Documentation for JSHOP 1.0.1

Füsun Yaman

Department of Computer Science

University of Maryland

College Park, MD 20742, USA

Last update: May 24, 2002

1 Introduction

The aim of this document is to present the design and implementation details of JSHOP, the Java version of SHOP. There is some difference between the syntax of the function calls in JSHOP and SHOP, since SHOP was implemented in Lisp rather than Java. Thus, this document not only discusses JSHOP, but also includes information (in section eight) about the differences between JSHOP and SHOP.

JSHOP does not include all of the optimizations that we have made in the Lisp version of SHOP, and does not run as quickly as the Lisp version of SHOP. Anyone wanting to run tests of SHOP's performance should use the Lisp version rather than the Java version.

The rest of the document is organized as follows:

- Section two discusses the development and test environment.
- Section three explains the notation used in this document.
- Section four presents the definitions and notation used in JSHOP.
- Section five explains the class hierarchy in JSHOP.
- Section six discusses the naming conventions used in JSHOP.
- Section seven explains the classes defined in JSHOP.
- Section eight summarizes the differences between SHOP and JSHOP.

1.1 Copyright

2 Development and Test Environment

JSHOP is implemented in JAVA using JDK 1.3 but it is compatible with JDK 1.2 and presumably with JDK 1.1 although we haven't tested it. The development tool that is used is Visual Cafe 4.0. Visual Cafe is not required to run JSHOP. We did our implementation and tests for JSHOP on the Windows NT and Windows98 operating systems.

3 Notation Used in this Document

In order to differentiate some words or expressions in the text, we used the following notation:

- Boldface is used to indicate that a term is being defined. For example:
  "An **axiom list** is a list of axioms intended to represent what we can infer from a state."
- Italic characters refer to special words or symbols. For example:
  " Let *a* be a *logical atom*."
- Typewriter characters are used to write computer code. For example:
  "`(call <= 7 (call  +  5  3))`"
- Square brackets indicate that a parameter is optional in an expression. For example, in the following expression, the $name_i$'s are optional parameters and thus the expression is still valid if any of the $name_i$'s are missing:
  "`(:-` a `[`$name_1$`]` C_1 `[`$name_2$`]` C_2 `[`$name_3$`]` C_3 ... `[`$name_n$`]` C_n`)`"

4 Definitions and Notations Used in JSHOP

The definitions and notation presented in this section are copied from SHOP documentation and modified slightly to describe the new syntax that JSHOP expects for domain and problem definitions. This change in the syntax was required since SHOP was implemented in LISP and the former syntax was designed to use the advantages of LISP which is no longer available in JSHOP environment. Java is a strongly typed language so the boundaries between terms, lists and preconditions should be clearer so that the parser (which SHOP did not require) will produce unambiguous domain and problem definitions.

4.1 Symbol

In the expressions 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, JSHOP uses the following conventions:

- a variable symbol can be any Lisp symbol whose name begins with a question mark (such as `?x` or `?hello-there`);
- a primitive task symbol can be any Lisp symbol whose name begins with an exclamation point (such as `!unstack` or `!putdown`);
- a constant symbol, function symbol, predicate symbol, or compound task symbol can be any Lisp symbol whose name does not begin with a question mark or exclamation point.

In everything that follows, a **ground** expression is one that contains no variable symbols.

4.2 Term, Logical Atom, Literal

A **term** is a variable symbol, a constant symbol, or an expression having of the form

(f t_1 t_2 ... t_n)

where f is a function symbol and each t_i is a term.

A **list** is a term having either of the following forms

(list t_1 t_2 ... t_n)

where *list* is a reserved word that specifies that t_1 t_2 ... t_n form an ordinary list, and each t_i is a term;

(. t l)

where t is a term and l is either a list or the constant symbol `nil`.

The term (list t) is semantically equivalent to (. t nil), the term (list t_1 t_2) is semantically equivalent to (. t_1 (. T_2 nil)), and so forth. Internally, JSHOP translates all occurrences of "`list`" terms into the equivalent "**.**" terms.

Here are some examples of terms, showing how they would be written in both SHOP and JSHOP:

SHOP definition	JSHOP definition	JSHOP Internal Representation
`( 1 g ?y 6 )`	`(list 1 g ?y 6 )`	`(. 1 (. g (. ?y (. 6 nil))) )`
`(goal ?x ?y )`	`(goal ?x ?y)`	`(goal ?x ?y)`
`((on a b) (e 5 (?t u 9)))`	`(list (on a b) (list e 5 (list ?t u 9)))`	`(. (on a b) (. (. e ( . 5 (. (. ?t ( .u ( .9 nil))) nil))) nil))`

Table 1 ***Examples for representing terms***

Note that unlike the Lisp version of SHOP, the following syntactic forms are errors in JSHOP:

```
(1  2  3  4  .  5)       ( 1  2 . ?rest )
```

These forms should instead be written as

```
(list 1 2 3 4 . 5)   (list 1 2.?rest ) .
```

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

(call f t_1 t_2 ... t_n)

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 JSHOP, because it tells JSHOP that *f* is an attached procedure, i.e., that whenever JSHOP needs to evaluate a precondition or task list that contains a call-term, JSHOP should replace the call term with the result of applying the function *f* on the arguments t_1, t_2, ..., t_n. (We later will define what preconditions and task lists are). For example, the following call-term would have the value 8:

```
(call + 5 3)
```

`Call` can be used for limited function names like +, -, *, / , < , >, <=, >=, *ceil, floor, min, max, equal, not* and the common *member* function which tests whether its first parameter is a member of the list in second parameter. Furthermore a *call- term* should be ground before the evaluation. Thus all of the variables appearing in a *call-term* should be bounded.
A **logical atom** is an expression of either of the forms:

(p t_1 t_2 ... t_n) or (call p u_1 u_2 ... u_n)

where p is a predicate symbol each u_i is a term, and each t_i is a term that is not a call-term and does not contain any call-terms. The second form specifies that p is an attached procedure; i.e., whenever JSHOP needs to evaluate a logical atom of the second form, it will use a procedure to evaluate the predicate p on the arguments u_1 u_2 ... u_n, and if the procedure returns any value other than *nil,* JSHOP will use a truth value of "true" for the atom. For example, the following logical atom would evaluate to "true":

```
(call <= 7 (call + 5 3))
```

A **literal** is any of the following:

- a logical atom a;
- an expression of the form (not a) where a is a logical atom (the intended meaning is that the expression is true if a is false).

4.3 Conjunct

A **conjunct** is either of the following:
- an **ordinary conjunct**, which is list of literals (l_1 l_2 l_3 ... l_n);
- a **tagged conjunct**, which is a list of the form (:first l_1 l_2 l_3 ... l_n) where l_1, l_2, l_3, ..., l_n are literals.

The intent of a tagged conjunct is to tell the theorem-prover that we only want to see the first proof of (l_1 l_2 l_3 ... l_n), rather than every possible proof. This is discussed in more detail later, at the end of Section 4.6.

4.4 Axiom

An **axiom** is an expression of the following form, where *a* is a logical atom and each C_i is a conjunct:

(:- *a* [$name_1$] C_1 [$name_2$] C_2 [$name_3$] C_3 ... [$name_n$] C_n)

The axiom's **head** is the atom *a*, and its **tail** is the list ([$name_1$] C_1 [$name_2$] C_2 [$name_3$] C_3 ... [$name_n$] C_n) where each Ci is a conjunct and $name_i$ is a symbol called the *name* of C_i. The names of the conjuncts are optional. A unique name will be generated for each conjunct if no name was given. These names have no semantic meaning to JSHOP, but are provided in order to help the user debug domain descriptions by looking at traces of JSHOP's behavior.

