JSetL User's Manual

Version 1.0

ELIO PANEGAI, ELISABETTA POLEO, GIANFRANCO ROSSI
Università di Parma, Dip. di Matematica
Via M. D'Azeglio, 85/A, 43100 Parma (Italy)
gianfranco.rossi@unipr.it panegai@cs.unipr.it

28th December 2004

Abstract

This is the first edition of the user's manual for JSETL, a Java library that offers a number of facilities to support declarative programming like those usually found in constraint logic programming languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism. JSETL is intended to be used as a general-purpose tool, not devoted to any specific application. The manual describes all features of JSETL and it shows, through simple examples, how to use them.

JSETL has been developed at the Department of Mathematics of the University of Parma (Italy). It is completely written in Java. The full Java code of the JSETL library, along with sample programs and related material, is available at the JSETL WEB page `http://www.math.unipr.it/~gianfr/JSetL`.

Contents

1 Introduction

Generally speaking, declarative programming means focusing on *what* a program does, rather than on *how* it does.

Fundamental facilities of a programming language to support this programming paradigm are:

- high-level data abstractions (e.g., lists, sets, ...);
- powerful control abstractions (e.g., recursion, non-determinism, ...)
- facilities to abstract from the order in which statements and expressions are elaborated (e.g., side-effect freeness, constraint solving, logical variables, ...).

These facilities can be provided as built-in mechanisms of the language (e.g., in logic and functional programming languages, such as Haskell and Prolog) and/or they can be provided by a library written in the language itself (e.g., in ILOG Solver [9, 8] or in JSolver [3]).

JSetL is a Java library that endows Java with a number of facilities to support declarative programming like those usually found in constraint logic programming languages [7]: list and set data structures (possibly partially specified), nondeterminism, logical variables, unification and constraint solving over sets.

Logical variables differ from conventional programming language variables in that their value is not modifiable (neither by assignment nor by side-effects). Logical variables, along with unification and constraint solving, are fundamental tools to allow execution order to be immaterial, hence to support a declarative reading of programs. In JSetL declarative programming facilities, however, coexist with conventional and object-oriented programming facilities of Java (see, e.g., [1] for a general discussion about declarative vs. conventional programming).

The library is carried out as a Java package, and as such it is subject to the common rules of use defined by the language. The classes of the library must be saved into a folder named `JsetL`. To use JSetL in a program it is necessary to import the library by inserting the statement

```
import JSetL.*;
```

at the beginning of the source file. JSetL must be a sub-folder of the folder in which the classes that import JSetL are saved. Otherwise, the path from root to the library folder must be added to the variable `CLASSPATH`.

2 Logical variables and data structures

JSetL provides logical variables and two new kinds of data structures: sets and lists. These new features are implemented by three new classes, `Lvar`, `Lst`, and `Set`.

2.1 Logical variables: the class `Lvar`

A *logical variable* is an instance of the class `Lvar`. A logical variable with name VarName can be created by the following Java statement (a *logical variable declaration*)

```
Lvar VarName = new Lvar(VarNameExt,VarValue);
```

where VarNameExt is an optional external name, and VarValue is an optional value for the variable.

The *external name* VarNameExt is a string value which can be useful when printing the variable and the possible constraints involving it (if omitted, a default external name of the form "Lvar_n", where n is a unique integer, is assigned to the variable automatically).

The value associated with a logical variable is called a Lvar-*value*. The type of an Lvar-value can be either a primitive type (namely, boolean, char, byte, short, int, long, float, double), or a wrapper class (namely, Boolean, Character, Byte, Short, Integer, Long, Float, Double), or any library or user defined class, provided it supplies a method equals for testing equality between two instances of the class itself. In particular, an Lvar-value can be an instance of Lvar, Lst, or Set.

We say that a logical variable is *uninitialized* (or that it is *unknown*) if it has no Lvar-value associated with it or if its Lvar-value is an uninitialized logical variable or list or set. Otherwise, we say that the logical variable is *initialized*. Lvar-values other than uninitialized logical variables (or lists or sets) are said to be *known values*. A Lvar-*value* for a logical variable can be specified either when the variable is created or as the result of processing constraints involving the variable itself, in particular, equality constraints.

Example 1. Lvar *definitions*

```
Lvar x = new Lvar();               // uninitialized
Lvar y = new Lvar("y");            // uninitialized,
                                   // with ext'l name "y"
Lvar z = new Lvar(2);              // initialized (value 2)
Lvar v = new Lvar("w",'a');        // initialized (value 'a'),
                                   // with ext'l name "w"
Lvar w = new Lvar(new Integer(7)); // initialized (value 7)
Lvar t = new Lvar(x);              // uninitialized;
                                   // same as variable x
```

□

Note that, in the last definition of Example 1, the value of the Lvar t is the other Lvar x: both x and t are uninitialized. If later either x or t get some known Lvar-value v, both x and t become initialized by (the same value) v.

Basically, Lvar objects are manipulated through constraints (see Sect. 3). Constraints can be used to set the value of an uninitialized logical variable or to inspect it.

No constraint is allowed to modify the value of a logical variable, by replacing it with a new one. However, the standard Java assignment between Lvar objects is available and can be used to modify the value of a logical variable in a non-logical way.

Besides constraints, the class Lvar provides a number of utility methods that allow to read/write its value, to inspect it (e.g., to test if it is completely specified), to get the external name of a logical variable, and so on. These methods will be described in detail in Sect. 2.6.

2.2 Lists: the class Lst

A *list* is an instance of the class Lst whose value is a finite (possibly empty) sequence of arbitrary Lvar-values (the *elements* of the list), not necessarily all of the same type. A list with name LstName can be created in JSETL by the Java statement (a *list declaration*)

Lst LstName = **new** Lst(LstNameExt,LstElemValues);

where LstNameExt is an optional *external name* of the list (the default external name has the form "Lst_n", where n is a unique integer), and LstElemValues is an optional part which is used to specify the elements of the list (see below).

The *empty list* is denoted by the constant `Lst.empty`. For instance,

```
Lst e = Lst.empty;
```

defines an empty list with name `e`.

Alternatively, a named list can be introduced as the value of a logical variable and accessed through the logical variable itself. The statement

`Lvar LvarName =` **new** `Lvar(`**new** `Lst(LstNameExt,LstElemValues));`

defines a logical variable with name `LvarName` whose value is a list.

JSETL allows methods dealing with lists to be applied indifferently both to lists (i.e., instances of the `Lst` class), possibly through the associated list name, and to logical variables initialized to lists.[1]

Hereafter, we will usually use the term *list*–`Lvar` to refer to the type of a logical variable whose value, if any, must be a `Lst` object. This check is performed necessarily at run-time, and it possibly causes a `NotLstException` exception to be generated. Analogously, we will use the term *set*–`Lvar` to refer to the type of a logical variable whose value, if any, must be a `Set` object. Also this check is performed at run-time, and it possibly causes a `NotSetException` exception to be generated. All exceptions defined in JSETL (see Appendix A) are extensions of the class `java.lang.Exception` and, like any other exception in Java, they can be caught, thus avoiding the computation to terminate.

Lists can be created through the **new** operator, possibly using list declarations, or as the result of executing utility methods dealing with lists (in particular the element insertion and removal methods—see Sections 2.3 and 2.6). Lists can be either initialized or uninitialized, like logical variables. The value of a list (i.e., a `Lst`-*value*) is the sequence of its elements. In particular, when a list is created through the **new** operator, the value of the list can be specified in the following ways:

- by passing an array `l_elems` of elements $c_1, \ldots, c_n$ of type t, t any `Lvar`-value type,
  t `[] l_elems =` $\{c_1, \ldots, c_n\}$`;`
  as a parameter of the `Lst` constructor:
  new `Lst(LstNameExt,l_elems).`

- by passing the limits l and u of an interval $[l, u]$ of integer numbers which constitute the elements of the list as parameters of the `Lst` constructor:
  new `Lst(LstNameExt,l,u).`
  If $u > l$ the created list is the empty one.

- by passing a `Lst` or a *list*–`Lvar` object as a parameter of the `Lst` constructor,
  new `Lst(LstNameExt,LObj).`

In order to write lists in a more convenient way, we will make use of an abstract notation, which closely resembles that of Prolog. Specifically,

$$[e_1, e_2, \ldots, e_n]$$

[1] Using `Lst` objects—instead of logical variables initialized to `Lst` objects—allows in general a more controlled usage of the objects. For example, if m is a method not defined for `Lst`, trying to apply m to a list will be detected as an error at compile-time, whereas applying m to a logical variable initialized to a list will only cause an exception to be raised at run-time. Moreover, if l is an uninitialized list, any attempt to initialize it with a non-list object raises an exception, whereas if l is just an uninitialized logical variable, no error is detected.

is used to denote the list containing n elements e_1, e_2, ..., e_n, while

$$[\,]$$

is used to denote the *empty list*. Moreover,

$$[e_1, e_2, \ldots, e_n \mid R],$$

where R is a list, is used to denote a list containing the n elements e_1, e_2, ..., e_n, plus elements in R. In particular, if R is uninitialized, $[e_1, e_2, \ldots, e_n \mid R]$ represents an "*unbounded*" list, with elements e_1, ..., e_n and an unknown part R. Similar abstract notation will be introduced also to represent sets. Unbounded lists and sets are created using element insertion operations and constraints (see next sections).

Example 2. *List declarations*

```
Lst e = Lst.empty;               // the empty list
Lst l = new Lst("l");            // an uninitialized list, with ext'l name "l"
Lst i = new Lst("i",4,4+3);      // an initialized list,
                                 // with ext'l name "i" and value [4,5,6,7]
int[] v_elems = {2,4,8,3};
Lst v = new Lst("v",v_elems);    // an initialized list,
                                 // with ext'l name "v" and value [2,4,8,3]
```

□

Note that in a list the order of elements and the repetitions do matter (whereas they do not matter in a set): for instance, the list $[1, 2]$ is different from the list $[2, 1]$ or from the list $[1, 2, 2]$. No typing information on elements of a list are provided, that is elements can be values of any type allowed for `Lvar` objects, including (initialized or uninitialized) logical variables, lists or sets.

A list that contains some elements which are uninitialized logical variables (or lists, or sets) is said to be a *partially specified list*. Note that, even if some elements of the list are not specified, the cardinality of the list is known (i.e., the list is *bounded*). Elements of a list which are themselves lists are said to be *nested lists*. Lists can be *nested* at any depth in JSetL.

Example 3. *Partially specified and nested lists*

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Lst pl = new Lst(pl_elems);                 // the list [1,x]

Lst[] nl_elems = {l,i,e,v};                 // l, i, e, and v are the lists
                                            // defined in Example 2
Lst nl = new Lst(nl_elems);
```

`pl` *is a partially specified list, containing an* `Integer` *object and an uninitialized logical variable* `x`. `nl` *is a list containing four nested lists (both initialized and uninitialized).* □

Lists can be manipulated through a number of `public` methods which are made available by both the `Lst` and the `Lvar` classes. The current version of the JSetL library does not support constraint over lists, apart from list equality and inequality constraints .[2] Lists can

[2]This limitation will be possibly removed in future releases by integrating other basic constraints over lists with set constraints as proposed for instance in [5].

be manipulated only through a number of utility methods. As such, they require that their arguments are sufficiently instantiated to be processed. Utility methods for lists will be completely described in Section 2.6. In the next subsection, however, we already introduce the element insertion and removal methods for lists, since they play an important role in the definition and manipulation of lists.

2.3 List element insertion and removal

List element insertion and removal allow the user to add/remove one or more elements to/from the head or the tail of a given list. These methods do not modify the list on which they are invoked: rather they build and return a new list obtained by adding/removing the elements to/from the input list.

In what follows, t represents a `Lvar`-value type, while `l` is an expression of type `Lst` or *list*–`Lvar` used to denote the invocation object.

Note that, for the sake of simplicity, usually we will not distinguish expressions from the values they denote (i.e., the values obtained by their evaluation). Thus, for instance, if `l` is an expression of type `Lst`, we will say that *the list* `l` has some property p instead of saying that *the list denoted by the expression* `l` has some property p. If necessary, however, we can refer to the value denoted by an expression e by using the notation *val(e)*.

`public Lst ins1(`t` elem)`

> `l.ins1(e)` returns a list obtained by adding `e` as the head element of the list `l`.

`public Lst ins1All(`t`[] elem_array)`

> `l.ins1All(a)` returns a list obtained by adding elements of `a` as the head elements of the list `l`, respecting the order they have in `a`.

`public Lst insn(`t` elem)`

> Same as `ins1` but with element `e` added as the tail element of `l`.

`public Lst insnAll(`t`[] elem_array)`

> Same as `ins1All` but with elements of `a` added as the tail elements of `l`.

List insertion (and extraction) methods can be concatenated (left associative). In fact these methods always return a `Lst` object, and the returned object can be used as the invocation object as well.

Example 4. *List element insertion*

```
Lvar nil = Lst.empty;                          // the empty list
Lst l1 = new Lst(nil.ins1(1));                 // the list [1]

Lvar x = new Lvar();
Lst l2 = new Lst(l1.ins1(2+3).ins1(x));        // the list [x,5,1]
                                               // (x uninitialized var.)

