $S$ , nil) = nil **then**

**return** find-satisfiers( $B$ ,  $S$ )

```
else
return nil
end if
else if l is an expression of the form (eval e) then
if eval(e) is not nil then
return find-satisfiers(B, S)
else
return nil
end if
else if l is an expression of the form (or p1 p2 ... pn) then
for every unifier u that unifies any pi with l
for every v in find-satisfiers(Bu, S)
insert compose-substitutions(u,v) into answers
end for
end for
else if l is an expression of the form (imply C1 C2) then
mgu = find-satisfiers(C1, S)
if mgu is null or there exist a unifier u in mgu such that
find-satisfiers(C2u, S) is not equal to nil then
return find-satisfiers(B, S)
else
return nil
end if
else if l is an expression of the form
(forall variables bounds conditions) then
mgu = find-satisfiers(bounds, S)
if mgu is null or for every unifier u in mgu,
find-satisfiers(conditionsu, S) is not equal to nil then
return find-satisfiers(B, S)
else
```



```

return nil
end if
else
for every atom  $s$  in  $S$  that unifies with  $l$ 
  let  $u$  be the unifier
  for every  $v$  in find-satisfiers( $B^u$ ,  $S$ )
    insert compose-substitutions( $u,v$ ) into answers
  end for
end for
for every axiom  $x$  in *axioms* whose head unifies with  $l$ 
  let  $u$  be the unifier
  if tail( $x$ ) contains a conjunct  $D$  such that
    find-satisfiers(append( $D^u$ ,  $B^u$ ),  $S$ ) is not nil then
      let  $D$  be the first such conjunct
      for every  $v$  in find-satisfiers(append( $D^u$ ,  $B^u$ ),  $S$ )
        insert compose-substitutions( $u$ ,  $v$ )
        into answers
      end for
    end if
  end for
end if
return answers
end find-satisfiers

```

In this pseudo-code, *\*axioms\** is an internal variable of SHOP3. It holds the list of the axioms defined for the domain under consideration.

(apply-method  $S$   $t$   $m$ )

If  $S$  is a state,  $t$  is a task, and  $m = (:method\ h\ C_1\ T_1\ C_2\ T_2\ \dots\ C_k\ T_k)$  is a method, then this function does the following:

- If  $m$  is not applicable to  $t$  in  $S$ , then the function returns the symbol FAIL.
- If  $m$  is applicable to  $t$  in  $S$  and  $C_i$  is the active precondition, then the function returns a list of all simple reductions of  $T_i$ , one for

each satisfier of  $C_i$  in  $S$ .

(apply-operator  $S$   $t$   $o$ )

If  $S$  is a state,  $t$  is a task, and  $o$  is an operator, then this function does the following:

- If there is an mgu  $u$  for  $o$  and  $t$ , then it returns the state produced by executing  $o^u$  in  $S$ .
- Otherwise, it returns FAIL.

(find-plans problem &key which verbose pshort gc pp state plan-tree optimize-cost time-limit explanation)

This function implements the SHOP3 planning algorithm. For more about the arguments to and use of this function, see [find-plans](#). A brief overview of the algorithm for this function is presented here. Calling find-plans is roughly equivalent to calling the following pseudocode, where  $S$  is the current state,  $T$  is a partially ordered set of tasks, and  $L$  is a list of protected conditions:

**procedure** find-plans ( $S, T, L$ )

**if**  $T$  is empty **then**

**return** NIL

**endif**

nondeterministically choose a task  $t$  in  $T$  that has no predecessors

$\langle r, R \rangle = \text{reduction}(S, t)$

**if**  $r = \text{FAIL}$  **then**

**return** FAIL

**endif**

nondeterministically choose an operator instance  $o$  applicable to  $r$  in  $S$

$S' =$  the state produced from  $S$  by applying  $o$  to  $r$

$L' =$  the protection list produced from  $L$  by applying  $o$  to  $r$

$T' =$  the partially ordered set of tasks produced from  $T$  by replacing  $t$  with  $R'$