The intended meaning of an axiom is that *a* is true if C_1 is true, or if C_1 is false but C_2 is true, or if both C_1 and C_2 are false but C_3 is true, ..., or if all of C_1, C_2, C_3, ..., C_{n-1} are false but C_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)))
```

4.5 Substitution

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

((x_1 . t_1) (x_2 . t_2) ... (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.

4.6 States and Satisfiers

A **state** is a list of ground atoms intended to represent some "state of the world". An **axiom list** is a list of axioms intended to represent what we can infer from a state. A conjunct *C* is a **consequent** of a state *S* and an axiom list *X* if every literal *l* in *C* is a consequent of *S* and *X*. A literal *l* is a consequent of *S* and *X* if one of the following is true:

- *l* is an atom in *S*;
- *l* is a ground expression of the form (call p t_1 t_2 .. t_n), and the evaluation of p with arguments t_1,t_2,, t_n returns a non-nil value;
- *l* is an expression of the form (not *a*), and the atom *a* is not a consequent of *S* and *X*;

- there exists a substitution v and an axiom `(:-` a n_1 C_1 n_2 C_2 ... n_n C_n`)` in X such that $l = a^v$ 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, ..., 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 several 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** (or **mgs**) 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))`.

4.7 Task

A **task atom** is an expression of the form

`(`s t_1 t_2 ... t_n`)`

where s is a task symbol and the arguments $t_1, t_2, ..., t_n$ are *terms* or *call-terms*. The task atom is **primitive** if s is a primitive task symbol, and it is **compound** if s is a compound task symbol.

4.8 Operator

An **operator** is a list having either of the following forms:

`(:operator` h P D A `)`

```
(:operator h P D A c)
```

where

- h (the operator's **head**) is a primitive task atom in which no call terms can appear;
- P (the operator's precondition) is a list of logical atoms;
- D (the operator's **delete list**) is a list of logical atoms that contain no variable symbols other than those in h;
- A (the operator's **add list**) is a list of logical atoms that contain no variable symbols other than those in h.
- c (the operator's **cost**) is a number. If c is omitted, its default value is 1.

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.

Let S be a state, X be the list of axioms, 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 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 produced by executing o^u (or equivalently, h^u) in S is the state:

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

Here is an example:

```
S =                     ((has-money john 40) (has-money mary 30))
t =                     (!set-money john 40 35)
o =                     (:operator (!set-money ?person ?old ?new)
                           ((has-money ?person ?old))
                           ((has-money ?person ?old))
                           ((has-money ?person ?new)))
u =                     ((?person . john) (?old . 40) (?new . 35))
o^u =                   (:operator (!set-money john 40 35)
                           ((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) )
```

4.9 Method

A **task list** is a list of task atoms. The intent of this definition is to specify that the task atoms should be performed in exactly the order given in the list. 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-terms can appear;
- each C_i (which is called a **precondition** for the method) is either a conjunct or a tagged conjunct;
- 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 name are optional and if omitted a unique name will be assigned for each pair. These names have no semantic meaning to JSHOP, but are provided in order to help the user debug domain descriptions by looking at traces of JSHOP's behavior.

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 (:method h C_1 T_1 C_2 T_2 ... 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:

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

where *Call* is JSHOP's evaluation function (the function that evaluates the values of the call-terms in the form (call f t1 t2 .. tn)). 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)

$m =$
```
(:method (transfer-money ?p1 ?p2 ?amount)
         ((has-money ?p1 ?m1)
          (has-money ?p2 ?m2)
```

