tuProlog Manual

tuProlog version: 2.5

Last Changes date: 2012-06-26

Alma Mater Studiorum—Università di Bologna, Italy

Contents

Chapter 1

What is tuProlog

tuProlog is an open-source, light-weight Prolog framework for distributed applications and infrastructures, released under the LGPL license, available from `http://tuprolog.apice.unibo.it`.

Originally developed in/upon Java, which still remains the main reference platform, tuProlog is currently available for several platforms/environments:

- plain JavaSE;
- Eclipse plugin;
- Android;
- Microsoft .NET.

While they all share the same core and libraries, the latter features an *ad hoc* library which extends the multi-paradigm approach to virtually any language available on the .NET platform (more on this in Section Chapter **??**).

Unlike most Prolog programming environments, aimed at providing a very efficient (yet monolithic) stand-alone Prolog system, tuProlog is explicitly designed to be *minimal*, dynamically *configurable*, straightforwardly *integrated* with Java and .NET so as to naturally support multi-paradigm/multi-language programming (MPP), and *easily deployable*.

Minimality means that its core contains only the Prolog engine essentials – roughly speaking, the resolution engine and some related basic mechanisms – for as little as 155KB: any other feature is implemented in *libraries*. So, each user can customize his/her prolog system to fit his/her own needs, and no more: this is what we mean by tuProlog *configurability*—the necessary counterpart of minimality.

Libraries provide packages of predicates, functors and operators, and can be loaded and unloaded in a tuProlog engine both statically and dynamically. Several standard libraries are included in the tuProlog distribution, and are loaded by default in the standard tuProlog configuration; however, users can easily develop their own libraries either in several ways – just pure Prolog, just pure Java[1], or a mix of the two –, as we will discuss in Chapter **??**.

Multi-paradigm programming is another key feature of tuProlog. In fact, the tuProlog design was intentionally calibrated from the early stages to support a straightforward, pervasive, multi-language/multi-paradigm integration, so as to enable users to:

- using any Java[2] class, library, object *directly from the Prolog code* (Chapter **??**) with no need of pre-declarations, awkward syntax, etc., with full support of parameter passing from the two worlds, yet leaving the two languages and computational models totally separate so as to preserve *a priori* their own semantics—thus bringing the power of the object-oriented platform (e.g. Java Swing, JDBC, etc) to the Prolog world for free;

- using any Prolog engine *directly from the Java/.NET code* as one would do with any other Java libraries/.NET assemblies, again with full support of parameter passing from the two worlds in a non-intrusive, simple way that does not alter any semantics—thus bringing the power of logic programming into virtually *any* Java/.NET application;

- augmenting Prolog by defining new libraries Chapter **??**) either in Prolog, or in the object-oriented language of the selected platform (again, with a straightforward, easy-to-use approach based on reflection which avoids any pre-declaration, language-to-language mapping, etc), or in a mix of both;

- augmenting Java[3] by defining new Java methods in Prolog (the so-called 'P@J' framework—Chapter **??**), which exploits reflection and type inference to provide the user with an easy-to-use way to implement Java methods declaratively.

[1]The .NET version of tuProlog supports other languages available on the .NET platform: more on this topic in Section Chapter **??**

[2]For the .NET version: any .NET class, library, object, etc.

[3]This feature is currently available only in the Java version: a suitable extension to the .NET platform is under study.

Last but not least, *easy deployability* means that the installation requirements are minimal, and that the installation procedure is in most cases[4] as simple as copying one archive to the desired folder. Coherently, a Java-based installation requires only a suitable Java Virtual Machine, and 'installing' is just copying a single JAR file somewhere—for as much as 474KB of disk usage (yes, minimality is not just a claim here). Of course, other components can be added (documentation, extra libraries, sources..), but are not necessary for a standard everyday use. The file size is quite similar for the Android platform – the single APK archive is 234KB – although an Android-compliant install is performed due to Android requirements. The install process is also quite the same on the .NET platform, although the files are slightly larger. The Eclipse platform also requires a different procedure, since plugin installation have to conform to the requirements of the Eclipse plugin manager: consequently, an update site was set up, where the tuProlog plugin is available as an Eclipse feature. Due to these constraints, file size increases to 1.5MB.

In order to manage all these platforms in a uniform way, a suitable *version numbering scheme* was recently introduced:

- the fist two digits represent the engine version;
- the last (third) digit is platform-specific and accounts for version differences which do not impact on the Prolog engine – that is, on the tuProlog behaviour – but simply on graphical aspects or platform-specific issues or bugs.

