tuProlog Guide

tuProlog version: 2.1

tuProlog IDE version: 2.0

Last Changes date: 2007-04-19

Alma Mater Studiorum—Università di Bologna a Cesena, Italy

Contents

1 What is tuProlog **4**

2 Installing tuProlog **6**

3 Getting Started **9**
- 3.1 Prolog Programmer Quick Start . . . 9
- 3.2 Developer Quick Start . . . 10

4 tuProlog Basics **12**
- 4.1 Structure of a tuProlog Engine . . . 12
- 4.2 Prolog syntax . . . 13
- 4.3 Configuration of a tuProlog Engine . . . 16
- 4.4 Built-in predicates . . . 17
  - 4.4.1 Control management . . . 17
  - 4.4.2 Term Unification and Management . . . 18
  - 4.4.3 Knowledge-base management . . . 18
  - 4.4.4 Operators and Flags Management . . . 19
  - 4.4.5 Libraries Management . . . 20
  - 4.4.6 Directives . . . 20

5 tuProlog Libraries **22**
- 5.1 BasicLibrary . . . 24
  - 5.1.1 Predicates . . . 24
  - 5.1.2 Functors . . . 32
  - 5.1.3 Operators . . . 32
- 5.2 ISOLibrary . . . 34
  - 5.2.1 Predicates . . . 34
  - 5.2.2 Functors . . . 35
  - 5.2.3 Operators . . . 36
  - 5.2.4 Flags . . . 36

5.3 DCGLibrary . . . 37
5.3.1 Predicates . . . 38
5.3.2 Operators . . . 38
5.4 IOLibrary . . . 38
5.4.1 Predicates . . . 39

6 Accessing Java from tuProlog 43
6.1 Mapping data structures . . . 43
6.2 General predicates description . . . 43
6.3 Predicates . . . 49
6.3.1 Method Invocation, Object and Class Creation . . . 49
6.3.2 Java Array Management . . . 51
6.3.3 Helper Predicates . . . 52
6.4 Functors . . . 52
6.5 Operators . . . 52
6.6 Java Library Examples . . . 53
6.6.1 RMI Connection to a Remote Object . . . 53
6.6.2 Java Swing GUI from tuProlog . . . 53
6.6.3 Database access via JDBC from tuProlog . . . 54
6.6.4 Dynamic compilation . . . 55

7 The IDE 58
7.1 Editing the theory . . . 60
7.2 Solving goals . . . 60
7.3 Debug Informations . . . 64
7.4 Dynamic library management . . . 64

8 Using tuProlog from Java 70
8.1 Getting started . . . 70
8.2 Basic Data Structures . . . 71
8.3 Engine, Theories and Libraries . . . 74
8.4 Some more examples of tuProlog usage . . . 77

9 How to Develop New Libraries 81
9.1 Implementation details . . . 81
9.2 Library Name . . . 83

10 Exceptions 85
10.1 Exceptions in ISO Prolog . . . 85
10.1.1 Examples . . . 86

10.1.2 Error classification 88
10.2 Implementing Exceptions in tuProlog 90
10.2.1 Java exceptions from tuProlog 91
10.2.2 Examples 92

Chapter 1

What is tuProlog

tuProlog is a light-weight Prolog framework for distributed applications and infrastructures. tuProlog is developed and maintained by the aliCE research group[1] at the Alma Mater Studiorum—Università di Bologna. It is built as an Open Source software, released under the LGPL license – thus allowing also for commercial derivative work –, and made available through the pages of the APICe web portal[2].

tuProlog is designed to be *minimal*, dynamically *configurable*, *interoperable*, straightforwardly *integrated* with Java and .NET, and easily *deployable*.

First of all, tuProlog is designed with *minimality* in mind. Accordingly, tuProlog core is a tiny Java object that contains only the essential properties of a Prolog engine. Only the required Prolog features – like, say, ISO compliance, I/O predicates, DCG operators – are then to be added to or removed from a tuProlog engine according to the contingent application needs.

The obvious counterpart of minimality is tuProlog *configurability*. In fact, a simple yet powerful mechanism based on the notion of tuProlog *library* is provided tha allows required predicates, functors and operators to be loaded and unloaded in a tuProlog engine, both statically and dynamically. Libraries can be either included in the standard tuProlog distribution, or defined *ad hoc* by the tuProlog user / developer.

A tuProlog library can be built in different ways. First of all, a tuProlog library could be straightforwardly written in Prolog. On the other hand, a tuProlog library could also be implemented using either Java or any language of .NET framework—depending on the chosen tuProlog implementation. Finally, a tuProlog library could be built by combining Prolog and Java /

[1] `http://www.alice.unibo.it`
[2] `http://tuprolog.apice.unibo.it`

.NET languages, thus paving the way for multi-language / multi-paradigm integration. Whatever the language(s) used, a tuProlog library can be either used to configure a tuProlog engine when this is started up, or loaded – and then unloaded – dynamically at any time during the engine execution.

tuProlog was first implemented upon Java, then ported upon .NET, and is now available on both platforms. *Deployability* of tuProlog owes a lot to Java and .NET. On the Java side, the requirements for tuProlog installation simply amount to the presence of a standard Java VM, and a Java invocation upon a single JAR file is everything needed to start a tuProlog activity. On the .NET side, ... ENRICO ENRICO ENRICO ENRICO

tuProlog *integration* with other languages and paradigms is kept as clean as possible, so that the components of a tuProlog application can be developed by choosing at any step the most suitable paradigm—either declarative/logic or imperative/object-oriented. On the Prolog side, thanks to the `JavaLibrary` library, any Java entity (object, class, package) can be represented as a Prolog term, and exploited from Prolog. So, for instance, Java packages like Swing and JDBC can be directly used from within Prolog, straightforwardly enhancing tuProlog with graphics and database access capabilities. In the same way, `DotNetLibrary` ... o come cavolo si chiama ENRICO ENRICO ENRICO ENRICO On the Java side, a tuProlog engine can be invoked and used as a simple Java object, possibly embedded in beans, or exploited in a multi-threaded context, according to the application needs. Also, a multiplicity of different tuProlog engines can be used from a Java program at the same time, each one configured with its own libraries and knowledge base. In the same way, sbrodolata .NET di ENRICO ENRICO ENRICO ENRICO

Interoperability misteriosa: cosa diciamo??? Ancora quanto segue???

Finally, *interoperability* is developed along two main lines: Internet standard patterns, and coordination models. So, tuProlog supports interaction via TCP/IP and RMI, and can be also provided as a CORBA service. In addition, tuProlog supports tuple-based coordination under many forms. First, components of a tuProlog application can be organised around Java-based tuple spaces, logic tuple spaces, and ReSpecT tuple centres [**?**]. Then, tuProlog applications can exploit Internet infrastructures providing tuple-based coordination services, like LuCe [**?**] and TuCSoN [**?**].

Chapter 2

Installing tuProlog

First, you need to have the tuProlog distribution. You can download it from the tuProlog web site:

`http://tuprolog.alice.unibo.it/`

You can find the latest version in the Download section. The distribution file has the form `2p-`*X*`.`*Y*`.`*Z*`.zip`, where *X*.*Y*.*Z* identifies the version of tuProlog: for instance, the distribution file `2p-2.0.zip` contains version 2.0 of the engine. After the download, unzip the distribution file in a folder of your choice in the file system; you should obtain the following directory tree:

```
2p-2.0
|---lib
|---doc
|   |---api
|---test
|---src
```

The `lib` directory contains the tuProlog Java binaries packaged in the JAR format:

- `2p.jar` contains everything you need to use tuProlog, such as the core API, the `Agent` application, libraries, IDE tools and other extensions.
- In addition, you find three other JAR files, provided as helper packages for users who would like to exploit some specific parts only from the tuProlog distribution:

- `tuprolog.jar` contains the core API, the `Agent` application and default libraries.
- `tuprolog-ide.jar` contains the IDE tools only.
- `tuprolog-extensions.jar` contains add-on libraries and other tuProlog extensions.

The `doc` directory contains this Guide and the Java documentation about tuProlog API, collected in the subdirectory `api`. The `test` directory contains the source code of unit and acceptance tests[1] for the software, as well as some demos to illustrate usage of libraries. Finally, the `src` directory contains the Java source for the tuProlog engine.

After downloading and unpacking the distribution on your system, you can install tuProlog in different ways, depending on how you want to use it.

- You may want to use tuProlog from a directory playing the role of a central repository where you usually install other programs and third-party libraries.[2] In this case, you have to move under the chosen filesystem tree the tuProlog directory you have already extracted. Then, you need to remember to add the `-cp <jar file>` option when invoking the Java interpreter, specifying the path to the `2p.jar` file contained in the `lib` subdirectory of the distribution. For instance, suppose that you unzipped the `2p-2.0.zip` distribution file in the `/java/tools` folder and you need to run your *ApplicationClass* application with tuProlog; then you should invoke the Java interpreter as follows:

  ```
  java -cp /java/tools/2p-2.0/lib/2p.jar ApplicationClass
  ```

  Alternatively, you can add the required tuProlog JAR file to your `CLASSPATH` environment variable,[3] thus avoiding to specify the `-cp` option every time you invoke the interpreter. In this way you can exploit tuProlog applications simply by invoking the Java intepreter as follows:

[1] tuProlog exploits JUnit (see `http://www.junit.org/`) for its unit testing needs and FIT (see `http://fit.c2.com/`) as its acceptance testing framework.

[2] Predefined examples of such a directory include `C:\Program Files` in Windows, `/Library/Applications` under Mac OS X, `/usr/share` under most *nix environments.

[3] Consult your operating system's manual for details regarding how to set and create environment variables.

`java` *`ApplicationClass`*

You can use the distribution content also by means of the scripts provided in the `bin` directory of the distribution; such scripts use the JAR located in the `lib` directory.

- You may want to use tuProlog from your current working directory. In this case, you have to copy the `2p.jar` file from the `lib` subdirectory in the extracted distribution to your working directory. Then, after you move directly in that directory, by means of a terminal or command line prompt, you can execute:

  `java -cp 2p.jar` *`ApplicationClass`*

  which invokes the Java interpreter and let it use the classes from tuProlog. As previously explained, you can also use the `CLASSPATH` environment variable to obtain the same effect.

- You may want to directly use the class files contained in the `2p.jar` archive from the tuProlog distribution. In this case, first copy the JAR file to your directory of choice; then, unfold it by means of the `jar` command provided by the Java distribution. For instance, open a terminal or a command line prompt from within that directory, and execute:

  ```
  jar -xvf 2p.jar
  ```

  After this operation, you can use tuProlog applications directly from that directory, with no need to specify any interpreter's option nor to exploit the operating system's environment variables.

Chapter 3

Getting Started

The tuProlog distribution offers some tools either to consult and execute already existing Prolog programs, or to help developing new Prolog theories and interact with a Prolog engine. Depending on the use you would like to make of tuProlog, you may want to start exploring the distribution tools along different directions.

3.1 Prolog Programmer Quick Start

As a Prolog programmer, you would like to start trying tuProlog by running your already existing Prolog programs. You can execute your programs in the form of source text files using the tuProlog Agent tool. This tool accepts as arguments the name of a text file containing a Prolog theory and, optionally, the goal to be solved; then it starts the demonstration. Once you have properly installed tuProlog in the *dir* directory, you can use the following template to invoke the Agent tool from the command line:

```
java -cp dir/2p.jar
         alice.tuprolog.Agent PrologTextFile {Goal}
```

For instance, suppose a text file named `hello.pl` in your current directory contains the following line:

```
go :- write('hello, world!'), nl.
```

In order to execute this Prolog program, you can type at the command prompt:

```
java -cp dir/2p.jar alice.tuprolog.Agent hello.pl go.
```

Then, the Agent tool tries to prove the goal `go` with respect to the theory contained in `hello.pl`. As a result, the string `hello, world!` should appear on your standard output.

Also, the goal to be proven can be embedded within the Prolog source by means of the `solve` directive. For instance, suppose that the text file `hellogo.pl` in your current directory contains the following lines:

```
:- solve(go).
go :- write('hello, world!'), nl.
```

Then, type:

```
java -cp dir/2p.jar alice.tuprolog.Agent hellogo.pl
```

Again, this will make `hello, world!` appear on your standard output.

3.2 Developer Quick Start

The first thing you may want to do as a developer would probably be to take advantage of the tools embedded in the Graphical User Interface included in the tuProlog distribution. The GUI can be obtained by issuing the following command:

```
java -cp dir/2p.jar alice.tuprologx.ide.GUILauncher
```

The development environment provided by the GUI makes standard Prolog features easily accessible, such as asking queries, viewing the current solution along with the related variable substitution, backtracking, and so on. Also, it enables you to view and edit the current Prolog theory contained in the engine, and to spy tuProlog at work during goal demonstrations. Finally, it also offers a facility to dynamically load and unload predicate libraries within the tuProlog engine.

It is worth remembering that the file `2p.jar` is an executable Java Archive, so by invoking the command:

```
java -jar 2p.jar
```

in the *dir* directory, or by double-clicking it under most operating systems, the graphic user interface console is automatically spawned.

You may also want to experience a pure interactive environment on a tuProlog engine. In this case, you need to get the tuProlog prompt using the command line shell provided within the distribution. To obtain it, just type:

```
java -cp dir/2p.jar alice.tuprologx.ide.CUIConsole
```

which starts a tuProlog interpreter to be used via console, in a sort of Command Line User Interface mode. To exit the tuProlog console, you have to issue the standard `halt.` command.

Chapter 4

tuProlog Basics

This chapter provides a brief introduction to the basic elements and structure of the tuProlog engine, covering syntax, programming support, and built-in predicates directly provided by the engine.

4.1 Structure of a tuProlog Engine

A tuProlog engine has a layered structure, where provided and recognised predicates are organised into three different categories:

built-in predicates — Predicates embedded in any tuProlog engine are called built-in predicates. Whatever modification is made to the engine either before or during execution time, it does not affect the number and properties of the built-in predicates.

library predicates — Predicates loaded in a tuProlog engine by means of a tuProlog library are called library predicates. Since libraries can be loaded and unloaded in tuProlog engines freely at the system start-up, or dynamically at execution time, the set of the library predicates of a tuProlog engine is not fixed, and can change from engine to engine, and in the same engine at different times. tuProlog libraries can be built by mixing Java and Prolog code. Prolog library predicates can be overridden by Prolog theory predicates. Both Java and Prolog library predicates cannot be individually retracted: if you want to remove a single library predicate from the engine, you need to unload the whole library containing that predicate.

theory predicates — Predicates loaded in a tuProlog engine by means of a tuProlog theory are called theory predicates. Since theories can be

loaded and unloaded in tuProlog engines freely at the system start-up, or dynamically at execution time, the set of the theory predicates of a tuProlog engine is not fixed, and can change from engine to engine, and in the same engine at different times. tuProlog theories are simple collections of Prolog clauses.

Even though they may seem similar, library and theory predicates are handled differently in a tuProlog engine.

First of all, they are conceptually different. In fact, while theory predicates should be used to axiomatically represent domain knowledge at the time the proof is performed, library predicates should more or less be used to represent what is required (procedural knowledge, utility predicates) in order to actually and effectively perform a (number of) proof(s) in the domain of interest: therefore, library predicates represent more "stable" knowledge, which is encapsulated once and for all (at least approximately) within a library container.

Since library and theory predicates are also structurally different, they are handled differently by the engine, and represented differently in the run-time: correspondingly, they have different level of observation when monitoring or debugging a working tuProlog engine. As a consequence, developer tools provided by tuProlog IDE typically show in a separate way the theory axioms or predicates and the loaded libraries or predicates. In addition, the debugging phase typically neglects library predicates (which, as mentioned above, are also conceived as more stable and well-tested), while the effect of the theory predicates is dutifully put in evidence during controlled execution.

4.2 Prolog syntax

The term syntax supported by tuProlog engine is basically ISO compliant,[1] and accounts for several elements:

Comments and Whitespaces – Whitespaces consist of blanks (including tabs and formfeeds), end-of-line marks, and comments. A whitespace can be put before and after any term, operator, bracket, or argument separator, as long as it does not break up an atom or number or separate a functor from the opening parenthesis that introduces its argument lists. For instance, atom `p(a,b,c)` can be written as

[1] Currently ISO exceptions, ISO I/O predicates and some ISO directives are not supported.

`p( a , b , c )`, but not as `p (a,b,c)`). Two types of comments are supported: one type begins with `/*` and ends with `*/`, the other begins with `%` and ends at the end of the line. Nested comments are not allowed.

Variables — A variable name begins with a capital letter or the underscore mark (_), and consists of letters, digits, and/or underscores. A single underscore mark denotes an anonymous variable.

Atoms — There are four types of atoms: *(i)* a series of letters, digit, and/or underscores, beginning with a lower-case letter; *(ii)* a series of one or more characters from the set {`#`, `$`, `&`, `*`, `+`, `-`, `.`, `/`, `:`, `<`, `=`, `>`, `?`, `@`, `^`, `~`}, provided it does not begin with `/*`; *(iii)* The special atoms `[]` and {}; *(iv)* a single-quoted string.

Numbers — Integers and float are supported. The formats supported for integer numbers are decimal, binary (with `0b` prefix), octal (with `0o` prefix), and hexadecimal (with `0x` prefix). The character code format for integer numbers (prefixed by `0'`) is supported only for alphanumeric characters, the white space, and characters in the set {`#`, `$`, `&`, `*`, `+`, `-`, `.`, `/`, `:`, `<`, `=`, `>`, `?`, `@`, `^`, `~`}. The range of integers is -2147483648 to 2147483647; the range of floats is -2E+63 to 2E+63-1. Floating point numbers can be expressed also in the exponential format (e.g. `-3.03E-05`, `0.303E+13`). A minus can be written before any number to make it negative (e.g. `-3.03`). Notice that the minus is the sign-part of the number itself; hence `-3.4` is a number, not an expression (by contrast, `- 3.4` is an expression).

Strings — A series of ASCII characters, embedded in quotes `'` or `"`. Within single quotes, a single quote is written double (e.g, `'don''t forget'`). A backslash at the very end of the line denotes continuation to the next line, so that:

```
'this is \
a single line'
```

is equivalent to `'this is a single line'` (the line break is ignored). Within a string, the backslash can be used to denote special characters, such as `\n` for a newline, `\r` for a return without newline, `\t` for a tab character, `\\` for a backslash, `\'` for a single quote, `\"` for a double quote.