int[] l3_elems = {1,2,3};
Lst l3 = new Lst(nil.ins1All(l3_elems));       // the list [1,2,3]

Lst r = new Lst();                             // an uninitialized list
Lst ul = new Lst(r.ins1(1));                   // the unbounded list [1 | r]
```

□

Using the insertion methods `ins1` and `ins1All` it is possible to build *unbounded* partially specified lists, that is lists with a certain number of (either known or unknown) elements $e_1, ..., e_n$ and a "rest" of the list, represented by an uninitialized list r (i.e., using the abstract notation, $[e_1, \ldots, e_n \mid r]$). List `ul` of Example 4, for instance, is an unbounded partially specified list.

Note that list $[e_1, \ldots, e_n|r]$ is completely different from list $[e_1, \ldots, e_n, r]$ that is a partially specified, though bounded, list. As a matter of fact, the number of elements of the second list is equal to $n+1$, whereas for the first list we can only say that the number of elements is $\geq n$.

The `ins1` and `ins1All` methods can be invoked on any kind of lists. Conversely, the `insn` and `insnAll` methods, which are intended to work on the tail of lists, are thought for initialized and bounded lists only, where the tail elements are known. However, we prefer to allow these methods to be invoked also on unbounded lists, without causing any error or raising any exception. If `l` is an unbounded list, methods `insn` and `insnAll` invoked on `l` are executed as usual but taking into account only the known part of `l`, while the rest is neglected. Thus, for instance, `ul.insn(2)`, where `ul` is the unbounded list defined above, returns the list `[1,2]`.

`public Lst ext1(Lvar elem)`

> `l.ext1(x)`, with `x` an uninitialized `Lvar`, returns a list obtained by extracting the head element from the list `l` and assigning its value to `x`. If `x` is already initialized, a `InitLvarException` exception is raised. If `l` is not initialized a `NotInitVarException` exception is raised. If `l` is the empty list, a `EmptyLstException` exception is raised.

`public Lst extn(Lvar elem)`

> Same as `ext1` but with element `x` extracted from the tail of the list `l`.

`public Lst ext1()` **and** `public Lst extn()`

> Same as `l.ext1(x)` and `l.extn(x)` but the element extracted from `l` is not assigned to any variable.

Example 5. *List element removal*

Let `l2` *and* `l3` *be the lists defined in Example 4.*

```
Lst l4 = new Lst(l2.ext1().ext1());      // the list [1]
Lst l5 = new Lst(l3.ext1(x).insn(x));    // the list [2,3,1]
                                         // (x uninitialized var.)
```

□

Like methods `insn` and `insnAll`, also `extn` is thought for bounded lists. If `l` is an unbounded list, `extn` is executed as usual but taking into account only the known part of `l`, while the rest is neglected. Thus, for instance, `ul.extn(x)`, where `x` is an uninitialized logical variable, returns the uninitialized list `r`.

2.4 Sets: the class Set

A set is a data structure, similar to a list, except that the order of elements and repetitions do not matter. In JSETL a *set* is defined as an instance of the class `Set`. Its value is a finite (possibly empty) collection of arbitrary `Lvar`-values (the *elements* of the set). A set with name SetName can be created in JSETL by the Java statement (a *set declaration*)

`Set SetName = new Set(SetNameExt,SetElemValues);`

where SetNameExt is an optional *external name* of the set (the default external name has the form `"Set_`n`"`, where n is a unique integer), and SetElemValues is an optional part which is used to specify the elements of the set when the set is created (see below).

The *empty set* is denoted by the constant `Set.empty`.

Like lists, a named set can be introduced also as the value of a logical variable and accessed through it. Methods and constraints dealing with sets, however, can be applied indifferently both to sets (i.e., instances of the `Set` class), possibly through the associated set name, and to logical variables initialized to sets.

Sets can be created through the **new** operator, possibly using set declarations, or as the result of executing utility methods dealing with sets (see Sections 2.5 and 2.6). Sets can be either initialized or uninitialized, like logical variables and lists. The value of a set (i.e., a `Set`-*value*) is the collection of its elements. In particular, when a set is created through the **new** operator, the value of the set can be specified in exactly the same ways seen for lists: by passing an array of its elements or the limits of an interval of integer numbers or a `Set` or a *set*–`Lvar` object.

The notation we will use to write sets is similar to the one used for lists, apart using the curly brackets instead of the square brackets. For instance, the set containing the three elements a, b, and c will be represented as $\{a, b, c\}$, and the empty set will be represented as $\{\,\}$. Moreover,

$$\{e_1, e_2, \ldots, e_n \mid S\},$$

where S is a possibly uninitialized set, is used to denote a set containing elements e_1, e_2, $\ldots$, e_n, plus elements in S.

Example 6. `Set` *definitions*

```
Set e = Set.empty;               // the empty set
Set s = new Set("s");            // an uninitialized set, with ext'l name "s"
Set i = new Set("i",4,4+3);      // an initialized set,
                                 // with ext'l name "i" and value {4,5,6,7}
int[] v_elems = {2,4,8,3};
Set v = new Set("v",v_elems);    // an initialized set,
                                 // with ext'l name "v" and value {2,4,8,3}
```

□

Note that in a set the order of elements and the repetitions do not matter. Thus two `Set` objects may have different values but denote the same set. For example, the `Set`-values `{1,2}`, `{2,1}` and `{1,2,2}` all represent the same set.[3]

Elements of a set can be also logical variables (or lists or sets), possibly uninitialized. Sets that contain uninitialized elements are said to be *partially specified sets*. Like lists, sets can also be *nested*, at any depth.

[3]This observation would require to distinguish in general between the *value of a* `Set` *object*—which may contain also duplicated elements—and the set it denotes. However, for the sake of simplicity, usually we will not distinguish between the two notions, using the term *set* to refer to both of them.

Example 7. *Partially specified and nested sets*

```
Lvar x = new Lvar();
Object[] ps_elems = {new Integer(1),x};
Set ps = new Set(ps_elems);              // the partially specified set {1,x}

Set[] ns_elems = {s,i,e,v};              // s, i, e, and v are the sets
                                         // defined in Example 6
Set ns = new Set(ns_elems);              // a set containing four nested sets
```

□

Note that, differently from lists, the cardinality of a partially specified set is not determined precisely, even if it is bounded. For example, the cardinality of the set `ps` of Example 7 can be 1 or 2 depending on whether `x` will be subsequently instantiated to a value equal to 1 or different from 1, respectively.

Sets are manipulated mainly through constraints (see Sect. 3). Besides constraints, element insertion operations, as well as a limited number of utility methods, are provided by JSetL for working with sets. Insertion operations, which are fundamental for the definition of sets, are described in the next subsection; other utilities methods will be described in Section 2.6.

2.5 Set element insertion

JSetL provides two methods for inserting elements in a set: `ins`, for the insertion of a single element, and `insAll` for the insertion of an array of elements. Since the order of elements in a set is not important, it is not necessary to provide distinct methods for head and tail insertion because they would produce the same set.

`ins` and `insAll` can be invoked on any kind of set (namely, completely or partially specified, bounded or unbounded), with no restriction on the presence of uninitialized logical variables in their arguments. Both methods do not modify the set on which they are invoked: rather they build and return a new set obtained by adding the elements to the set.

In what follows, t represents a `Lvar`-value type, while `s` is an expression of type `Set` or *set*–`Lvar` used to denote the invocation object.

`public Set ins(`*t* `elem)`

> `s.ins(e)` returns a set obtained by adding `e` to the set `s` (i.e., $\mathtt{s} \cup \{\mathtt{e}\}$).

`public Set insAll(`*t*`[] elem_array)`

> `s.ins1All(a)`, with `a` array of elements of type t, returns a set obtained by adding all elements of `a` to the set `s`.

The `ins` and `insAll` methods can be concatenated (left associative). In fact these methods always return a `Set` object, and the returned object can be used as the invocation object as well.

Example 8. *Set element insertion*

```
Lvar nil = new Lvar(Set.empty);           // the empty set
Set s1 = new Set(nil.ins(1));             // the set {1}

Lvar x = new Lvar();
Set s2 = new Set(s1.ins(2+3).ins(x));     // the set {5,1,x}
                                          // (x uninitialized var.)

int[] s3_elems = {1,2,3};
Set s3 = new Set(nil.insAll(s3_elems));   // the set {1,2,3}

Set r = new Set();                        // an uninitialized set
Set us = new Set(r.ins(1));               // the unbounded set {1 | r}
```

□

Note that using the insertion methods `ins` and `insAll` it is possible to build *unbounded* partially specified sets, that is sets with a certain number of (either known or unknown) elements $e_1, ..., e_n$, and a "rest" of the set, represented by an uninitialized set r (i.e., using the abstract notation, $\{e_1, \ldots, e_n \mid r\}$). Set `us` of Example 8, for instance, is an unbounded partially specified set.

The `Set` class does not provide any extraction methods like those seen for lists. Set element extraction, instead, is carried on by using constraints, in particular, the `less` and `differ` constraints (see Section 3). As a matter of fact, set element extraction can involve nondeterministic choices. For example, the extraction of the uninitialized `Lvar` x from the (known) set $\{1, 2, 3\}$ does not yield a unique solution, being 1, 2 or 3 all admissible values for x. Similarly, even the extraction of a known element from a set, can involve a nondeterministic choice. For example, the extraction of 1 from the set $\{1, 2, x\}$, with x an uninitialized logical variable, can return either the set $\{2\}$, if the value of x is 1 or 2, or the set $\{2, x\}$ if on the contrary the value of x is different from 1. The extraction of one or more elements from a set is deterministic only if the element is known, the set is bounded and all its elements are known; in the other cases it involves some nondeterministic choice. Therefore, instead of providing extraction methods that impose strong restrictions on the form of their arguments (to make the methods deterministic), we prefer to leave the duty of implementing set extraction to the constraint manager, which can handle correctly also the nondeterministic cases.

2.6 Utility methods

JSetL provides a number of utility methods which can be applied to logical variables, lists and sets. Some of them are *general* and can be applied to any logical variable, list or set, either instantiated or not. Others are specific either for lists or for sets.

2.6.1 General utility methods

In the description of the following methods, the invocation object `o` is an expression of type `Lvar` or `Lst` or `Set`.

`public boolean known()`

> `o.known()` returns true if `o` is initialized and false otherwise.

`public boolean unknown()`

> `o.unknown()` returns true if `o` is uninitialized and false otherwise.

```
public boolean isGround()
```

o.isGround() returns true if o is ground, that is if it does not contain any uninitialized variable, and false otherwise.

```
public String name()
```

o.name() returns the external name associated with o.

```
public String giveName(String name)
```

o.giveName(s) assigns the string s to the object o as its external name and returns s.

```
public Object value()
```

o.value() returns the value of o if o is initialized, null otherwise.

```
public void print()
```

o.print() prints the value of o to the standard output stream. If o is uninitialized, the output is the external name of o (either defined by the user or the default one), preceded by the character '_'.[4] Lists and sets are printed following the syntactic conventions of Prolog. For example, if o is the list [x,y,2|k], with x an initialized logical variable with value "alpha", y an uninitialized logical variable with external name "y", and k an uninitialized list with no external name specified for it, o.print() produces the following output:

```
[alpha,_y,2|_Lst_1]
```

where "Lst_1" is the default external name automatically assigned to k.

```
public void output()
```

o.output() prints the external name of the object o followed by its value, that is,

ObjName = ObjValue

or followed by the special constant unknown if the object is uninitialized. For example, given the logical variable x, the statement

```
x.output();
```

will produce the output

```
x = 2
```

if x has external name "x" and value 2; the output

```
X = unknown
```

if x has external name "x" and it is uninitialized; the output