So, as long as the first two digits are the same, a tuProlog application is guaranteed to behave identically on any supported platform.

Finally, tuProlog also supports *interoperability* with both Internet standard patterns (such as TCP/IP, RMI, CORBA) and coordination models and languages. The latter aspect, in particular, is currently developed in the context of the TuCSoN coordination infrastructure [5, 4], which provides logic-based, programmable tuple spaces (called *tuple centres*) as the coordination media for distributed processes and agents.[5]

[4] Exceptions are the Eclipse plugin and the Android versions, which need to be installed as required by the hosting platforms.

[5] An alternative infrastructure, LuCe [3], developed the same approach in a location-unaware fashion: this infrastructure is currently no longer supported.

Chapter 2

Installing tuProlog

Quite obviously, the installation procedure depends on the platform of choice. For Java, Microsoft .NET and Android, the first step is to manually download the desired distribution (or even just the single binary file) from the tuProlog web site, `tuprolog.alice.unibo.it`, or directly from the Google code repository, `tuprolog.googlecode.com`; for Eclipse the procedure is different, since the plug-in installation has to be performed via the Eclipse Plugin Manager.

As a further alternative, users wishing to have a look at tuProlog and trying it without installing anything on their computer can do so by exploiting the 'Run via Java Web Start' option, available on the tuProlog web site.

2.1 Installation in Java

The Java distribution has the form of a single `zip` file which contains everything (binaries, sources, documentation, examples, etc.) and unzips into a multi-level directory tree, similar to the following (only first-level sub-dirs are shown):

```
2p
|---ant
|---bin
|---build
|   |---archives
|   |---classes
|   |---release
|   |---reports
|   |---tests
|---doc
|   |---javadoc
|---lib
|---src
|---test
|   |---fit
|   |---unit
|---tmp
|---test
```

If you are only interested in the Java binaries, just look into the `build/archives` directory, which contains two JAR files:

- `2p.jar`, which contains everything you need to use tuProlog, such as the core API, the `Agent` application, libraries, GUI, etc.; this is a runnable JAR, that open the tuProlog IDE when double-clicked.
- `tuprolog.jar`, which contains only the core part of tuProlog, namely, what you will need to include in a Java application project to be able to access the tuProlog classes, and write multi-paradigm Java/Prolog applications.

The other folders contain project-specific files: `src` contains all the sources, `doc` all the documentation, `lib` the libraries used by the tuProlog project, `test` the sources for the tuProlog test suite (partly as FIT test, partly as JUnit tests), `ant` some Ant scripts to automate the build of parts of the tuProlog project, etc.

2.2 Installation in .NET

The .NET distribution has also the form of a single `zip` file containing everything; however, due to the automatic generation of `tu`Prolog .NET binaries via IKVM from Java (more on this in Section Chapter **??**), the unzipped directory tree is much simpler, as there are no sources (and therefore no tests, no ant tasks, etc), except for `OOLibrary`, which is .NET-specific and therefore is written in C#. So, the resulting tree is similar to the following:

```
2p
|---build
|   |---examples
|   |---lib
|---OOLibrary
|   |---Conventions
|   |---Fixtures
|   |---OOLibrary
```

The .NET binary, `2p.exe`, can be found in the `build` folder.

2.3 Installation in Android

The Android distribution has the form of a single `apk` file, to be installed via install mechanism provided by the Android OS. So, unless you are interested in the implementation details, there should be no need to download the whole project distribution. If, however, you like to do so, you will eventually get to a directory tree similar to the following (only the most relevant first-level sub-folders are shown):

```
2p
|---assets
|---bin
|   |---classes
|   |---res
|---doc
|---gen
|---lib
|---res
|---screenshots
|---src
```

The APK binary can be found into the `bin` folder.

As for the Java case, the other folders contain project-specific files: in particular, `src` contains the sources, `res` the Android resources automatically generated during the project build process, `lib` the libraries used by this project—mainly, the `tuprolog.jar` file of the corresponding Java version, imported here as an external dependency.

2.4 Installation in Eclipse

The installation procedure is different for the Eclipse platform due to the need to conform to the Eclipse standard procedure for plug-in installation via Plugin Manager. Please see the specific section on the tuProlog web site for detailed, screenshot-driven instruction.

Chapter 3

Getting Started

tuProlog can be enjoyed from different perspectives:

1. as a *Prolog user*, you can exploit its Integrated Development Environment (IDE) and Graphical User Interface (GUI) to consult, edit, and run Prolog programs, as you would do with any other Prolog system—and you can do so in any of the supported platforms (Java, .NET, Android, Eclipse).