Compounds — The ordinary way to write a compound is to write the functor (as an atom), an opening parenthesis, without spaces between

them, and then a series of terms separated by commas, and a closing parenthesis: `f(a,b,c)`. This notation can be used also for functors that are normally written as operators, e.g. `2+2 = '+'(2,2)`. Lists are defined as rightward-nested structures using the dot operator `'.'`; so, for example:

```
[a] = '.'(a,[])
[a,b] = '.'(a,'.'(b,[]))
[a,b|c] = '.'(a,'.'(b,c))
```

There can be only one `|` in a list, and no commas after it. Also curly brackets are supported: any term enclosed with { and } is treated as the argument of the special functor `'{}'`: `{hotel} = '{}'(hotel)`, `{1,2,3} = '{}'(1,2,3)`. Curly brackets can be used in the Definite Clause Grammars theory.

Operators — Operators are characterised by a name, a specifier, and a priority. An operator name is an atom, which is not univocal: the same atom can be an operator in more than one class, as in the case of the infix and prefix minus signs. An operator specifier is a string like `xfy`, which gives both its class (infix, postfix and prefix) and its associativity: `xfy` specifies that the grouping on the right should be formed first, `yfx` on the left, `xfx` no priority. An operator priority is a non-negative integer ranging from 0 (max priority) and 1200 (min priority).

Operators can be defined by means of either the `op/3` predicate or directive. No predefined operators are directly given by the raw `tu`Prolog engine, whereas a number of them is provided through libraries.

Commas — The comma has three functions: it separates arguments of functors, it separates elements of lists, and it is an infix operator of priority 1000. Thus `(a,b)` (without a functor in front) is a compound, equivalent to `','(a,b)`.

Parenthesis – Parenthesis are allowed around any term. The effect of parenthesis is to override any grouping that may otherwise be imposed by operator priorieties. Operators enclosed in parenthesis do not function as operators; thus `2(+)3` is a syntax error.

4.3 Configuration of a tuProlog Engine

Prolog developers have four different means to configure a tuProlog engine in order to fit their application needs. In fact, a tuProlog can be suitably configured by means of:

Theories — A tuProlog theory is represented by a text, consisting of a sequence of clauses and/or directives. Clauses and directives are terminated by a dot, and are separated by a whitespace character. Theories can be loaded or unloaded by means of suitable library predicates, which are described in Chapter 5.

Directives — A directive can be given by means of the `:-/1` predicate, which is natively supported by the engine, and can be used to configure and use a tuProlog engine (`set_prolog_flag/1`, `load_library/1`, `consult/1`, `solve/1`), format and syntax of read-terms[2] (`op/3`). Directives are described in detail in the following sections.

Flags — A tuProlog engine allows the dynamic definition of flags (or properties) describing some aspects of libraries and their predicates and evaluable functors. A flag is identified by a name (an alphanumeric atom), a list of possible values, a default value, and a boolean value specifying if the flag value can be modified. Dynamically, a flag value can be changed (if modifiable) with a new value included in the list of possible values.

Libraries — A tuProlog engine can be dynamically extended by loading or unloading libraries. Each library can provide a specific set of predicates, functors, and a related theory, which also allows new flags and operators to be defined. Libraries can be either pre-defined (see Chapter 5) or user-defined (see Chapter 9). A library can be loaded by means of the predicate `load_library` (Prolog side), or by means of the method `loadLibrary` of the tuProlog engine (Java side).

Currently tuProlog does not support exception management: actually, an exception causes the predicate/functor in which it occurred to fail and be false.

[2] As specified by the ISO standard, a read-term is a Prolog term followed by an end token, composed by an optional layout text sequence and a dot.

4.4 Built-in predicates

This section contains a comprehensive list of the built-in predicates provided by the tuProlog engine, that is, those predicates defined directly in its core.

Following an established convention in built-in argument template description, which takes root into an imperative interpretation, the symbol `+` in front of an argument means an *input argument*, `-` means *output argument*, `?` means *input/output* argument, `@` means *input argument* that must be bound.

4.4.1 Control management

- `true/0`
  `true` is true.

- `fail/0`
  `fail` is false.

- `','/2`
  `','(First,Second)` is true if and only if both `First` and `Second` are true.

- `!/0`
  `!` is true. All choice points between the cut and the parent goal are removed. The effect is a commitment to use both the current clause and the substitutions found at the point of the cut.

- `'$call'/1`
  `'$call'(Goal)` is true if and only if `Goal` represents a goal which is true. It is not opaque to cut.
  Template: `'$call'(+callable_term)`

- `halt/0`
  `halt` terminates a Prolog demonstration, exiting the Prolog processor and returning to the system that invoked the processor.

- `halt/1`
  `halt(X)` terminates a Prolog demonstration, exiting the Prolog processor and returning to the systems that invoked the processor passing the value of `X` as a message.
  Template: `halt(+int)`

4.4.2 Term Unification and Management

- `is/2`
  `is(X, Y)` is true iff `X` is unifiable with the value of the expression `Y`.
  Template: `is(?term, @evaluable)`

- `'='/2`
  `'='(X, Y)` is true iff `X` and `Y` are unifiable.
  Template: `'='(?term, ?term)`

- `'\='/2`
  `'\='(X, Y)` is true iff `X` and `Y` are not unifiable.
  Template: `'\='(?term, ?term)`

- `'$tolist'/2`
  `'$tolist'(Compound, List)` is true if `Compound` is a compound term, and in this case `List` is list representation of the compound, with the name as first element and all the arguments as other elements.
  Template: `'$tolist'(@struct, -list)`

- `'$fromlist'/2`
  `'$fromlist'(Compound, List)` is true if `Compound` unifies with the list representation of `List`.
  Template: `'$fromlist'(-struct, @list)`

- `copy_term/2`
  `copy_term(Term1, Term2)` is true iff `Term2` unifies with the a renamed copy of `Term1`.
  Template: `copy_term(?term, ?term)`

- `'$append'/2`
  `'$append'(Element, List)` is true if `List` is a list, with the side effect that the `Element` is appended to the list.
  Template: `'$append'(+term, @list)`

4.4.3 Knowledge-base management

- `'$find'/2`
  `'$find'(Clause, ClauseList)` is true if `ClauseList` is a list, and `Clause` is a clause, with the side effect that all the clauses of the database matching `Clause` are appended to the list.
  Template: `'$find'(@clause, @list)`

- `abolish/1`
  `abolish(PI)` completely wipes out the dynamic predicate matching the predicate indicator `PI`.
  Template: `abolish(@term)`

- `asserta/1`
  `asserta(Clause)` is true, with the side effect that the clause `Clause` is added to the beginning of database.
  Template: `asserta(@clause)`

- `assertz/1`
  `assertz(Clause)` is true, with the side effect that the clause `Clause` is added to the end of the database.
  Template: `assertz(@clause)`

- `'$retract'/1`
  `'$retract'(Clause)` is true if the database contains at least one clause unifying with `Clause`. As a side effect, the clause is removed from the database. It is not re-executable.
  Template: `'$retract'(@clause)`

4.4.4 Operators and Flags Management

- `op/3`
  `op(Priority, Specifier, Operator)` is true. It always succeeds, modifying the operator table as a side effect. If `Priority` is 0, then `Operator` is removed from the operator table; else, `Operator` is added to the operator table, with priority (lower binds tighter) `Priority` and associativity determined by `Specifier`. If an operator with the same `Operator` symbol and the same `Specifier` already exists in the operator table, the predicate modifies its priority according to the specified `Priority` argument.
  Template: `op(+integer, +specifier, @atom_or_atom_list)`

- `flag_list/1`
  `flag_list(FlagList)` is true and `FlagList` is the list of the flags currently defined in the engine.
  Template: `flag_list(-list)`

- `set_prolog_flag/2`
  `set_prolog_flag(Flag, Value)` is true, and as a side effect associates `Value` with the flag `Flag`, where `Value` is a value that is within the

implementation defined range of values for `Flag`.
Template: `set_prolog_flag(+flag, @nonvar)`

- `get_prolog_flag/2`
  `get_prolog_flag(Flag, Value)` is true iff `Flag` is a flag supported by the engine and `Value` is the value currently associated with it. Note that `get_prolog_flag/2` is not re-executable.
  Template: `get_prolog_flag(+flag, ?term)`

4.4.5 Libraries Management

- `load_library/1`
  `load_library(LibraryName)` is true if `LibraryName` is the name of a tuProlog library available for loading. As side effect, the specified library is loaded by the engine. Actually `LibraryName` is the full name of the Java class providing the library.
  Template: `load_library(@string)`

- `unload_library/1`
  `unload_library(LibraryName)` is true if `LibraryName` is the name of a library currently loaded in the engine. As side effect, the library is unloaded from the engine. Actually `LibraryName` is the full name of the Java class providing the library.
  Template: `unload_library(@string)`

4.4.6 Directives

Directives are used in Prolog text only as queries to be immediately executed when loading it. When a corresponding predicate with the same procedure name as a directive exists, they perform the same actions. Their arguments will satisfy the same constraints as those required for an errorless execution of the corresponding predicate, otherwise their behaviour is undefined.

In tuProlog, directives are not composable: each query must contain one and only one directive. When you need to use multiple directives, you must employ multiple queries as well.

- `:- op/3`
  `op(Priority, Specifier, Operator)` adds `Operator` to the operator table, with priority (lower binds tighter) `Priority` and associativity determined by `Specifier`.
  Template: `op(+integer, +specifier, @atom_or_atom_list)`

- `:- flag/4`
  `flag(FlagName, ValidValuesList, DefaultValue, IsModifiable)` adds to the engine a new flag, identified by the `FlagName` name, which can assume only the values listed in `ValidValuesList` with `DefaultValue` as default value, and that can be modified if `IsModifiable` is true.
  Template: `flag(@string, @list, @term, @true, false)`

- `:- initialization/1`
  `initialization(Goal)` sets the starting goal to be executed just after the theory has been consulted.
  Template: `initialization(@goal)`

- `:- solve/1`
  Synonym for `initialization/1`. *Deprecated.*
  Template: `solve(@goal)`

- `:- load_library/1`
  `load_library(LibraryName)` is a valid directive if true if `LibraryName` is the name of a tuProlog library available for loading. This directive loads the specified library in the engine. Actually `LibraryName` is the full name of the Java class providing the library.
  Template: `load_library(@string)`

- `:- include/1`
  `include(Filename)` immediately loads the theory contained in the file specified by `Filename`.
  Template: `include(@string)`

- `:- consult/1`
  Synonym for `include/1`. *Deprecated.*
  Template: `consult(@string)`

Chapter 5

tuProlog Libraries

Libraries are the means by which tuProlog achieves its fundamental characteristics of minimality and configurability. The engine is by design choice a minimal, purely-inferential core: as such, it only includes a few *built-in* predicates, intended as predicates statically defined inside the core, to establish the foundation which the mechanisms of the engine are based on. Instead, each and every other piece of functionality, in the form of predicates, functors, flags and operators, is delivered by libraries, and can be added to or subtracted from the engine at any time. Thus, a tuProlog engine can be dynamically extended by loading (and unloading) any number of libraries. Each library can provide a specific set of predicates, functors and a related theory, which can be used to define new flags and operators. Besides built-in and library predicates, new functionalities can also be added to an engine by feeding it with a user-defined Prolog theory.

Libraries can be loaded at any time in the tuProlog engine, both from the Java side, by means of the `loadLibrary` method of the `Prolog` object representing a tuProlog engine, and from the Prolog side, using the `load_library/1` predicate. For example, suppose you want to exploit some features defined in a library whose name is `ExampleLibrary`. If, on the Java side, you want to load the library immediately afterwards building a tuProlog engine, you would write the following code, using the fully qualified Java class name for the library:

```
Prolog engine = new Prolog();
try {
    engine.loadLibrary("com.example.ExampleLibrary");
} catch (InvalidLibraryException e) {
}
```

If, on the other hand, you just want to load the library on the Prolog side for those clauses which actually make use of its predicates, you would write the following code, using just the name of the library, which can be different from its fully qualified class name:

```
% println/1 is defined in ExampleLibrary
run_test(Test, Result) :- run(Test, Result),
                          load_library('ExampleLibrary'),
                          println(Result).
```

Correspondingly, means for unloading libraries are provided, in the form of the `unloadLibrary` method of the `Prolog` class on the Java side, and the `unload_library/1` predicate on the Prolog side. It must be noted that predicates for loading or unloading libraries are also available in the form of directives: they perform the same actions, but as directives they are immediately executed when the Prolog text containing them is feeded to the engine.

Since the core comes as a pure inferential engine, tuProlog includes in its distribution some standard libraries which are loaded by default into the engine at construction time. While it is possible to create an engine with no default libraries preloaded, those standard libraries provide the fundamental bricks of a Prolog engine, in the form of basic functionalities, ISO compliant predicates and evaluable functors, I/O predicates and predicates for interoperability and integration between Java and Prolog. More user-defined libraries can be then loaded or unloaded, thus exploiting the dynamic configurability of tuProlog engines which can be reconfigured on the fly enriching or reducing the set of available functionalities by need.

The standard libraries are:

BasicLibrary (class `alice.tuprolog.lib.BasicLibrary`) — provides common Prolog predicates and functors, and operators. No I/O predicates are included.

DCGLibrary (class `alice.tuprolog.lib.DCGLibrary`) — provides support for Definite Clause Grammar, an extension of context free grammars used for describing natural and formal languages.

IOLibrary (class `alice.tuprolog.lib.IOLibrary`) — provides some basic and classic I/O predicates.

ISOLibrary (class `alice.tuprolog.lib.ISOLibrary`) — provides predicates and functors that are part of the built-in section in the ISO standard [**?**], and are not provided by previous libraries.

JavaLibrary (class `alice.tuprolog.lib.JavaLibrary`) — provides predicates and functors to create, access and deploy (existent or new) Java resources, like objects and classes.

The description of each library is provided by discussing in the order: predicates, functors, operators and flags defined by the library. For each library the dependencies with other libraries are specified: that is, which other libraries are required in order to provide the correct computational behaviour.

5.1 BasicLibrary

Library Dependencies: none.

This library provides common Prolog built-in predicates, functors, and operators. No I/O predicates are included.

Please note that in the following `string` means a single or double quoted string, as detailed in Chapter 4; `expr` means an evaluable expression, that is a term that can be interpreted as a value by some library functors.

5.1.1 Predicates

Here follows a list of predicates implemented by this library, grouped by category.

Type Testing

- `constant/1`
  `constant(X)` is true iff `X` is a constant value.
  Template: `constant(@term)`

- `number/1`
  `number(X)` is true iff `X` is an integer or a float.
  Template: `number(@term)`

- `integer/1`
  `integer(X)` is true iff `X` is an integer.
  Template: `integer(@term)`

- `float/1`
  `float(X)` is true iff `X` is an float.
  Template: `float(@term)`

- `atom/1`
  `atom(X)` is true iff `X` is an atom.
  Template: `atom(@term)`

- `compound/1`
  `compound(X)` is true iff `X` is a compound term, that is neither atomic nor a variable.
  Template: `compound(@term)`

- `var/1`
  `var(X)` is true iff `X` is a variable.
  Template: `var(@term)`

- `nonvar/1`
  `nonvar(X)` is true iff `X` is not a variable.
  Template: `nonvar(@term)`

- `atomic/1`
  `atomic(X)` is true iff `X` is atomic (that is is an atom, an integer or a float).
  Template: `atomic(@term)`

- `ground/1`
  `ground(X)` is true iff `X` is a ground term.
  Template: `ground(@term)`

- `list/1`
  `list(X)` is true iff `X` is a list.
  Template: `list(@term)`

Term Creation, Decomposition and Unification

- `'=..'/2` : *univ*
  `'=..'(Term, List)` is true if `List` is a list consisting of the functor and all arguments of `Term`, in order.
  Template: `'=..'(?term, ?list)`

- `functor/3`
  `functor(Term, Functor, Arity)` is true if the term `Term` is a compound term, `Functor` is its functor, and `Arity` (an integer) is its arity; or if `Term` is an atom or number equal to `Functor` and `Arity` is 0.
  Template: `functor(?term, ?term, ?integer)`

- `arg/3`
  `arg(N, Term, Arg)` is true if `Arg` is the `N`th arguments of `Term` (counting from 1).
  Template: `arg(@integer, @compound, -term)`

- `text_term/2`
  `text_term(Text, Term)` is true iff `Text` is the text representation of the term `Term`.
  Template: `text_term(?text, ?term)`

- `text_concat/3`
  `text_concat(TextSource1, TextSource2, TextDest)` is true iff `TextDest` is the text resulting by appending the text `TestSource2` to `TextSource1`.
  Template: `text_concat(@string, @string, -string)`

- `num_atom/2`
  `num_atom(Number, Atom)` succeeds iff `Atom` is the atom representation of the number `Number`
  Template: `number_codes(+number, ?atom)`
  Template: `number_codes(?number, +atom)`

Occurs Check

When the process of unification takes place between a variable S and a term T, the first thing a Prolog engine should do before proceeding is to check that T does not contain any occurences of S. This test is known as *occurs check* [**?**] and is necessary to prevent the unification of terms such as $s(X)$ and X, for which no finite common instance exists. Most Prolog implementations omit the occurs check from their unification algorithm for reasons related to speed and efficiency: tuProlog is no exception. However, they provide a predicate for occurs check augmented unification, to be used when the programmer wants to never incur on an error or an undefined result during the process.

- `unify_with_occurs_check/2`
  `unify_with_occurs_check(X, Y)` is true iff `X` and `Y` are unifiable.
  Template: `unify_with_occurs_check(?term, ?term)`

Expression and Term Comparison

- expression comparison (generic template: *pred*(@expr, @expr)):
  `'=:='`, `'=\='`, `'>'`, `'<'`, `'>='`, `'=<'`;

- term comparison (generic template: *pred*(@term, @term)):
  `'=='`, `'\=='`, `'@>'`, `'@<'`, `'@>='`, `'@=<'`.

Finding Solutions

- `findall/3`
  `findall(Template, Goal, List)` is true if and only if `List` unifies with the list of values to which a variable X not occurring in `Template` or `Goal` would be instantiated by successive re-executions of
  `call(Goal), X = Template`
  after systematic replacement of all variables in X by new variables.
  Template: `findall(?term, +callable_term, ?list)`

- `bagof/3`
  `bagof(Template, Goal, Instances)` is true if `Instances` is a non-empty list of all terms such that each unifies with `Template` for a fixed instance W of the variables of `Goal` that are free with respect to `Template`. The ordering of the elements of `Instances` is the order in which the solutions are found.
  Template: `bagof(?term, +callable_term, ?list)`

- `setof/3`
  `setof(Template, Goal, List)` is true if `List` is a sorted non-empty list of all terms that each unifies with `Template` for a fixed instance W of the variables of `Goal` that are free with respect to `Template`.
  Template: `setof(?term, +callable_term, ?list)`

Control Management

- `(->)/2` : *if-then*
  `'->'(If, Then)` is true if and only if `If` is true and `Then` is true for the first solution of `If`.