$P = \text{find-plans}(S', T', L')$

**return** cons( $o, P$ )

**end** find-plans

**procedure** reduction ( $S, t$ )

**if**  $t$  is a primitive task **then**

**return**  $\langle t, \text{NIL} \rangle$

**else if** no method is applicable to  $t$  in  $S$  **then**

**return** <FAIL,NIL>

**endif**

nondeterministically let  $m$  be any method applicable to  $t$  in  $S$

$R$  = the decomposition (partially ordered set of tasks) produced by  $m$  from  $t$

$r$  = any task in  $R$  that has no predecessors

< $r',R'$ > = reduction ( $S,r$ )

**if**  $r' = \text{FAIL}$  **then**

**return** <FAIL,NIL>

**endif**

$R''$  = the partially ordered set of tasks produced from  $R$  by replacing  $r$  with  $R'$

**return** < $r',R''$ >

**end** reduction

(defdomain *domain-name*  $D$ )

This macro gives the name *domain-name* to planning domain  $D$ . (More specifically, what it does is to store  $D$ 's axioms, operators, and methods on *domain-name*'s property list.)

(defproblem *problem-name* *domain-name*  $S$   $T$ )

This macro gives the name *problem-name* to the planning problem  $(S,T,D)$ , where  $D$  is the planning domain whose name is *domain-name*. (More specifically, what it does is to store  $S$ ,  $T$ , and *domain-name* on *problem-name*'s property list.)

(def-problem-set *set-name* *list-of-problems*)

This macro gives the name *set-name* to the set of planning problems in *list-of-problems*. (More specifically, what it does is to store *list-of-problems* on *set-name*'s property list.)

Note that for backwards compatibility, SHOP3 also accepts the forms *make-domain*, *make-problem*, and *make-problem-set*, which were employed in SHOP 1.x, using the same arguments as *defdomain*, *defproblem*, and *def-problem-set*. The difference between the *make-X* and *def-X* forms is that in the latter case since the form is a macro, the arguments are not evaluated. This changes the syntax one uses. Thus in a SHOP 1.x domain one might define a problem as

```
(make-problem problem-name domain-name
  list of state atoms
  list of tasks to be accomplished)
```

whereas in SHOP3 the syntax becomes

```
(defproblem problem-name domain-name
  (list of state atoms)
  (list of tasks to be accomplished))
```

where the arguments are all quoted in the SHOP 1.x make-problem function, they are unquoted when using the SHOP3 defproblem macro.

```
(print-axioms &optional name)
```

This function prints a list of the axioms for the domain whose name is *name*; defaults to the most recently defined domain.

```
(print-operators &optional name)
```

This function prints a list of the operators for the domain whose name is *name*; defaults to the most recently defined domain.

```
(print-methods &optional name)
```

This function prints a list of the methods for the domain whose name is *name*; defaults to the most recently defined domain.

```
(get-state name)
```

This function returns the initial state for the problem whose name is *name*.

```
(get-tasks name)
```

This function returns the list of tasks for the problem whose name is *name*.

```
(get-problems name)
```

This function returns the list of problem names for the problem set whose name is *name*.

```
(do-problems name-or-list &rest keywords)
```

*name-or-list* should be either a list of problem names or the name of a problem set. This function runs `find-plans` on each planning problem specified by the list or problem set, and then returns nil. The keywords are simply passed on to `find-plans`.

---

Next: [General Notes on SHOP3](#), Previous: [Internal Technical Information](#),  
Up: [Top](#) [\[Contents\]](#)[\[Index\]](#)

## 10 Differences between SHOP 1.x, SHOP2 and SHOP3

The differences between SHOP3 and SHOP 1.x can be grouped in two sets: syntactic changes in the domain definitions and differences in functionality.