2. as a *Java user*, you can include tuProlog in any Java project, thus bringing the power of Artificial Intelligence to the Java world; the tuProlog API provides many classes and methods for exchanging data between the Java and the Prolog worlds. Though you can do so using your preferred IDE, the tuProlog plugin for the Eclipse platform is probably the most practical choice for this purpose, as the tuProlog perspective provides all the views over the Prolog world in an Eclipse-compliant, effective way.

3. as a *.NET user*, analogously, you can add tuProlog to any Visual Studio project (including the related IKVM libraries, as detailed in chapter **??**, or just manually compile your .NET application with the necessary DLL files in the build path. The tuProlog API, which is nearly identical to the Java one, provides for proper data exchange between the .NET and the Prolog worlds.

4. finally, as an *Android user*, you can both enjoy the tuProlog app to consult, edit, and run Prolog programs, as you would do with any other Prolog system, and –perhaps more interestingly– exploit the tuProlog Java API for developing Android applications, adding intelligence to your next Android app.

3.1 tuProlog for the Prolog User

As a Prolog user/programmer, you might want to start running your existing programs. There are three ways to do so:

- by using the graphical tuProlog GUI (both in Java and .NET)
- by using the console-based tuProlog CUI (Java only)
- by using the `Agent` class to execute a Prolog program in a 'batch' form—that is, running the program provided as a text file (Java only).

The first two forms are rather obvious: after starting the GUI/CUI, you will get a rather standard graphical/character-based Prolog user interface (Figure 3.1).

The GUI includes an editing pane with syntax highlighting, a toolbar providing facilities to load/save/create theories, load/unload libraries, and show/hide the the debug information window; at the bottom, the status bar provides information, as detailed below.

The GUI can be launched either by double-clicking the tuProlog executable (`2p.jar` in Java, `2p.exe` in .NET), or by manually issuing the commands

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

or

```
2p.exe
```

in .NET, respectively.

Analogously, the command-line CUIconsole (available in Java only) can be launched by issuing the command:

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

The CUIconsole can be quitted issuing the standard `halt.` command.

The third form, available in Java only, is basically an auxiliary tool to batch-execute a Prolog program: it takes the name of a text file containing a Prolog theory as its first (mandatory) argument and optionally the goal to be solved as its second argument, then starts a new Prolog virtual machine, performs the demonstration, and ends. The `Agent` tool is invoked from the command line as follows:

```
java -cp dir/2p.jar alice.tuprolog.Agent theoryfile {goal}
```

For instance, if the file `hello.pl` contains the mini-theory:

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

the following command causes its execution:

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

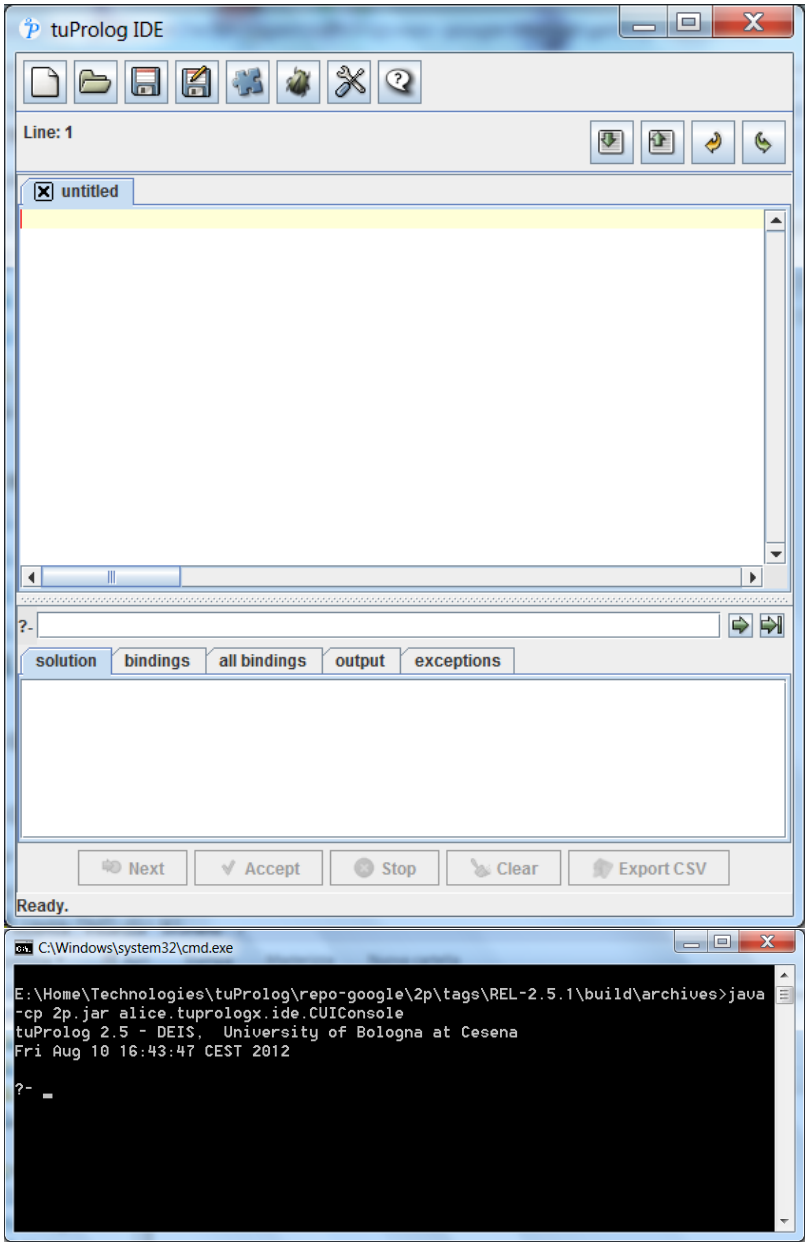

Figure 3.1: The standard tuProlog GUI and CUI.

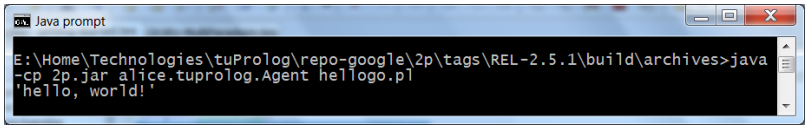

Figure 3.2: The tuProlog Agent tool.

resulting in the string `hello, world!` being printed on the standard output. Alternatively, the goal to be proven can be embedded in the Prolog source by means of the `solve` directive, as follows (Figure 3.2):

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

Quite obviously, in this case no second argument is required.

3.1.1 Editing theories

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.

The toolbar contains four buttons: two are used to upload/download a theory to/from the Prolog engine, two support the classical Undo/Redo actions. Explicit uploading/downloading of theories to/from the Prolog engine is a consequence of tuProlog's choice to maintain a clear separation between the engine and the currently-viewed theories: in this way,

- theories can be edited without affecting the engine content: they can also be in an inconsistent state, since syntax checking is performed only upon loading;
- changes in the current database performed by the Prolog program via the `assert/retract` do not affect the theory shown in the editor, which maintains the original user theory.

Accordingly, the *set theory* button uploads the text in the editor window to the engine, while the *get theory* button downloads the current engine theory (possibly changed by the program) from the engine to a new editor tab.

However, for the user convenience, a logical shortcut is provided that automatically uploads the current theory to the engine whenever a new query is issued: obviously, if the theory is invalid, the query will not be executed.