- `(;)/2` : *if-then-else*
  `';'(Either, Or)` is true iff either `Either` or `Or` is true.

- `call/1`
  `call(Goal)` is true if and only if `Goal` represents a goal which is true. It is opaque to cut.
  Template: `call(+callable_term)`

- `once/1`
  `once(Goal)` finds exactly one solution to `Goal`. It is equivalent to `call((Goal, !))` and is opaque to cuts.
  Template: `once(@goal)`

- `repeat/0`
  Whenever backtracking reaches `repeat`, execution proceeds forward again through the same clauses as if another alternative has been found.
  Template: `repeat`

- `'\+'/1` : *not provable*
  `'\+'(Goal)` is the negation predicate and is opaque to cuts. That is, `'\+'(Goal)` is like `call(Goal)` except that its success or failure is the opposite.
  Template: `'\+'(@goal)`

- `not/1`
  The predicate `not/1` has the same semantics and implementation as the predicate `'\+'/1`.
  Template: `not(@goal)`

Clause Retrival, Creation and Destruction

Every Prolog engine lets programmers modify its logic database during execution by adding or deleting specific clauses. The ISO standard [**?**] distinguishes between static and dynamic predicates: only the latter can be modified by asserting or retracting clauses. While typically the *dynamic/1* directive is used to indicate whenever a user-defined predicate is dynamically modifiable, tuProlog engines work differently, establishing two default behaviors: library predicates are always of a static kind; every other user-defined predicate is dynamic and modifiable at runtime. The following list contains library predicates used to manipulate the knowledge base of a tuProlog engine during execution.

- `clause/2`
  `clause(Head, Body)` is true iff `Head` matches the head of a dynamic predicate, and `Body` matches its body. The body of a fact is considered to be `true`. `Head` must be at least partly instantiated.
  Template: `clause(@term, -term)`

- `assert/1`
  `assert(Clause)` is true and adds `Clause` to the end of the database.
  Template: `assert(@term)`

- `retract/1`
  `retract(Clause)` removes from the knowledge base a dynamic clause that matches `Clause` (which must be at least partially instantiated). Gives multiple solutions upon backtracking.
  Template: `retract(@term)`

- `retractall/1`
  `retractall(Clause)` removes from the knowledge base all the dynamic clauses matching with `Clause` (which must be at least partially instantiated).
  Template: `retractall(@term)`

Operator Management

- `current_op/3`
  `current_op(Priority, Type, Name)` is true iff `Priority` is an integer in the range [0, 1200], `Type` is one of the `fx`, `xfy`, `yfx`, `xfx` values and `Name` is an atom, and as side effect it adds a new operator to the engine operator list.
  Template: `current_op(?integer, ?term, ?atom)`

Flag Management

- `current_prolog_flag/3`
  `current_prolog_flag(Flag,Value)` is true if the value of the flag `Flag` is `Value`
  Template: `current_prolog_flag(?atom,?term)`

Actions on Theories and Engines

- `set_theory/1`
  `set_theory(TheoryText)` is true iff `TheoryText` is the text representation of a valid tuProlog theory, with the side effect of setting it as the new theory of the engine.
  Template: `set_theory(@string)`

- `add_theory/1`
  `add_theory(TheoryText)` is true iff `TheoryText` is the text represen-

tation of a valid tuProlog theory, with the side effect of appending it to the current theory of the engine.
Template: `add_theory(@string)`

- `get_theory/1`
  `get_theory(TheoryText)` is true, and `TheoryText` is the text representation of the current theory of the engine.
  Template: `get_theory(-string)`

- `agent/1`
  `agent(TheoryText)` is true, and spawns a tuProlog agent with the knowledge base provided as a Prolog textual form in `TheoryText` (the goal is described in the knowledge base).
  Template: `agent(@string)`

- `agent/2`
  `agent(TheoryText, Goal)` is true, and spawn a tuProlog agent with the knowledge base provided as a Prolog textual form in `TheoryText`, and solving the query `Goal` as a goal.
  Template: `agent(@string, @term)`

Spy Events

During each demonstration, the engine notifies to interested listeners so-called *spy events*, containing informations on its internal state, such as the current subgoal being evaluated, the configuration of the execution stack and the available choice points. The different kinds of spy events currently corresponds to the different states which the virtual machine realizing the tuProlog's inferential core can be found into. *Init* events are spawned whenever the machine initialize a subgoal for execution; *Call* events are generated when a choice must be made for the next subgoal to be executed; *Eval* events represent actual subgoal evaluation; finally, *Back* events are notified when a backtracking occurs during the demonstration process.

- `spy/0`
  `spy` is true and enables the notification of spy events occurring inside the engine.
  Template: `spy`

- `nospy/0`
  `nospy` is true and disables the notification of the spy events inside the

engine.
Template: `nospy`

Auxiliary predicates

The following predicates are provided by the library's theory.

- `member/2`
  `member(Element, List)` is true iff `Element` is an element of the list `List`
  Template: `member(?term, +list)`

- `length/2`
  `length(List, NumberOfElements)` is true in three different cases: (1) if `List` is instantiated to a list of determinate length, then `Length` will be unified with this length; (2) if `List` is of indeterminate length and `Length` is instantiated to an integer, then `List` will be unified with a list of length `Length` and in such a case the list elements are unique variables; (3) if `Length` is unbound then `Length` will be unified with all possible lengths of `List`.
  Template: `member(?list, ?integer)`

- `append/3`
  `append(What, To, Target)` is true iff `Target` list can be obtained by appending the `To` list to the `What` list
  Template: `append(?list, ?list, ?list)`

- `reverse/2`
  `reverse(List, ReversedList)` is true iff `ReversedList` is the reverse list of `List`
  Template: `reverse(+list, -list)`

- `delete/3`
  `delete(Element, ListSource, ListDest)` is true iff `ListDest` list can be obtained by removing the element `Element` from the list `ListSource`.
  Template: `delete(@term, +list, -list)`

- `element/3`
  `element(Position, List, Element)` is true iff `Element` is the `Position`th element of the list `List` (starting the count from 1).
  Template: `element(@integer, +list, -term)`

- `quicksort/3`
  `quicksort(List, ComparisonPredicate, SortedList)` is true iff `SortedList` is the list `List` sorted by the comparison predicate `ComparisonPredicate`.
  Template: `element(@list, @pred, -list)`

5.1.2 Functors

Functors for expression evaluation (with usual semantics):

- unary: `+, -, ~, +`
- binary: `+, -, *, \, **, <<, >>, /\, \/`

5.1.3 Operators

Name	Assoc.	Prio.
:-	fx	1200
:-	xfx	1200
?-	fx	1200
;	xfy	1100
->	xfy	1050
,	xfy	1000
not	fy	900
\+	fy	900
=	xfx	700
\=	xfx	700
==	xfx	700
\==	xfx	700
@>	xfx	700
@<	xfx	700
@=<	xfx	700
@>=	xfx	700
=:=	xfx	700
=\=	xfx	700
>	xfx	700
<	xfx	700
>=	xfx	700
=<	xfx	700
is	xfx	700
=..	xfx	700
+	yfx	500
-	yfx	500
/\	yfx	500
\/	yfx	500
*	yfx	400
/	yfx	400
//	yfx	400
>>	yfx	400
<<	yfx	400
>>	yfx	400
**	xfx	200
^	xfy	200
\\	fx	200
-	fy	200

5.2 ISOLibrary

Library Dependencies: BasicLibrary.

This library contains almost[1] all the built-in predicates and functors that are part of the ISO standard and that are not part directly of the tuProlog core engine or other core libraries. Moreover, some features are added, not currently ISO, such as the support for definite clause grammars (DCGs).

5.2.1 Predicates

Here follows a list of predicates implemented by this library, grouped by category.

Type Testing

- `bound/1`
  `bound(Term)` is a synonym for the `ground/1` predicate defined in BasicLibrary.
  Template: `bound(+term)`

- `unbound/1`
  `unbound(Term)` is true iff `Term` is not a ground term.
  Template: `unbound(+term)`

Atoms Processing

- `atom_length/2`
  `atom_length(Atom, Length)` is true iff the integer `Length` equals the number of characters in the name of atom `Atom`.
  Template: `atom_length(+atom, ?integer)`

- `atom_concat/3`
  `atom_concat(Start, End, Whole)` is true iff the `Whole` is the atom obtained by concatenating the characters of `End` to those of `Start`. If `Whole` is instantiated, then all decompositions of `Whole` can be obtained by backtracking.
  Template: `atom_concat(?atom, ?atom, +atom)`
  Template: `atom_concat(+atom, +atom, -atom)`

[1] Currently ISO exceptions, ISO I/O predicates and some ISO directives are not supported.

- `sub_atom/5`
  `sub_atom(Atom, Before, Length, After, SubAtom)` is true iff `SubAtom` is the sub atom of `Atom` of length `Length` that appears with `Before` characters preceding it and `After` characters following. It is re-executable.
  Template: `sub_atom(+atom, ?integer, ?integer, ?integer, ?atom)`

- `atom_chars/2`
  `atom_chars(Atom,List)` succeeds iff `List` is a list whose elements are the one character atoms that in order make up `Atom`.
  Template: `atom_chars(+atom, ?character_list)`
  Template: `atom_chars(-atom, ?character_list)`

- `atom_codes/2`
  `atom_codes(Atom, List)` succeeds iff `List` is a list whose elements are the character codes that in order correspond to the characters that make up `Atom`.
  Template: `atom_codes(+atom, ?character_code_list)`
  Template: `atom_chars(-atom, ?character_code_list)`

- `char_code/2`
  `char_code(Char, Code)` succeeds iff `Code` is a the character code that corresponds to the character `Char`.
  Template: `char_code(+character, ?character_code)`
  Template: `char_code(-character, +character_code)`

- `number_chars/2`
  `number_chars(Number, List)` succeeds iff `List` is a list whose elements are the one character atoms that in order make up `Number`.
  Template: `number_chars(+number, ?character_list)`
  Template: `number_chars(-number, ?character_list)`

- `number_codes/2`
  `number_codes(Number, List)` succeeds iff `List` is a list whose elements are the codes for the one character atoms that in order make up `Number`.
  Template: `number_codes(+number,?character_code_list)`
  Template: `number_codes(-number,?character_code_list)`

5.2.2 Functors

- Trigonometric functions: `sin(+expr)`, `cos(+expr)`, `atan(+expr)`.

- Logarithmic functions: `exp(+expr)`, `log(+expr)`, `sqrt(+expr)`.
- Absolute value functions: `abs(+expr)`, `sign(+Expr)`.
- Rounding functions: `floor(+expr)`, `ceiling(+expr)`, `round(+expr)`, `truncate(+expr)`, `float(+expr)`, `float_integer_part(+expr)`, `float_fractional_part(+expr)`.
- Integer division functions: `div(+expr, +expr)`, `mod(+expr, +expr)`, `rem(+expr, +expr)`.

5.2.3 Operators

Name	Assoc.	Prio.
mod	yfx	400
div	yfx	300
rem	yfx	300
sin	fx	200
cos	fx	200
sqrt	fx	200
atan	fx	200
exp	fx	200
log	fx	200

5.2.4 Flags

Flag Name	Possible Values	Default Value
bounded	true	true
max_integer	2147483647	2147483647
min_integer	-2147483648	-2147483648
integer_rounding_function	down	down
char_conversion	off	off
debug	off	off
max_arity	2147483647	2147483647
undefined_predicates	fail	fail
double_quotes	atom	atom

5.3 DCGLibrary

Library Dependencies: BasicLibrary.

This library provides support for Definite Clause Grammar [**?**], also known as DCG,[2] an extension of context free grammars that have proven useful for describing natural and formal languages, and that may be conveniently expressed and executed in Prolog. Note that this library is not loaded by default when a tuProlog engine is created.

A Definite Clause Grammar rule has the general form:

```
Head --> Body
```

with the declarative interpretation that a possible form for `Head` is `Body`. A non-terminal symbol may be any term other than a variable or a number. A terminal symbol may be any term. In order to distinguish terminals from nonterminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list, with the empty sequence written as the empty list `[]`. The body can contain also executable blocks – interpreted according to normal Prolog rule – enclosed by the `{` and `}` parenthesis. A simple example of DCG follows:

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> verb, noun_phrase.
noun_phrase --> [charles].
noun_phrase --> [linda].
verb --> [loves].
```

So, you can verify that a phrase is correct according to the grammar simply by the query:

```
?- phrase(sentence, [charles, loves, linda]).
```

But also:

```
?- phrase(sentence, [Who, loves, linda]).
```

which would give, according to the grammar, two solutions, `Who` bound to `charles`, and `Who` bound to `linda`.

[2]The DCG formalism is not defined as an ISO standard at the time of writing this document.

5.3.1 Predicates

The classic built-in predicates provided for parsing DCG sentences are:

- `phrase/2`
  `phrase(Category, List)` is true iff the list `List` can be parsed as a phrase (i.e. sequence of terminals) of type `Category`. `Category` can be any term which would be accepted as a nonterminal of the grammar (or in general, it can be any grammar rule body), and must be instantiated to a non-variable term at the time of the call. This predicate is the usual way to commence execution of grammar rules. If `List` is bound to a list of terminals by the time of the call, then the goal corresponds to parsing `List` as a phrase of type `Category`; otherwise if `List` is unbound, then the grammar is being used for generation.
  Template: `phrase(+term, ?list)`

- `phrase/3`
  `phrase(Category, List, Rest)` is true iff the segment between the start of list `List` and the start of list `Rest` can be parsed as a phrase (i.e. sequence of terminals) of type `Category`. In other words, if the search for phrase Phrase is started at the beginning of list `List`, then `Rest` is what remains unparsed after `Category` has been found. Again, `Category` can be any term which would be accepted as a nonterminal of the grammar (or in general, any grammar rule body), and must be instantiated to a non variable term at the time of the call.
  Template: `phrase(+term, ?list, ?rest)`

5.3.2 Operators

Name	Assoc.	Prio.
`-->`	`xfx`	1200

5.4 IOLibrary

Library Dependencies: BasicLibrary.

The IOLibrary defines classic Prolog built-ins predicates to enable interaction between Prolog programs and external resources, typically files and I/O channels.

5.4.1 Predicates

Here follows a list of predicates implemented by this library, grouped by category.

General I/O

- `see/1`
  `see(StreamName)` is used to create/open an input stream; the predicate is true iff `StreamName` is a string representing the name of a file to be created or accessed as input stream, or the string `stdin` selecting current standard input as input stream.
  Template: `see(@atom)`

- `seen/0`
  `seen` is used to close the input stream previously opened; the predicate is true iff the closing action is possible.
  Template: `seen`

- `seeing/1`
  `seeing(StreamName)` is true iff `StreamName` is the name of the stream currently used as input stream.
  Template: `seeing(?term)`

- `tell/1`
  `tell(StreamName)` is used to create/open an output stream; the predicate is true iff `StreamName` is a string representing the name of a file to be created or accessed as output stream, or the string `stdout` selecting current standard output as output stream.
  Template: `tell(@atom)`

- `told/0`
  `told` is used to close the output stream previously opened; the predicate is true iff the closing action is possible.
  Template: `told`

- `telling/1`
  `telling(StreamName)` is true iff `StreamName` is the name of the stream

currently used as input stream.
Template: `telling(?term)`

- `put/1`
  `put(Char)` puts the character `Char` on current output stream; it is true iff the operation is possible.
  Template: `put(@char)`

- `get0/1`
  `get0(Value)` is true iff `Value` is the next character (whose code can span on the entire ASCII codes) available from the input stream, or -1 if no characters are available; as a side effect the character is removed from the input stream.
  Template: `get0(?charOrMinusOne)`

- `get/1`
  `get(Value)` is true iff `Value` is the next character (whose code can span on the range 32..255 as ASCII codes) available from the input stream, or -1 if no characters are available; as a side effect the character (with all the characters that precede this one not in the range 32..255) is removed from the input stream.
  Template: `get(?charOrMinusOne)`

- `tab/1`
  `tab(NumSpaces)` inserts `NumSpaces` space characters (ASCII code 32) on output stream; the predicate is true iff the operation is possible.
  Template: `tab(+integer)`

- `read/1`
  `read(Term)` is true iff `Term` is Prolog term available from the input stream. The term must ends with the `.` character; if no valid terms are available, the predicate fails. As a side effect, the term is removed from the input stream.
  Template: `read(?term)`

- `write/1`
  `write(Term)` writes the term `Term` on current output stream. The predicate fails if the operation is not possible.
  Template: `write(@term)`

- `print/1`
  `print(Term)` writes the term `Term` on current output stream, removing

apices if the term is an atom representing a string. The predicate fails if the operation is not possible.
Template: `print(@term)`

- `nl/0`
`nl` writes a new line control character on current output stream. The predicate fails if the operation is not possible.
Template: `nl`

I/O and Theories Helpers

- `text_from_file/2`
`text_from_file(File, Text)` is true iff `Text` is the text contained in the file whose name is `File`.
Template: `text_from_file(+string, -string)`

- `agent_file/1`
`agent_file(TheoryFileName)` is true iff `TheoryFileName` is an accessible file containing a Prolog knowledge base, and as a side effect it spawns a tuProlog agent provided with that knowledge base.
Template: `agent_file(+string)`

- `solve_file/2`
`solve_file(TheoryFileName, Goal)` is true iff `TheoryFileName` is an accessible file containing a Prolog knowledge base, and as a side effect it solves the query `Goal` according to that knowledge base.
Template: `solve_file(+string, +goal)`

- `consult/1`
`consult(TheoryFileName)` is true iff `TheoryFileName` is an accessible file containing a Prolog knowledge base, and as a side effect it consult that knowledge base, by adding it to current knowledge base.
Template: `consult(+string)`

Random Generation of Numbers

The random generation of number can be regarded as a form of I/O.

- `rand_float/1`
`rand_float(RandomFloat)` is true iff `RandomFloat` is a float random number generated by the engine between 0 and 1.
Template: `rand_float(?float)`

- `rand_int/2`
  `rand_int(Seed, RandomInteger)` is true iff `RandomInteger` is an integer random number generated by the engine between 0 and `Seed`.
  Template: `rand_int(?integer, @integer)`

Chapter 6

Accessing Java from tuProlog

One of the main advantages of tuProlog open architecture is that any Java component can be directly accessed and used from Prolog, in a simple and effective way, by means of the `JavaLibrary` library: this delivers all the power of existing Java components and packages to tuProlog sources. In this way, all Java packages involving interaction (such as Swing, JDBC, the socket package, RMI) are immediately available to increase the interaction abilities of tuProlog: "one library for all libraries" is the basic motto.

6.1 Mapping data structures

Complete bi-directional mapping is provided between Java primitive types and tuProlog data types. By default, tuProlog integers are mapped into Java `int` or `long` as appropriate, while `byte` and `short` types are mapped into tuProlog's `Int` instances. Only Java `double` numbers are used to map tuProlog reals, but `float` values returned as result of method invocations or field accesses are handled properly anyway, without any loss of information. Boolean Java values are mapped into specific tuProlog `Term` constants. Java `char`s are mapped into Prolog atoms, but atoms are mapped into Java `String`s by default. The *any* variable (_) can be used to specify the Java `null` value.

6.2 General predicates description

The library offers the following predicates:

(i) the `java_object/3` predicate is used to create a new Java object of the specified class, according to the syntax:

Figure 6.1: A sample Java class (a counter) used to explain JavaLibrary predicates behaviour.