```
                 (call >= ?m1 ?amount))
                ((!set-money ?p1 ?m1 ( call - ?m1 ?amount))
                 (!set-money ?p2 ?m2 (call + ?m2 ?amount))))
```

$u =$ `((?p1 . john) (?p2 . mary) (?amount . 5))`

$h^u =$ `(transfer-money john mary 5)`

$C_1{}^u =$
```
((has-money john ?m1)
 (has-money mary ?m2)
 (call >= ?m1 5))
```

$T_1{}^u =$
```
((!set-money john ?m1 (call- ?m1 5))
  (!set-money mary ?m2 (call + ?m2 5))))
```

$v =$ `((?m1 . 40) (?m2 . 30))`

$(C_1{}^u)^v =$
```
((has-money john 40)
 (has-money mary 30)
 (call >= 40 30)))
```

$(T^u)^v =$
```
((!set-money john 40 (call - 40 5))
  (!set-money mary 30 (call + 30 5)))
```

$call((T^u)^v) =$ `((!set-money john 40 35) (!set-money mary 30 35))`

4.10 Plan

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

$$(\; h_1 \;\; c_1 \;\; h_2 \;\; c_2 \;\; \ldots \;\; h_n \;\; c_n \;)$$

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

4.11 Planning Domain

A **planning domain** is a list of axioms, operators, and methods. A **planning problem** is a 3-tuple (S,T,D), where S is a state, T is a task list, and D is a domain representation. Suppose that (S,T,D) is a multi-planning problem, where T is the multi-task list $(t_1 \;\; t_2 \;\; \ldots \;\; t_k)$. If $P = (p_1 \;\; p_2 \;\; \ldots \;\; p_n)$ is a plan, then we say that P **solves** (S,T,D), or equivalently, that P achieves T 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: T and P are empty (i.e., k=0 and n=0).
- Case 2: t_1 is a primitive task, p_1 is a simple plan for t_1, and $(p_2 \;\; \ldots \;\; p_n)$

achieves $(t_2 \ldots\ t_k)$ from result(S,p_1).

- Case 3: t_1 is a compound task, and there is a simple reduction $(r_1\ \ r_2\ \ \ldots\ \ r_l)$ of t_1 in S such that P achieves $(r_1\ \ r_2\ \ \ldots\ \ r_l\ \ t_2\ \ \ldots\ \ t_k)$ from S.

5 Class Hierarchy in JSHOP

The simplified class hierarchy for JSHOP classes can be seen in Figure 1. This figure omits some built in classes of Java to make the figure easier to understand.

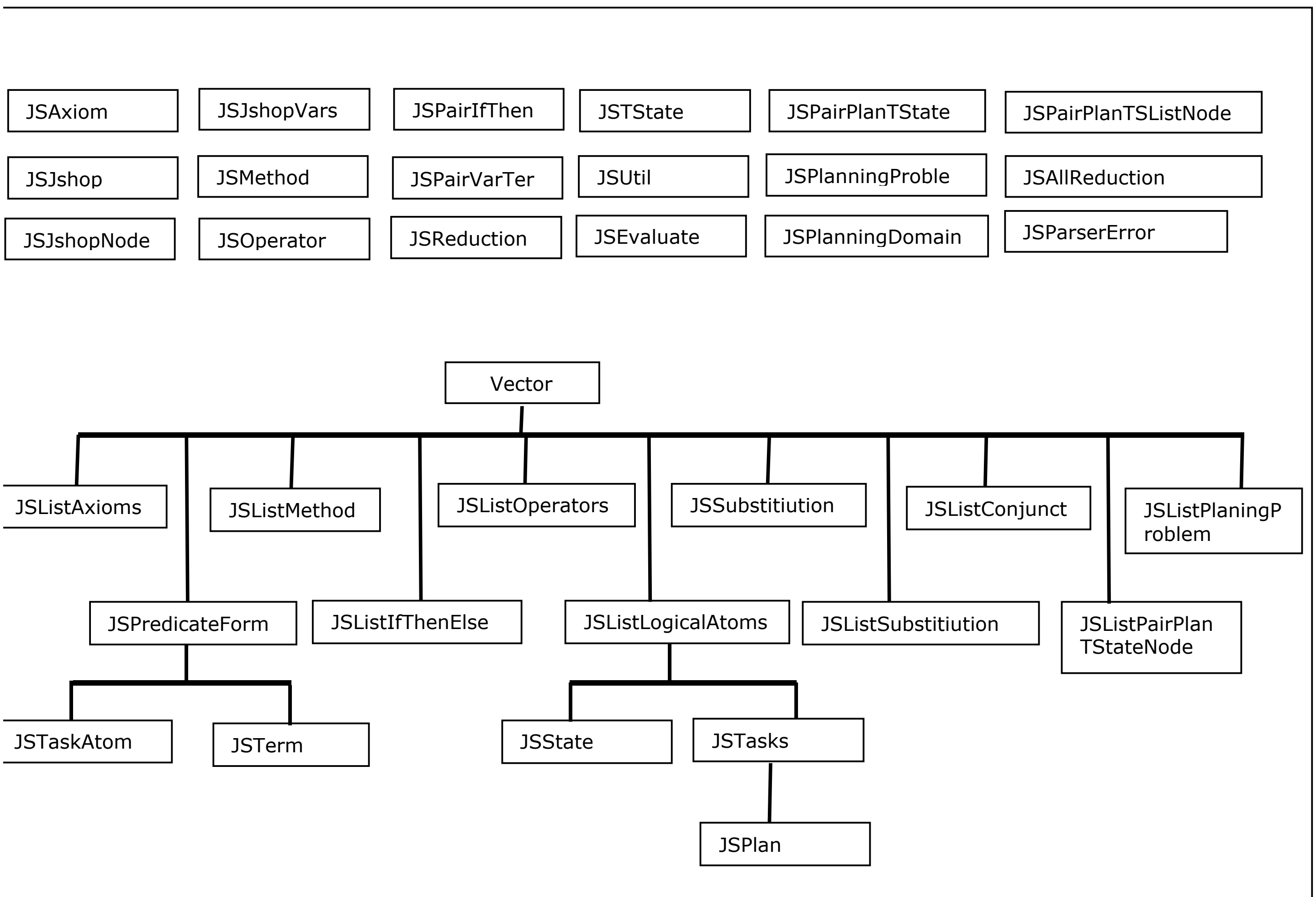

Figure 1 *Class Hierarchy of JSHOP. Vector is a built in class for representing Lists.*

6 Naming Conventions in JSHOP

As the reader can easily detect there is a kind of systematic naming of the classes defined in JSHOP. This system most of the time make the code more readable and understandable. When adding new classes to JSHOP please pay attention to the following three rules :

- All the class names start with "*JS*" letters. Remember that Java is a case sensitive language so capital letters make difference. If the rest of the class name is a compound word that contain more than one meaningful words each word should start with a capital letter (example : *JSTaskAtom*)
- The class name *JSListX* stands for a class that has a data structure , list of objects of type *X*. (example : *JSListMethod* , *JSListOperator*)
- Class named as *JSPairXY* represents a class that has two variables named *X* and *Y* such that the main algorithm of the class depends on those two variables. Usually it is the case that the name of the class is self-explanatory for the functionality of the class (example *: JSPairIfThen*, *JSPairVarTerm*)

7 Classes defined in JSHOP

This section lists and explains the functionality of each class defined in JSHOP. For each class its functionality, instance variables, the methods and the corresponding SHOP definition (if there is any) will be explained.

Any method that has the same name as an instance variable returns the value of that variable. Almost all of the instance variables defined in JSHOP classes are private variables so in all classes you may find such methods. Those methods will not be explained for the sake of simplicity.

7.1 JSJshopVars

The global variables for JSHOP are defined in this class. There is no method defined in this class. As usual, global variables are defined as class variables.

7.1.1 Class Variables

`static char LastCharRead` The last character read by the parser. This variable is not used.

static int VarCounter
This variable is used to create unique variable names for methods that we refer to as *standarizers*. Whenever a method, an operator or an axiom is applied the value of the counter is increased. To create unique names VarCounter is appended to the name of the variable

---

static String errorMsg
The error message to be displayed on the standard output.

---

static boolean flagParser
If true flags will appear when parsing the file. This varible is not used

---

static boolean flagPlanning
If true flags will appear during planning . This varible is not used.

---

static int flagLevel
Corresponds to the verbose level in SHOP. The greater it is the more information will be printed on the standard output.

---

static boolean flagExit
If a parser error occours and the value of *flagExit* is true then the program terminates. If the value of flagExit is false, the program returns from the main procedure.

```
static int leftPar = 0x0028
static int rightPar = 0x0029
static int apostrophe = 0x0027
static int colon = 0x003A
static int semicolon = 0x003B
static int exclamation = 0x0021;
static int interrogation = 0x003F;
static int minus = 0x002D;
static int equalT = 0x003D;
static int greaterT = 0x003E;
static int lessT = 0x003C;
static int coma = 0x002C;
static int astherisk = 0x002A;
static int rightBrac = 0x005D;
static int leftBrac = 0x005B;
static int verticalL = 0x007C;
static int plus = 0x002B;
static int whiteSpace = 0x0020;
static int percent = 0x0025;
```

The Unicode values for the special characters used in the parser.

7.2 JSPredicateForm

The class represents a **literal** which is any of the following:

- a logical atom (predicate) *a*;
- an expression of the form `(not a)` where *a* is a logical atom (the intended meaning is that the expression is true if *a* is false).

This class is extended from Vector class to store a list of JSTerm objects, which will be the arguments of a predicate. The JSPredicateForm class has a constructor that can read an input file and initialize its contents from that file.

7.2.1 Instance Variables

No instance variables are declared for this class.

7.2.2 Method Details

`JSPredicateFormInit( StreamTokenizer tokenizer)`

Method to parse the input predicate

`void print()`

Prints the predicate to standard output

`StringBuffer toStr()`

Prints the contents of the predicate into a string buffer.

`JSPredicateForm clonePF()`

Returns a new JSPredicateForm object that has the same content as this one

---

`JSPredicateForm applySubstitutionPF(JSSubstitution alpha)`

Returns a new JSPredicateForm object such that the variables appearing in the substitution (alpha) are replaced with the corresponding terms.

`JSSubstitution matches(JSPredicateForm t)`

Returns a substitution that will unify this predicate with the one given as the parameter provided that there are no current bindings for the variables.

`JSSubstitution matches(JSPredicateForm t, JSSubstitution alpha)`

Returns a substitution that will unify this predicate with the one given as the parameter provided that there is a current binding (alpha) for some of the variables. Below is the pseudo-code for this function:

> **if** number of terms in two predicates are not same
>
> **return** a failure substitution (a substitution, such that subs.fail() is true)