**Note:** *This section has not been updated since the extensive modifications were made to SHOP3 (since as time goes by, the number of people who would find it useful dwindles more and more). It provides an effective description of the difference between SHOP1 and the **original** SHOP3 syntax, but does not reflect updates in the past decade. This*

*section should probably be removed. – rpg December 22, 2017*

Next: [SHOP 1x Functionality Comparison](#), Previous: [Version Differences](#),  
Up: [Version Differences](#) [[Contents](#)][[Index](#)]

## 10.1 SHOP 1.x Syntax Comparison

The table below gives examples of SHOP 1.x syntax, comparing it to the syntax used in SHOP3:

SHOP1 syntax	SHOP2 Syntax
(make-domain 'travel	(defdomain travel
'(	(
(:- (have-taxi-fare ? distance))	(:- (have-taxi-fare ?distance))
((have-cash ?m)	((have-cash ?m)
(eval (>= ?m (+ 1.5 ? distance))))	(call >= ?m (call + 1.5 ?distance)))
(:- (walking-distance ?u ?v)	(:- (walking-distance ?u ?v)
((weather-is'good) (distance ?u ?v ?w) (eval (<= ?w 3)))	good ((weather-is good) (distance ?u ?v ?w) (call <= ?w 3))
((distance ?u ?v ?w)	bad ( (distance ?u ?v ?w)
(eval (<= ?w 0.5)))	(call <= ?w 0.5)))
(:method (pay-driver ?fare)	(:method (pay-driver ?fare)
((have-cash ?m) (eval (>= ?m ?fare)))	((have-cash ?m) (call >= ?m ?fare))
'(!set-cash ?m ,(- ?m ? fare)))	((!set-cash ?m ( call - ?m ?fare)))
(:method (travel-to ?q)	(:method (travel-to ?q)
((at ?p) (walking-distance ? p ?q))	((at ?p) (walking-distance ?p ?q))
'(!walk ?p ?q)))	((!walk ?p ?q)))
(:method (travel-to ?y)	(:method (travel-to ?y)
((at ?x) (at-taxi-stand ?t ?x)	((at ?x) (at-taxi-stand ?t ?x)
(distance ?x ?y ?d) (have- taxi-fare ?d))	(distance ?x ?y ?d) (have-taxi-fare ?d))
'(!hail ?t ?x) (!ride ?t ?x ?y)	((!hail ?t ?x) (!ride ?t ?x ?y)
(pay-driver (call + 1.50 ?d)))	(pay-driver (call + 1.50 ?d)))

```

(pay-driver ,(+ 1.50 ?d))) by-bus

((at ?x) (bus-route ?bus ?x ?y))
?y))
((!wait-for ?bus ?x) (pay-driver 1.00)
(!ride ?bus ?x ?y)))
(:operator (!hail ?vehicle ?location)
() ())
((at ?vehicle ?location)))
()
(:operator (!wait-for ?bus ?location)
((at ?vehicle ?location)))
() ()
(:operator (!wait-for ?bus ?location)
((at ?bus ?location)))
()
(:operator (!ride ?vehicle ?a ?b)
((at ?bus ?location)))
()
((at ?a) (at ?vehicle ?a))
(:operator (!ride ?vehicle ?a ?b)
((at ?b) (at ?vehicle ?b)))
((at ?a) (at ?vehicle ?a))
(:operator (!set-cash ?old ?new)
((at ?b) (at ?vehicle ?b)))
((have-cash ?old))
(:operator (!set-cash ?old ?new)
((have-cash ?old))
((have-cash ?new)))
((have-cash ?old))
(:operator (!walk ?here ?there)
((have-cash ?new)))
((at ?here))
(:operator (!walk ?here ?there)
((at ?here))
((at ?here))
((at ?there)))
((at ?there)))
))

(make-problem 'go-park-rich 'travel

'((distance downtown park
2)
(defproblem go-park-rich travel
(distance downtown uptown
8)
((distance downtown park 2)
(distance downtown suburb
12)
(distance downtown uptown 8)
(distance downtown suburb 12)
(at-taxi-stand taxi1
downtown)
(at-taxi-stand taxi1 downtown)