```
public class Counter {
    public String name;
    private long value = 0;

    public Counter() {}
    public Counter(String aName) { name = aName; }

    public void setValue(long val) { value=val; }
    public long getValue() { return value; }
    public void inc() { value++; }

    static public String getVersion() { return "1.0"; }
}
```

java_object(*ClassName*, *ArgumentList*, *ObjectRef*)

ClassName is a Prolog atom bound to the name of the proper Java class (e.g. `'Counter'`, `'java.io.FileInputStream'`), while the parameter *ArgumentList* is a Prolog list used to supply the required arguments to the class constructor: the empty list matches the default constructor. Also Java arrays can be instantiated, by appending `[]` at the end of the *ClassName* string. The reference to the newly-created object is bound to *ObjectRef*, which is typically a ground Prolog term; alternatively, an unbound term may be used, in which case the term is bound to an automatically-generated Prolog atom `'$obj_N'`, where `N` is a progressive integer. In both cases, these atoms are interpreted as object references – and therefore used to operate on the Java object from Prolog – *only* in the context of `JavaLibrary`'s predicates. The predicate fails whenever *ClassName* does not identify a valid Java class, or the constructor does not exists, or arguments in *ArgumentList* are not ground, or *ObjectRef* already identifies an object in the system.

According to the default behaviour of `java_object`, when a ground term is bound to a Java object by means of the predicate, the binding is kept for the full time of the demonstration (even in the case of

backtracking). This behaviour can be changed, getting the bindings created by the `java_object` undone by backtracking, by changing the value of the flag `java_object_backtrackable` to `true` (the default is `false`).

(ii) the `<-/2` predicate is used to invoke a method on a Java object according to a send-message pattern:

```
ObjectRef <- MethodName(Arguments)
ObjectRef <- MethodName(Arguments) returns Term
```

`ObjectRef` is an atom interpreted as a Java object reference as explained above, while *`MethodName`* is the Java name of the method to be invoked, along with its *`Arguments`*. The `returns` keyword is used to retrieve the value returned from non-void Java methods and bind it to a Prolog term: if the type of the returned value can be mapped onto a primitive Prolog data type (a number or a string), *`Term`* is unified with the corresponding Prolog value; if, instead, it is a Java object other than the ones above, *`Term`* is handled as *`ObjectRef`* in the case of `java_object/3`. Static methods can be invoked using the compound term `class(`*`ClassName`*`)` in the place of *`ObjectRef`*. If *MethodName* does not identify a valid method for the object (class), or arguments in *`ArgumentList`* are not valid (because of a wrong signature or not ground values) the predicate fails.

(iii) the `.` infix operator is used, in conjunction with the `set` / `get` pseudo-method pair, to access the public fields of a Java object. The syntax is based on the following constructs:

```
ObjectRef . Field <- set(GroundTerm)
ObjectRef . Field <- get(Term)
```

As usual, *`ObjectRef`* is the Prolog identifier for a Java object. The first construct set the public field *`Field`* to the specified *`GroundTerm`*, which may be either a value of a primitive data type, or a reference to an existing object: if *`GroundTerm`* is not ground, the infix predicate fails. The second construct retrieves the value of the public field *`Field`*, where *`Term`* is handled once again as *`ObjectRef`* in the case of `java_object/3`. As for methods, static fields of classes can be accessed using the compound term `class(`*`ClassName`*`)` in the place of

ObjectRef. Some helper predicates are provided to access Java array elements:
`java_array_set(`*ArrayRef*`,` *Index*`,` *Object*`)`
`java_array_set_Basic Type(`*ArrayRef*`,` *Index*`,` *Value*`)`
to set elements,
`java_array_get(`*ArrayRef*`,` *Index*`,` *Object*`)`
`java_array_get_Basic Type(`*ArrayRef*`,` *Index*`,` *Value*`)`
to get elements,
`java_array_length(`*ArrayObject*`,` *Size*`)` to get the array length.
It is worth to point out that the `set` and `get` formal pseudo-methods above are *not* methods of some class, but just part of the construct of the `.` infix operator, according to a JavaBeans-like approach.

(iv) the `as` infix operator is used to explicitly specify (i.e., cast) method argument types:

ObjectRef `as` *ClassName*

By writing so, the object represented by *ObjectRef* is considered to belong to class *Classname*: both *ObjectRef* and *Classname* have the usual meaning explained above. The operator works also with primitive Java types, specified as *Classname* (for instance, `myNumber` *`as int`*). The purpose of this predicate is both to provide methods with the exact Java types required, and to solve possible overloading conflicts a-priori.

(v) The `java_class/4` predicate makes it possible to create and load a new Java class from a source text provided as an argument, thus supporting *dynamic compilation* of Java classes:

`java_class(`*SourceText*`,` *FullClassName*`,` *ClassPathList*`,`
ObjectRef`)`

SourceText is a string representing the text source of the Java class, *FullClassName* is the full Java class name, and *ClassPathList* is a (possibly empty) Prolog list of class paths that may be required for a successful dynamic compilation of this class. *ObjectRef* is a reference to an instance of the class `java.lang.Class` that represents the newly-created class. The predicate fails whenever *SourceText* contains errors, or the class cannot be located in the package hierarchy as specified, or *ObjectRef* already identifies an object in the system.

Generally, exceptions thrown by method or constructor calls cannot be explicitly managed and cause the failure of the related predicate.

To taste the flavour of `JavaLibrary`, let us consider the example below (refer to Figure 6.1 for `Counter` class definition):

```
?-  java_object('Counter', ['MyCounter'], myCounter),
    myCounter <- setValue(5),
    myCounter <- inc,
    myCounter <- getValue returns Value,
    write(X),

    class('Counter') <- getVersion return Version,

    myCounter.name <- get(Name),
    class('java.lang.System') . out <- get(Out),
    Out <- println(Name),

    myCounter.name <- set('MyCounter2'),

    java_object('Counter[]', [10], ArrayCounters),
    java_array_set(ArrayCounters, 0, myCounter).
```

Here, a `Counter` object is created providing the `MyCounter` name as constructor argument: the reference to the new object is bound to the Prolog atom `myCounter`. This reference is then used for method invocation via the `<-` operator, calling the `setValue(5)` method (which is void and therefore returns nothing) first, incrementing the counter (no arguments are specified) and invoking the `getValue` method just after. Since `getValue` returns an integer value, the `returns` operator retrieves the method result (hopefully, 5) and binds it to the `X` Prolog variable, which is printed via the Prolog `write/1` predicate. Of course, if the Prolog variable `X` is already bound to 5, the predicate succeeds as well, while fails if `X` is bound to anything else. Then, the static method `getVersion` is called, retrieving the version of the class `Counter`, and printed using the method `println` provided by the static `out` field in the `java.lang.System` class. The `name` public field of `myCounter` object is then accessed, setting the `MyCounter2` value. Finally, an array of 10 counters is created, and the `myCounter` object assigned to its first element.

The key point here is that the only requirement for this example to run is the presence of the `Counter.class` file in the proper position in the file system, according to Java naming conventions: no other auxiliary information is needed – no headers, no pre-compilations, etc. This enables the seamless

Figure 6.2: Using a Swing componen from a tuProlog program. Note the _ Prolog value used to represent the Java `null` value.

```
test_open_file_dialog(FileName) :-
    java_object('javax.swing.JFileChooser', [], Dialog),
    Dialog <- showOpenDialog(_),
    Dialog <- getSelectedFile returns File,
    File <- getName returns FileName.
```

reuse and exploitation of the large amount of available Java libraries and resources, starting from the standard ones, such as Swing to manage GUI components, JDBC to access databases, RMI and CORBA for distributed computing, and so on. Figure 6.2 shows an example, where Java Swing API is exploited to graphically choose a file from Prolog: a Swing `JFileChooser` dialog is instantiated and bound to the Prolog variable `Dialog` (a univocal Prolog atom of the form `'$obj_N'`, to be used as the object reference, is automatically generated and bounded to the variable) which is then used to invoke methods `showOpenDialog` and `getSelectedFile` of `JFileChooser`'s interface. Further examples about exploiting standard Java libraries from tuProlog can be found in [**?**].

Besides the Prolog predicates, `JavaLibrary` embeds the `register` function, which, unlike the previous functionalities, is to be used on the Java side. Its purpose is to associate an existing Java object *`obj`* to a Prolog identifier *`ObjectRef`*, according to the syntax:

```
boolean register(Struct ObjectRef, Object obj) throws
                InvalidObjectIdException;
```

`ObjectRef` is a ground term (otherwise an exception is raised) that represents the Java object *`obj`* in the context of `JavaLibrary`'s predicates: the function returns `false` if the object represented by *`obj`* is already registered under a different *`ObjectRef`*. As an example of use, let us consider the following case:[1]

```
Prolog core = new Prolog();
Library lib = core.loadLibrary("alice.tuprolog.lib.JavaLibrary");
((alice.tuprolog.lib.JavaLibrary)lib).register(new Struct("stdout"),
                                               System.out);
```

[1] An explicit cast to `alice.tuprolog.lib.JavaLibrary` is needed because `loadLibrary` returns a reference to a generic `Library`, while the `register` primitive is defined in `JavaLibrary` only.

Here, the Java object `System.out` is registered for use in `tu`Prolog under the name `stdout`. So, within the scope of the `core` engine, a Prolog program can now contain

```
stdout <- println('What a nice message!')
```

as if `stdout` was a pre-defined `tu`Prolog identifier.

6.3 Predicates

Here follows a list of predicates implemented by this library, grouped in categories corresponding to the functionalities they provide.

6.3.1 Method Invocation, Object and Class Creation

- `java_object/3`
  `java_object(ClassName, ArgList, ObjId)` is true iff `ClassName` is the full class name of a Java class available on the local file system, `ArgList` is a list of arguments that can be meaningfully used to instantiate an object of the class, and `ObjId` can be used to reference such an object; as a side effect, the Java object is created and the reference to it is unified with `ObjId`. It is worth noting that `ObjId` can be a Prolog variable (that will be bound to a ground term) as well as a ground term (not a number).
  Template: `java_object(+full_class_name, +list, ?obj_id)`

- `java_object_bt/3`
  `java_object_bt(ClassName, ArgList, ObjId)` has the same behaviour of `java_object/3`, but the binding that is established between the `ObjId` term and the Java object is destroyed with backtracking.
  Template: `java_object_bt(+full_class_name, +list, ?obj_id)`

- `destroy_object/1`
  `destroy_object(ObjId)` is true and as a side effect the binding between `ObjId` and a Java object, possibly established, by previous predicates is destroyed.
  Template: `destroy_object(@obj_id)`

- `java_class/4`
  `java_class(ClassSourceText, FullClassName, ClassPathList, ObjId)` is true iff `ClassSouceText` is a source string describing a valid Java

class declaration, a class whose full name is `FullClassName`, according to the classes found in paths listed in `ClassPathList`, and `ObjId` can be used as a meaningful reference for a `java.lang.Class` object representing that class; as a side effect the described class is (possibly created and) loaded and made available to the system.
Template: `java_class(@java_source, @full_class_name, @list, ?obj_id)`

- `java_call/3`
  `java_call(ObjId, MethodInfo, ObjIdResult)` is true iff `ObjId` is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term `MethodInfo` and possible arguments the arguments of `MethodInfo` as a compound, and `ObjIdResult` can be used as a meaningful reference for the Java object that the method possibly returns. As a side effect the method is called on the Java object referenced by the `ObjId` and the object possibly returned by the method invocation is referenced by the `ObjIdResult` term. The anonymous variable used as argument in the `MethodInfo` structure is interpreted as the Java `null` value.
  Template: `java_call(@obj_id, @method_signature, ?obj_id)`

- `'<-'/2`
  `'<-'(ObjId, MethodInfo)` is true iff `ObjId` is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term `MethodInfo` and possible arguments the arguments of `MethodInfo` as a compound. As a side effect the method is called on the Java object referenced by the `ObjId`. The anonymous variable used as argument in the `MethodInfo` structure is interpreted as the Java `null` value.
  Template: `'<-'(@obj_id, @method_signature)`

- `return/2`
  `return('<-'(ObjId, MethodInfo), ObjIdResult)` is true iff `ObjId` is a ground term currently referencing a Java object, which provides a method whose name is the functor name of the term `MethodInfo` and possible arguments the arguments of `MethodInfo` as a compound, and `ObjIdResult` can be used as a meaningful reference for the Java object that the method possibly returns. As a side effect the method is called on the Java object referenced by the `ObjId` and the object possibly returned by the method invocation is referenced by the `ObjIdResult` term. The anonymous variable used as argument in the `MethodInfo` structure is interpreted as the Java `null` value.

It is worth noting that this predicate is equivalent to the `java_call` predicate.
Template: `return('<-'(@obj_id, @method_signature), ?obj_id)`

6.3.2 Java Array Management

- `java_array_set/3`
  `java_array_set(ObjArrayId, Index, ObjId)` is true iff `ObjArrayId` is a ground term currently referencing a Java array object, `Index` is a valid index for the array and `ObjId` is a ground term currently referencing a Java object that could inserted as an element of the array (according to Java type rules). As side effect, the object referenced by `ObjId` is set in the array referenced by `ObjArrayId` in the position (starting from 0, following the Java convention) specified by `Index`. The anonymous variable used as `ObjId` is interpreted as the Java `null` value. This predicate can be used for arrays of Java objects: for arrays whose elements are Java primitive types (such as `int`, `float`, etc.) the following predicates can be used, with the same semantics of `java_array_set` but specifying directly the term to be set as a `tu`Prolog term (according to the mapping described previously):
  ```
  java_array_set_int(ObjArrayId, Index, Integer)
  java_array_set_short(ObjArrayId, Index, ShortInteger)
  java_array_set_long(ObjArrayId, Index, LongInteger)
  java_array_set_float(ObjArrayId, Index, Float)
  java_array_set_double(ObjArrayId, Index, Double)
  java_array_set_char(ObjArrayId, Index, Char)
  java_array_set_byte(ObjArrayId, Index, Byte)
  java_array_set_boolean(ObjArrayId, Index, Boolean)
  ```
  Template: `java_array_set(@obj_id, @positive_integer, +obj_id)`

- `java_array_get/3`
  `java_array_get(ObjArrayId, Index, ObjIdResult)` is true iff `ObjArrayId` is a ground term currently referencing a Java array object, `Index` is a valid index for the array, and `ObjIdResult` can be used as a meaningful reference for a Java object contained in the array. As a side effect, `ObjIdResult` is unified with the reference to the Java object of the array referenced by `ObjArrayId` in the `Index` position. This predicate can be used for arrays of Java objects: for arrays whose elements are Java primitive types (such as `int`, `float`, etc.) the following predicates can be used, with the same semantics of `java_array_get` but

binding directly the array element to a tuProlog term (according to the mapping described previously):

```
java_array_get_int(ObjArrayId, Index, Integer)
java_array_get_short(ObjArrayId, Index, ShortInteger)
java_array_get_long(ObjArrayId, Index, LongInteger)
java_array_get_float(ObjArrayId, Index, Float)
java_array_get_double(ObjArrayId, Index, Double)
java_array_get_char(ObjArrayId, Index, Char)
java_array_get_byte(ObjArrayId, Index, Byte)
java_array_get_boolean(ObjArrayId, Index, Boolean)
```

Template: `java_array_get(@obj_id, @positive_integer, ?obj_id)`

- `java_array_length/2`
  `java_array_length(ObjArrayId, ArrayLength)` is true iff `ArrayLength` is the length of the Java array referenced by the term `ObjArrayId`.
  Template: `java_array_length(@term, ?integer)`

6.3.3 Helper Predicates

- `java_object_string/2`
  `java_object_string(ObjId, String)` is true iff `ObjId` is a term referencing a Java object and `PrologString` is the string representation of the object (according to the semantics of the `toString` method provided by the Java object).
  Template: `java_object_string(@obj_id, ?string)`

6.4 Functors

No functors are provided by the `JavaLibrary` library.

6.5 Operators

Name	Assoc.	Prio.
<-	xfx	800
returns	xfx	850
as	xfx	200
.	xfx	600

6.6 Java Library Examples

The following examples are designed to show `JavaLibrary`'s ease of use and flexibility.

6.6.1 RMI Connection to a Remote Object

Here we connect via RMI to a remote Java object. In order to allow the reader to try this example with no need of other objects, we connect to the remote Java object identified by the name `'prolog'`, which is an RMI server bundled with the tuProlog package, and can be spawned by typing:

```
java -Djava.security.all=policy.all alice.tuprologx.runtime.rmi.Daemon
```

Then, we invoke the object method whose signature is

```
SolveInfo solve(String goal);
```

```
?- java_object('java.rmi.RMISecurityManager', [], Manager),
   class('java.lang.System') <- setSecurityManager(Manager),
   class('java.rmi.Naming') <- lookup('prolog') returns Engine,
   Engine <- solve('append([1],[2],X).') returns SolInfo,
   SolInfo <- success returns Ok,
   SolInfo <- getSubstitution returns Sub,
   Sub <- toString returns SubStr, write(SubStr), nl,
   SolInfo <- getSolution returns Sol,
   Sol <- toString returns SolStr, write(SolStr), nl.
```

The Java version of the same code would be:

```
System.setSecurityManager(new RMISecurityManager());
PrologRMI core = (PrologRMI) Naming.lookup("prolog");
SolveInfo info = core.solve("append([1],[2],X).");
boolean ok = info.success();
String sub = info.getSubstiturion();
System.out.println(sub);
String sol = info.getSolution();
System.out.println(sol);
```

6.6.2 Java Swing GUI from tuProlog

What about creating Java GUI components from the tuProlog environment? Here is a little example, where a standard Java Swing open file dialog windows is popped up:

```
open_file_dialog(FileName):-
    java_object('javax.swing.JFileChooser', [], Dialog ),
```

```
    Dialog <- showOpenDialog(_) returns Result,
    write(Result),
    Dialog <- getSelectedFile returns File,
    File <- getName returns FileName,
    class('java.lang.System') . out <- get(Out),
    Out <- println('you want to open file '),
    Out <- println(FileName).
```

6.6.3 Database access via JDBC from tuProlog

This example shows how to access a database via the Java standard JDBC interface from tuProlog. The program computes the minimum path between two cities, fetching the required data from the database called 'distances'. The entry point of the Prolog program is the `find_path` predicate.