```
if the first elements of the predicates (names) are not literally the same then
        return a failure substitution
Create a new substitution beta that is cloned from alpha (current bindings)
for every term in this predicate do
    if the current term is equal to the corresponding term in the second predicate
    then
        continue with the next term
    else if the current term matches with the corresponding term in the second
    predicate then
        add the substitution that will unify them to beta
    else
        return a failure substitution
end for
return beta
```

`boolean equals( JSPredicateForm t)`

Checks if this predicate is equal to the predicate given in the parameter

---

`public JSPredicateForm standarizerPredicateForm()`

standarizer function for predicates.

`public JSPredicateForm applySubstitutionPF(JSSubstitution alpha)`

Replaces each occurrence of each variable in predicate form with the corresponding term in substitution *`alpha`*.

7.3 JSTerm

As defined in section 4.2 a *term* is either a variable symbol, a constant symbol, or an expression of either of the forms

$(list\ t_1\ t_2\ \ldots\ t_n)$ or $(f\ t_1\ t_2\ \ldots\ t_n)$

where *f* is a function symbol and each t_i is a term. Also *call-term* is an expression of the form:

$(call\ f\ t_1\ t_2\ \ldots\ t_n)$

where *f* is the name of an attached procedure and each t_i is a term or a call-term. JSTerm class implements these two definitions. JSTerm is an extended form of JSPredicateForm class. JSTerm class has some flags that identify whether its content is a constant, variable, a function or a *call-term*. It has a constructor that can initialize its contents from an input file.

7.3.1 Instance Variables

`boolean isVariable`
True if the term is a variable

`boolean isConstant`
True if the term is a constant

`boolean isFunction`
True if the term is a function

`boolean isEval`
True if the term should be evaluated (i.e., the term is defined as $(call\ f\ t_1\ t_2\ \ldots\ t_n)$)

7.3.2 Method Details

`public void print()`
Prints the term.

`public void printList()`
Prints the term that is in the form *(list* $t_1\ t_2 \ldots t_n$*)* which is internally represented as $(.\ t_1\ (.\ t_2\ .\ (.t_3\ nil\ ))\ .$

`public JSTerm parseList(InputStream)`
Parses the term that is in the form *(list* $t_1\ t_2 \ldots t_n$*)* which is internally represented as $(.\ t_1\ (.\ t_2\ .\ (.t_3\ nil\ ))\ .$

`public JSTerm cloneT()`
Creates a copy of the term

`public JSTerm applySubstitutionT(JSSubstitution alpha)`
Returns a new term object such that the variables in this object are instantiated with the corresponding terms in the substitution `alpha` and the term is evaluated if `isEval` for this object is `true`.

`public JSSubstitution matches(JSTerm t)`
Returns the most general unifier of this term and the parameter term when there is no prior binding for the variables.

`public JSSubstitution matches(JSTerm t, JSSubstitution alpha)`

Returns the most general unifier of this term and the parameter term unify when there are prior bindings for the variables given in `alpha`. .

`public boolean equals(JSTerm t)`

Returns *true* if the two terms are lexically equal

```
public boolean isVariable()
public boolean isConstant()
public boolean isFunction()
public boolean isEval()
public void makeFunction()
public void makeVariable()
public void makeConstant()
public void makeEval()
```

Methods for checking and setting the values of instance variables.

`public boolean isGround()`

Returns true if there is no variable in the term.

`public JSTerm standardizerTerm()`

Replaces the names of the variables with unique new names and returns a new term

---

`public JSTerm call()`

If the term is a constant term then it just returns itself. If it is a variable term it returns failure because any term to be evaluated should be grounded. If this is a function then the first element gives the operator name and the rest of them are the arguments. Returns the term computed by the JSEvaluate.applyOperator function.

7.4 JSListLogicalAtoms

This class represents a conjunct as defined in section 4.3. The class is an extension from the Vector class and contains a list of JSPredicateForm objects. It can be seen as a form:

$(P_1\ \ P_2\ \ldots\ P_n)$ or $(\text{:first}\ P_1\ \ P_2\ \ldots\ P_n)$

where the P_i's are literals.

It has a constructor that can initialize its contents from an input file. There are functions for printing and standardizing its contents.

7.4.1 Instance Variables

`string label`

If this is a tagged conjunct, then *label* contains the string "first"; otherwise *label* is null.

`string name`

The name of the conjunct.

`boolean varlist`

The value is true if the whole list is to be constructed from a variable that contains a list of logical atoms. For example, consider the following operator definition:

```
(:operator (!assert ?g)
                ()
                ()
                ?g
                0)
```

In the above expression, the operator's *add list* is given by the variable `?g`.The value of `?g` should have the form $(P_1 \quad P_2 \; ... \; P_n)$, where the P_i's are literals. When parsing an operator definition that contains a variable such as `?g`, JSHOP sets the value of *varlist* to *true*, and when searching for a plan, JSHOP creates the addlist of the *operator* using the bound value of the variable `?g`.

7.4.2 Method Details

`void addElements(JSListLogicalAtoms l)`

Appends the contents of `l` to this list. Used for creating new conjuncts.

`void print()`

Usual print method.

`JSListLogicalAtoms standarizerListLogicalAtoms()`

Usual standarizer function

`JSListLogicalAtoms`
`ApplySubstitutionListLogicalAtoms (JSSubstitution alpha)`

This method returns a new object such that in this new object the variables that are defined in the substitution `alpha` are replaced with the corresponding terms.

`JSListLogicalAtoms Cdr()`

Returns a new object such that it is same as this one but it does not have the first element in the list.

`public String Label()`
`public String Name()`
`public void setName( String newName)`

Methods for checking and setting the values of instance variables.

7.5 JSListConjuncts

The JSListConjuncts class is extended from the Vector class, It represents a structure of the form:

(C_1 C_2 C_3 ... C_n)

where each C_i . is a conjunct (list of logical atoms). This structure is used for defining the tail of an axiom. The constructor of the class reads from an input file and initializes the contents of its list.

7.5.1 Instance Variables

No instance variable is declared.

7.5.2 Method Details

void ***print****()*

Prints all conjuncts.

JSListConjuncts ***standarizerListConjuncts()***

Returns a new axiom in which all the variable names are changed with the new unique names.

7.6 JSAxiom

This class represents an axiom defined in the planning domain. The expected form is:

(:- a [n_1] C_1 [n_2] C_2 [n_3] C_3 ... [n_n] C_n)

7.6.1 Instance Variables

JSPredicateForm head

Represents the head of the axiom

JSConjuncts tail

Tail of the axiom

7.6.2 Method Details

public void ***print****()*

Prints the axiom to standard output in the form: (:- a C_1 C_2 C_3 ... C_n)

JSPredicateForm ***head****()*

JSListConjuncts ***tail****()*

JSAxiom ***standarizerMet****()*

Returns a new axiom in which all the variable names are replaced with new unique names.

7.7 JSListAxioms

JSListAxioms class stores all the axioms defined in the planning domain. It is extended from the Vector class.

7.7.1 Instance Variables

No instance variable is declared.

7.7.2 Method Details

`void` **`print`**`()`

Prints all axioms

`JSListSubstitution` **`TheoremProver`***`(JSListLogicalAtoms conds,JSState S, JSSubstitution alpha,Boolean findall)`*

Finds all the substitutions that make *`conds`* (a list of conjuncts) true in the current state *S* with the bindings *alpha* of the variables. If the parameter *`findall`* is false it returns only the first substitution.