```

(at-taxi-stand taxi2 downtown)	(at-taxi-stand taxi2 downtown)
(bus-route bus1 downtown park)	(bus-route bus1 downtown park)
(bus-route bus2 downtown uptown)	(bus-route bus2 downtown uptown)
(bus-route bus3 downtown suburb)	(bus-route bus3 downtown suburb)
(at downtown)	(at downtown)
(weather-is good)	(weather-is good)
(have-cash 80))	(have-cash 80))
(weather-is good)	((travel-to park)))
(have-cash 80))	
'((travel-to park)))	
(make-problem-set 'travel	(def-problem-set travel
'( go-park-broke	(go-park-broke
go-park-rich	go-park-rich
))	))
	*def-problem-set is not available in JSHOP
(do-problems 'travel :which :all)	Works in SHOP3 without any changes. Not available in JSHOP.

To summarize, the changes in syntax are as follows:

- Quotes, back quotes and commas are not used in SHOP3 except as needed in general Lisp expressions.
- *make-domain* is replaced with *defdomain*.
- *make-problem* is replaced with *defproblem*.
- Operators may have preconditions in SHOP3.
- The tail of an axiom can have names for each of the conjuncts, and the tail of a method can have names for each of the precondition-tail pairs. These names are optional.
- An ordinary list can be differentiated from a predicate or a function by inserting the optional label, *list*, before the first element of the list.
- In SHOP 1.x it was valid to have methods of the following form, where *e* is any Lisp expression:

```
(:method (varSubtasks ?tasklist)
  ((precondition))
  e)
```

This kind of method is not supported in SHOP3.

Note that in many cases, SHOP3 is able to process SHOP 1.x syntax (i.e., SHOP3 is partially backward compatible). For example, the old *make-domain* and *make-problem* forms can be handled by SHOP3. It is recommended, however, that new domains be written using the new SHOP3 forms.

---

Previous: [SHOP 1x Syntax Comparison](#), Up: [Version Differences](#)  
[\[Contents\]](#)[\[Index\]](#)

## 10.2 SHOP 1.x Functionality Comparison

The following are some key differences between the functionality of SHOP 1.x and SHOP3:

- Unlike SHOP 1.x, SHOP3 allows the combination of partially ordered and fully ordered tasks through the use of the `:unordered` and `:ordered` keywords.
- The following keywords in SHOP3 domain definitions are not supported in SHOP 1.x: `or`, `assign`, `sort-by`, `forall`. The functionality associated with those keywords is not available in SHOP 1.x.
- The `not` keyword in SHOP 1.x can only be applied to individual atoms, not to arbitrary logical expressions.
- The `:optimize-cost`, `:time-limit`, and `:plan-tree` keyword arguments for SHOP3 are not supported in SHOP 1.x. The functionality associated with those keywords is not available in SHOP 1.x.
- The debugging features in SHOP3 (see Sections 5.2 and 5.3) are not present in SHOP 1.x.
- SHOP3 version 1.1 includes a Java interface. SHOP 1.x has no such interface.

SHOP3 is largely backward compatible with SHOP 1.x. SHOP3 can run most SHOP 1.x knowledge bases with only little or no modification.

---

Next: [Acknowledgments](#), Previous: [Version Differences](#), Up: [Top](#)  
[\[Contents\]](#)[\[Index\]](#)

## 11 General Notes on SHOP3

1. Since the null conjunct is always true, an axiom of the form `(:- a nil)` is equivalent to asserting the atom *a* as a basic fact. The difference is that the expression `(:- a nil)` is what one would put into the set of axioms for the problem domain, whereas the atom *a* is what one would put into a state description. An atom *a* in the state description can be deleted by an operator. However, if we have an axiom `(:- a nil)`, then *a* is always true, no operator can change that.