```
find_path(From, To) :-
    init_dbase('jdbc:odbc:distances', Connection, '', ''),
    exec_query(Connection,
      'SELECT city_from, city_to, distance FROM distances.txt',
      ResultSet),
    assert_result(ResultSet),
    findall(pa(Length,L), paths(From,To,L,Length), PathList),
    current_prolog_flag(max_integer, Max),
    min_path(PathList, pa(Max,_), pa(MinLength,MinList)),
    outputResult(From, To, MinList, MinLength).

paths(A, B, List, Length) :-
    path(A, B, List, Length, []).

path(A, A, [], 0, _).
path(A, B, [City|Cities], Length, VisitedCities) :-
    distance(A, City, Length1),
    not(member(City, VisitedCities)),
    path(City, B, Cities, Length2, [City|VisitedCities]),
    Length is Length1 + Length2.

min_path([], X, X) :- !.
min_path([pa(Length, List) | L],  pa(MinLen,MinList), Res) :-
    Length < MinLen, !,
    min_path(L, pa(Length,List), Res).
min_path([_|MorePaths], CurrentMinPath, Res) :-
    min_path(MorePaths, CurrentMinPath, Res).

writeList([]) :- !.
writeList([X|L]) :- write(','), write(X), !, writeList(L).
```

```
outputResult(From, To, [], _) :- !,
    write('no path found from '), write(From),
    write(' to '), write(To), nl.
outputResult(From, To, MinList, MinLength) :-
    write('min path from '), write(From),
    write(' to '), write(To), write(': '),
    write(From), writeList(MinList),
    write('  - length: '), write(MinLength).

% Access to Database

init_dbase(DBase, Username, Password, Connection) :-
    class('java.lang.Class') <- forName('sun.jdbc.odbc.JdbcOdbcDriver'),
    class('java.sql.DriverManager') <- getConnection(DBase, Username, Password)
        returns Connection,
    write('[ Database '), write(DBase), write(' connected ]'), nl.

exec_query(Connection, Query, ResultSet):-
    Connection <- createStatement returns Statement,
    Statement <- executeQuery(Query) returns ResultSet,
    write('[ query '), write(Query), write(' executed ]'), nl.

assert_result(ResultSet) :-
    ResultSet <- next returns Valid, Valid == true, !,
    ResultSet <- getString('city_from') returns From,
    ResultSet <- getString('city_to') returns To,
    ResultSet <- getInt('distance') returns Dist,
    assert(distance(From, To, Dist)),
    assert_result(ResultSet).
assert_result(_).
```

6.6.4 Dynamic compilation

As already said, the `java_class` predicate performs *dynamic compilation*, creating an instance of a Java `Class` class that represents the public class declared in the source text provided as argument. The created `Class` instance, referenced by a Prolog term, can be used to create instances via the `newInstance` method, to retrieve specific constructors via the `getConstructor` method, to analyze class methods and fields, and for other above-mentioned meta-services: a sketch is reported in Figure 6.3. The `java_class` arguments in the example specify, besides the source text and the binding variable, the full class name (`Counter`), which is necessary to locate the class in the pack-

Figure 6.3: Predicate `java_class` performing dynamic compilation of Java code in tuProlog.

```
?- Source = 'public class Counter { ... }',
   java_class(Source, 'Counter', [], counterClass),
   counterClass <- newInstance returns myCounter,
   myCounter <- setValue(5),
   myCounter <- getValue returns X,
   write(X).
```

age hierarchy, and possibly a list of class paths required for a successful compilation (if any).

Figure 6.4 shows a more complex example, where a Java source is retrieved via FTP and then exploited first to create a new (previously unknown) class, and then a new instance of that class. (The FTP service is provided by a shareware Java library.) Though a lot remains to explore, `java_class` features seem quite interesting: in perspective one might think, for instance, of a Prolog intelligent agent that dynamically acquires information on a Java resource, and then autonomously builds up, at run-time, the proper Java machinery enabling efficient interaction with the resource.

Figure 6.4: A new Java class is compiled and used after being retrieved via FTP.

```
% A user whose name is 'myName' and whose password is 'myPwd' gets the content of the file
% 'Counter.java' from the server whose IP address is 'srvAddr', creates the corresponding
% Java class and exploits it to instantiate and deploy an object

test :-
    get_remote_file('alice/tuprolog/test', 'Counter.java', srvAddr, myName, myPwd, Content),
    % creating the class
    java_class(Content, 'Counter', [], CounterClass),
    % instantiating (and using) an object of such a class
    CounterClass <- newInstance returns MyCounter,
    MyCounter <- setValue(303),
    MyCounter <- inc,
    MyCounter <- inc,
    MyCounter <- getValue returns Value,
    write(Value), nl.

% +DirName: Directory on the server where the file is located
% +FileName: Name of the file to be retrieved
% +FTPHost: IP address of the FTP server
% +FTPUser: User name of the FTP client
% +FTPPwd: Password of the FTP client
% -Content: Content of the retrieved file