```
Lvar_1 = 3
```

if x has no external name and its value is 3.

[4]The character '_' is used to distinguish variable names from other strings and characters printed on the output stream.

`public Object read()`

> `o.read()` reads from the standard input stream an `Lvar`-value, assigns it to the object `o` and returns the read value as its result. `o` must be an uninitialized logical variable (or list, or set); otherwise, a `InitLvarException` (or `InitLstException`, or `InitSetException`) exception is raised.
>
> `Lvar`-values which are read through the `read` method must be written according to the following syntactic rules: numbers have the usual form of numerical literals in Java; (single) characters must be preceded (and possibly followed) by a single quote, while strings are preceded (and possibly followed) by double quotes; lists and sets are sequences of readable `Lvar`-values, separated by commas, and delimited between square and curly brackets, respectively. Sets and lists can be nested at any level, but they must be bounded and completely specified, that is uninitialized variables cannot occur as their elements. If the input value does not satisfy these rules an `IOException()` is raised.
>
> As an example, if `x` is an uninitialized logical variable, the statement `x.read()` initializes `x` as follows:
>
> | *input* | *value assigned to* `x` |
> |---|---|
> | `{1,'e',["aaa",2]}` | the set `{1,e,[aaa,2]}` |
> | `'alpha` | the character `a` |
> | `"alpha"` | the string `alpha` |
>
> If `x` would be declared as a `Set` variable, only the first input value in the above example would be acceptable (whereas the others will cause an exception `NotSetException` to be raised).
>
> If some other symbols appear after the right-hand delimiter of the input value (e.g., after the closed curly bracket), these symbols, up to the next carriage-return or new-line symbol, are simply ignored. For example, the input value `{1,'e',["aaa",2]}xyw,zjk` is considered equivalent to `{1,'e',["aaa",2]}`, while `'a'aabbaabb` or `'aaabbaabb` are considered equivalent to `'a'`.

`public Character readChar()`
`public String readString()`

> `o.readChar()` and `o.readString()` read from the standard input stream a character value and a string value, respectively, assign the read values to the object `o` and return the read values as their result. `o` must be an uninitialized logical variable, otherwise, a `InitLvarException` exception is raised.
>
> Differently from the `read` method, `o.readChar()` and `o.readString()` do not require that the character and string values to be read are written between quotes. Note that, using the `readChar` method, if more than one character is typed in, those following the first one are ignored.

2.6.2 Utility methods for lists

In the description of the following methods, t represents a `Lvar`-value type, while the invocation object `l` is an expression of type `Lst` or *list*–`Lvar`. When the method is invoked, `l` must be initialized, otherwise a `NotInitVarException` exception is generated.

`public boolean inl(`*t* `elem)`

`l.inl(e)` returns true if `e` is an element of the list `l`, otherwise it returns false. `e` must be ground, that is it must not contain any uninitialized variable when the method is executed; otherwise, a `NotInitVarException` exception is raised.

`public boolean ninl(`*t* `elem)`

`l.ninl(e)` returns true if `e` is not an element of the list `l`, and false otherwise. `l` and `e` must be ground; otherwise, a `NotInitVarException` exception is raised.

`public Object get(int index)`

`l.get(i)` returns the `i`-th element of the list `l`. If the value of `i` is smaller than 0 or greater than the length of the list `l`, an `ArrayIndexOutOfBoundsException` exception is raised.

`public Lst concat(Lst list)`

If `l` is the list $[a_1, ..., a_n]$ and `m` is the list $[b_1, ..., b_m]$, `l.concat(m)` returns the list obtained by concatenating `l` and `m`, i.e., $[a_1, ..., a_n, b_1, ..., b_m]$. Lists `l` and `m` remain unchanged.

This method is thought for initialized and bounded lists only. However, we prefer to allow the method to be invoked also on unbounded lists, without causing any error or raising any exception. If `l` is an unbounded list, `l.concat(m)` returns the list obtained by concatenating `l` and `m` but taking into account only the known parts of the two lists, while the unknown rests are neglected. Thus, for instance, `l1.concat(l2)`, where `l1` is the unbounded list `[1,2,3|x]` and `l2` is the list `[6,7]`, returns the bounded list `[1,2,3,6,7]`.

`public boolean isEmpty()`

`l.isEmpty()` returns true if `l` is the empty list, false otherwise.

`public int size()`

`l.size()` returns the length of the list `l`, that is the number of elements of the list. For example, if `l` is the list `[1,2,x]`, with `x` an uninitialized logical variable, `l.size()` returns `3`. In particular, if `l` is the empty list, `l.size()` returns `0`.

`public Set toSet()`

`l.toSet()` returns a `Set` object whose elements are those of the list `l`.

`public Set toSet(String name)`

`l.toSet(s)` returns a `Set` object whose elements are those of the list `l` and whose external name is the string `s`.

`public static Lst mkLst(int n)`

`Lst.mkLst(n)` returns a list composed of `n` uninitialized `Lvar` objects.

2.6.3 Utility methods for sets

In the description of the following methods, t represents a `Lvar`-value type, while the invocation object `s` is an expression of type `Set` or *set*–`Lvar`. When the method is invoked, `s` must be initialized, otherwise a `NotInitVarException` exception is generated.

`public Set concat(Set T)`

If `s` is a set with value $\{a_1, ..., a_n\}$ and `t` is a set with value $\{b_1, ..., b_m\}$, `s.concat(t)` returns a set with value $\{a_1, ..., a_n, b_1, ..., b_m\}$. Sets `s` and `t` remain unchanged.

Note that $a_1, ..., a_n, b_1, ..., b_m$ are not necessarily all distinct elements. For example, if the value of `s` is {`1,2`}, and the value of `t` is {`2,1`}, `s.concat(t)` returns the `Set`-value {`1,2,2,1`} which actually denotes the set containing two elements, `1` and `2`. As another example, if the value of `s` is {`1,x`}, and the value of `t` is {`2,y`}, with `x` and `y` uninitialized variables, `s.concat(t)` returns the `Set`-value {`1,x,2,y`} which actually denotes many sets, depending on the values that `x` and `y` will possibly assume: equal to `1` or `2`, different from `1` and `2` but with the same value for both `x` and `y`, and so on.

The `concat` method for sets is thought for initialized and bounded sets only. However, like in the case of lists, we prefer to allow the `concat` method to be invoked also on unbounded sets, taking into account only the known parts of the two sets. Thus, for instance, `s1.concat(s2)`, where `s1` is the unbounded set {`1,2,3|x`} and `s2` is the set {`6,7`}, returns the bounded set {`1,2,3,6,7`}.

`public boolean isEmpty()`

`s.isEmpty()` returns true if `s` is the empty set. false otherwise.

`public static Set mkSet(int n)`

`Set.mkSet(n)` returns a set composed of `n` uninitialized `Lvar` objects.

Other operations on sets, such as equality, membership and not membership, union (which generalizes the `concat` method shown above), and so on, which are inherently nondeterministic, are provided as constraints, and will be presented in the next section.

3 Constraints

Basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraint expressions are evaluated even if they contain uninitialized variables. Their evaluation is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the *constraint store*) using a powerful (set) constraint solver, which allows us to compute with partially specified data. Basically, constraints are added to the constraint store using the `add` method and their resolution is performed by calling the `solve` method, which implements the constraint solver. The solver nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. If there is no solution, a `Failure` exception is generated; otherwise the computation terminates with *success* when the first solution is found. Constraints that are not solved are left in the constraint store: they will be used later to narrow the set of possible values that can be assigned to uninitialized variables.

In this section, we give the definition of the constraints that JSetL is able to deal with and we introduce the methods that are used to add constraints to the constraint store. Constraint solving and the relevant methods (in particular, the `solve` method), will be introduced in the next section.

3.1 Constraint definition

In JSetL an *atomic constraint* is an expression of one of the forms:

- $e_1.op\,(e_2)$
- $e_1.op\,(e_2, e_3)$

where *op* is one of a collection of predefined methods provided by classes `Lvar`, `Lst` and `Set`, and e_1, e_2 and e_3 are expressions whose type depends on *op*.

More precisely, the types of e_1, e_2 and e_3 are defined as follows. Let `LvarExpr` be the type of an object representing an arithmetic expression possibly involving *int*–`Lvar` objects and built using the arithmetic methods `sum`, `sub`, `mul`, `div`, and `mod` (see Sect. 3.4).[5] Let $\mathtt{l}_1$ be an expression of type `Lvar` or `Lst` or `Set` or `LvarExpr`, $\mathtt{l}_2$ be an expression of type `Lvar`-value (see Section 2.1) or `LvarExpr`, $\mathtt{s}$, $\mathtt{s}_1$, $\mathtt{s}_2$, $\mathtt{s}_3$ be expressions of type `Set` or *set*–`Lvar`, $\mathtt{i}_1$ be an expression of type *int*–`Lvar` or `LvarExpr`, and $\mathtt{i}_2$ be an expression of type `int` or `Integer` or *int*–`Lvar` or `LvarExpr`.

JSetL provides the following atomic constraints (here we use the *val* operator to make explicit the distinction between expressions and values):

- **Equality** and **inequality**: $\mathtt{l}_1.\mathsf{eq}\,(\mathtt{l}_2)$, $\mathtt{l}_1.\mathsf{neq}\,(\mathtt{l}_2)$,
  whose meaning is $val(\mathtt{l}_1) = val(\mathtt{l}_2)$, $val(\mathtt{l}_1) \neq val(\mathtt{l}_2)$
- **Membership** and **not membership**: $\mathtt{l}_1.\mathsf{in}\,(\mathtt{s})$, $\mathtt{l}_1.\mathsf{nin}\,(\mathtt{s})$,
  whose meaning is $val(\mathtt{l}_1) \in val(\mathtt{s})$, $val(\mathtt{l}_1) \notin val(\mathtt{s})$
- **Inclusion** and **not inclusion**: $\mathtt{s}_1.\mathsf{subset}\,(\mathtt{s}_2)$, $\mathtt{s}_1.\mathsf{nsubset}\,(\mathtt{s}_2)$,
  whose meaning is $val(\mathtt{s}_1) \subseteq val(\mathtt{s}_2)$, $val(\mathtt{s}_1) \not\subseteq val(\mathtt{s}_2)$
- **Union** and **not union**: $\mathtt{s}_1.\mathsf{union}\,(\mathtt{s}_2, \mathtt{s}_3)$, $\mathtt{s}_1.\mathsf{nunion}\,(\mathtt{s}_2, \mathtt{s}_3)$,
  whose meaning is $val(\mathtt{s}_1) = val(\mathtt{s}_2) \cup val(\mathtt{s}_3)$, $val(\mathtt{s}_1) \neq val(\mathtt{s}_2) \cup val(\mathtt{s}_3)$
- **Intersection** and **not intersection**: $\mathtt{s}_1.\mathsf{inters}\,(\mathtt{s}_2, \mathtt{s}_3)$, $\mathtt{s}_1.\mathsf{ninters}\,(\mathtt{s}_2, \mathtt{s}_3)$,
  whose meaning is $val(\mathtt{s}_1) = val(\mathtt{s}_2) \cap val(\mathtt{s}_3)$, $val(\mathtt{s}_1) \neq val(\mathtt{s}_2) \cap val(\mathtt{s}_3)$
- **Disjunction** and **not disjunction**: $\mathtt{s}_1.\mathsf{disj}\,(\mathtt{s}_2)$, $\mathtt{s}_1.\mathsf{ndisj}\,(\mathtt{s}_2)$,
  whose meaning is $val(\mathtt{s}_1) \cap val(\mathtt{s}_2) \neq \emptyset$, $val(\mathtt{s}_1) \cap val(\mathtt{s}_2) = \emptyset$
- **Difference** and **not difference**: $\mathtt{s}_1.\mathsf{differ}\,(\mathtt{s}_2, \mathtt{s}_3)$, $\mathtt{s}_1.\mathsf{ndiffer}\,(\mathtt{s}_2, \mathtt{s}_3)$,
  whose meaning is $val(\mathtt{s}_1) = val(\mathtt{s}_2) \backslash val(\mathtt{s}_3)$, $val(\mathtt{s}_1) \neq val(\mathtt{s}_2) \backslash val(\mathtt{s}_3)$
- **Less**: $\mathtt{s}_1.\mathsf{less}\,(\mathtt{l}_1, \mathtt{s}_2)$,
  whose meaning is $val(\mathtt{l}_1) \in val(\mathtt{s}_1) \wedge val(\mathtt{s}_2) = val(\mathtt{s}_1) \backslash \{val(\mathtt{l}_1)\}$
- **Comparison** constraints: $\mathtt{i}_1.\mathsf{le}\,(\mathtt{i}_2)$, $\mathtt{i}_1.\mathsf{lt}\,(\mathtt{i}_2)$, $\mathtt{i}_1.\mathsf{ge}\,(\mathtt{i}_2)$, $\mathtt{i}_1.\mathsf{gt}\,(\mathtt{i}_2)$,
  whose meaning is $val(\mathtt{i}_1) \leq val(\mathtt{i}_2)$, $val(\mathtt{i}_1) < val(\mathtt{i}_2)$, $val(\mathtt{i}_1) \geq val(\mathtt{i}_2)$, $val(\mathtt{i}_1) > val(\mathtt{i}_2)$.

[5] Similarly to lists and sets, we use the term *int*–`Lvar` to refer to the type of a logical variable whose value, if any, must be of type `int` or `Integer`. As usual this check is performed at run-time, and it possibly causes a `NotIntLvarException` exception to be generated.

A *constraint* is either an atomic constraint or (recursively) the conjunction of two or more atomic constraints c_1, c_2,…, c_n:

c_1.and (c_2) … .and (c_n)

Example 9. JSETL *constraints*

Let x, y, *and* z *be logical variables and* r, s, *and* t *be sets.*

```
r.eq(s);                               // equality between sets
t.union(r,s);                          // t = r ∪ s
x.eq(y).and(x.eq(3)).and(y.neq(z));    // x = y ∧ x = 3 ∧ y ≠ z
```

□

Constraints are implemented as public methods of the `Lvar`, `Lst` and `Set` classes. All atomic constraint methods return a result of type `StoreElem`, that is a data structure that contains all information concerning the relevant constraint. The `and` method returns a `Vector` of `StoreElem`. `StoreElem` objects constitute the elements of the constraint store. In what follows, however, we will not distinguish between the expression that represents a constraint and the `StoreElem` object obtained by the evaluation of this expression, using the term (atomic) *constraint* for both of them.

The addition of a constraint C to the constraint store is mainly performed by calling the `add` method of the `Solver` class:

`Solver.add(C)`

This method, along with a few other methods for adding constraints to the constraint store, will be examined in detail in Section 4. After constraints have been added to the constraint store, one can invoke their resolution by calling the `solve` method of the `Solver` class:

`Solver.solve()`

The `solve` method searches for a solution that satisfies all the constraints introduced in the constraint store. If a solution is found then the invocation of the `solve` method terminates with success; otherwise, if there is no solution, a `Failure` exception is generated. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred. Constraint solving and the related methods will be described in detail in Section 5.

3.2 Equality and inequality constraints

Let us see some simple examples involving equality and inequality constraints.

Example 10. *Equality and inequality constraints.*

Let x, y, *and* z *be uninitialized logical variables.*

```
Solver.add(x.eq(3));     // equality constraint x = 3
Solver.add(y.neq(x));    // inequality constraint y ≠ x
Solver.add(x.eq(z));     // equality constraint x = z
```

The constraint added to the constraint store is (using an intuitive abstract notation): x = 3 ∧ y ≠ x ∧ x = z*. The same effect can be obtained by executing:*

```
Solver.add(x.eq(3).and(y.neq(x)).and(x.eq(z)));
```

The constraint is satisfiable. After the execution of the `solve` *method, the logical variables* x *and* z *have both value* 3*, while* y *remains uninitialized.* □

Example 11. *Unsatisfiable equality and inequality.*

Let x *be an uninitialized logical variable and* y *a logical variable initialized to* 1.