2. An axiom with several conjuncts in its tail has a different semantics than what you would get by making each conjunct the tail of a separate axiom. For example, consider the following axiom lists:

$$X_1 = ((:- (a \text{ ?}x) ((b \text{ ?}x)) \\ ((c \text{ ?}x))))$$

$$X_2 = ((:- (a \text{ ?}x) ((b \text{ ?}x)) \\ (:- (a \text{ ?}x) ((c \text{ ?}x))))$$

In  $X_1$ , the single axiom acts like an *if-then-else*: if  $((b \text{ ?}x))$  is true then find-satisfiers returns the satisfiers for  $(b \text{ ?}x)$ ; otherwise if  $((c \text{ ?}x))$  is true then it returns the satisfiers for  $(c \text{ ?}x)$ . For example,

```
(find-satisfiers '(a ?u) '((b 2) (c 3)))
```

would return

```
((?u . 2)).
```

On the other hand, in  $X_2$ , the set of axioms acts like a logical "or": find-satisfiers returns every satisfier for  $(b \text{ ?}x)$  and every satisfier for  $(c \text{ ?}x)$ . In this case,

```
(find-satisfiers '(a ?u) '((b 2) (c 3)))
```

would return

```
((?u . 2) (?u . 3)).
```

3. Since a primitive task name is basically a call to an operator, you should never create a set of methods and operators that has more than one operator for the same primitive task. Otherwise, your plans will be ambiguous.
4. The following two calls to find-plans SHOP3 will find the same set of all shallowest plans, but in the first case SHOP3 will use a depth-first search and in the second case it will use an iterative-deepening search:

```
(find-plans 'p :which :all-shallowest)
```

```
(find-plans 'p :which :id-all)
```

Likewise, the following two calls to SHOP3 will both find the same shallowest plan, but in the first case SHOP3 will use a depth-first search and in the second case it will use an iterative-deepening search:

```
(find-plans 'p :which :shallowest)
```

```
(find-plans 'p :which :id-first)
```

Next: [References](#), Previous: [General Notes on SHOP3](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 12 Acknowledgments

Original University of Maryland work on SHOP3 was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F30602-99-1-0013 and F30602-00-2-0505, Army Research Laboratory DAAL01-97-K0135, and National Science Foundation DMI-9713718, and the University of Maryland General Research Board.

SIFT, LLC work on SHOP3 has been supported by DARPA SBIR contract DAAH01-03-C-R177, Army AFDD contract NAS-0155(MJH), Delivery Order 920, by the Air Force Research Laboratory (AFRL) under contract FA8750-16-C-0182, and by Contract FA8650-11-C-7191 with the US Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory.

SIFT LLC work was also supported by Internal Research and Development.

Joint SIFT, LLC and University of Maryland work was supported by DARPA contract FA8650-06-C-7606.

Recent SIFT work on the manual was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under Contract No. FA8750-17-C-0184.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), the Department of Defense, the National Science Foundation, the University of Maryland, SIFT, or the United States Government.

---

Next: [Function Index](#), Previous: [Acknowledgments](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## 13 References

[Dean and Boddy, 1998] T. Dean and M. Boddy. An analysis of time-dependent planning. In *AAAI-88*, 1988.

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI-99*, 1999.

[Nau *et al.*, 2000] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP and M-SHOP: Planning with Ordered Task Decomposition. Tech report TR 4157, University of Maryland, College Park, MD, June 2000.

[Nau *et al.*, 2001] D. S. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Totally Ordered Planning with Partially Ordered Subtasks. In *IJCAI-01*, 2001.

---

Next: [Type and Class Index](#), Previous: [References](#), Up: [Top](#) [[Contents](#)]

[\[Index\]](#)

## Function and Macro Index

Jump to: [A](#) [B](#) [D](#) [E](#) [F](#) [G](#) [M](#) [P](#) [Q](#) [S](#) [T](#) [U](#) [V](#)