get_remote_file(DirName, FileName, FTPHost, FTPUser, FTPPwd, Content) :-
    java_object('com.enterprisedt.net.ftp.FTPClient', [FTPHost], Client),
    % get file
    Client <- login(FTPUser, FTPPwd),
    Client <- chdir(DirName),
    Client <- get(FileName) returns Content,
    Client <- quit.
```

Chapter 7

The IDE

The tuProlog system comes with a simple application providing an user friendly integrated development environment to interact with a tuProlog engine, manipulate its knowledge base, make queries and explore solutions. In addition, means to dynamically manage the loading and unloading of tuProlog libraries are provided. After a proper installation of the tuProlog distribution, the application is spawned by launching the executable class `alice.tuprologx.ide.GUILauncher`. The console user interface version, providing a command-line shell, can be accessed by launching the executable class `alice.tuprologx.ide.CUIConsole`.

The main window of the tuProlog IDE is shown in Figure 7.1. It is divided in two sections:

- an editing area on the middle, providing means to edit the engine's current theory;
- a console on the bottom, providing means to ask queries and display their solutions.

In the main window there also are: a toolbar at the top, providing facilities to manage theories, such as load, save as well as create a new theory, to load and unload libraries into and from the tuProlog engine, and to view in a separate window the debug informations activated by means of the `spy/0` predicate; and a status bar at the very bottom, providing status informations for the IDE and the engine.

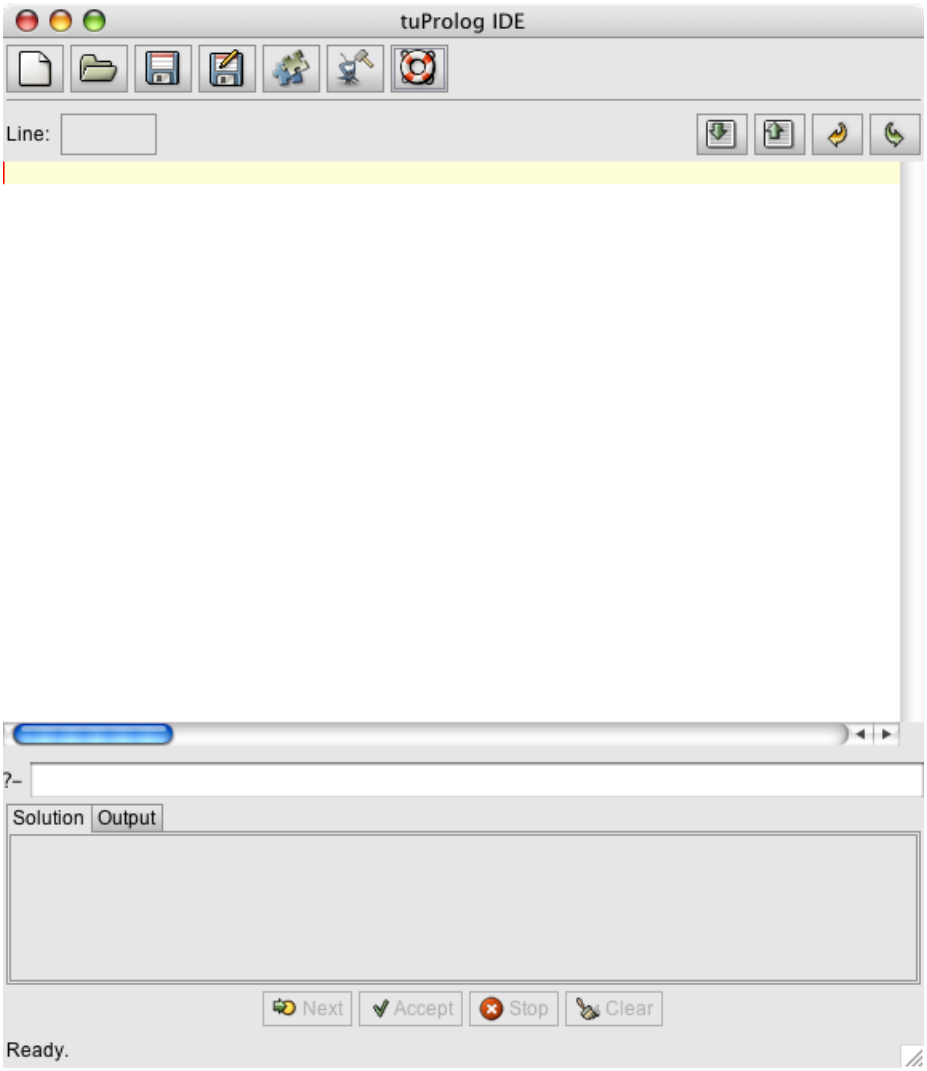

Figure 7.1: tuProlog IDE.

7.1 Editing the theory

The editing area allows multiple theories to be created and modified at the same time, by allocating a tab with a new text area for each theory. The text area provides syntax highlighting for comments, string and list literals, and predefined predicates. Undo and Redo actions are supported through the usual **Ctrl**+**Z** and **Ctrl**+**Shift**+**Z** key bindings.

Above the editing tabs, a control area is found, where two buttons are provided to get the text of the engine's current theory into a new tab and to set the text contained in the editor of the selected tab as a new theory for the engine, and two buttons are provided for mouse-clicking support of Undo and Redo actions. An apposite action for retrieving the engine's current theory in an editor (shown in Figure 7.2) is needed because whenever that theory gets modified by other means, such as calling the `consult/1` predicate, the changes are not automatically reflected in any text area. On the left side of the control area, there also is an indicator of the line where the caret is currently positioned in the edit area. Informations about the result of the action issued by the control area are provided in the status bar at the very bottom of the IDE's window: for instance, when setting an invalid theory to the engine because of syntax errors, details about the error are provided.

7.2 Solving goals

The console at the bottom of the tuProlog IDE's window is subdivided in two logical panes:

- a query pane composed by a textfield where queries can be inserted, and two buttons to trigger the solving process. The leftmost (Solve) button asks the engine to find the first solution to the query, allowing the user to possibly navigate through further solutions; the rightmost (Solve All) button forces the engine to find all solutions to the given query. Pressing the **Enter** key in the textfield has the same effect as pressing the Solve button.

- an answer pane, where answers and output informations are visualized. Answers to Prolog queries are composed by both solutions, showed in a free form within a read-only text area, and bindings, displayed in tabular form. The output tab provides a read-only view on the standard output where informations are possibly written by Prolog

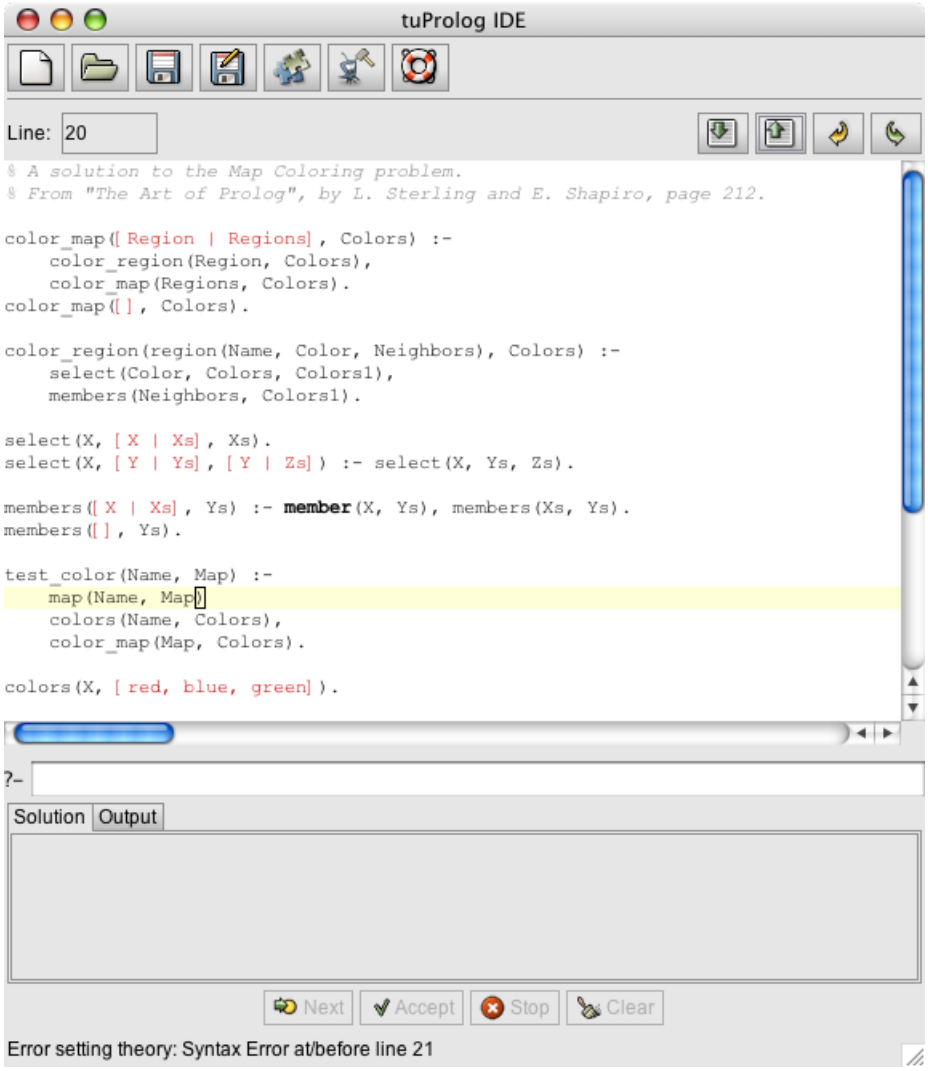

Figure 7.2: A syntax error is found when setting the content of the editor area as the new engine's theory.

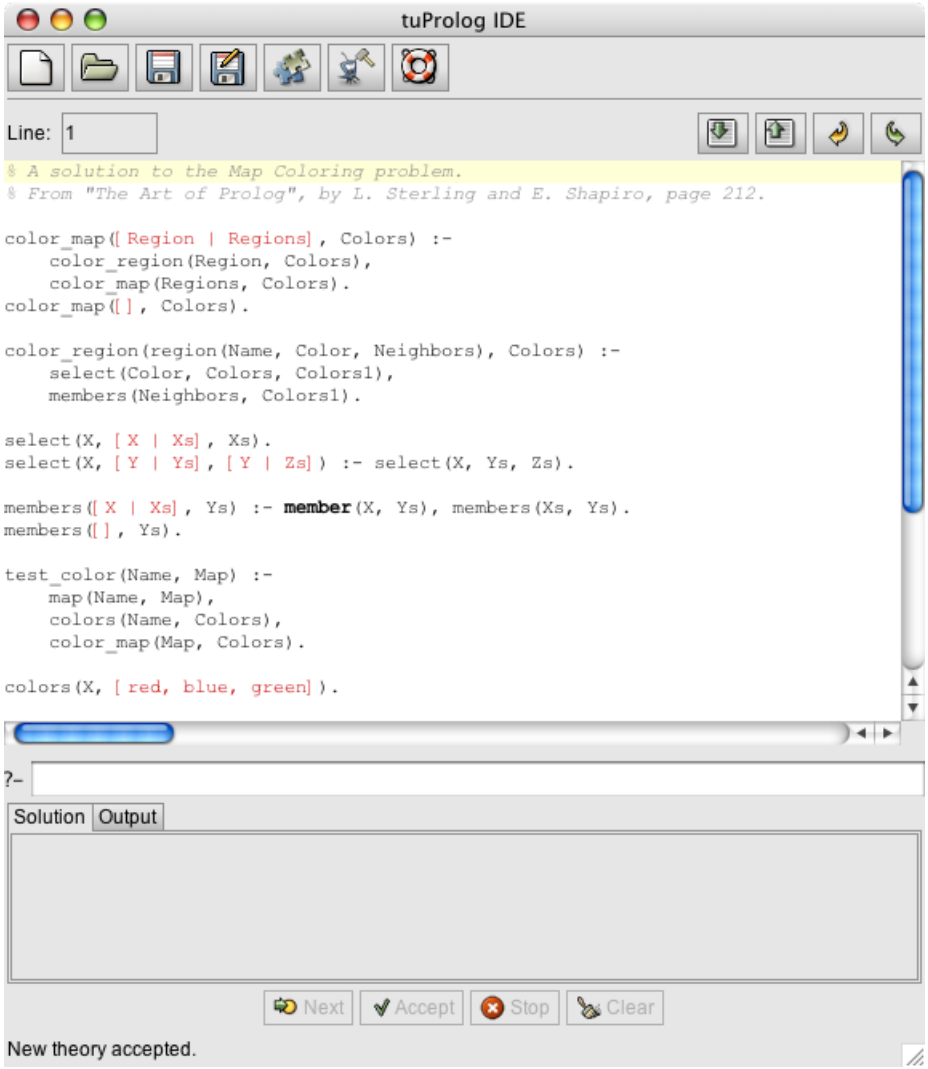

Figure 7.3: The syntax error is removed and the Set Theory operation succeeds.

programs, by means of the I/O predicates supplied by the `IOLibrary`.[1] Control buttons are also provided to iterate through possibly multiple solutions, clear the bindings and output panes, and export tabular data in a convenient CSV format.

Goals are asked through the query input box, and answers (bindings and solutions) are provided in the related text area. Query and answers are traced in a proper chronological history, that can be explored by means of **Up** and **Down** arrow keys from the query input textfield. When open alternatives are found solving a goal, the Next and Accept buttons are enabled in the answer pane to interact with the engine, in order to let the user specify if the current solution is accepted or if other alternatives need to be explored.

Note that the theory contained in the currently selected edit pane does not have to be explicitly feeded to the Prolog engine before it could be possible to solve queries against that theory's knowledge base. In fact, any time a goal is asked to be solved, the theory contained in the active edit area is automatically feeded to the engine if its knowledge base has been modified since the last solved goal. (This obviously happens also on the first time a query is asked.) However, whenever the engine's theory is modified by other means than the editor, it does need to be explicitly acquired and presented to the programmer in the text area. In fact, if the theory in the engine is augmented by a call to the predicate `consult/1` issued from a query, for example, the contents of the newly consulted theory will not be automatically inserted in the editor: when the programmer needs an up-to-date view of the knowledge base contained in the underlying tuProlog engine, its display has to be explicitly triggered by means of the GetTheory button, available in the editing area.

An example of the user interaction involving multiple solutions is shown in the following sequence of figures: in Figure 7.4, the user issued the query `test_color(test, X).`, using the knowledge base written in the edit area (a solution to the Map Coloring problem,[2] with a test map composed of six areas). The first solution is displayed, and multiple open alternatives can be explored: in Figure 7.5, the user asked to get the next possible solution by pressing the Next button, and another solution is provided; finally, in Figure 7.6, the user, after having explored the first two solutions, accepts

[1] The information written on standard output by methods invoked on Java objects from the `JavaLibrary` – for instance using the `stdout` object – are not displayed on this view.

[2] The problem is to color a planar map so that no two adjoining regions have the same color. A famous conjecture was proved in 1976 showing that four colors are sufficient to color any planar map.

the third one by pressing the Accept button. During the resolution of a goal, all the theory-related buttons are disabled, included the Library Manager button, since each library can have its theory to be feeded into the engine.

Near to the Next and Accept buttons, a Stop button is found, providing the user with a means to halt the engine if a computation takes too long or a bug in the knowledge base feeded to the engine results in an infinite loop.

Finally, a Clear button is provided, with the aim of allowing the user to clear the bindings and output panes when they get overfull with informations. The button is enabled only when the proper tabs are selected.

7.3 Debug Informations

By pressing the View Debug Information button, a new window is opened, providing a view on the warnings, produced by events such as the attempt at redefining a library predicate, and the spy information, concerning basic steps of the engine computation and state, possibly supplied by the engine during a goal demonstration: Warnings are always active; in order to activate the spy information notification, the `spy/0` built-in predicate (provided by `BasicLibrary`) must be issued; `nospy/0` can be used to stop this notification. As an example, Figure 7.7 shows the content of the spy information view after the execution of a goal involving the activation of spy inspection.

It is worth noting that a computation may contain a huge number of traced steps. For this reason, a toolbar at the top of the window allows to collapse and expand all nodes in the spy information pane, or to expand and collapse selected nodes only. Finally, the content of the warnings and spy panes can be cleared using the Clear button at the leftmost end of the toolbar.

7.4 Dynamic library management

A tuProlog engine can be extended by loading any number of libraries, each provinding a specific set of built-in predicates and functors, and a related theory. The tuProlog IDE allows a dynamic management of libraries through a GUI dialog, which can be displayed by pressing the Open Library Manager button in the toolbar. The Library Manager dialog is shown in Figure 7.8.

This dialog displays a list of the libraries currently loaded into the tuProlog engine. For a new instance of the engine, that list will typically contain the four standard libraries coming with the application core, that is `BasicLibrary`, `IOLibrary`, `ISOLibrary`, `JavaLibrary`, along with their

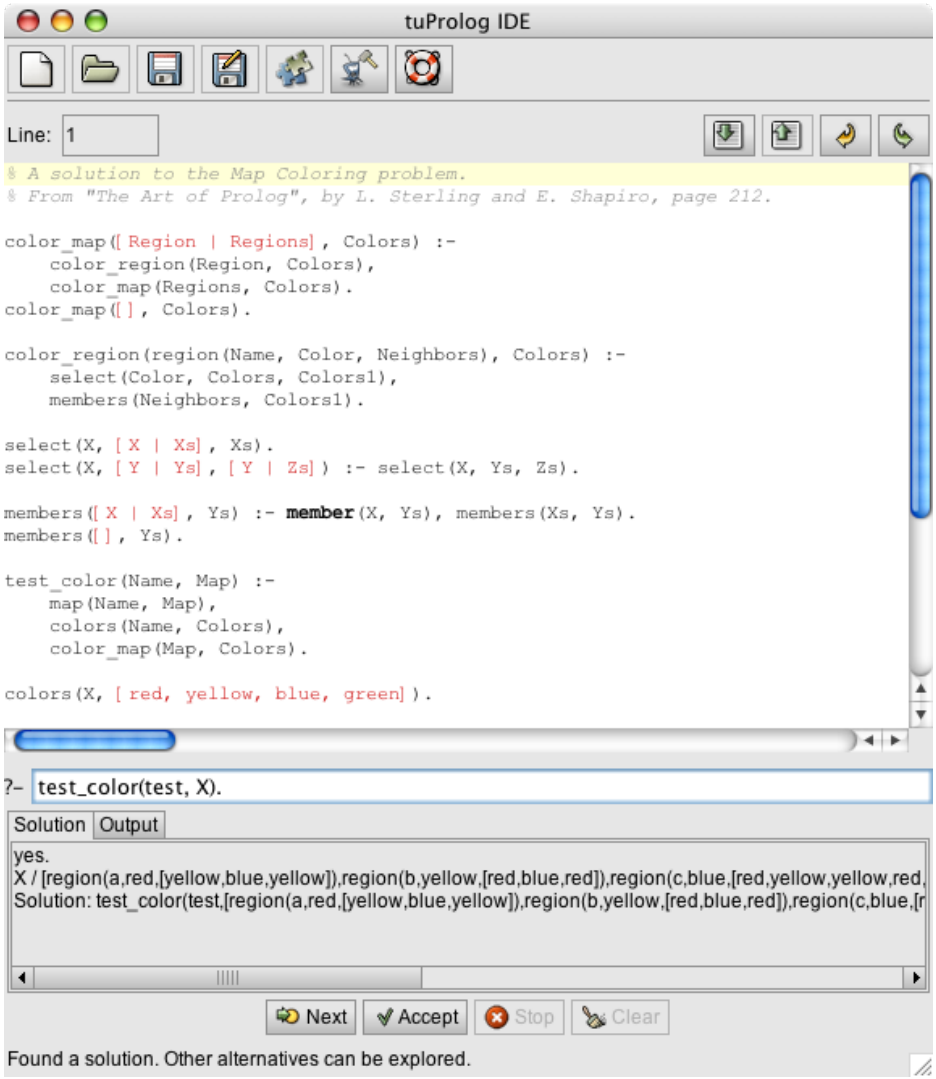

Figure 7.4: The user issued a query `test_color(test, X).` and the first solution is displayed.

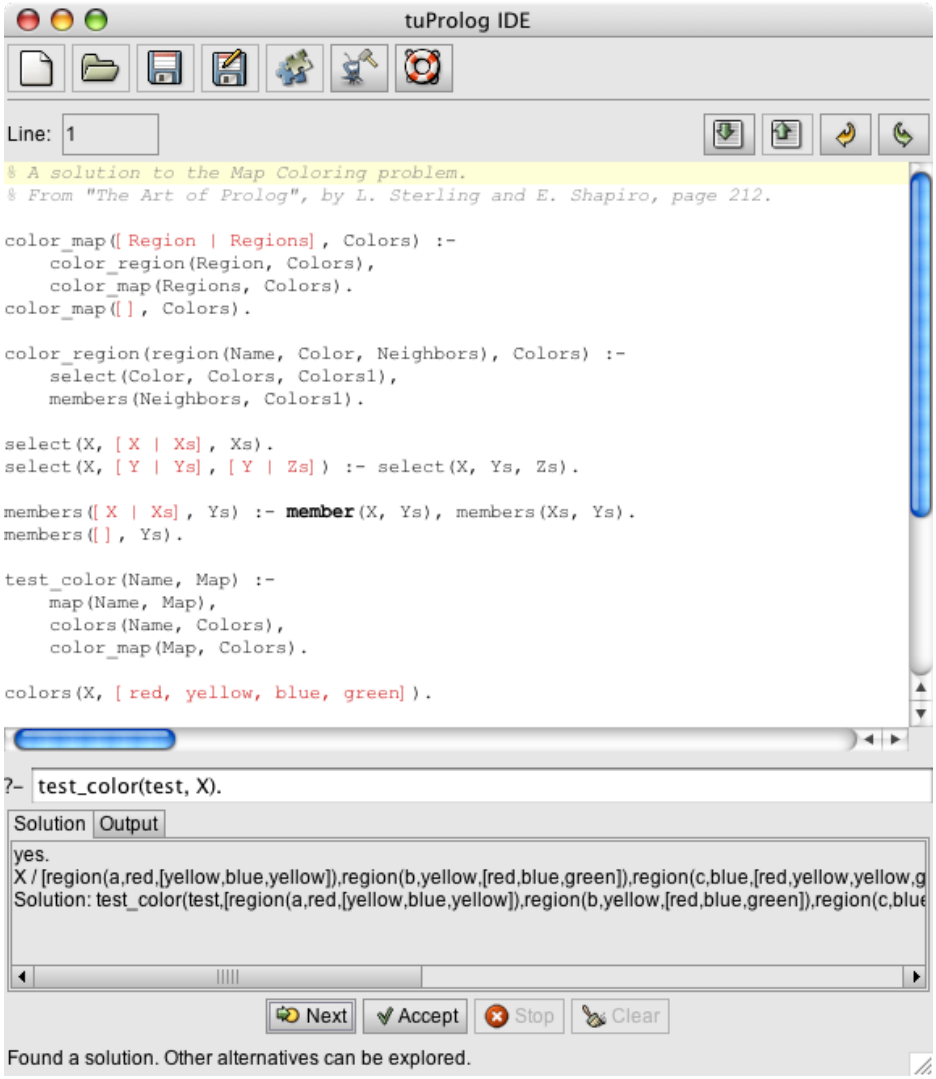

Figure 7.5: The user issued a Next command and got another solution.

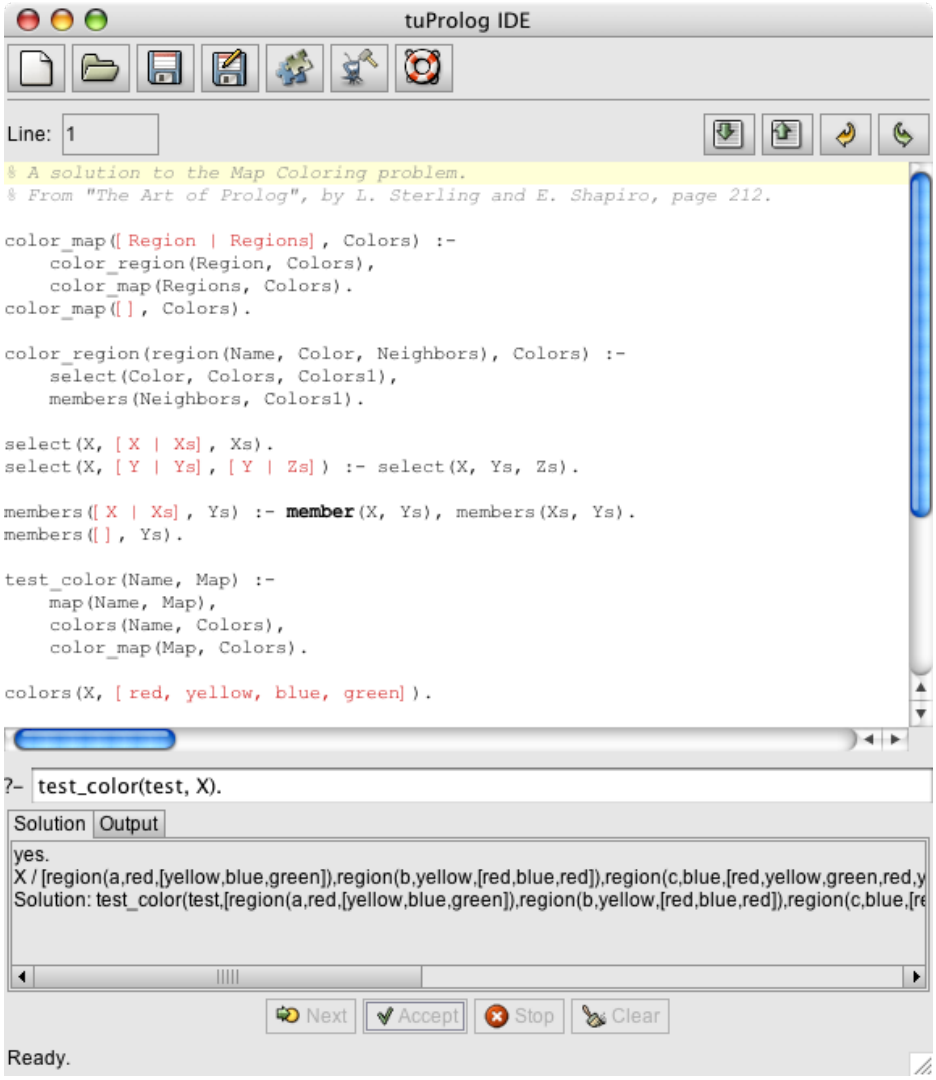

Figure 7.6: The user accepted the third solution by pressing the Accept button.

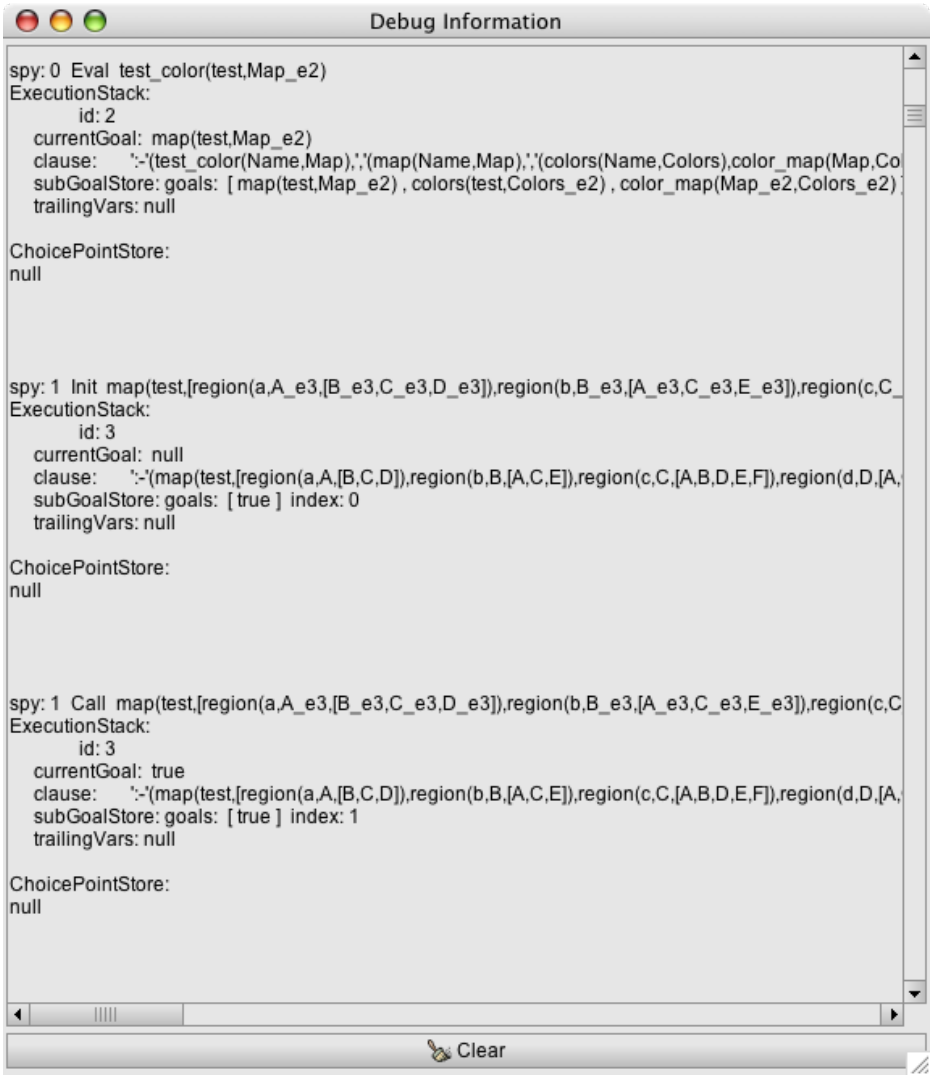

Figure 7.7: Debug Information View after the execution of a goal.

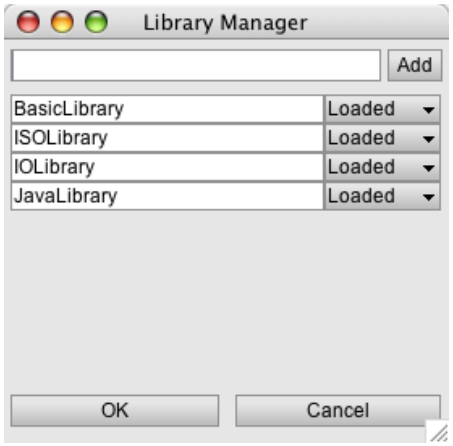

Figure 7.8: The Library Manager dialog.

current status. The user can add a library to the Library Manager simply by providing the fully qualified name of the library's class in the textfield on the top of the dialog, then pressing the Add button: the added library will be displayed with an initial Unload status. The user can further select the status of each library in the list, and commit changes to the tuProlog engine by pressing the OK button, or dismiss the dialog by pressing the Cancel button.

The library manager is also capable of updating itself accordingly to the events of libraries load and unload fired by the tuProlog engine. Such events are triggered by the use of the `load_library/1` and `unload_library/1` predicates or directives in query issued or theories feeded to the engine. So, if an user asks to solve the goal `load_library('TestLibrary'), test(X).`, for example, the manager would immediately reflect the change occurred in the engine's libraries pool, adding a new entry if `TestLibrary` had not been previously loaded or, if necessary, changing the library's entry status to show the result of the loading action.

Both the action of adding a library to the manager and the action of loading a library into the engine can fail. If, for example, the classname provided does not identify a tuProlog library (i.e. it identifies a class not extending the `alice.tuprolog.Library` class) or the identified class does not exist, an appropriate message will appear in the status bar at the bottom of the dialog. When adding or loading a library, please remember that every class needed by that library must be in the classpath in order to have the library correctly added to the manager's list or loaded into the engine.

Chapter 8

Using tuProlog from Java

8.1 Getting started

Let's begin with your first Java program using tuProlog.