```
Solver.add(x.neq(1));   // inequality constraint x ≠ 1
Solver.add(y.eq(x));    // equality constraint 1 = x
```

The constraint added to the constraint store is : x $= 1 \wedge 1 \neq$ x*. Clearly, the constraint is unsatisfiable, and a* Failure *exception will be raised by the constraint solver.* □

The logical variables involved in equality and inequality constraints can also be instances of classes Lst and Set or they can be Lvar initialized to lists or sets.

Example 12. *Equalities and inequalities involving sets.*

Let x, y*, and* z *be uninitialized logical variables and* s *be a* Set *variable initialized to the set* {1,2,3}.

```
Solver.add(x.eq(3));    // equality constraint x = 3
Solver.add(y.neq(x));   // inequality constraint y ≠ x
Solver.add(z.eq(y));    // equality constraint z ≠ y
Solver.add(z.eq(s));    // equality constraint z = s
```

The constraint added to the constraint store is: x $= 3 \wedge$ y $\neq$ x $\wedge$ z $\neq$ y $\wedge$ z $= \{1, 2, 3\}$*. The constraint is satisfiable. After the execution of the* solve *method, the logical variable* x *is initialized to* 3*, while* y *and* z *have value equal to the set* {1,2,3}. □

Solving equality constraints, in particular equalities between lists and sets, implies in general the ability to solve *unification problems* (in particular, *set unification*; cf., e.g., [6]). Unification of two possibly partially specified lists/sets means finding an assignment of values to uninitialized variables occurring in them (if any), that makes them equal in the underlying list/set theory. For lists, two lists are equal simply if each element of one list is equal to the element that appears in the same position in the other list (i.e., the lists are identical). For sets, in any reasonable set theory, two sets are equal if their elements are equal disregarding their order and possible repetitions. Thus, for instance, the equality between the lists $[x, 2]$ and $[1, y]$, x and y uninitialized logical variables, is satisfied by $x = 1$ and $y = 2$, while the equality between the lists $[x, 2]$ and $[y, 1]$ or between $[x, 2]$ and $[1, 1, y]$ are not satisfiable, since there are no values for x and y that make the two lists identical. Conversely, the set equalities $\{x, 2\} = \{y, 1\}$ and $\{x, 2\} = \{1, 1, y\}$ are satisfied by $x = 1$ and $y = 2$.

It is important to notice that a set unification problem can admit more than one solution. For example, the constraint $\{x, y\} = \{z, 2\}$ is satisfied by x, y and z equal to 2 or by $x = z$ and $y = 2$ or by $y = z$ and $x = 2$. The constraint solver is able to compute, one after the other, all solutions for a given unification problem.

The most interesting (and complex) cases for list and set unification, however, are those dealing with unbounded lists and sets. Let us see some examples involving such kinds of lists and sets.

Example 13. *Unification between unbounded lists.*

Let lr, mr *be uninitialized lists, and* x, z *be uninitialized logical variables.*

```
Lst l = new Lst(lr.ins1(x).insn(2));                    // l is [x,2 | lr]
Lst m = new Lst(mr.ins1(1).insn(z).insn(3).insn(4));    // m is [1,z,3,4 | mr]
Solver.add(l.eq(m));                                    // equality constraint
```

The constraint added to the constraint store is: $[\mathtt{x}, 2 \mid \mathtt{lr}] = [1, \mathtt{z}, 3, 4 \mid \mathtt{mr}]$. *This constraint is then rewritten by the constraint solver to the new constraint (in solved form)* $\mathtt{x} = 1 \wedge \mathtt{z} = 2 \wedge \mathtt{lr} = [3, 4 \mid \mathtt{mr}]$. *The constraint is trivially satisfiable.* □

Example 14. *Unification between unbounded sets.*

Let `sr` *and* `tr` *be uninitialized sets, and* `x`, `y` *and* `z` *be uninitialized logical variables.*

```
Set s = new Set(sr.ins(y).ins(1));            // s is {y,1 | sr}
Set t = new Set(tr.ins(z).ins(2).ins(x));     // t is {z,2,x | tr}
Solver.add(s.eq(t));                          // equality constraint
```

The constraint added to the constraint store is: $\{\mathtt{y}, 1 \mid \mathtt{sr}\} = \{\mathtt{z}, 2, \mathtt{x} \mid \mathtt{tr}\}$. *A possible solution for it nondeterministically computed by the solver is:* $\mathtt{x} = 1 \wedge \mathtt{y} = \mathtt{z} \wedge \mathtt{sr} = \{2 \mid \mathtt{tr}\}$, *with* `y` *and* `z` *uninitialized variables.* □

Note that also inequality constraints between partially specified lists and sets can have more than one solution. For example, the constraint $[x, y] \neq [1, 2]$ is satisfied if at least one of the constraints $x \neq 1$ or $y \neq 2$ is satisfied (that is, logically, if the formula $x \neq 1 \vee y \neq 2$ is true). In practice, the constraint solver computes each one of the possible solutions one at a time, nondeterministically. As another example, the similar constraint $\{x, y\} \neq \{1, 2\}$ admits four distinct solutions: $x \neq 1 \wedge x \neq 2$, $y \neq 1 \wedge y \neq 2$, $x \neq 1 \wedge y \neq 1$, $x \neq 2 \wedge y \neq 2$. All of them are nondeterministically computed by the constraint solver.

3.3 Set constraints

Let us analyze some particularly interesting cases involving atomic constraints based on set-theoretical operations (i.e., *set constraints*).

In all these constraints, expressions `s`, $\mathtt{s}_1$, $\mathtt{s}_2$, and $\mathtt{s}_3$ can be *uninitialized* `Lvar` or `Set` objects. The constraint solver is always able to solve all these cases, leaving the solved form constraint in the constraint store or generating a `Failure` exception.

As an example, let us consider a membership constraint $\mathtt{x} \in \mathtt{s}$, where `s` is an uninitialized set, to be added to the constraint store:

```
Solver.add(x.in(s));
```

The constraint solver rewrites this constraint to the equality constraint

$$\mathtt{s} = \{\mathtt{x} \mid \mathtt{N}\}$$

where `N` is a newly generated uninitialized `Set` object, which states, in terms of equality, that `s` must contain `x` plus "something else", denoted by `N`.

Notice that the `in` and `nin` constraints are quite different from the `inl` and `ninl` methods for lists. The latter are simple tests that allow to establish whether a known element belongs or not to a known list. Conversely, the `in` and `nin` set constraints can be invoked also on uninitialized sets with unknown elements. Thus, for instance, the `in` constraint can be used to obtain nondeterministically all elements of a set, or to build a set that contains a given element (as in the above example).

Sets involved in set constraints can also be partially specified, either bounded or unbounded. Let us consider the following simple example.

Example 15. *Constraints using unbounded sets.*

Let `sr` *and* `tr` *be uninitialized sets, and* `x` *be an uninitialized logical variable.*

```
Set s = new Set(sr.ins(1));   // s is {1 | sr}
Set t = new Set(tr.ins(x));   // t is {x | tr}
Solver.add(s.disj(t));        // disjointness constraint
```

$\mathtt{s}$ and $\mathtt{t}$ are unbounded sets. The constraint added to the constraint store admits one solution, specifically: $\mathtt{x} \neq 1 \wedge 1 \notin \mathtt{tr} \wedge \mathtt{x} \notin \mathtt{sr} \wedge \mathtt{tr} \cap \mathtt{sr} = \emptyset$. $\mathtt{sr}$ and $\mathtt{tr}$ remain uninitialized but a new constraint stating that $\mathtt{tr}$ and $\mathtt{sr}$ must be disjoint sets is added to the constraint store. □

All set constraints are *invertible*, that is all parameters, as well as the invocation object, of the constraints can be used either for input or for output. In the following example, we consider the constraint $\mathtt{t} = \mathtt{r} \cup \mathtt{s}$ with `s` an uninitialized set, and `r` and `t` two sets with a known value. Thus, `r` and `t` serve as input parameter and `s` as the output parameter.

Example 16. *Invertibility.*

```
Set r = new Set(Set.empty.ins(1).ins(2));  // s is {1,2}
Set t = new Set(r.ins(3).ins(4));;         // t is {1,2,3,4}
Set s = new Set();                         // an uninitialized set
Solver.add(t.union(r,s));                  // union constraint
```

The constraint added to the constraint store is $\{1,2,3,4\} = \{1,2\} \cup \mathtt{s}$. One possible solution for this constraint (actually, the first one returned by the JSetL *constraint solver) is $\mathtt{s} = \{3,4\}$.*

Note that this is not the only possible solution. In fact if we add also the constraint $\mathtt{s} \neq \{3,4\}$, the previous solution does not fulfill the new constraint, and the solver looks for another solution that satisfies the whole constraint. One such solution is $\mathtt{s} = \{2,3,4\}$. □

3.4 Comparison constraints

Besides equality, inequality and set constraints, JSetL provides a limited support for comparison constraints over integer numbers. Comparison constraints have the form $\mathtt{i_1.op(i_2)}$, where $\mathtt{i_1}$ is an expression of type *int*–`Lvar`, $\mathtt{i_2}$ is an expression of type `int` or `Integer` or *int*–`Lvar`, and `op` is one of the following comparison operators: `le`, `lt`, `ge`, `gt`.

Comparison constraints in JSetL can be used only as simple tests of known values, not as real constraints over integers. As a matter of fact, comparison constraints are evaluated only if all their arguments are known values, that is no uninitialized logical variable occurs in them. Otherwise, the constraint is simply left unchanged in the constraint store: its solution is *delayed* until both its arguments possibly become completely known. If some comparison constraints are still unresolved at the end of the invocation of the constraint solver, then either a `NotIntLvarException` exception is generated or the invocation terminates normally, depending on which method of the `Solver` class has been used to invoke the solver (see Sect. 5). Anyway, the comparison constraints left in the constraint store are not further elaborated. For example, if we add the constraint $\mathtt{x} < \mathtt{y} \wedge \mathtt{x} > \mathtt{y}$ (e.g., by executing the statement `Solver.add(x.le(y).and(x.gt(y)))`), and `x` and `y` are *uninitialized* logical variables, the JSetL solver is not able to detect the inconsistency and the added constraints are simply left unchanged in the constraint store.[6]

[6]Extensions of the constraint solving capabilities of the JSetL constraint solver are planned for future releases of the library. In particular, following the approach in [4], we plan to integrate the current set constraint solver with a solver over *finite domains*[2], which would allow us to deal with basic operations over integers as constraints with almost no restriction on the instantiation of expressions that can occur in them.

Like all other constraints, however, comparison constraints are "sensible" to backtracking, that is, values of logical variables possibly occurring in them can be modified due to backtracking and the constraint re-evaluated accordingly. Let us consider the following example.

Example 17. *Comparison constraints.*

Let x, y, *and* z *be logical variables and* s *be the set* {2,5,8}.