Index Entry	Section
<b>A</b>	
<a href="#">anonymous-var-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">anonymous-var-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">apply-substitution [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">apply-substitution [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<b>B</b>	
<a href="#">binding-list-value [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding-list-value [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding-val [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding-val [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding-var [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding-var [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<b>D</b>	
<a href="#">def-problem-set:</a>	<a href="#">Planning Problem Set</a>
<a href="#">defproblem:</a>	<a href="#">Planning Problem</a>
<a href="#">defproblem:</a>	<a href="#">Planning Problem</a>
<a href="#">defproblem:</a>	<a href="#">Planning Problem</a>
<a href="#">do-problems:</a>	<a href="#">do-problems</a>
<a href="#">do-problems:</a>	<a href="#">do-problems</a>
<b>E</b>	
<a href="#">external-access-hook:</a>	<a href="#">Hook Routines</a>
<b>F</b>	
<a href="#">find-plans:</a>	<a href="#">find-plans</a>
<a href="#">find-plans:</a>	<a href="#">find-plans</a>
<a href="#">find-plans-stack:</a>	<a href="#">find-plans-stack</a>
<a href="#">find-plans-stack:</a>	<a href="#">find-plans-stack</a>
<a href="#">find-satisfiers [shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>
<a href="#">find-satisfiers [shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>
<b>G</b>	
<a href="#">graph-enhanced-plan-tree [spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">graph-enhanced-plan-tree [spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">graph-plan-tree [spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">graph-plan-tree [spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">groundp [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">groundp [shopunif]:</a>	<a href="#">The SHOP Unifier</a>

---

**M**

<a href="#">make-binding-list [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">make-binding-list [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">make-problem:</a>	<a href="#">Planning Problem</a>
<a href="#">make-problem:</a>	<a href="#">Planning Problem</a>

---

**P**

<a href="#">plan-found-hook:</a>	<a href="#">Hook Routines</a>
<a href="#">print-current-plan:</a>	<a href="#">Other Debugging Features</a>
<a href="#">print-current-state:</a>	<a href="#">Other Debugging Features</a>
<a href="#">print-current-tasks:</a>	<a href="#">Other Debugging Features</a>

---

**Q**

<a href="#">query [shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>
<a href="#">query [shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>

---

**S**

<a href="#">shop-trace:</a>	<a href="#">Tracing</a>
<a href="#">shop-trace:</a>	<a href="#">Tracing</a>
<a href="#">shop-trace:</a>	<a href="#">Tracing</a>
<a href="#">shop-untrace:</a>	<a href="#">Tracing</a>
<a href="#">shorter-plan:</a>	<a href="#">Plans</a>
<a href="#">shorter-plan:</a>	<a href="#">Plans</a>
<a href="#">standardize [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">standardize [shopunif]:</a>	<a href="#">The SHOP Unifier</a>

---

**T**

<a href="#">trace-query-hook:</a>	<a href="#">Hook Routines</a>
-----------------------------------	-------------------------------

---

**U**

<a href="#">unify [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-fail [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-fail [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-fail-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-fail-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">unify-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>

---

**V**

<a href="#">variable-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">variable-p [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">variablep [shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">variablep [shopunif]:</a>	<a href="#">The SHOP Unifier</a>

---

**Jump to: [A](#) [B](#) [D](#) [E](#) [F](#) [G](#) [M](#) [P](#) [Q](#) [S](#) [T](#) [U](#) [V](#)**

---

Next: [Variable Index](#), Previous: [Function Index](#), Up: [Top](#) [[Contents](#)]  
[\[Index\]](#)

## Type and Class Index

Jump to: [A](#) [B](#) [C](#) [D](#) [E](#) [N](#) [P](#) [Q](#) [S](#) [T](#) [U](#)

Index Entry	Section
<b>A</b>	
<a href="#">ADL-DOMAIN:</a>	<a href="#">PDDL Domain Classes</a>
<a href="#">ADL-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>B</b>	
<a href="#">binding_[shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<a href="#">binding_[shopunif]:</a>	<a href="#">The SHOP Unifier</a>
<b>C</b>	
<a href="#">CONDITIONAL-EFFECTS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<a href="#">COSTS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>D</b>	
<a href="#">DISJUNCTIVE-PRECONDITIONS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>E</b>	
<a href="#">enhanced-plan-tree-graph_[spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">enhanced-plan-tree-graph_[spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">EQUALITY-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<a href="#">EXISTENTIAL-PRECONDITIONS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>N</b>	
<a href="#">NEGATIVE-PRECONDITIONS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>P</b>	
<a href="#">PDDL-DOMAIN:</a>	<a href="#">PDDL Domain Classes</a>
<a href="#">PDDL-TYPING-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<a href="#">plan-tree-graph_[spg]:</a>	<a href="#">Plan Grapher</a>
<a href="#">plan-tree-graph_[spg]:</a>	<a href="#">Plan Grapher</a>
<b>Q</b>	
<a href="#">QUANTIFIED-PRECONDITIONS-MIXIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>S</b>	
<a href="#">SIMPLE-PDDL-DOMAIN:</a>	<a href="#">PDDL Domain Classes</a>
<b>T</b>	
<a href="#">thpr-domain_[shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>
<a href="#">thpr-domain_[shopthpr]:</a>	<a href="#">The SHOP Theorem Prover</a>