```
import alice.tuprolog.*;

public class Test2P {
    public static void main(String[] args) throws Exception {
        Prolog engine = new Prolog();
        SolveInfo info = engine.solve("append([1],[2,3],X).");
        System.out.println(info.getSolution());
    }
}
```

In this first example a tuProlog engine is created, and asked to solve a query, provided in a textual form. This query in the Java environment is equivalent to the query

```
?- append([1],[2,3],X).
```

in a classic Prolog environment, and accounts for finding the list that is obtained by appending the list `[2,3]` to the list `[1]` (`append` is included in the theory provided by the `alice.tuprolog.lib.BasicLibrary`, which is downloaded by the default when instantiating the engine).

By properly compiling and executing this simple program,[1] the string

[1] Save the program in a file called `Test2P.java`, then compile it with `javac -classpath tuprolog.jar Test2P.java` and then execute it with `java -cp .;tuprolog.jar Test2P.java`

`append([1],[2,3],[1,2,3])` – that is the solution of out query – will be displayed on the standard output.
Then, let's consider a little bit more involved example:

```
public class Test2P {
    public static void main(String[] args) throws Exception {
        Prolog engine = new Prolog();
        SolveInfo info = engine.solve("append(X,Y,[1,2]).");
        while (info.isSuccess()) {
            System.out.println("solution: " + info.getSolution() +
                               " - bindings: " + info);
            if (engine.hasOpenAlternatives()) {
                info = engine.solveNext();
            } else {
                break;
            }
        }
    }
}
```

In this case, all the solutions of a query are retrieved and displayed, with also the variable bindings:

```
solution: append([],[1,2],[1,2]) - bindings: Y / [1,2]  X / []
solution: append([1],[2],[1,2]) - bindings: Y / [2]  X / [1]
solution: append([1,2],[],[1,2]) - bindings: Y / []  X / [1,2]
```

8.2 Basic Data Structures

All Prolog data objects are mapped onto Java objects: `Term` is the base class for Prolog untyped terms such as atoms and compound terms (represented by the `Struct` class), Prolog variables (`Var` class) and Prolog typed terms such as numbers (`Int`, `Long`, `Float`, `Double` classes). In particular:

- `Term` – this abstract class represents a generic Prolog term, and it is the root base class of `tu`Prolog data structures, defining the basic common services, such as term comparison, term unification and so on. It is worth noting that it is an abstract class, so no direct `Term` objects can be instantiated;

- `Var` – this class (derived from `Term`) represents `tu`Prolog variables. A variable can be anonymous, created by means of the default constructor, with no arguments, or identified by a name, that must starts with an upper case letter or an underscore;

- `Struct` – this class (derived from `Term`) represents un-typed `tu`Prolog terms, such as atoms, lists and compound terms; `Struct` objects are characterised by a functor name and a list of arguments (which are `Term`s themselves), while `Var` objects are labelled by a string representing the Prolog name. Atoms are mapped as `Struct`s with functor with a name and no arguments; Lists are mapped as `Struct` objects with functor `'.'`, and two `Term` arguments (head and tail of the list); lists can be also built directly by exploiting the 2-arguments constructor, with head and tail terms as arguments. Empty list is constructed by means of the no-argument constructor of `Struct` (default constructor).

- `Number` – this abstract class represents typed, numerical Prolog terms, and it is the base class of `tu`Prolog number classes;

- `Int, Long, Double, Float` – these classes (derived from `Number`) represent typed numerical `tu`Prolog terms, respectively integer, long, double and float numbers. Following the Java conventions, the default type for integer number is `Int` (integer, not long, number), and for `Double` (and so double), for floating point number.

Some examples of term creation follow:

```
// constructs the atom vodka
Struct drink = new Struct("vodka");

// constructs the number 40
Term degree = new alice.tuprolog.Int(40);

// constructs the compound degree(vodka, 40)
Term drinkDegree = new Struct("degree",
                              new Struct("vodka"),
                              new Int(40));
// second way to constructs the compound degree(vodka,40)
Struct drinkDegree2 = new Struct("degree", drink, degree);

// constructs the compound temperature('Rome', 25.5)
```

```
Struct temperature = new Struct("temperature",
                                new Struct("Rome"),
                                new alice.tuprolog.Float(25.5));

// constructs the compound equals(X, X)
Struct t1 = new Struct("equals", new Var("X"), new Var("X"));
t1.resolveTerm();

// mother(John,Mary)
Struct father = new Struct(new Struct("John"), new Struct("Mary")));

// father(John, _)
Term  father = new Struct(new Struct("John"), new Var());

// p(1, _, q(Y, 3.03, 'Hotel'))
Term  t2 = new Struct("p",
                      new Int(1),
                      new Var(),
                      new Struct("q",
                                 new Var("Y"),
                                 new Float(3.03f),
                                 new Struct("Hotel")));

// The Long number 130373303303
Term t3 = new alice.tuprolog.Long(130373303303h);

// The double precision number 1.7625465346573
Term t4 = new alice.tuprolog.Double(1.7625465346573);

// an empty list
Struct empty = new Struct();

// the list [303]
Struct l = new Struct(new Int(303), new Struct());

// the list [1,2,apples]
Struct alist = new Struct(
                    new Int(1),
                    new Struct(
                        new Int(2),
                        new Struct("apples")));

// fruits([apple, orange | _ ])
Term list2 = new Struct("fruits", new Struct(
                                       new Struct("apple",
```

```
                                        new Struct("orange"),
                                        new Var())));

// complex_compound(1, _, q(Y, 3.03, 'Hotel', k(Y,X)), [303, 13, _, Y])
Term t5 = Term.parse(
     "complex_compound(1, _, q(Y, 3.03, 'Hotel', k(Y,X)), [303, 13, _, Y])"
);
```

The name of the tuProlog number classes (`Int`, `Float`, `Long`, `Double`) follows the name of the primitive Java data type they represents. Note that due to class name clashes (for instance between classes `java.lang.Long` and `alice.tuprolog.Long`), it could be necessary to use the full class name to identify tuProlog classes.

8.3 Engine, Theories and Libraries

Then, the other main classes that make tuProlog Core API concern tuProlog engines, theories and libraries. In particular:

- `Prolog` – this class represent tuProlog engines. This class provides a minimal interface that enables users to:
  – set/get the theory to be used for demonstrations;
  – load/unload libraries;
  – solve a goal, represented either by a `Term` object or by a textual representation (a `String` object) of a term.
  A tuProlog engine can be instantiated either with some standard default libraries loaded, by means of the default constructor, or with a starting set of libraries, which can be empty, provided as argument to the constructor (see JavaDoc documentation for details). Accordingly, a raw, very lightweight, tuProlog engine can be created by specifying an empty set of library, providing natively a very small set of built-in primitives.

- `Theory` – this class represent tuProlog theories. A theory is represented by a text, consisting of a series of clauses and/or directives, each followed by a dot and a whitespace character. Instances of this class are built either from a textual representation, directly provided as a string or taken by any possible input stream, or from a list of terms representing Prolog clauses.

- `Library` – this class represents tuProlog libraries; A `tuprolog` engine can be dynamically extended by loading (and unloading) any number of libraries; each library can provide a specific set of of built-ins predicates, functors and a related theory. A library can be loaded by means of the built-in by means of the method `loadLibrary` of the tuProlog engine. Some standard libraries are provided in the `alice.tuprolog.lib` package and loaded by the default instantiation of a tuProlog engine: `alice.tuprolog.lib.BasicLibrary`, providing basic and well-known Prolog built-ins, `alice.tuprolog.lib.IOLibrary` providing *de facto* standard Prolog I/O predicates, `alice.tuprolog.lib.ISOLibrary` providing some ISO predicates/functors not directly provided by `BasicLibrary` and `IOLibrary`, and `alice.tuprolog.lib.JavaLibrary`, which enables the creation and usage of Java objects from tuProlog programs, in particular enabling the reuse of any existing Java resources.

- `SolveInfo` – this class represents the result of a demonstration and instances of these class are returned by the `solve` methods the `Prolog` engines; in particular `SolveInfo` objects provide services to test the success of the demonstration (`isSuccess` method), to access to the term solution of the query (`getSolution` method) and to access the list of the variable with their bindings.

Some notes about tuProlog terms and the services they provide:

- the static `parse` method provides a quick way to get a term from its string representation.

- tuProlog terms provides directly methods for unification and matching:
  ```
  public boolean unify(Term t)
  public boolean match(Term t)
  ```
  Terms that have been subject to unification outside a demonstration context (that is invoking directly these methods, and not passing through the solving service of an engine) should not be used then in queries to engines.

- some services are provided to compare terms, according to the Prolog rules, and to check their type; in particular the standard Java method `equals` has the same semantics of the method `isEqual` which follows the Prolog comparison semantics.

- some services makes it possible to copy a term as it is or to get a renamed copy of the term (`copy` and `getRenamedCopy`); it is worth

noting that the design of tuProlog promotes a stateless usage of terms; in particular, it is good practice not to reuse the same terms in different demonstration contexts, as part of different queries.

- the method `getTerm` is useful in the case of variables, providing the term linked possibly considering all the linking chain in the case of variables referring other variables.

- when a term is created by means of the proper constructor, consider as example:

```
Struct myTerm = new Struct("p", new Var("X"), new Int(1), new Var("X"))
```

  it is *not resolved*, in the sense that possible variable terms with the same name in the term do not refer each other; so in the example the first and the third argument of the compound `myTerm` point to different variable objects. A term is resolved the first time it is involved in a matching or unification context.

Some notes about tuProlog engines, theories, libraries and the services they provide:

- tuProlog engines support natively some *directives*, that can be defined by means of the :-/1 predicate in theory specification. Directives are used to specify properties of clauses and of the engine (*solve/1*, *initialization/1*, *set_prolog_flag/1*, *load_library/1*, *consult/1*), format and syntax of read-terms (*op/3*, *char_conversion/2*).

- tuProlog engines support natively the dynamic definition and management of *flags* (or property), used to describe some aspects of libraries and their built-ins. A flag is identified by a name (an alphanumeric atom), a list of possible values, a default value and a boolean value specifying if the flag value can be modified.

- tuProlog engines are thread-safe. The methods that could create problems in being used in a multi-threaded context are now synchronised.

- tuProlog engines have no (static) dependencies with each other, multiple engines can be created independently as simple objects on the same Java virtual machine, each with its own configuration (theory and loaded libraries). Moreover, accordingly to the design of tuProlog system in general, engines are very lightweight, making suitable the use of multiple engines in the same execution context.

- tuProlog engines can be serialised and stored as a persistent object or sent through the network. This is true also for engines with pre-loaded standard libraries: in the case that other libraries are loaded, these must be serializable in order to have the engine serializable.

8.4 Some more examples of tuProlog usage

Creation of an engine (with default libraries pre-loaded):

```
import alice.tuprolog.*;

...
Prolog engine = new Prolog();
```

Creation of an engine specifying only the `BasicLibrary` as pre-loaded library:

```
import alice.tuprolog.*;

...
Prolog engine = new Prolog(new String[]{"alice.tuprolog.lib.BasicLibrary"});
```

Creation and loading of a theory from a string:

```
String theoryText = "my_append([],X,X).\n" +
                    "my_append([X|L1],L2,[X|L3]) :- my_append(L1,L2,L3).\n";

Theory theory = new Theory(theoryText);
try {
    engine.setTheory(theory);
} catch(InvalidTheoryException e) {
}
```

Creation and loading of a theory from an input stream:

```
Theory theory = new Theory(new FileInputStream("test.pl");
try {
    engine.setTheory(theory);
} catch(InvalidTheoryException e) {
}
```

Goal demonstration (provided as a string):

```
// ?- append(X,Y,[1,2,3]).
try {
    SolveInfo info = engine.solve("append(X,Y,[1,2,3]).");
```

```
        Term solution = info.getSolution();
    } catch(MalformedGoalException mge) {
        ...
    } catch(NoSolutionException nse) {
        ...
    }
```

Goal demonstration (provided as a Term):

```
    try {
        Term goal = new Struct("p", new Int(1), new Var("X"));
        try {
            // ?- p(1,X).
            SolveInfo info = engine.solve(goal);
            Term solution = info.getSolution();

        } catch (NoSolutionException nse) {
        }
    } catch (InvalidVarNameException ivne) {
    }
```

Getting another solution:

```
    try {
        SolveInfo info = engine.solve(goal);
        info = engine.solveNext();
    } catch(NoMoreSolutionException e)
```

Loading a library:

```
    try {
        engine.loadLibrary('alice.tuprologx.lib.TucsonLibrary');
    } catch(InvalidLibraryException e) {
    }
```

Here, a complete example of interaction with a tuProlog engine is shown (refer to the JavaDoc documentation for details about interfaces):

```
import alice.tuprolog.*; import java.io.*;

public class Test2P {
    public static void main (String args[]) {
        Prolog engine = new Prolog();
        try {

            // solving a goal
            SolveInfo info = engine.solve(new Struct("append",
                                          new Var("X"),
                                          new Var("Y"),
                                          new Struct(new Term[]{new Struct("hotel"),
                                                                new Int(303),
```

```
                                                                    new Var()})));

            // note we could use strings:
            // SolveInfo info = engine.solve("append(X, Y, [hotel, 303, _]).");

            // test for demonsration success
            if (info.isSuccess()) {

                // acquire solution and substitution
                Term sol = info.getSolution();
                System.out.println("Solution: " + sol);

                System.out.println("Bindings: " + info);

                // open choice points?
                if (engine.hasOpenAlternatives()) {

                    // ask for another solution
                    info = engine.solveNext();

                    if (info.isSuccess()) {
                        System.out.println("An other substitution: " + info);
                    }
                }
            }

            // other frequent interactions

            // setting a new theory in the engine
            String theory = "p(X,Y) :- q(X), r(Y).\n" +
                            "q(1).\n" +
                            "r(1).\n" +
                            "r(2).\n";
            engine.setTheory(new Theory(theory));

            SolveInfo info2 = engine.solve("p(1,X).");
            System.out.println(info2);

            // retrieving the theory from a file
            FileOutputStream os=new FileOutputStream("test.pl");
            os.write(theory.getBytes());
            os.close();
            engine.setTheory(new Theory(new FileInputStream("test.pl")));
            info2 = engine.solve("p(X,X).");
            System.out.println(info2.getSolution());

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

With the program execution, the following string are displayed on the standard output:

```
Solution: append([],[hotel,303,_],[hotel,303,_])
Bindings: Y /[hotel,303,_] X / []
An other substitution: Y / [303,_]  X / [hotel]
X / 1
p(1,1)
```

Chapter 9

How to Develop New Libraries

Libraries are tuProlog's way to achieve the desired characteristics of minimality, dynamic configurability, and straightforward Prolog-to-Java integration. Libraries are reflection-based, and can be written both in Prolog and Java: other languages may be used indirectly, via JNI (Java Native Interface). At the tuProlog side, exploiting a library written in Java requires no pre-declaration of the new built-ins, nor any other special mechanism: all is needed is the presence of the corresponding `.class` library file in the proper location in the file system.

9.1 Implementation details

Syntactically, a library developed in Java must extend the base abstract class `alice.tuprolog.Library`, provided within the tuProlog package, and define new *predicates* and/or *evaluable functors* and/or *directives* in the form of methods, following a simple signature convention. In particular, new predicates must adhere to the signature:

```
public boolean <pred name>_<N>(T1 arg1, T2 arg2, ...,Tn argN)
```

while evaluable functors must follow the form:

```
public Term <eval funct name>_<N>(T1 arg1, T2 arg2, ...,Tn argN)
```

and directives must be provided with the signature:

```
public void <dir name>_<N>(T1 arg1, T2 arg2, ..., Tn argN)
```

where *T1*, *T2*, ... *Tn* are `Term` or derived classes, such as `Struct`, `Var`, `Long`, etc., defined in the tuProlog package, constituting the Java counterparts of the corresponding Prolog data types. The parameters represent the arguments actually passed to the built-in predicate, functor, or directive.

A library defines also a new piece of theory, which is collected by the Prolog engine through a call to the library method `String getTheory()`. By default, this method returns an empty theory: libraries which need to add a Prolog theory must override it accordingly. Note that only the external representation of a library's theory is constrained to be in `String` form; the internal implementation can be freely chosen by the library designer. However, using a Java `String` for wrapping a library's Prolog code guarantees self-containment while loading libraries through remote mechanisms such as RMI.

Table 9.1: Predicate and functor definitions in Java and their use in a tuProlog program.

```
// sample library

import alice.tuprolog.*;

public class TestLibrary extends Library {
  // builtin functor sum(A,B)
  public Term sum_2(Number arg0, Number arg1){
    float arg0 = arg0.floatValue();
    float arg1 = arg1.floatValue();
    return new Float(arg0+arg1);
  }
  // builtin predicate println(Message)
  public boolean println_1(Term arg){
    System.out.println(arg);
    return true;
  }
}
```

```
% tuProlog test
program


test :-
  N is sum(5,6),
  println(N).
```

Table 9.1 shows a couple of examples about how a predicate (such as `println/1`) and an evaluable functor (such as `sum/2`) can be defined in Java and exploited from tuProlog. The Java method `sum_2`, which imple-

ments the evaluable functor `sum/2`, is passed two `Number` terms (5 and 6) which are then used (via `getTerm`) to retrieve the two (float) arguments to be summed. In the same way, method `println_1`, which implements the predicate `println/1`, receives `N` as `arg`, and retrieves its actual value via `getTerm`: since this is a predicate, a boolean value (`true` in this case) is returned.