```
procedure TheoremProver(C, S, alpha , all)
let  answers be an empty JSListofSubstitution
 if C is empty then
     add an empty substitution to answers
     return  answers
end

 l = the first literal in C; B = the remaining literals in C

 if l is an expression of the form (not e) then
          if TheoremProver(e, S, alpha, false)  is failure  then
                 return TheoremProver(B, S, alpha , all)
          else
                 return empty ListofSubstitution indicating failure
          end
else if l is an expression of the form (call e) then
   if  call( applySubstitution (e)) is not failure  then
       return TheoremProver(B, S, alpha,all)
   else
       return nil
```

```
 end
end

for every atom s in S that unifies with l
   let u be the unifier
   for every v in TheoremProver( B, S, compose-substitutions(alpha, u))
        insert compose-substitutions(u,v) into answers
end

for every axiom x whose head unifies with l
    let u be the unifier
    if tail(x) contains a conjunct D such that TheoremProver(append(D,B), S,
    compose-substitutions(alpha, u)) is not failure then
        let D be the first such conjunct
        for every v in TheoremProver(append(D,B),
S,composesubstitutions(alpha,u))
              insert compose-substitutions(u, v) into answers
         end
end
return answers
end TheoremProver
```

7.8 JSMethod

This class represents a method defined in the planning domain. Methods have the form:

`(:method` $h\ C_1\ T_1\ C_2\ T_2\ \ldots\ C_k\ T_k$`)`

7.8.1 Instance Variables

`JSTaskAtom head`

Head of the method (`h`)

`JSListIfThenElse ifThenElseList`

The tail of the method where the preconditions and the task lists are stored.

`boolean notDummy`

For any method that is defined in the domain, *notDummy* is false. For an empty method it is true indicating that it is meaningless.

7.8.2 Method Details

`JSMethod` **`standarizerMet`***`()`*

Returns a new method in which all the variable names are replaced with the new unique names.

---

void ***print***()

Prints the method to the standard output in the form: `(:method` $h\ C_1\ \ T_1\ \ C_2\ T_2$ ... `)`

JSTaskAtom ***head***()

JSListIfThenElse ***ifThenElseList***()

---

boolean ***notDummy()***

7.9 JSListMethods

This is a subclass of *Vector*. JSListMethod class stores all the methods defined in the planning domain. .

7.9.1 Instance Variables

No instance variables declared

7.9.2 Method Details

void ***print()***

Prints all the methods defined in the domain .

JSReduction ***findReduction****(JSTaskAtom task, JSState s, JSReduction red , JSListAxioms axioms)*

This method returns a reduction of task relative to the state s and a list of axioms. If red is a dummy reduction, it will search all the methods. If red is the reduction of a method m, it will start searching in all methods listed after m

---

JSAllReduction ***findAllReduction****(JSTaskAtom task, JSState s, JSAllReduction red , JSListAxioms axioms)*

This method returns all the reductions of task relative to the state s and a list of axioms. If red is a dummy reduction, it will search all the methods. If red is the reduction of a method m, it will start searching in all methods listed after m

7.10 JSOperator

This class stands for an operator in SHOP. It has a constructor that can initialize the head, precondition, add list, and the delete list of the operator. With the exception of the print() method, all other methods in this class access the instance variables.

7.10.1 Instance Variables

JSTaskAtom head;
The head of the operator

---

double cost
Cost of the operator

JSListLogicalAtoms deleteList;
Delete list for the operator

JSListLogicalAtoms addList;
Add list for the operator

7.10.2 Method Details

JSTaskAtom head()
JSListLogicalAtoms addList()
JSListLogicalAtoms deleteList()
Methods for accessing instance variables

---

void print()
Prints the operator in the SHOP format.

JSOperator standarizerOp()
This method is used for replacing all the variables in the operator with the unique variable names.

7.11 JSListOperators

This class extends the Vector class. Its purpose is solely to hold all the operators defined in the domain.

7.11.1 Instance Variables

No instance variables declared.

7.11.2 Method Details

void print()
Prints all operators defined in the domain.

7.12 JSPlanningDomain

This class stores the domain definition that consists of the methods, operators and the axioms. Other than the methods for accessing and printing its variables, it has a constructor that can read an input file and initialize its variables. An outstanding method provided by this class is "solveAll" (see below).

7.12.1 Instance Variables

`String name;`

Name of the planning domain.

`JSListAxioms axioms`

The axioms defined for this domain.

`JSListOperators operators`

The operators defined for this domain.

`JSListMethods methods`

The methods defined for this domain.

7.12.2 Method Details

`void parserOpsMethsAxs (StreamTokenizer tokenizer)`

Method that parses the input file and initializes the methods, operators and axioms.

`JSPairPlanTSListNodes solve(JSPlanningProblem prob, Vector listNodes)`

This method calls the *seekplan* function defined in the Tasks class to solve a given problem "prob" and returns the first plan along with the derivation tree.

`JSListPairPlanTSListNodes solveAll(JSPlanningProblem prob, boolean All)`

This method calls the *seekplanAll* function defined in the Tasks class to solve a given problem "prob" and returns all the plans along with the derivation trees. If the value of *All* is *false* it returns only the first plan.

`public void print()`

Prints the axioms, operators and methods defined in this domain.

`public JSListMethods methods()`
`public JSListAxioms axioms()`
`public JSListOperators operators()`

Methods for accessing instance variables.

7.13 JSPairVarTerm

This class represents the dotted pairs in the substitution. The form of such an expression is :

$(x_1\ .\ t_1)$

where x1 is a variable and t1 is the corresponding term.

7.13.1 Instance Variables

JSTerm var
Contains the variable

JSTerm term
Contains the term

7.13.2 Method Details

*public JSTerm **var**()*

---

*JSTerm **term**()*

---

*JSPairVarTerm **clonePVT**()*
Returns a new pair that has the same variable and term values

---

*void **print**()*
Usual print function

---

*JSPairVarTerm **standarizerPVT**()*
Usual standarizer function

7.14 JSSubstitution

This class represents for a list of dotted pairs of the form

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

JSSubstitution class is extended from Vector class to hold a list of objects of the type JSPairVarTerm. It is used as a list of current variable bindings. When two terms are checked for unification a list of bindings that makes the two unify will be generated, an empty list does not mean that the two terms don't unify, it just shows that no variables have to be bound. If the objects don't unify a failed substitution is returned.

7.14.1 Instance Variables

`boolean fail`

True if the substitution is a failure

7.14.2 Method Details

`JSTerm instance(JSTerm var)`

Returns the corresponding term for the given *var* if *var* appears in one of the dotted pairs

---

`JSSubstitution cloneS()`

Returns a new JSSubstitution object that has the same content with this one.

`boolean fail()`

Indicates if this is a failed substitution

`void assignFailure()`

Makes a failed substitution

`void addElements(JSSubstitution Sub2)`

Applies Sub2 to the right-hand-side of each item in this substitution, and appends all items in Sub2 whose left-hand-sides are not in this one.

`void removeElements(JSSubstitution l)`

Removes the elements from this list if they are also listed in "l"

`void print()`

Prints the contents of the substitution

`JSSubstitution standarizerSubs`

Replaces all the variables in the list with the unused variable names.

7.15 JSListSubstitution

This is an extension of Vector class. The aim of the class is to store a list of substitution objects. It is generally the return type for the functions that check whether the current state satisfies a condition. Such functions return all the substitutions that will satisfy that condition. So if there is no element in this list this indicates a failure.

7.15.1 Instance Variables

There are no instance variables for this class.

7.15.2 Method Details

`boolean fail()`

Returns true if this is a failed substitution .

`void print()`

Prints all the substitutions in list.

7.16 JSPairIfThen

In SHOP a method can have different decompositions for different preconditions.

(:method head C_1 T_1 C_2 T_2 ... C_n T_n)

JSPairIfThen class stands for the C_i T_i pairs in the form above. C_i (ifPart) is a conjunct and T_i (thenPart) is a list of tasks. This class has a constructor that can initialize the ifPart and the thenPart from an input file. It also has methods that allow to accesses instance variables, print them and standardize the variables used in this pair.

7.16.1 Instance Variables

`JSListLogicalAtoms ifPart;`

The conjunct part of the pair.

`JSTasks thenPart;`

The task list for the pair.

`String name;`

The name of the pair.

7.16.2 Method Details

`JSListLogicalAtoms ifPart()`
`JSTasks thenPart()`
`String name();`
`String setName(String newName);`

methods for accesing the instance variables

`void print()`

Usual print function

`JSPairIfThen standarizerPIT()`

Returns a new pair that has its variable name changed with a new name.

7.17 JSListIfThenElse

This class stores a list of PairIfThen objects. The class represents the tail part (C_1 T_1 C_2 T_2 ... C_n T_n) of the method. It has a constructor that can initialize its elements from an input file. Like other classes it has the functions for printing and standarizing its contents. One outstanding method defined in this class is the "evalPrec" function (see details below).

7.17.1 Instance Variables

No instance variable is declared.

7.17.2 Method Details

`void print()`

Prints the contents

`JSTasks evalPrec(JSState s, JSSubstitution alpha,JSListAxioms axioms)`

Given the State, axioms and current bindings (alpha) this function checks all the pairs in order to find one pair whose "ifPart" can be satisfied. Whenever such a pair found, the substitution that makes the "ifPart" true is applied to the "thenPart" and those list of tasks are returned. If none of the pairs can be satisfied then a task list of failure is returned.

`Vector evalPrecAll(JSState s, JSSubstitution alpha,JSListAxioms axioms)`

Given the State, axioms and current bindings (alpha) this function checks all the pairs in order to find one pair whose "ifPart" can be satisfied. Whenever such a pair found, for every substitution *v* that makes the "ifPart" satisfied, *v* is applied to the "thenPart" and the new *thenPart* is added to a list *l* . If none of the pairs are satisfiable then a list will be empty indicating the failure. The return value of this function is *l*

`JSListIfThenElse standarizerListIfTE()`

Standarizes the variables

7.18 JSReduction

This is a data structure that stores a list of tasks and a method that produced this task list. All the functions defined in this class are for accessing the instance variables. There are two constructors for the class one takes a method and a task list and one with no parameters. The one with no parameters creates a dummy reduction.

7.18.1 Instance Variables

`JSMethod selectedMethod;`

The method used for creating this reduction.

`JSTasks reduction;`

The resulting subtask if the selected method is applied.

7.18.2 Method Details

`JSMethod selectedMethod()`
`public JSTasks reduction()`

Methods for accessing instance variables.

`boolean isDummy()`

Returns true if the selected method is a dummy method.

7.19 JSAllReduction

This class contains a list of tasks and a method that produced this list of task list. All the functions defined in this class access the instance variables.

7.19.1 Instance Variables

`JSMethod selectedMethod;`

The method used for creating this reduction.

`Vector reduction;`

The resulting subtask if the selected method is applied.

7.19.2 Method Details

`JSMethod selectedMethod()`
`public Vector reduction()`

Method for accessing instance variables

`boolean isDummy()`

Returns true if the selected method is a dummy method.

7.20 JSTaskAtom

This class represents a task in SHOP and extends the JSPredicateForm class. The outstanding method implemented in this class is *seeksimpleplan*. The contents of the class can be initialized from an input file.

7.20.1 Instance Variables

boolean isPrimitive

True if the task is a primitive task.

---

boolean isCompound

True if the task is a compound task.

7.20.2 Method Details

JSPairPlanTState seekSimplePlan(JSPlanningDomain dom, JSTState ts)

This method searches for all the operators defined in the planning domain for one that is applicable to this task atom. If it finds one, it returns a PairPlanTstate object such that the plan variable of that object will contain the grounded operator head and the Tstate variable of that object contains the state that will result from applying this operator.

`JSReduction reduce(JSPlanningDomain dom,JSState s, JSReduction red)`

Calls the *findReduction* function defined in ListMethods class to find a reduction for this compound task.

JSTaskAtom applySubstitutionTA(JSSubstitution alpha)

Returns a new task atom object that is same as this one except the variables bounded in alpha are substituted in the new one with their corresponding values.

JSTaskAtom cloneTA()

Returns a new TaskAtom object that has the same content as this one.

---

boolean isGround()

Returns true if this task atom is ground.

JSTaskAtom standarizerTA()

Standarizes this task atom.

JSJshopNode findInList(Vector list)

Searches in a List of JshopNodes to find the one with the same TaskAtom in it.

---

boolean isPrimitive()
void makePrimitive()
void makeCompound()
Methods changing the truth values of instance variables.

7.21 JSTasks

This class extends JSListLogicalAtoms, the difference is it has a list of task atoms. The main function of JSHOP, *seekplanAll,* is implemented in this class. The contents of this class can be initialized from an input file. An empty task list is not a failing plan, it may be the case that the plan is doing nothing. So an instance variable "fail" keeps track of this situation.

7.21.1 Instance Variables

boolean fail

True if the task list indicates a failure.

7.21.2 Method Details

JSPairPlanTState seekPlan(JSTState ts, JSPlanningDomain dom, JSPlan pl, Vector listNodes)
This method finds the first plan for the list of tasks in this object.
The pseudo code for this function:

```
Let t be the  first task atom in list and rest the rest of the atoms
        if  t is primitive then
           Answer= t.seek_simpleplan
           if  answer is a failure
                return failure
           else
                return rest.seekplan()
        else
                Find the first reduction  "red" for t by calling t.reduce
                while (red is not a dummy reduction)
                        Subtasks= red.reduction
                        Add the task atoms in "rest"  to subtasks.
                        Answer= subtasks.seekplan
                        if  Answer has not a falied plan
                                return  Answer
                        else
                                red is the next reduction
                end while
        return  an empty PairPlanTState object with failed plan
```

JSListPairPlanTStateNodes seekPlan(JSTState ts, JSPlanningDomain dom, boolean All)

This method finds all plans for the tasks in this object. If *All* is false it returns the first plan. It also returns a list that contain tuples of plan, state, global additions list, global deletions list and the derivation tree. This can be useful if one is interesting in observing the whole derivation of the plan.

---

boolean fail()
void makeFail()
void makeSucceed()

Methods for changing the truth value of the instance variables.

JSTasks applySubstitutionTasks(JSSubstitution alpha)

Returns a new task list that is same as this one except the variables bounded in alpha are changed in the new one to their corresponding values.

---

boolean contains(JSTaskAtom t)

Returns true if the task atom "t" is in the list of this object.

JSTasks cloneTasks()

Returns a new task list that is the same as this one.

JSTasks cdr()

Returns a new task list that has the same content with this one except the first element is removed.

JSTasks standarizerTasks()

Returns anew task list such that all the variables names in the list are replaced with new names.

7.22 JSPlan

This class extends JSTasks (a plan is a list of primitive tasks) . The only extension to JSTasks class (which represents a list of tasks) is an instance variable that shows whether a plan failed or not. There are also some functions to manipulate this variable.

7.22.1 Instance Variables

boolean isFailure

True when there is no plan, an empty list of tasks does not mean a failure.

7.22.2 Method Details

void assignFailure()
Assigns true value to "isFailure"

---

boolean isFailure()
Returns the value of "isFailure"

---

void addElements(JSPlan pl)
Append the contents of the plan p1 to this plan

7.23 JSState

This class is extends the JSListLogicalAtoms class. It is used for representing the current state of the world. The functions defined in this class can check whether a predicate, conjunct can be satisfied with in the current state. The effects of applying an operator are also defined in this class.

7.23.1 Instance Variables

No instance variables are declared.

7.23.2 Method Details

JSTState applyOp(JSOperator op, JSSubstitution alpha, JSListLogicalAtoms addL,JSListLogicalAtoms delL)
Returns a new TState object that has the new state resulting from applying the operator and a delete list and add list that shows which state atoms should be deleted or added from this state. This function does not change current state. "addL" and "delL" can be empty or may contain elements.

---

public JSSubstitution satisfies(JSListLogicalAtoms conds, JSSubstitution alpha, JSListAxioms axioms)

Tests if "conds" can be inferred from this (the current state) state and axioms relative to the substitution alpha. If "conds" can be inferred, it returns the first matching substitution else it returns the failed substitution

---

public JSListSubstitution satisfiesAll(JSListLogicalAtoms conds, JSSubstitution alpha, JSListAxioms axioms)
Tests if "conds" can be inferred from this (the current) state and axioms modulo the substitution alpha. If 'conds' can be inferred, it returns all of the satisfying substitutions else it returns an empty list indicating a failure.

---

public JSListSubstitution satisfiesTAm(JSPredicateForm t, JSSubstitution alpha)

It searches all atoms in the current state to find the ones that unify with *t* under the current variable bindings given in *alpha*. The list unifiers or an empty list is returned. –

7.24 JSTState

This class stores a target state and the list of additions and deletions that must be performed on the current state to reach the target state. This class has no use in the SHOP algorithm. This is useful for systems integrated with SHOP which may need the list of changes in the state. When searching for a plan a list for global additions and deletions are propagated along the plan seeking functions so that whenever an operator is applied these lists are also updated.

7.24.1 Instance Variables

`JSState state`

Target state

`JSListLogicalAtoms addList`

Additions to current state to reach the target state

`JSListLogicalAtoms deleteList`

Deletions from current state to reach the target state

7.24.2 Methods Details