```
Solver.add(x.in(s).and(y.in(s)).and(z.in(s)));   // membership constraints
Solver.add(x.gt(y).and(y.lt(z)));                // comparison constraints
```

According to the first constraint, x, y, *and* z *must belong to the set* {2,5,8}. *The first solution computed by the constraint solver is likely to be* $\mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 2$. *However, when also the second constraint, i.e.,* $\mathtt{x} > \mathtt{y} \wedge \mathtt{y} < \mathtt{z}$, *is required to be satisfied, then a failure is detected. Then, new values for* x, y, *and* z *are computed and the comparison constraints evaluated again, using the new values for the logical variables. Finally, the constraint solver determines a possible solution for the given constraints:* $\mathtt{x} = 5 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 5$. □

Expressions $\mathtt{i}_1$ and $\mathtt{i}_2$ in comparison constraints can be also compound arithmetic expressions, e.g. $(x + y) * 2$. Variables possibly occurring in these expressions must be logical variables to allow the constraint solver to handle them properly (in particular during backtracking).[7]

JavaSet provides five methods which can be used to write complex arithmetic expressions involving logical variables:

```
public LvarExpr sum(t e)
public LvarExpr sub(t e)
public LvarExpr mul(t e)
public LvarExpr div(t e)
public LvarExpr mod(t e)
```

where e is an expression of type int or Integer or *int*–Lvar or LvarExpr. All these methods return a LvarExpr object as their result and they can be concatenated to represent compound expressions. LvarExpr objects can be used in comparison constraints and in equality and inequality constraints, both as the invocation object and as the argument of the invocation. LvarExpr objects can be used also in membership and not membership constraints as the invocation object.

Example 18. *Constraints using* LvarExpr *objects.*

Let us consider the following system:

$$\begin{cases} \mathtt{L1} + \mathtt{L2} - \mathtt{L3} = \mathtt{L4} - \mathtt{1} \\ \mathtt{L1} + \mathtt{m} - \mathtt{L2} * \mathtt{L3} \neq \mathtt{n} + \mathtt{m} \\ \mathtt{L1}/\mathtt{m} + \mathtt{L2} * \mathtt{n} \geq \mathtt{L3} \end{cases}$$

where L1, L2, L3, L4 *are the unknowns to be computed. This system can be written using constraints as follows:*

```
Solver.add(L1.sum(L2).sub(L3).eq(L4.sub(1)));
Solver.add(L1.sum(m).sub(L2.mul(L3)).neq(n + m));
Solver.add(L1.div(m).sum(L2.mul(n)).ge(L3));
```

[7]Conventional programming variables can be used in comparison constraints as well, but their values will be fixed to the values they have when the constraint is added to the constraint store, with no possibility to be modified through backtracking.

where `L1`, `L2`, `L3`, `L4` *are logical variables, and* `n`, `m` *are integer variables.* □

Finally, an implementation remark. Comparison constraints are implemented as methods of the `Lvar` class. Actually, if a more complete treatment of integer constraints would be provided, it could be convenient to provide also a new class, which extends the `Integer` class, and which encapsulates all methods used to implement comparison constraints and `LvarExpr` objects (similarly to the `Lst` and `Set` classes).

4 The Constraint Store

The constraint store (hereafter, abbreviated CS) contains the collection of all active constraints in a running program. JSetL provides methods through which the program can add new constraints to the CS, visualize the content of the CS, and remove (all) constraints from the CS. General constraints are added to the CS by using either the `add` method or the `forall` method. Moreover, the `allDifferent` method is used to deal with a special case of inequality constraints. Constraint visualization and removal is performed by the `showStore` and `clearStore` methods, respectively. All these methods are implemented as static methods of the `Solver` class.

4.1 The `add` methods

`public static void add(StoreElem AtomicConstr)`

> `Solver.add(`c`)`, where c is an atomic constraint, adds c to the CS.

`public static void add(Vector Constr)`

> `Solver.add(`c_1`.and(`c_2`)....and(`c_n`))`, adds the n atomic constraints c_1, c_2,...,c_n to the CS (note that the `and` method, which is used to define a conjunction of atomic constraints, returns a vector of `StoreElem` objects).

The statement `Solver.add(`c_1`.and(`c_2`)....and(`c_n`))` is equivalent to the sequence of statements:

`Solver.add(`c_1`);`
`Solver.add(`c_2`);`
...
`Solver.add(`c_n`);`

The order in which atomic constraints are added to the constraint store is completely immaterial.

4.2 The `allDifferent` method

Let *AggrType* be either a `Lst`, a `Set`, a *list*–`Lvar` or a *set*–`Lvar` type.

`public static void allDifferent(`*AggrType* `A)`

> `Solver.allDifferent(a)`, where `a` is either a (bounded) list or set containing n elements e_1,...,e_n, adds the atomic constraints $e_i \neq e_j$ to the CS, for all i, j in $1, \ldots, n$. If `a` is not initialized or it is an unbounded list or set, no exception is raised and, in the second case, only the known part is used to add inequality constraints.

As an example, if `r` is the set {`x`,`y`,`z`}, with `x`, `y`, `z` uninitialized logical variables, the invocation `Solver.allDifferent(r)` adds to the CS the new constraint $\mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \neq \mathtt{z} \wedge \mathtt{y} \neq \mathtt{z}$. A more complex example is presented in Section 7.1.

4.3 The `forall` methods

The `forall` methods allow to state that a given constraint C must hold for each element of a given set s.

Let *StoreElemType* be either a `StoreElem` or a vector of `StoreElem` type.

`public static void forall (Lvar x, Set s,` *StoreElemType* `c) throws Failure`

> `Solver.forall(x,s,`C`)`, where `x` is an uninitialized logical variable and C is a constraint containing `x`, adds to the CS an instance of C for each element e of `s` with the value of `x` in the instance of C initialized to e. Logically, this is the so-called Restricted Universal Quantifier (cf., e.g., [6]), $(\forall x \in s)C$, whose logical meaning is $\forall x((x \in s) \rightarrow C)$.

C can be a constraint of any type, either atomic or a conjunction of atomic constraints. The only restriction on C is that C must contain at least one occurrence of the variable `x`[8].

Example 19. *Simple use of* `forall`.

```
Lvar x = new Lvar();          // an uninitialized l. var.
int[] s_elems = {2,8,5};
Set s = new Set(s_elems);     // the set {2,8,5}
Solver.forall(x,s,x.ge(0));   // (∀x ∈ {2,8,5})x ≥ 0
```

The constraint added to the CS is $2 \geq 0 \wedge 8 \geq 0 \wedge 5 \geq 0$ *which is clearly satisfiable. If the invocation of the* `forall` *method is replaced by*

```
Solver.forall(x,s,(x.ge(0)).and(x.le(5)));   // (∀x ∈ {2,8,5})x ≥ 0 ∧ x ≤ 5
```

the constraint added to the CS is $2 \geq 0 \wedge 2 \leq 5 \wedge 8 \geq 0 \wedge 8 \leq 5 \wedge 5 \geq 0 \wedge 5 \leq 5$ *which is clearly not satisfiable.* □

The set `s` in an invocation `Solver.forall(x,s,`C`)` can be also uninitialized or, if initialized, it can be either bound or unbound, and in both cases it can contain uninitialized logical variables as its elements. In all these cases, the subsequent call of the constraint solver will nondeterministically compute all possible values for the uninitialized variables occurring in `s` such that the constraint C is satisfied. Hence, the `forall` method can be used not only to "iterate" over all elements of a given set, but also to *generate* the set, or part of the set, whose elements satisfy a given property.

Example 20. `forall` *over partially specified sets.*

Let `x`, `y`, *and* `z` *be uninitialized logical variables.*

```
int[] r_elems = {2,4};
Set r = new Set(s_elems);                      // the set {2,4}
Set s = new Set(Set.empty.ins(y).ins(z));      // the partially specified set {y,z}
Solver.forall(x,s,x.in(r));                    // (∀x ∈ {y,z})x ∈ {2,4}
```

The constraint added to the CS is $\mathtt{y} \in \{2,4\} \wedge \mathtt{z} \in \{2,4\}$ *which is clearly satisfiable and the first solution found by the constraint solver will be* $\mathtt{y} = 2 \wedge \mathtt{z} = 4$ *(see Section 5.5).* □

[8] In the current version, no automatic check is performed to enforce the invocation of a `forall` method to satisfy all the restrictions on its format. Rather, this is left to users's accuracy.

The constraint C that occurs in the invocation of a `forall` method, `Solver.forall(x,s,` C`)`, can contain also other uninitialized logical variables y_1, y_2,...,y_n. In this case, the instances of C which are added to the CS all share the same variables y_1, y_2,...,y_n. For example, `Solver.forall(x,{1,2,3},x.neq(y))`, with `x` and `y` uninitialized logical variables, adds to the CS the constraint $1 \neq \mathtt{y} \wedge 2 \neq \mathtt{y} \wedge 3 \neq \mathtt{y}$.

There are situations, however, in which one wants to have different instances of the variables y_1, y_2,...,y_n for each different instance of the constraint C added to the CS by the `forall` method. Logically, this corresponds to consider variables y_1, y_2,...,y_n in C as existentially quantified within the scope of the Restricted Universal Quantifier, that is, $\forall x((x \in s) \rightarrow \exists y_1, \ldots, y_n(C))$.

For this purpose, JSetL provides also a variant of the `forall` method that allows the user to specify which logical variables possibly occurring in C must be considered as "local". Let *LVarsType* be either a `Lvar` or an array of `Lvar` type.

`public static void forall(Lvar x, Set s,` *LVarsType* `Y,` *StoreElemType* `c)`
`throws Failure`

> `Solver.forall(x,s,Y,`C`)`, where `Y` is an uninitialized logical variable or an array of uninitialized logical variables, adds to the CS an instance of C for each element e of `s` with the value of `x` in the instance of C initialized to e and all occurrences of the variables `Y` replaced by newly created logical variables.
>
> For example, `Solver.forall(x,{1,2,3},y,x.neq(y))`, with `x` and `y` uninitialized logical variables, adds to the CS the constraint $1 \neq \mathtt{y}_1 \wedge 2 \neq \mathtt{y}_2 \wedge 3 \neq \mathtt{y}_3$, with $\mathtt{y}_1, \mathtt{y}_2, \mathtt{y}_3$ newly generated uninitialized logical variables.

As a more concrete example, let us consider the problem of checking whether all elements of a given set `s` are pairs, i.e., they have the form `[y1,y2]`, for any `y1` and `y2`.

Example 21. `forall` *with local variables.*

```
Lvar x = new Lvar();                       // an uninitialized l. var.
Lvar y1 = new Lvar();                      // an uninitialized l. var.
Lvar y2 = new Lvar();                      // an uninitialized l. var.
Lvar[] Y = {y1,y2};                        // an array of two l. var's.
Lst pair = Lst.empty.ins1(y1).insn(y2);    // the list [y1,y2]
Solver.forall(x,s,Y,x.eq(pair));           // ∀x(x ∈ s → ∃y1, y2 x = [y1, y2])
```

Let `s` *be the set* `{[1,3],[1,2],[2,3]}`. *The constraint added to the CS is:* $[1,3] = [\mathtt{y1}_1, \mathtt{y2}_1] \wedge [1,2] = [\mathtt{y1}_2, \mathtt{y2}_2] \wedge [2,3] = [\mathtt{y1}_3, \mathtt{y2}_3]$ *which is clearly satisfiable. Note that, if* `y1` *and* `y2` *were not defined as "local" to the invocation of the* `forall` *method (i.e., the* `Y` *parameter is omitted), all equality constraints added to the CS would have the same variables* `y1`*,* `y2` *and the whole constraint turns out to be unsatisfiable.* □

4.4 Constraint visualization

Constraints stored in the constraint store can be visualized by using the static method `showStore()` of class `Solver`.

At each moment, the CS is constituted by all atomic constraints that have been added to the CS and not yet processed by the constraint solver, and by the *irreducible* (or solved form) constraints, that is those atomic constraints that have been processed by the solver and that cannot be further simplified (see Sect. 5). In fact, constraints can be completely eliminated by the constraint solver (logically, they are rewritten to `true`) or they can be rewritten to a simplified irreducible form, and as such left in the CS.

`public static void showStore()`

If the CS contains the atomic constraints $C_1, C_2, \ldots, C_n$, `Solver.showStore()` prints on the standard output the line:

$$\texttt{Constraint store:}\ \ \tilde{C_1}\ \tilde{C_2}\ \ldots\tilde{C_n}$$

where $\tilde{C_i}$ is the string obtained from C_i by replacing the names of all uninitialized variables which possibly occur in it with the associated external names, either user defined or the default ones. If the CS is empty, instead, the following line is printed:

```
Constraint store:  empty
```

This method can be invoked everywhere in the program and it shows the CS state at the moment it is invoked.

Example 22. *Constraint visualization.*

```
Lvar x = new Lvar("x");                    // an uninitialized l. var.
Lvar y = new Lvar("y");                    // an uninitialized l. var.
Lvar z = new Lvar();                       // an uninitialized l. var.
Solver.add(x.neq(Lst.empty.ins1(y)));      // the constraint x ≠ [y]
Solver.add(x.eq(Lst.empty.ins1(z)));       // the constraint x = [z]
Solver.add(z.in(Set.empty.ins(1)));        // the constraint z ∈ {1}
Solver.showStore();
```

When the `showStore` *method is invoked the CS contains the constraint* $\mathtt{x} \neq [\mathtt{y}] \wedge \mathtt{x} = [\mathtt{z}] \wedge \mathtt{z} \in \{1\}$ *which is printed by the* `showStore` *as follows:*

```
Constraint store:  x.neq([y]) x.eq([Lvar_1]) Lvar_1.in({1})
```

If the following statements are executed subsequently:

```
Solver.solve();         // calling the constraint solver
Solver.showStore();
```

we get:

```
Constraint store:  y.neq(1)
```

□

y.neq(1) (i.e. $\mathtt{y} \neq 1$) is an irreducible constraint and as such it is left in the CS, while all other constraints are eliminated. In particular, equality constraints of the form $x = v$, where x is an uninitialized logical variable and v any `Lvar`-value, are always eliminated by the constraint solver, and x becomes initialized to v.

4.5 Constraint deletion

The static method `clearStore()` of class `Solver` allows to delete all constraints contained in the constraint store. After the invocation of this method the constraint store is empty. `clearStore|` can be invoked everywhere in the program.

5 The Constraint Solver

The approach adopted for constraint solving in JSetL is the same developed for CLP($\mathcal{SET}$) [6] and used also in Singleton [10]. The CLP($\mathcal{SET}$) constraint solver tries to reduce any conjunction of atomic constraints to a simplified form - called the *solved form* - which is proved to be satisfiable. The success of this reduction allows one to conclude the satisfiability of the original collection of constraints. On the contrary, the detection of a failure (logically, the reduction to false) implies the unsatisfiability of the original constraints.

A conjunction of atomic constraints C is in *solved form* if it is empty or all its constituting constraints have one of the following forms (where x, x_i are uninitialized logical variables or sets, and t is an expression of any `Lvar`-value type):

1. $x = t$, and x does not occur neither in t, nor in the other constraints of C;
2. $x \neq t$, and x does not occur in t;
3. $t \notin x$, and x does not occur in t;
4. $x_3 = x_1 \cup x_2$, with $x_1 \not\equiv x_2$ and there are no inequalities of the form $x_i \neq t$ or $t \neq x_i$ in C for all $i = 1, 2, 3$;
5. $x_1 \parallel x_2$, and $x_1 \not\equiv x_2$.

A solved form constraint obtained from a constraint C represent a *solution* for C. A constraint C can have more than one solution. The JSetL constraint solver is able to find nondeterministically all correct solutions for a given constraint. Nondeterminism is handled through *choice points* and *backtracking*. Once the constraint reduction process detects a failure, the computation backtracks to the most recently created choice point (chronological backtracking). If no choice point is left open the whole reduction process fails and the `Failure` exception is generated.

We say that the invocation of a method calling (directly or indirectly) the constraint solver terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred.

Solved form constraints are *irreducible* constraints. As such, they are left in the current constraint store and possibly passed ahead to a new invocation of the constraint solver. Conversely, constraints not in solved form are always eliminated (i.e., rewritten to either true or false) or reduced to solved form constraints by the constraint solving process.

The JSetL constraint solver is implemented by the class `Solver`. The main method for constraint solving is the `solve` method. Besides this, the class `Solver` supplies other five methods for constraint solving: `finalSolve`, `solve1`, `boolSolve`, `nextSolution`, `Setof`.

5.1 The `solve` method

`public static void solve()`

> `Solver.solve()` either detects that the constraint in the CS is unsatisfiable and raises a `Failure` exception, or it transforms nondeterministically the constraint in the CS into a constraint in weak solved form.

The *weak solved form* for a constraint C is defined exactly as the solved form except that:

- C can contain comparison constraints involving uninitialized logical variables;
- the conditions over inequality and `union` constraints in the definition of the solved form (item 4) are relaxed.

The weak solved form is not guaranteed to be satisfiable in general (see examples below). However, if a constraint in weak solved form does not contain neither unresolved comparison constraints nor union constraints, then it is in solved form and as such it is guaranteed to be satisfiable (see also the description of the `finalSolve` method.

The constraint solving process performed by the `solve` method involves in general choice points and backtracking, as shown in the following example.

Example 23. *Constraint solving.*

Let `s` *be the set* {`x,y,z`}*, where* `x`*,* `y`*,* `z` *are uninitialized logical variables, and* `r` *be the set* {`1,2,3`}.

```
Solver.add(r.eq(s));       // set unification r = s
Solver.add(x.neq(1));      // x ≠ 1
Solver.solve();            // calling the constraint solver
x.output();
```

The output generated by this code fragment is:

```
x = 2
```

The value for `x` *is computed through backtracking. Solving the set unification problem* {`x,y,z`}={`1,2,3`} *nondeterministically returns one of the six different solutions:*

```
x = 1, y = 2, z = 3,
x = 1, y = 3, z = 2,
x = 2, y = 3, z = 1, ...
```

and so on. Assuming the first computed value for `x` *is* `1`*, then the other constraint,* `x.neq(1)`*, turns out to be not satisfied. Thus, backtracking forces the solver to find another solution for* `x`*, namely* `x = 2`*. In this case, the conjunction of the two given constraints is satisfied, and the invocation of the* `solve` *method terminates successfully.* □

The constraint solver can be invoked more than once. Constraint solving is *incremental*. Every time the `solve` method is invoked it does not restart solving the constraint from the beginning but it restarts from the point reached by the last invocation of `solve`. In particular, the choice points left open by a previous call to the solver are still open and they are explored by the new invocation if needed (see also the `nextSolution` method). For instance, if at the end of the code fragment of Example 23, we add the following statements:

```
Solver.add(Lst.empty.ins1(x).neq(Lst.empty.ins1(2))); // add constraint [x] ≠ [2]
Solver.solve();
x.output();
```

the output generated is:

```
x = 3
```

Often a single invocation of the `solve` method is enough. Sometimes however it may be useful or necessary to invoke the `solve` method more than once. For instance, when the considered problem requires the introduction of a large number of constraints it can be useful to add "intermediate" calls to the `solve` method so that possible unsatisfiability can be detected without having to explore the whole search space. On the other hand, in the case of satisfiable constraint the computational time is not increased by the addition of some calls to the `solve` method, since every time it is invoked the `solve` method restarts to solve the constraint from the point reached by the last invocation.

There are also cases in which the invocation of more than one `solve` is necessary. This happens, generally, whenever the results of the evaluation of the constraints in the CS—in particular the initializations of logical variables—need to be used in the subsequent computation. The following method exemplifies this situation.

Example 24. *Incremental constraint solving.*

Compute the sum of all elements in a set of integers `s`.

```
public static int sumAll(Set s) throws Failure
    {
        int sum = 0;
        while (!s.isEmpty())
        {
            Lvar a = new Lvar();
            Set r = new Set();
            Solver.add(s.less(a,r));
            Solver.solve();
            int newElem = ((Integer)a.value()).intValue();
            sum = sum + newElem;
        }
        return sum;
    }
```

At each step a `less` *constraint is added to the CS to extract an element from the set* `s` *and the extracted element is added to the partial sum obtained by the preceding step. After the introduction of each* `less` *constraint it is necessary to invoke the* `solve` *method, because at the subsequent step it is necessary to know the new set on which the* `sumAll` *method is invoked.* □

The following example shows how the user can handle the situation in which the constraint solver has detected a failure (i.e., the constraint in the CS is unsatisfiable).

Example 25. *Failure handling.*

```
public static void testFailure()
    {
    try {
        Lvar z1 = new Lvar();
        Lvar z2 = new Lvar();
        Solver.add(z1.eq(z2));
        Solver.add(z1.neq(z2));
        Solver.solve();
        return;
        }
    catch(Failure e)
        {
        System.out.print("No solution found");
        }
    }
```

□

The weak solved form generated by the solve method is not guaranteed to be satisfiable.

Example 26. *Unsatisfiable weak solved form.*

If the following sequence of statements is executed

```
Solver.add(x.le(y).and(x.gt(y)));
Solver.solve();
```

with x *and* y *uninitialized logical variables, the CS will contain at the end the weak solved form constraint* $\mathtt{x} < \mathtt{y} \wedge \mathtt{y} > \mathtt{x}$ *which is clearly unsatisfiable.*

Analogously, the following sequence of statements

```
Solver.add(x.subset(y).and(y.subset(x)).and(x.neq(y)));
Solver.solve();
```

with x *and* y *uninitialized sets, causes the constraint solver to produce the following constraint (left in the CS):* $\mathtt{y} = \mathtt{x} \cup \mathtt{y} \wedge \mathtt{x} = \mathtt{y} \cup \mathtt{x} \wedge \mathtt{x} \neq \mathtt{y}$ *which is in weak solved form but it is clearly unsatisfiable.* □

All these situations can be handled properly by using the finalSolve method (see Sect. 5.2) which performs all checks necessary to guarantee that the output constraint is in solved form (hence satisfiable).

5.2 The finalSolve method

The finalSolve method is defined exactly as the solve method, but it generates a solved form constraint. If the constraint solving process is successful, therefore, the constraint left in CS by a finalSolve invocation is guaranteed to be satisfiable.

The finalSolve method actually invokes the solve method but it performs also a final global check on the generated constraint. If the constraint turns out to be unsatisfiable then a Failure exception is raised. If, on the contrary, the constraint solver detects that the CS still contains comparison constraints with uninitialized logical variables—i.e., the constraint solver finds out that it has not enough information to decide—then a Uninitialized_Variable_in_arithmetical_expression exception is raised.[9] Otherwise, it terminates successfully.

For example, using the finalSolve method (instead of the solve method) in the programs of Example 26, we get an Uninitialized_Variable_in_arithmetical_expression exception for the first call, and the detection of a failure (hence, a Failure exception) for the second one.

The finalSolve method can be used in place of the solve method or it can be used in conjunction with it. In the second case, the finalSolve method is usually called at the end of the constraint solving process, after one or more invocations of the solve method. This technique can be useful, on the one hand, to gain some execution efficiency since the global check is performed only once. On the other hand, it can be useful whenever comparison constraints are involved as a way to delay constraints that are not enough specified: specifically, comparison constraints which contain uninitialized variables are simply postponed by the solve method, until the finalSolve method is invoked.

[9] As already observed, the current version of the JSetL constraint solver is not able to deal with integer arithmetic constraints in a complete way. These limitations will be possibly overcome in future releases.

5.3 The `solve1` method

The `solve1` method is defined exactly as the `solve` method, but it does not keep track of the possible unexplored alternatives. In case of backtracking, those alternatives are simply not considered.[10]

The `solve` method and the `solve1` method have exactly the same behavior as concerns the first solution that is possibly found. In particular, in both cases they perform a nondeterministic search of the whole search space, using choice points and backtracking as needed. The difference between the two methods becomes evident if the problem admits more than one solution and we want to get also (some of) the other solutions. Using `solve1` we are not able to find any other solution: any attempt to force backtracking will cause a `Failure` exception to be raised.

The `solve1` method, therefore, can be useful whenever one is not interested in computing all solutions but just the first one. In these cases the use of the `solve1` allows one to drastically prune the search tree, possibly leading to better execution efficiency.

Example 27. *Using* `solve1`.

Check whether `s` *is a subset of* `t`*, i.e.,* `s` $\subseteq$ `t`*.*

```
public static boolean mySubset(Set s, Set t)
    {
    try {
        if(s.isEmpty()) return true;
        else
        {
            Lvar x = new Lvar();
            Set r = new Set();
            Solver.add(s.less(x,r));
            Solver.add(x.in(t));
            Solver.solve1();
            return mySubset(r,t);
        }}
    catch(Failure e)
        {
        return false
        }
    }
```

Using `solve1` *instead of* `solve` *allows us to extract elements from set* `s` *in a deterministic way, that is with a fixed ordering. Thus, in case of backtracking, all other possible orderings for set* `s` *are not taken into account. If, for instance,* `s1` *is the set* {`1,2,3,4`} *and* `s2` *is the set* {`5`}*, the invocation* `mySubset(s1,s2)` *returns* false *after 4 attempts to solve the membership constraint* `x.in(t)`*. Using* `solve`*, in contrast, would cause all possible permutations of set* `s1` *to be considered (4! possibilities), before concluding that no solution can satisfy the given constraints.* □

The `solve1` method must be used with care, since it can cause (useful) solutions to be lost. In the above example, if `s1` is the set {`x,y`}, with `x` and `y` uninitialized variables, and `s2` is the set {`1,2`}, the invocation `mySubset(s1,s2)` returns true with `x` and `y` initialized to 1, while all other solutions (namely, `x` = `1` $\wedge$ `y` = `2`, `x` = `2` $\wedge$ `y` = `1`, `x` = `2` $\wedge$ `y` = `2`) are

[10] This is akin to the "cut" built-in predicate of Prolog, whose purpose is that of "cutting" open alternatives in the search space.

not taken into account, even if we force backtracking (with constraint handling) in order to explore the whole search space. All possible solutions are found if, in contrast, we use the `solve` method.

5.4 The `boolSolve` method

`public static boolean boolSolve()`

> The behavior of the `boolSolve` method is like that of the `solve` method, except that it returns `true` if the constraint solving process is successful and `false` if the constraint solver detects that the constraint in the CS is unsatisfiable (in this case, differently from the `solve` method no exception is raised).

Note that the `boolSolve` method can be always replaced by the proper use of the `solve` method and the exception handling facilities to catch the `Failure` exception possibly generated by the constraint solver.

5.5 The `nextSolution` method: finding one solution at a time

The `nextSolution` method is defined like the `boolSolve` method, but, in case of multiple solutions, it allows to find a new solution each time it is called.

`nextSolution` can be invoked only after a `solve` (or a `boolSolve` or a `finalSolve`) has been invoked. The `solve` (or `boolSolve` or `finalSolve`) invocation finds the first solution (if any) for the constraint in the CS. The subsequent invocation of the `nextSolution` method forces a failure and a new solution, if any, is computed through backtracking. At any time, the result of the `nextSolution` call can be tested to check whether a solution has been found or not. Thus, repeated calls to `nextSolution` allow the user to find one at a time all possible solutions for a given constraint.

Example 28. *Getting one solution at a time.*

Let `x` *be an uninitialized logical variable with eternal name* `"x"` *and* `s` *be the set* `{1,2,3}`.