The developer of a library may face two corner case as far as method naming is concerned: the first happens when the name of the predicate, functor or directive she is defining contains a symbol which cannot legally appear in a Java method's name; the second occurs when he has to define a predicate and a directive with the same Prolog signature, which Java would not be able to tell apart because it cannot distinguish signatures of methods differing for their return type only. To overcome this kind of issues, a *synonym map* can be constructed under the form of an array of `String` arrays, and returned by the appropriate `getSynonymMap` method, defined as abstract by the `Library` class. In both the cases described above, another name must be chosen for the Prolog executable element the library's developer want to define: then, by means of the synonym map, that fake name can be associated with the real name and the type of the element, be it a predicate, a functor or a directive. For example, if a definition for an evaluable functor representing addition is needed, but the symbol `+` cannot appear in a Java method's name, a method called `add` can be defined and associated to its original Prolog name and its function by inserting the array {`"+", "add", "functor"`} in the synonym map.

9.2 Library Name

By default, the name of the library coincides with the full class name of the class implementing it. However, it is possible to define explicitly the name of a library by overriding the `getName` method, and returning as a string the real name. For example:

```
package acme;
import alice.tuprolog.*;
public class MyLib_ver00 extents Library {
    public String getName(){
        return "MyLibrary";
    }
    ...
}
```

This class defines a library called `MyLibrary`. It can be loaded into a Prolog engine by using the `loadLibrary` method on the Java side, or a `load_library` built-in predicate on the Prolog side, specifying the full class name (`acme.MyLib_ve00`). It can be unloaded then dynamically using the `unloadLibrary` method (or the corresponding `unload_library` built-in), specifying instead the *library name* (`MyLibrary`).

Chapter 10

Exceptions

This chapter describes the new support for exceptions in tuProlog.

10.1 Exceptions in ISO Prolog

The ISO Prolog standard (ISO/IEC 13211-1) has been published in 1995. Among the many additions, it introduces the `catch/3` e `throw/1` constructs for exception handling. The first distinction has to be made between *errors* and *exceptions*. An error is a particular circumstance that interrupts the execution of a Prolog program: when a Prolog engine encounters an error, it raises an exception. The exception handling support is supposed to intercept the exception and transfer the execution flow to a suitable exception handler, with any relevant information. Two basic principles are followed during this operation:

- *error bounding* – an error must be bounded and not propagate through the entire program: in particular, an error occurring inside a given component must either be captured at the component's frontier, or remain invisible and be reported nicely. According to ISO Prolog, this can be done via the `catch/3` predicate.
- *atomic jump* – the exception handling mechanism must be able to exit atomically from any number of nested execution contexts. According to ISO Prolog, this is done via the `throw/1` predicate.

In practice, `throw(`*`Error`*`)` raises an exception, while the controlled execution of a goal is launched via the `catch(`*`Goal, Catcher, Handler`*`)` predicate, which is very much like the `try/catch` construct of many imperative

languages. Here, *Goal* is first executed: if an error occurs, the subgoal where the error occurred is replaced by the corresponding `throw(`*Error*`)`, which raises the exception. Then, a matching `catch/3` clause – that is, a clause whose second argument unifies with *Error* – is searched among the antenate nodes in the resolution tree: if one is found, the path in the resolution tree is cut, the catcher itself is removed (because it only applies to the protected goal, not to the handler), and the *Handler* predicate is executed. If, instead, no such matching clause is found, the execution simply fails. So, `catch(`*Goal*`, `*Catcher*`, `*Handler*`)` performs exactly like *Goal* if no exception are raised: otherwise, all the choicepoints generated by *Goal* are cut, a matching *Catcher* is looked for, and if a one is found then *Handler* is executed, maintaining the substitutions made during the previous unification process. In the very end, the execution continues with the subgoal which follows `catch/3`. However, any side effects possibly occurred during the execution of a goal are not undone in case of exceptions, exactly as it normally happens when a predicate fails. Summing up, `catch/3` is true if:

- `call(`*Goal*`)` is true;

  or

- `call(`*Goal*`)` is interrupted by a call to `throw(`*Error*`)` whose *Error* unifies with *Catcher*, and the subsequent `call(`*Handler*`)` is true;

If *Goal* is non-deterministic, it can obviously be executed again in backtracking. However, it should be clear that *Handler* *is possibly executed just once*, since all the choicepoints of *Goal* are cut in case of exception.

10.1.1 Examples

As a first, basic example, let us consider the following toy program:

```
p(X):- throw(error), write('---').
p(X):- write('+++').
```

and let us consider the behaviour of the program in response to the execution of the goal:

```
?:- catch(p(0), E, write(E)), fail.
```

which tries to execute `p(0)`, catching any exception `E` and handling the error by just printing it on the standard output (`write(E)`).

Perhaps surprisingly, the program will just print `'error'`, not `'error---'` or `'error+++'`. The reason is that once the exception is raised, the execution of `p(X)` is aborted, and after the handler terminates the execution proceeds with the subgoal which follows `catch/3`, i.e. `fail`. So, `write('---')` is never reached, nor is `write('+++')` since all the choicepoints are cut upon exception.

In the following we report a small yet complete set of mini-examples, thought to put in evidence one single aspect of tuProlog compliance to the ISO standard.

Example 1: *`Handler` must be executed maintaining the substitutions made during the unification process between `Error` and `Catcher`*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, atom_length(E, Length)).`
Answer: `yes.`
Substitutions: `E/error`, `Length/5`

Example 2: *the selected `Catcher` must be the nearest in the resolution tree whose second argument unifies with `Error`*

Program:
```
p(0) :- throw(error).
p(1).
```
Query: `?- catch(p(1), E, fail), catch(p(0), E, true).`
Answer: `yes.`
Substitutions: `E/error`

Example 3: *execution must fail if an error occurs during a goal execution and there is no matching `catch/3` predicate whose second argument unifies with `Error`*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), error(X), true).`
Answer: `no.`

Example 4: *execution must fail if `Handler` is false*

Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, false).`
Answer: `no.`

Example 5: *if `Goal` is non-deterministic, it is executed again on backtracking, but in case of exception all the choicepoints must be cut, and `Handler` must be executed only once*

Program:
```
p(0).
p(1) :- throw(error).
p(2).
```

Query: `?- catch(p(X), E, true).`
Answer: `yes.`
Substitutions: `X/0`, `E/error`
Choice: `Next solution?`
Answer: `yes.`
Substitutions: `X/1`, `E/error`
Choice: `Next solution?`
Answer: `no.`

Example 6: *execution must fail if an exception occurs in* `Handler`
Program: `p(0) :- throw(error).`
Query: `?- catch(p(0), E, throw(err)).`
Answer: `no.`

10.1.2 Error classification

So far we have just said that, when an exception is raised, `throw(`*`Error`*`)` is executed, and a matching `catch/3` is looked for, but no specifications have been given about the possible structure of the *`Error`* term. According to the ISO Prolog standard, such a term should follow the pattern `error(`*`Error_term`*`,` *`Implementation_defined_term`*`)` where *`Error_term`* is constrained by the standard to a pre-defined set of possible values, in order to represent the error category: *`Implementation_defined_term`*, instead, is left for implementation-specific details, and could also be omitted.

The error classification induced by *`Error_term`* is flat, so as to easily support pattern matching. Ten error classes are identified by the ISO standard:

1. `instantiation_error`: when the argument of a predicate or one of its components is a variable, while it should be instantiated. A typical example is `X is Y+1` if `Y` is not instantiated when `is/2` is evaluated.

2. `type_error(`*`ValidType`*`,` *`Culprit`*`)`: when the type of an argument of a predicate, or one of its components, is instantiated, but nevertheless incorrect. In this case, *`ValidType`* represents the expected data type (one of `atom`, `atomic`, `byte`, `callable`, `character`, `evaluable`, `in_byte`, `in_character`, `integer`, `list`, `number`, `predicate_indicator`, `variable`), while *`Culprit`* is the wrong type found. For instance, if a predicate operates on dates and expects months to be represented as integers between 1-12, calling the predicate with an argument like `march` instead of `3` would raise a `type_error(integer, march)`, since an integer was expected and `march` was found instead.

3. `domain_error(`*`ValidDomain, Culprit`*`)`: when the argument type is correct, but its value falls outside the expected range. *`ValidDomain`* is one of `character_code_list`, `close_option`, `flag_value`, `not_empty_list`, `not_less_than_zero`, `io_mode`, `operator_priority`, `operator_specifier`, `prolog_flag`, `read_option`, `source_sink`, `stream`, `stream_option`, `stream_or_alias`, `stream_position`, `stream_property`, `write_option`. In the example above, a domain error could be raised if, for instance, a value like 13 was provided for the month argument.

4. `existence_error(`*`ObjectType, ObjectName`*: when the referenced object to be accessed does not exist. Again, *`ObjectType`* is the type of the unexisting object, and *`ObjectName`* its name. *`ObjectType`* is `procedure`, `source_sink`, or `stream`. If, for instance, the file `'usr/goofy'` does not exist, an `existence_error(stream, 'usr/goofy')` would be raised.

5. `permission_error(`*`Operation, ObjectType, Object`*`)`: when *`Operation`* is not allowed on *`Object`*, which is of type *`ObjectType`*. *`Operation`* is one of `access`, `create`, `input`, `modify`, `open`, `output`, or `reposition`, while *`ObjectType`* falls among `binary_stream`, `operator`, `past_end_of_stream`, `private_procedure`, `static_procedure`, `source_sink`, `stream`, `flag`, and `text_stream`.

6. `representation_error(`*`Flag`*`)`: when an implementation-defined limit, whose category is given by *`Flag`*, is violated during execution. *`Flag`* is one of `character`, `character_code`, `in_character_code`, `max_arity`, `max_integer`, `min_integer`.

7. `evaluation_error(`*`Error`*`)`: when the evaluation of a function produces an exceptional value. Accordingly, *`Error`* is one of `float_overflow`, `int_overflow`, `undefined`, `underflow`, `zero_divisor`.

8. `resource_error(`*`Resource`*`)`: when the Prolog engine does not have enough resources to complete the execution of the current goal. Typical examples are the reach of the maximum number of opened files, no further available memory, etc. Accordingly, `resource_error(Resource)` can be any valid term.

9. `syntax_error(`*`Message`*`)`: when external data, read from an external source, have an incorrect format or cannot be processed for some reason. This kind of error typically occurs during *read* operations.

Message can be any valid (simple or compound) term describing the occurred problem.

10. system_error: this latter category represents any other unexpected error which does not fall in any of the above categories.

10.2 Implementing Exceptions in tuProlog

Implementing exceptions in tuProlog does not mean just to extend the engine to support the above mechanisms: given its library-based design, and its intrinsic support to multi-paradigm programming, adding exceptions in tuProlog has also meant (1) to revise all the existing libraries, modifying any library predicate so that it raises the appropriate type of exception instead of just failing; and (2) to carefully define and implement a model to make Prolog exceptions not only coexist, but also fruitfully operate with the Java (or C#/.NET) imperative world, which brings its own concept of exception and its own handling mechanism.

As a preliminary step, the finite-state machine which constitutes the core of the tuProlog engine was extended with a new *Exception* state, between the existing *Goal Evaluation* and *Goal Selection* states [**?**].

Then, all the tuProlog libraries were revised, according to clearness and efficiency criteria — that is, the introduction of the new checks required for proper exception raising should not reduce performance unacceptably. This issue was particularly relevant for runtime checks, such as existence_errors or evaluation_errors; moreover, since tuProlog libraries could also be implemented partly in Prolog and partly in Java, careful choices had to be made so as to introduce such checks at the most adequate level in order to intercept all errors while maintaining code readability and overall organisation, while guaranteeing efficiency. This led to intervene with extra Java checks for libraries fully implemented in Java, and with new "Java guards" for predicates implemented in Prolog, keeping the use of Prolog meta-predicates (such as integer/1) to a minimum.

With respect to the third aspect, which will be discussed more in depth below, one key aspect to be put in evidence right now concerns the handling of Java objects accessed from the Prolog world via Javalibrary. At a first sight, one might think of re-mapping Java exceptions and constructs onto the Prolog one, but this approach is unsatisfactory for three main reasons:

- the semantics of the Java mechanism should not be mixed with the Prolog one, and vice-versa;

- the Java construct admits also a `finally` clause which has no counterpart in ISO Prolog;
- the Java catching mechanisms operates hierarchically, while the `catch/3` predicate operates via pattern matching and unification, allowing for multiple granularities.

For these reasons, supporting Java exceptions from tuProlog programs called for two further, *ad hoc* predicates *which are not present in ISO Prolog* because ISO Prolog does not consider multi-paradigm programming: `java_throw/1` and `java_catch/3`.

10.2.1 Java exceptions from tuProlog

The `java_throw/1` predicate has the form

```
java_throw(JavaException(Cause, Message, StackTrace))
```

where *JavaException* is named after the specific Java exception to be launched (e.g., `'java.io.FileNotFoundException'`, and its three arguments represent the typical properties of any Java exception. More precisely, *`Cause`* is a string representing the cause of the exception, or `0` if the cause is unknown; *`Message`* is the message associated to the error (or, again, `0` if the message is missing); *`StackTrace`* is a list of strings, each representing a stack frame.

The `java_catch/3` predicate takes the form

```
java_catch(JGoal, [(Catcher1, Handler1),
                   ...,
                   (CatcherN, HandlerN)], Finally)
```

where *`JGoal`* is the goal (representing a Java operation in the Java world) to be executed under the protection of the handlers specified in the subsequent list, each associated to a given type of Java exception and expressed in the form `java_exception(`*`Cause, Message, StackTrace`*`)`, with the same argument semantics explained above. The third argument *`Finally`* expresses the homonomous Java clause, and therefore represents the predicate to be executed at the very end either of the *`Goal`* or one of the *`Handler`*s. If no such a clause is actually needed, the conventional atom (`'0'`) has to be used as a placeholder.

The predicate behaviour can be informally expressed as follows. First, *`JGoal`* is executed. Then, if no exception is raised via `java_throw/1`, the *`Finally`* goal is executed. If, instead, an exception is raised, all the choicepoints generated by *`JGoal`* (in the case of a non-deterministic predicate like

java_object_bt/3, of course) are cut: if a matching handler exists, such a handler is executed, maintaining the variable substitutions. If, instead, no such a handler is found, the resolution tree is backsearched, looking for a matching java_catch/3 clause: if none exists, the predicate fails. Upon completion, the *Finally* part is executed anyway, then the program flow continues with the subgoal following java_catch/3. As already said above, side effects possibly generated during the execution of *JGoal* are *not* undone in case of exception.

So, summing up, java_catch/3 is true if:

- *JGoal* and *Finally* are both true;

  or

- call(*JGoal*) is interrupted by a call to java_throw/1 whose argument unifies with one of the *Catcher*s, and both the execution of the catcher and of the *Finally* clause are true.

Even if *JGoal* is a non-deterministic predicate, like java_object_bt/3, and therefore the goal itself can be re-executed in backtracking, in case of exception only one handler is executed, then all the choicepoints generated by *JGoal* are removed: so, no further handler would ever be executed for that exception. In other words, java_catch/3 only protects the execution of *JGoal*, *not* the handler execution or the *Finally* execution.

10.2.2 Examples

First, let us consider the following program:

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
       [('java.lang.ClassNotFoundException'(Cause, Msg, StackTrace),
         write(Msg))],
       write(+++)).
```

which tries to allocate an instance of Counter, bind it to the atom c, and – if everything goes well – print the '+++' message on the standard output. Indeed, this is precisely what happens if, at runtime, the class Counter is actually available in the file system. However, it might also happen that, for some reason, the required class is *not* present in the file system when the above predicate is executed. Then, a 'java.lang.ClassNotFoundException'(Cause, Msg, StackTrace) exception is raised, no side effects occur – so, no object is actually created – and the *Msg* is printed on the standard output, followed

by '+++' as required by the *Finally* clause. Since the *Msg* in this exception is the name of the missing class, the global message printed on the console is Counter+++.

In the following we report a small yet complete set of mini-examples, thought to put in evidence one single aspect of tuProlog compliance to the ISO standard.

Example 1: *the handler must be executed maintaining the substitutions made during the unification process between the exception and the catcher: then, the* *Finally* *part must be executed.*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
      [('java.lang.ClassNotFoundException'(Cause, Message, _),
        X is 2+3)], Y is 2+5).
```

Answer: yes.
Substitutions: Cause/0, Message/'Counter', X/5, Y/7.

Example 2: *the selected* *java_catch/3* *must be the nearest in the resolution tree whose second argument unifies with the launched exception*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
      [('java.lang.ClassNotFoundException'(Cause, Message, _),
        true], true),
   java_catch(java_object('Counter', ['MyCounter2'], c2),
      [('java.lang.ClassNotFoundException'(Cause2, Message2, _),
        X is 2+3], true).
```

Answer: yes.
Substitutions: Cause/0, Message/'Counter', X/5, C/0, Message2/'Counter'.

Example 3: *execution must fail if an exception is raised during the execution of a goal and no matching* *java_catch/3* *can be found*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.Exception'(Cause, Message, _), true)], true)).
```

Answer: no.

Example 4: *java_catch/3 must fail if the handler is false*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.Exception'(Cause, Message, _), false)], true)).
```

Answer: `no.`

Example 5: *`java_catch/3` must fail also if an exception is raised during the execution of the handler*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
     [('java.lang.ClassNotFoundException'(Cause, Message, _),
       java_object('Counter', ['MyCounter'], c))], true).
```

Answer: `no.`

Example 6: *the `Finally` must be executed also in case of success of the goal*

```
?- java_catch(java_object('java.util.ArrayList', [], l),
      [E, true], X is 2+3).
```

Answer: `yes.`
Substitutions: `X/5`.

Example 7: *the `Handler` to be executed must be the proper one among those available in the handlers' list*

```
?- java_catch(java_object('Counter', ['MyCounter'], c),
    [('java.lang.Exception'(Cause, Message, _), X is 2+3),
     ('java.lang.ClassNotFoundException'(Cause, Message, _), Y is 3+5)],
     true).
```

Answer: `yes.`
Substitutions: `Cause/0`, `Message/'Counter'`, `Y/8`.