```
JSState state()
JSListLogicalAtoms addList()
JSListLogicalAtoms deleteList()
```

Methods for accessing instance variables

`void print()`

Prints the target state, add and delete lists.

7.25 JSPairPlanTState

This is a data structure that has two main variables of type JSPlan and JSTSate. It does not correspond to a form defined in SHOP. This class is used to store the plan and the resulting state if the plan is applied The methods defined in this class are as usual for accessing and printing the instance variables.

7.25.1 Instance Variables

`JSPlan plan`

The plan generated for a given problem

`JSTState tState`

The final state that will be produced when the plan is applied.

7.25.2 Method Details

JSPlan plan()
JSTState tState()
Functions for accessing instance variables

---

void print()
Prints the plan and the TState.

7.26 JSPlanningProblem

This class stores a planning problem, initial state and the task list to be accomplished. It has a constructor that can initialize its contents from an input file that contains a make-problem statement. The methods declared in this class are for assigning values to the instance variables and for accessing and printing them.

7.26.1 Instance Variables

String name
Name of the problem

JSState state
Initial state of the world

JSTasks tasks
Task list to be planned

String domainName
Name of the domain this problem belongs to

7.26.2 Method Details

void assignState(JSState aState)
Assigns the value of parameter "astate" to variable "state"

---

void makeTask(JSTaskAtom pred)
Assigns the value of parameter "pred" to variable "tasks"

JSState state()
JSTasks tasks()

Functions for accessing instance variables

---

void print()

Prints the initial state and the task list to be planned.

7.27 JSListPlanningProblem

This class is extended from Vector class to store a list of planning problem.

7.27.1 Instance Variables

No instance variables declared.

7.27.2 Method Details

void print()

Prints all the planning problems.

7.28 JSPairPlanTSListNodes

This is a data structure that has two main variables of type JSPairPlanTSate and a list (vector) of JSJShopNode .This class is used to store the plan, the resulting state if the plan is applied and the tree that shows the whole decompositions to generate that plan. The methods defined in this class are as usual for accessing and printing the instance variables.

7.28.1 Instance Variables

JSPairPlanTState planS

The plan and the final state generated for a given problem

Vector listNodes

Tree that shows the decomposition of tasks to generate that plan.

7.28.2 Method Details

JSPairPlanTState planS()
Vector listNodes()

Functions for accessing instance variables

---

void print()

Prints the PairPlanTState and the tree.

7.29 JSListPairPlanTStateNodes

This is a data structure extended from Vector class. The class represents a list of objects of type *JSPairPlanTStateListNodes.*

7.29.1 Instance Variables

No instance variables

7.29.2 Method Details

void print()

Prints the PairPlanTState and the tree for every element.

7.30 JSJShop

This is the main class that parses and initializes the domain and problems.

7.30.1 Instance Variables

JSPlanningDomain dom
Contains the planning domain

JSPlanningProblem prob
Contains the planning problem

JSPlan sol;
Contains the solution plan

JSJshopNode tree
Contains the decomposition tree that generates the solution plan

---

JSPairPlanTSListNodes solution
Contains the solution plan and the final state that will be reached if the plan is applied

7.30.2 Method Details

JSJshopNode getTree()
Returns the decomposition tree

JSPairPlanTSListNodes getSolution()
Returns the solution plan

JSListLogicalAtoms getAddList()
The list of atoms that should be added the current state if the plan is applied.

JSListLogicalAtoms getDeleteList()
The list of atoms that should be deleted from the current state if the plan is applied

---

void testParser()
void parserFile(String libraryFile)
void processToken(StreamTokenizer tokenizer)
Methods for parsing the input file.

JSPlanningDomain dom()
JSPlanningProblem prob()
JSPlan sol()
JSJshopNode tree()
Methods for accessing the instance variables.

7.31 JSEvaluate

This class is used for evaluating the expressions in a predicate form or a term. For the time being it accepts the binary operations namely – "+,-,*,/,<,<=,>,>=, member , min , max" and unary operations " floor, ceil,not ".

7.31.1 Instance Variables

boolean fail;
True if the evaluation fails.

---

boolean BothInt;
True if the both parameters are integers.

7.31.2 Method Details

static float numericValue (JSTerm operant1)
Returns the numeric value that is stored in *operant1*. If the conversion fails it sets the *fail* flag.

JSTerm addsub (float operant1, float operant2,int optype)
Performs addition if optype is 1 and subtraction otherwise.

JSTerm mult*(float operant1 , float operant2)*
Performs multiplication operation

JSTerm div*(float operant1 , float operant2)*
Performs division operation

JSTerm greater(float operant1 , float operant2)
Checks if the value of operant1 is greater than operant2

JSTerm greaterequal(float operant1 , float operant2)
Checks if the value of operant1 is greater than or equal to operant2

JSTerm equal(float operant1 , float operant2)
Checks if the value of operant1 is equal to operant2

JSTerm floor(float operant)
Returns a term containing the floor value of the operant

JSTerm ceil(float operant)

Returns a term containing the ceiling value of the operant

JSTerm minOf*(float operant1 , float operant2)*
Returns the term that contains the minimum of operant1 and operant2

JSTerm maxOf*(float operant1 , float operant2)*

Returns the term that contains the maximum of operant1 and operant2

JSTerm member(JSTerm operant1 , JSTerm operant2)
Checks if the term operant1 is a member of the list that is in operant2

JSValue applyOperator (String op, JSTerm operant1, JSTerm operant2)
Checks the operant values and calls the appropriate binary function depending on the value of *op*.

---

int OperantNum (String op)
Returns the the number of operants for the given operator *op*.

JSTerm applyOperatorUnary (String op ,JSTerm operant1)
Calls the appropriate binary function depending on the value of *op*

7.32 JSJshopNode

This is a data structure that corresponds to a node of a tree. The data on each node is a task atom T and the children of a node contains the nodes for subtasks generated by reducing T. The whole derivation tree is represented as a adjacency list of JSJshopNode.

7.32.1 Instance Variables

JSTaskAtom atom
Task atom that the node stands for

Vector children
List of subtasks reduced from atom

7.32.2 Method Details

`JSJshopNode( JSTaskAtom a, Vector c)`
Class constructor that initializes atom to a and children to c

---

`public void print()`

Prints the atom and the children

`public JSTaskAtom atom()`

`public Vector children()`
Method for accessing instance variables.

7.33 JSParserError

This class is extended from the Error class. When the parser encounters something it does not expect it throws an error of type JSParserError. There are no instance variables or methods defined in this class.

7.34 JSUtil

This class includes a list of functions to read tokens from an input file and to print messages on standard error. It serves as a library function for I/O operations.

8 Difference between SHOP and JSHOP

The differences between SHOP and JSHOP can be can be grouped in two sets: syntactic changes in the domain definitions, and differences in functionality.

8.1 JSHOP Syntax

The table below gives examples of JSHOP's syntax, comparing it to the syntax used in SHOP:

Previous syntax	**New 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)	good ((weather-is good)
(distance ?u ?v ?w)	(distance ?u ?v ?w)