**U**

[UNIVERSAL-PRECONDITIONS-MIXIN](#): [PDDL Domain Classes](#)

Jump to: [A](#) [B](#) [C](#) [D](#) [E](#) [N](#) [P](#) [Q](#) [S](#) [T](#) [U](#)

Next: [Concept Index](#), Previous: [Type and Class Index](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## Variable Index

Jump to: [\\*](#)

Index Entry	Section
<a href="#">*</a>	
<a href="#">*break-on-backtrack*</a> :	<a href="#">Other Debugging Features</a>
<a href="#">*current-plan*</a> :	<a href="#">Other Debugging Features</a>
<a href="#">*current-state*</a> :	<a href="#">Other Debugging Features</a>
<a href="#">*current-tasks*</a> :	<a href="#">Other Debugging Features</a>
<a href="#">*domain*</a> :	<a href="#">Extended form</a>
<a href="#">*problem*</a> :	<a href="#">Planning Problem</a>
<a href="#">*which*</a> :	<a href="#">Common Keyword Arguments</a>

Jump to: [\\*](#)

Next: [Colophon](#), Previous: [Variable Index](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## Concept Index

Jump to: [:](#)  
[A](#) [D](#) [E](#) [H](#) [I](#) [P](#) [S](#)

Index Entry	Section
<a href="#">:</a>	
<a href="#">:include:</a>	<a href="#">Inclusion directives</a>
<b>A</b>	
<a href="#">anonymous variables:</a>	<a href="#">Syntax Checks</a>
<a href="#">assignment:</a>	<a href="#">Assignments</a>
<b>D</b>	
<a href="#">domain inclusion:</a>	<a href="#">Inclusion directives</a>
<b>E</b>	
<a href="#">explanation:</a>	<a href="#">Common Keyword Arguments</a>
<b>H</b>	
<a href="#">hook-routines:</a>	<a href="#">Hook Routines</a>

---

**I**[include directives:](#)[Inclusion directives](#)[internal operators:](#)[Operators](#)

---

**P**[PDDL method preconditions:](#)[Enhanced Preconditions](#)[planning problem:](#)[Planning Problem](#)[plans:](#)[Plans](#)

---

**S**[singleton variables:](#)[Syntax Checks](#)[Sorting in PDDL method preconditions:](#) [Enhanced Preconditions](#)

---

**Jump to: :**[A](#) [D](#) [E](#) [H](#) [I](#) [P](#) [S](#)

---

Previous: [Concept Index](#), Up: [Top](#) [[Contents](#)][[Index](#)]

## Colophon

This manual is maintained in Texinfo, and automatically translated into other forms (e.g. HTML or pdf). If you're *reading* this manual in one of these non-Texinfo translated forms, that's fine, but if you want to *modify* this manual, you are strongly advised to seek out a Texinfo version and modify that instead of modifying a translated version.

This manual was made possible in part by the SB-TEXINFO library provided by Nikodemus Siivola and Rudi Schlatte, for which we are very grateful.

---

## Footnotes

**(1)**

From *Automated Planning: Theory and Practice*, Ghallab, Nau and Traverso, Morgan Kaufmann, 2004.

**(2)**

The term `aclass` here is meant *informally*; the reader should draw no conclusions about programming language classes in the SHOP implementation. – *rpg*

**(3)**

If you don't know what this means, you may safely ignore it.

---