```
Solver.add(x.in(s));
Solver.solve();
x.output();
Solver.nextSolution();
x.output();
Solver.nextSolution();
x.output();
```

The output produced by this program is:

```
x = 1
x = 2
x = 3
```

Note that if in the above code the calls to `nextSolution` *are replaced by calls to* `solve` *then the output produced is:*

```
x = 1
x = 1
x = 1
```

□

5.6 The `setof` method: all solutions

The `setof` method allows us to explore the whole search space generated by the (nondeterministic) solution of a given constraint C and to collect into a set all the computed solutions for a specified variable `x`.

Let *LType* be either a `Lvar` or a `Lst` or a `Set` type.

`public static void setof(`*LType* `x) throws Failure`

> `Solver.setof(x)`, where `x` is an uninitialized logical variable (or list, or set), returns as its result the set of all possible values of `x` for which the constraint in the CS turns out to be satisfiable.

In other words, the `setof` method allows us to define a set *by property* (or *intensionally*) rather than by enumeration of all its elements (or *extensionally*). The property is represented by the current content of the CS. Specifically, if C is the constraint in the CS, the set denoted by `Solver.setof(x)` is the set `s` of all `x` such that C is true, that is, in logical terms,

$$\forall \mathtt{x}((\mathtt{x} \in \mathtt{s}) \leftrightarrow C).$$

The following two examples show simple usages of the `setof` method.

Example 29. *Using the* `setof` *method: collecting a set of integer.*

Compute the set of all even numbers greater than 0 and less or equal to 10.

```
Set s = new Set(1,10);
Lvar x = new Lvar();
Solver.add(x.in(s));            // add the constraint x ∈ s
Solver.add(x.mod(2).eq(0)));    // add the constraint x mod 2 = 0
Solver.setof(x).output();       // collect and print all solutions for x
```

The output generated is: {`2,4,6,8,10`}. □

Example 30. *Using the* `setof` *method: collecting a set of sets .*

Compute the set of all subsets (i.e., the powerset) of a given set `s`.

```
public static Set powerset(Set s) throws Failure
    {
    Set r = new Set();          // an uninitialized set
    Solver.add(r.subset(s));    // add constraint r ⊆ s
    return Solver.setof(r);     // collect all solutions for r
    }
```

If `s` *is the set* {`'a','b'`} *(and the CS is empty), the set returned by* `powerset(s)` *is* {{}, {`'a'`}, {`'b'`}, {`'a','b'`}}. □

Very often, the constraint in CS contains uninitialized logical variables when the `setof` method is called. From a logical point of view, these variables must be considered as existentially quantified within the scope of the intensional definition of the set s. That is, if C is the constraint in the CS and y_1, y_2,…,y_n are the uninitialized logical variables in C, the set `s` denoted by `Solver.setof(x)` is

$$\forall \mathtt{x}((\mathtt{x} \in \mathtt{s}) \leftrightarrow \exists y_1, \ldots, y_n(C)).$$

Example 31. `setof` *over constraints with uninitialized variables.*

Given a set `s` *compute the set of all pairs* `[x,y]` *such that* `x` *and* `y` *belong to* `s` *and* $x \neq y$.

```
Lvar x = new Lvar();
Lvar y = new Lvar();
Lst pair = new Lst(Lst.empty.ins1(x).insn(y));     // a pair [x,y]
Solver.add(x.in(s));          // add the constraint x ∈ s
Solver.add(y.in(s));          // add the constraint y ∈ s
Solver.add(x.neq(y));         // add the constraint x ≠ y
Solver.setof(pair).output();  // collect and print all solutions for x
```

If `s` *is the set* {1,2,3,4} *(and the CS is empty), the set printed by this code fragment is*

{[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],[4,1],[4,2],[4,3]}.

□

The `setof` method uses the `finalSolve` method to compute one after the other all solutions that satisfy the constraint in the CS. After each solution has been computed, `setof` forces a failure and a new choice (if any) is considered through backtracking. Each value computed for the parameter of the `setof` invocation is collected into a set which is returned as the result of the invocation. At the end of the `setof` execution the constraint in the CS is in solved form (possibly empty).[11]

Variables `x` and `y` in the above example are intended to be used as "local" to the `setof` invocation. As such their values after this invocation are unspecified (implementation dependent) and cannot be relied on.

5.7 Computation time

The `Solver` class provides also two methods for getting and printing the execution time.

`public static double getTime()`

> `Solver.getTime()` returns as its result the time in seconds from the beginning of the main thread execution.

`public static void printTime()`

> `Solver.printTime()` computes the time in seconds from the beginning of the main thread execution and prints it on the standard output stream with the format `time =` n `sec`, where n is the computed time.

6 User-defined constraints

JSetL allows the user to define new constraints, and in particular constraints whose solution involves nondeterminism. User-defined constraints are dealt with as the built-in constraints: they can be added to the constraint store using the `add` method and solved using the `Solver` methods for constraint solving.

[11] Precisely, if the given constraint admits more than one solution—each one represented by a different solved form—the last computed solved form is the one left in the CS at the end of the execution of the `setof` method.

Definitions of user-defined constraints, however, require some programming conventions to be respected.[12] In particular, all definitions must be provided as part of the `NewConstraints` class. This class extends the `Solver` class and must be defined as part of the JSetL package:

```
package JsetL;
public class NewConstraints extends Solver
    {
        New constraint definitions
    }
```

6.1 New constraint definitions

Let us assume the user wants to define a new constraint named `newc` with n arguments `a1`, ..., `an` of type $t1$, ..., tn, respectively. To do this, the user has to define three new methods in the class `NewConstraints`.

The first method provides the definition of `newc` as a constraint:

```
public static StoreElem newc(t1 a1, ..., tn an)
    {
    StoreElem s = new StoreElem(k,a1,...,an);
    return s;
    }
```

This method creates an instance of the class `StoreElem` which is used to store the new constraint within the CS. The instance contains all parameters for the `newc` constraint, along with an integer k which will be used by the solver to uniquely identify the new constraint. k is called the *constraint internal code* and must be an integer (greater than 100 and not equal to the ones previously used).

Whenever a program wants to add a `newc` constraint to the CS it can invoke the `add` method of the `Solver` class as follows:

```
Solver.add(NewConstraints.newc(e1,...,en))
```

where $e1$,...,en are expressions of type $t1$,...,tn respectively.

The second method that must be included in the `NewConstraints` class, called `user_code`, is used to associate the new constraint internal code with the actual implementation of the constraint itself:

```
protected static void user_code(int c, StoreElem s) throws Failure
    {
    switch(c)
        { ...
        case k:  newc(s); break;
        ...}
    return s;
    }
```

For each new user-defined constraint defined in the program, with associated internal code j, the `user_code` method must provide a new **case** alternative for value j in the

[12] This task will be simplified in next releases by the adoption of suitable preprocessing tools that automatically force the program to conform to these conventions.

switch statement. The purpose of this alternative is simply to invoke the method which implements the associated constraint. The `user_code` method is called directly by the solver whenever it detects that the constraint internal code of the atomic constraint it is processing does not correspond to any built-in constraint. Conversely, the `user_code` method can not be invoked by any method in the user program.

Finally, the `NewConstraints` class must provide a method which implements the constraint solving procedure for the new constraint:

```
protected static void newc(StoreElem s) throws Failure
    {
    t1 a1 = (t1)s.arg1;
    ...
    tn an =(tn)s.argn;
    implementation of constraint newc
    }
```

The values of the parameters passed to the constraint when added to the CS can be retrieved by accessing the protected data members `arg1`, `arg2`,...,`argn` of the object `s` of class `StoreElem` (in the current version, the number of arguments is limited to 4).

6.2 Implementing nondeterministic new constraints

Constraint solving for a new constraint can require nondeterminism. JSetL provides a number of facilities for nondeterminism handling which can be exploited in the implementation of the constraint within the `NewConstraints` class (since these facilities are implemented as protected they can be used in the `NewConstraints` class, not in the user program code). In order to exploit these facilities, however, the user has to adhere to some programming conventions.

Let the definition of the constraint solving procedure for the `newc` constraint to be based on m different nondeterministic alternatives. Then the user must provide a **switch** statement with m case blocks (numbered from 0 to $m-1$), one for each nondeterministic alternative, as follows:

```
protected static void newc(StoreElem s) throws Failure
    {
    t1 a1 = (t1)s.arg1;
    ...
    tn an =(tn)s.argn;
    switch(s.caseControl)
        {
        case 0:
            Backtracking.add_ChoicePoint(s);
            alternative 1
            return;
        case 1:
            Backtracking.add_ChoicePoint(s);
            alternative 2
            return;
            ...
        case m:
            alternative m
            return;
        }
```

```
    }
```

The control expression of the **switch** statement is the `caseControl` attribute of the store element `s` associated with the `newc` constraint (default value: 0). Each `case` block, but the last one, creates a choice point and adds it to the stack of the alternatives by executing the statement `add_ChoicePoint(s)`. The remaining code of the `case` block provides the actual implementation of this alternative.

As an example we show a fully nondeterministic recursive definition of the classical list concatenation operation (`concat`). `concat` takes three lists as its parameters, `l1`, `l2`, `l3`, and succeeds if `l3` is the concatenation of `l1` and `l2`.

Example 32. *The new constraint* `concat`.

```
class NewConstraints
  {
  public static StoreElem concat(Lst l1, Lst l2, Lst l3)
     {
     StoreElem s = new StoreElem(n,l1,l2,l3);
     return s;
     }

  protected static void user_code(int c, StoreElem s)
  throws Failure
     {
     switch(c)
        { ...
        case n:  concat(s); break;
        ...}
     return s;
     }

  protected static void concat(StoreElem s)
  throws Failure
     {
     Lst l1 = (Lst)s.arg1;
     Lst l2 = (Lst)s.arg2;
     Lst l3 =(Lst)s.arg3;
     switch(s.caseControl)
        {
        case 0:
           Backtracking.add_ChoicePoint(s);
           add(l1.eq(Lst.empty));
           add(l2.eq(l3));
           return;
        case 1:
           Lvar x = new Lvar();
           Lst l1new = new Lst();
           Lst l3new = new Lst();
           add(l1.eq(l1new.ins1(x)));     // l1 = [x | l1new]
           add(l3.eq(l3new.ins1(x)));     // l3 = [x | l3new]
           add(concat(l1new,l2,l3new));  // concat(l1new,l2,l3new)
           return;
        }
```

```
    }
}
```

concat *can be used both to check if a given concatenation of lists holds and to build any of the three lists, given any of the other two. Such flexibility is obtained by using unification and nondeterminism. The first (nondeterministic) alternative of the* switch *statement states that when* l1 *is the empty list,* l2 *and* l3 *must be equal. The second alternative deals with the case in which the first element of* l1 *is* x *so that* l3 *is obtained by inserting* x *as the head element of the list* l3new *which is obtained (recursively) by concatenating the rest of* l1 *(i.e.,* l1new*) with* l2.[13]

If, for instance, l1 *is* [1,2,3]*,* l2 *is* [4,5]*, and* l3 *is an uninitialized list, execution of the statement*

Solver.add(NewConstraints.concat(l1,l2,l3))

and the subsequent call to Solver.solve()*, will set* l3 *equal to* [1,2,3,4,5]*. If, on the contrary,* l1 *and* l2 *are uninitialized lists and* l3 *is the list* [1,2,3,4,5]*, execution of the same two statements will cause* l1 *and* l2 *to get the values* [] *and* [1,2,3,4,5]*, respectively. This is just one of the six different solutions for the given constraint (the others being,* $\texttt{l1} = [1] \wedge \texttt{l2} = [2,3,4,5]$, $\texttt{l1} = [1,2] \wedge \texttt{l2} = [3,4,5]$*, and so on).* □

6.3 Using the NewConstraints class

Besides the definition of new methods to be used as (real) constraints in the current computation, the user can exploit the facilities provided by the NewConstraints class whenever he/she wants a part of his/her program to be "sensible" to backtracking.

As an example, let us consider the following trivial program fragment, where we assume that x is an uninitialized logical variable (with external name "x") and s is the set {0,1}.

```
Solver.add(x.in(s));
Solver.solve();
x.output();
Solver.add(x.neq(0));
Solver.solve();
```

One could expect that this program (and in particular the output statement) prints all values nondeterministically assigned to x, namely, 0 and 1. Unfortunately, this is not true: indeed only the first value assigned to x is printed.

The problem is caused by the fact that nondeterminism in JSetL is confined to constraint solving: backtracking allows the computation to go back to the nearest open choice point *within* the constraint solver, but it does not allow to "re-execute" user program code. In this example, the output statement is not re-executed when the second alternative for the x.in(s) constraint is taken into account.

The problem can be avoided using the NewConstraints class. The piece of code that requires to be re-executed is defined as the body of a new constraint in the NewConstraints class, with proper parameter passing. The code in the original program then is replaced by a call to the add method which adds the new constraint to the CS. With reference to the above example, we can define a new constraint in NewConstraints, named printVar, which simply invokes the output method on its argument x:[14]

[13] Note that if we use an **if** statement in place of the **switch**, with the **if** condition simply checking whether l1 is the empty list, then concat continues to work correctly whenever the first and second list are given and the third list has to be computed out of them. Conversely, this method can no longer be used in the other direction, i.e. to split the third list into its two components lists.

[14] Besides the new methods that implements the new constraint also the other two interface methods

```
protected static void printVar(StoreElem s) throws Failure
    {
    Lvar x = (Lvar)s.arg1;
    x.output();
    return;
    }
```

Then the original program fragment is modified as follows:

```
Solver.add(x.in(s));
Solver.solve();
Solver.add(NewConstraints.printVar(x));
Solver.add(x.neq(0));
Solver.solve();
```

With these changes, execution of this program causes all values assigned to `x` to be printed, that is:

```
x = 0
x = 1
```

As a more complex example, let us consider the following program fragment, where we assume that `x` and `s` are defined as in the above example, `y` is an uninitialized logical variable, and `s1` and `s2` are the sets {'a','b'} and {'c','d'}, respectively.