```
                (eval (<= ?w 3)))

             (  (distance ?u ?v ?w)
                (eval (<= ?w 0.5))))

    (:method (pay-driver ?fare)
           ((have-cash ?m)
              (eval (>= ?m ?fare)))
           `((!set-cash ?m ,(- ?m ?fare))))

    (:method (travel-to ?q)
           ((at ?p) (walking-distance ?p ?q))
            '((!walk ?p ?q)))

    (:method (travel-to ?y)
        ((at ?x) (at-taxi-stand ?t ?x)
         (distance ?x ?y ?d) (have-taxi-fare ?d))
       `((!hail ?t ?x) (!ride ?t ?x ?y)
          (pay-driver ,(+ 1.50 ?d)))
        ((at ?x) (bus-route ?bus ?x ?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 ?bus ?location)))

     (:operator (!ride ?vehicle ?a ?b)
          ((at ?a) (at ?vehicle ?a))
          ((at ?b) (at ?vehicle ?b)))

     (:operator (!set-cash ?old ?new)
          ((have-cash ?old))
          ((have-cash ?new)))

     (:operator (!walk ?here ?there)
          ((at ?here))
          ((at ?there)))
 ))
```

```
                (call <= ?w 3))

        bad    (  (distance ?u ?v ?w)
                  (call <= ?w 0.5)))

    (:method (pay-driver ?fare)
           ((have-cash ?m)
              (call >= ?m ?fare))
           ((!set-cash ?m ( call - ?m ?fare))))

    (:method (travel-to ?q)
           ((at ?p) (walking-distance ?p ?q))
            ((!walk ?p ?q)))

    (:method (travel-to ?y)
        by-taxi
        ((at ?x) (at-taxi-stand ?t ?x)
        (distance ?x ?y ?d) (have-taxi-fare ?d))
        ((!hail ?t ?x) (!ride ?t ?x ?y)
         (pay-driver (call + 1.50 ?d)))
        by-bus
       ((at ?x) (bus-route ?bus ?x ?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 ?bus ?location)))

     (:operator (!ride ?vehicle ?a ?b)
          ()
          ((at ?a) (at ?vehicle ?a))
          ((at ?b) (at ?vehicle ?b)))

     (:operator (!set-cash ?old ?new)
          ((have-cash ?old))
          ((have-cash ?old))
          ((have-cash ?new)))

     (:operator (!walk ?here ?there)
          ((at ?here))
          ((at ?here))
          ((at ?there)))
 ))
```

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

   `((distance downtown park 2)
    (distance downtown uptown 8)
    (distance downtown suburb 12)
    (at-taxi-stand taxi1 downtown)
    (at-taxi-stand taxi2 downtown)
    (bus-route bus1 downtown park)
    (bus-route bus2 downtown  uptown)
    (bus-route bus3 downtown  suburb)
```

```
(defproblem go-park-rich travel

   ((distance downtown park 2)
   (distance downtown uptown 8)
   (distance downtown suburb 12)
   (at-taxi-stand taxi1 downtown)
   (at-taxi-stand taxi2 downtown)
   (bus-route bus1 downtown park)
   (bus-route bus2 downtown uptown)
   (bus-route bus3 downtown  suburb)
```

(at downtown) (weather-is good) (have-cash 80))) '((travel-to park)))	(at downtown) (weather-is good) (have-cash 80))) ((travel-to park)))
(make-problem-set 'travel '(go-park-broke go-park-rich))	Not defined in JSHOP. Instead JSHOP parses and solves all the problems in a problem file.
(defparameter *downtown-broke-bad* \`((distance downtown park 2) (distance downtown uptown 8) (distance downtown suburb 12) (at-taxi-stand taxi1 downtown) (at-taxi-stand taxi2 downtown) (bus-route bus1 downtown park) (bus-route bus2 downtown uptown) (bus-route bus3 downtown suburb) (at downtown) (weather-is bad) (have-cash 0)))	Not defined in JSHOP
(do-problems travel :which ':all)	Not defined in JSHOP
(1 g ?y 6) an ordinary list	(list 1 g ?y 6)

Table 2 Simple travel domain defined in JSHOP's syntax and SHOP's syntax.

To summarize, the changes in syntax are as follows:

- Quotes, back quotes or commas are not used in JSHOP.
- *make-domain* is replaced with *defdomain.*
- *make-problem* is replaced with *defproblem.*
- *defparameter,make-problem-set, do-problem* and *do-problem-set* are not implemented in JSHOP. Instead JSHOP parses and solves all the problems in a problem file.
- Operators have preconditions in JHOP.
- Instead of using *eval (e)* where e is a lisp expression, JSHOP uses *(call f* t_1 t_2 *..*t_n*)*. *Call* is not as powerful as *eval*, because it can only compute a small subset of functions that *eval* can.
- 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-decomposition pairs. These names are optional.
- An ordinary list should be differentiated from a predicate or a function by inserting the label "list" before the first element of the list.

- "." can be used in JSHOP only for ordinary lists. Predicates and functions are treated different from ordinary lists. Thus, unlike in SHOP, one can not use a list to serve as a function in JSHOP.
- The JSHOP parser expects only alphanumeric characters and the special characters defined in JSJshopVars class (see section 7.1). The only whitespace characters that the parser expects, are the space and newline characters. If the parser encounters a tab or any other nonexpected characters, it returns an error message.
- In SHOP the following is valid.

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

  This kind of method, in which the sunbtasks are created on the fly, is not supported in JSHOP.

8.2 The usage of JSHOP

The command line below will run JSHOP :

```
java JSJshop  domaindef-file-name   problem-def-file-name   [numer-of-plans]  [flag-level]
```

where

- JSJshop is the name of the main class
- `domaindef-file-name` is the name of the file in which the domain definiton is specified,
- `problem-def-file-name` is the name of the file in which a list of problems are defined
- `numer-of-plans` is an optional parameter. It can be *one* or *all* . If the value of this parameter is *one* then only the first plan will be displayed, if the value is *all* then all of the plans will be displayed. The default value is *one*.
- Another optional parameter is `flag-level`, it can have any value in the range 1 to 10. The higher the value the more output is displayed.

Given the command line above JSHOP will parse the domain definition and problem definition files. If any parsing error occurs the program terminates. Otherwise JSHOP will solve all the problems defined in the problem definition file.

8.3 Functionality differences in JSHOP

The following functions are implemented in SHOP but not in JSHOP;

- Iterative Deepening search for a plan

- Statistics like depth and cost of the plan
- The check-for-loop option, which tells SHOP to checks for loops in the plan.
- JSHOP's unification mechanism does include the "occurs check."
- The depth first search for the plan has no depth limit.

When JSHOP finds a plan, it also returns the following items that SHOP does not return:

- The final state that will be reached upon applying the plan ,
- The list of state atoms that will be added and deleted from current state to reach the final state,
- The derivation tree for the plan