```
Solver.add(x.in(s));
Solver.solve();
if (x.value().equals(new Integer(0)) Solver.add(y.in(s1));
else Solver.add(y.in(s2));
Solver.add(y.eq('c'));
Solver.solve();
x.output();
```

Assume that when evaluating the `if` condition the value of the logical variable `x` is `0` and therefore that the constraint `y.in(s1)` is added to the constraint store. When, subsequently, a failure is detected (due to the inconsistent constraints on `y` stored in the CS), backtracking will allow to consider a different value for `x`, namely `1`, but the `if` condition is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for `x`, but still with the constraint `y.in(s1)` added to it and it will detect a failure again. Observe that if the first value assigned to `x` would be `1` (instead of `0`), the computation would terminate successfully.

Again the problem is caused by the fact that backtracking allows to consider all possible values for `x`, but the `if` statement is no longer re-executed after backtracking took place.

The problem can be solved in the same way shown before. The piece of code that requires to be re-executed, that is actions of the `if` statement, are implemented as a new constraint (called `condAdd`) within the `NewConstraints` class. The `if` condition is rendered through equality and inequality constraints, while the two alternatives of the `if` statement are implemented as two nondeterministic cases of the new constraint definition.

```
protected static void condAdd(StoreElem c)
throws Failure
    {
    Lvar x = (Lvar)c.arg1;
```

required by the definition of a new constraint must be suitably defined in the `NewConstraints` class, as shown in Sect. 6.1.

```
Lvar y = (Lvar)c.arg2;
Set s1 = (Set)c.arg3;
Set s2 = (Set)c.arg4;
switch(c.caseControl)
    {
    case 0:
        Backtracking.add_ChoicePoint(c);
        add(x.eq(0));
        add(y.in(s1));
        return;
    case 1:
        add(x.eq(1));
        add(y.in(s2));
        return;
    }
}
```

The original code fragment is modified as follows

```
Solver.add(x.in(s));
Solver.solve();
Solver.add(NewConstraints.condAdd(x,y,s1,s2));
Solver.add(y.eq('c'));
Solver.solve();
x.output();
```

With these changes, the behaviour is the expected one: the program terminates successfully, printing `x = 1`.

7 Examples

In this section we present two simple examples that show how to use the JSetL library to solve constraint satisfaction problems, and in particular how to use sets and set constraints. The considered problema are the well known *n-queens* problem and the *traveling salesman* problem.

7.1 n-queens problem

The n-queens problem consists in trying to place N queens on a $N \times N$ chess board such that no two queens attack each other, i.e., no two queens are placed on the same row, the same column or the same diagonal.

To model the problem we observe that any solution will have a queen on each row and one on each column. We can then represent each row with a logical variable, $r_0, ..., r_{N-1}$. For every row there are N columns on which we can place the queen. If we identify the columns with the integer $0, ..., N-1$, the resolution of the problem consists in assigning a value among 0 and $N-1$ to each of the N logical variables $r_0, ..., r_{N-1}$ such that

$$
\begin{array}{ll}
(i) & r_i \neq r_j \\
(ii) & r_j + j - r_i \neq i \\
(iii) & r_i + j - r_j \neq i
\end{array}
$$

for each $i \neq j$, $0 \leq i, j \leq N-1$. The first inequality states that two queens can not be placed on the same column, while the other two inequalities state that two queens can not be placed on the same diagonal.

The following is the Java program for n-queens problem using JSetL (with $N = 4$).

```
import JSetL.*;
import java.io.*;
class Queens
{
  public static final int N = 4;
  public static void main (String[] args) throws IOException,Failure
  {
    Set colmns = new Set(0,N-1); // a set of N integers 0..N − 1 representing columns
    Lst rows = Lst.mkLst(N);     // a list of N l. var's representing rows

    for(int i = 0; i < N; i++)  // assign a distinct int. in 0..N − 1 to each var. in rows
       Solver.add(((Lvar)rows.get(i)).in(colmns));
    Solver.allDifferent(rows);

    for(int i = 0; i < N-1; i++)  // add constraints on diagonals
       for(int j = i+1; j < N; j++){
          Solver.add(((Lvar)rows.get(j)).sum(j).sub((Lvar)rows.get(i)).neq(i));
          Solver.add(((Lvar)rows.get(i)).sum(j).sub((Lvar)rows.get(j)).neq(i));
       }
    Solver.solve(); // check constraints

    for(int i = 0; i < N; i++) { // print a solution
       System.out.print("\n Queen "+i+" = ");
       ((Lvar)rows.get(i)).print();
    }
    return;
  }
}
```

Integers in set `colmns` indicate the columns on which the queens can be placed. Logical variables in list `rows` indicate the rows of the chess board. A value c in the i-th `Lvar` of `rows` indicates a queen placed in row i at column c. The `allDifferent` method assures that only one queen is placed on each different column. The other two constraints added to the CS through the `add` method implement inequalities (*ii*) and (*iii*) (note that `rows.get(i)` and `rows.get(j)` denote the i-th and the j-th `Lvar` of the list `rows`, respectively).

A possible solution printed by this program is:

```
Queen0 = 1
Queen1 = 3
Queen2 = 0
Queen3 = 2
```

that represents the situation depicted in Figure 1.

7.2 Travelling Salesman Problem

As a second example we consider a simplified version of the traveling salesman problem.

Given a directed labeled graph G, the *traveling salesman problem (TSP)* consists in determining whether there is a path in G starting from a source node, passing exactly once for every other node, and returning in the initial node (here we are making the simplification

	$\mathcal{Q}$		
			$\mathcal{Q}$
$\mathcal{Q}$			
		$\mathcal{Q}$	

Figure 1: A solution for the n-queen problem with $n = 4$.

of not considering costs attached to edges; the problem, and the proposed solution, however, can be straightforwardly extended to weighted graphs as well, where the goal is that of finding a path of global cost less than a constant k).

A directed labeled graph G can be represented as a pair $\langle N, E \rangle$ where N is the set of nodes and E is the set of edges, and each edge has the form $\langle n_1, n_2 \rangle$, with $n_1, n_2 \in N$. This representation of graphs has an immediate implementation using JSetL 's data structures: N can be implemented as a `Set` object containing simple elements (e.g., strings), and E as a `Set` object whose elements are lists (i.e., `Lst` objects) of two nodes. If E contains the list $[a, b]$ it means that the graph contains an arc from node a to node b (note that arcs can be conveniently implemented using sets—instead of lists—if the graph if undirected).

The following is the Java program which is able to solve the (simplified) TSP problem using JSetL.

```
import JSetL.*;
import java.io.*;
class Tsp {
   static Set visited = Set.empty;
   static Lst path = Lst.empty;
   static int n;

public static void main (String[] args)
throws IOException, Failure
{
   InputStreamReader reader = new InputStreamReader(System.in);
   BufferedReader in = new BufferedReader(reader);
   Set nodes = readNodes(in);
   Set edges = readArcs(in,nodes);
   Lvar source = readSource(in);

   Solver.add(source.in(nodes));
   Solver.solve();
   visited = visited.ins(source);
   path = path.ins1(source);
   tsp(edges,source,source);

   showPath();
   return;
}
```

```
 public static void tsp(Set edges,Lvar source,Lvar startNode)
 throws Failure
 {
    Lvar nextNode = new Lvar();
    Lst newArc = new Lst(Lst.empty.ins1(startNode).insn(nextNode));
    Solver.add(newArc.in(edges));
    Solver.add(nextNode.nin(visited));
    Solver.solve();
    visited = visited.ins(nextNode);
    path = path.insn(nextNode);
    if(path.size() < n) tsp(edges,source,nextNode);
    else tspLast(edges,source,nextNode);
    return;
 }

 public static void tspLast(Set edges,Lvar source,Lvar lastNode)
 throws Failure
 {
    Lst backArc = new Lst(Lst.empty.ins1(source).ins1(lastNode));
    Solver.add(backArc.in(edges));
    Solver.solve();
    return;
 }

 public static Set readNodes(BufferedReader in)
 throws IOException
 { ... }

 public static Set readArcs(BufferedReader in,Set nodes)
 throws IOException
 { ... }

 public static Lvar readSource(BufferedReader in)
 throws IOException
 { ... }

 public static void showPath()
 throws IOException
 { ... }
}
```

The set `visited` is the set of all the already examined nodes, while the list `path` is used to store the computed path. Initially they are both empty. The `main` method first asks the user to supply the nodes and the edges of the graph, and then the initial node which is stored in the logical variable `source`. We assume the interaction with the user is implemented by methods `readNodes`, `readArcs`, `readSource` in some reasonable way.

Methods `tsp` and `tspLast` provide our solution to the TSP. Statements in the `main` method preceding the invocation of the `tsp` method state that the source node must belong to the set of nodes of the given graph G and that it must be added to the set of visited nodes and (as the first element) to the list that represents the computed path. Then the `tsp` method is called with `source` as the start node `startNode`. The definition of the `tsp` method simply states that an edge leaving from node `startNode` and ending in a node `nextNode` not yet visited must belong to the set of edges of G. If these constraints are satisfiable, then `nextNode` is added to the set of visited nodes as well as to the computed path, and the `tsp` method is called recursively with `nextNode` as the new initial node, unless all nodes have been already visited. In the last case, the `tspLast` method is called instead; this method simply states that there must be an edge (`backEdge`) from the last visited node to the initial one.

If the problem admits at least one solution the invocation of the `tsp` method within the `main` method terminates successfully. Thus the `main` method can invoke `showPath()` which prints the solution stored in `path`.

The following is an example of a program execution (assuming suitable user interaction is implemented by methods `readNodes`, `readArcs`, `readSource` and `showPath`):

```
Insert the set of all nodes (separated by blanks): a b c d e

Insert the nodes reachable from a : b c
Insert the nodes reachable from b : a e
Insert the nodes reachable from c : b
Insert the nodes reachable from d : a
Insert the nodes reachable from e : d

Insert the initial node: a

Path: a -> c -> b -> e -> d -> a
```

References

[1] K.R. Apt and A. Schaerf. Programming in Alma-0, or Imperative and Declarative Programming Reconciled. In *Frontiers of Combining Systems 2*, D.M. Gabbay and M. de Rijke (editors),Research Studies Press Ltd, 1-16, 2000.

[2] D. Diaz, P. Codognet. A minimal extension of the Wam for CLP(fd). In *Proc. of the 10th International Conference on Logic Programming*, 1993.

[3] A. Chun. Constraint programming in Java with JSolver. In *Proc. of Practical Applications of Constraint Logic Programming PACLP99*, 1999.

[4] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.

[5] A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets Research Report "Quaderni del Dipartimento di Matematica", Università di Parma, n.235, September 200.

[6] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[7] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19–20*, 1994, 503–581.

[8] ILOG Optimisation Suite - White Paper. Available at `www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.

[9] M. Leconte and J.-F. Puget. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 International Symposium on Logic Programming*, MIT press, pp. 513-527.

[10] G. Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Electronic Notes in Theoretical Computer Science, Vol. 76, Elsevier Science B. V., 17 pages, 2002.

A Exception classes

In this appendix we give a brief description of all the exception classes of JSetL. Class `Failure` extends class `Exception`: it defines a "controlled" exception, hence it has to be declared in the `throws` clause of all methods that may (either directly or indirectly) generate exceptions of this type. All other classes extend class `RuntimeException`: they define "uncontrolled" exceptions, that is they can be generated without being declared in the `throws` clauses.

EmptyLstException

This exception ir raised whenever the `ext1` and the `extn` methods are applied to an empty list.

Failure

This exception is raised whenever the constraint solver finds that the constraint in the CS has no solution.

InitLvarException, InitLstException, InitSetException

These exceptions are raised whenever we try to apply to an initialized `Lvar` or `Lst` or `Set` object a method that requires an uninitialized one.

NotDefinedConstraint

This exception is raised whenever we try to add to the CS a constraint that is neither a predefined constraint nor a user-defined constraint.

NotInitVarException

This exception is raised whenever we try to apply to an uninitialized `Lvar` or `Lst` or `Set` object a method that requires an initialized one.

NotIntLvarException

This exception is raised whenever an arithmetic operator or a comparison constraint is applied to an `Lvar` initialized with a not integer value.

NotLstException

This exception is raised whenever a method of class `Lst` is applied to an `Lvar` initialized with a not `Lst`-value.

NotSetException

This exception is raised whenever a method of class `Set` is applied to an `Lvar` initialized with a not `Set`-value.

Uninitialized_Variable_in_arithmetical_expression

This exception is raised whenever the `finalSolve` method finds that unresolved comparison constraints are still present in CS at the end of the computation.