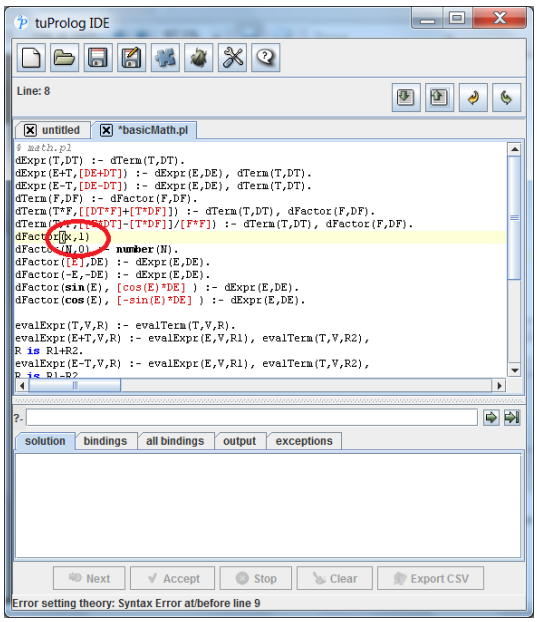

Figure 3.3: Syntax error found when setting a theory

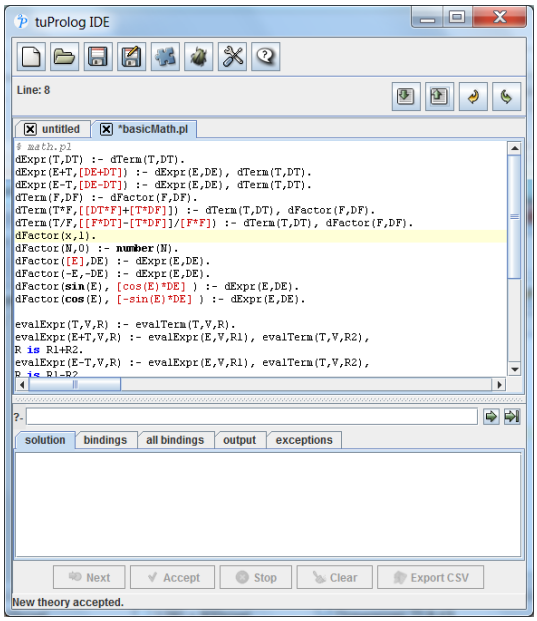

Figure 3.4: Set theory operation succeeded

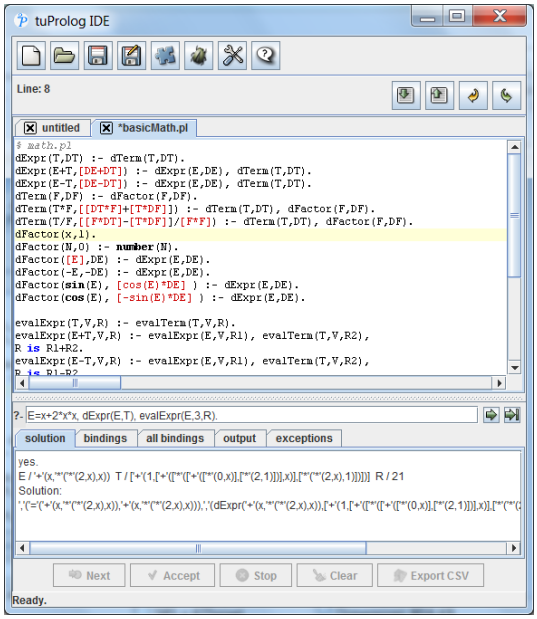

Figure 3.5: The solutions tab showing the query solution.

Manual uploading is still needed whenever the theory in the editor window is modified via other other means than the built-in editor—for instance, after a `consult/1` goal, or via other editors.

The status bar at the bottom of the window reports information such as the cursor line number or syntax errors when setting an invalid theory. For instance, Figure 3.3 shows the error message due to a missing dot at line 8, while Figure 3.4 shows the status message after the error has been corrected, and the theory successfully uploaded.

3.1.2 Solving goals

The console at the bottom of the window contains the *query textfield* and a multi-purpose, tabbed information panel.

The *query textfield* is where to write and execute queries: the leftmost (Solve) button triggers the engine to find the first (and then the subsequent) solution(s) interactively, while the rightmost (Solve All) button forces the engine to find all the solutions at once. Pressing the **Enter** key in the textfield has the same effect as pressing the Solve button.

The subsequent area below contains five panes:

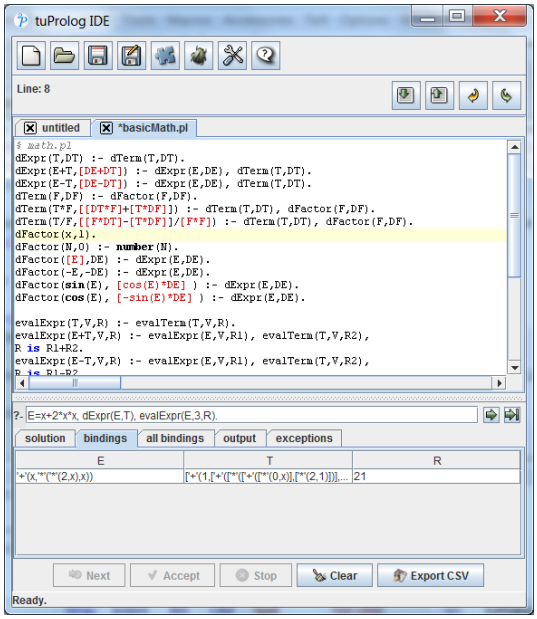

Figure 3.6: The bindings tab showing the bindings of query solution.

- the *solution pane* shows the query solutions (see Figure 3.5): proper control buttons are provided to iterate through multiple solutions;
- the *binding* and the *all bindings* panes show the variable bindings in tabular form, for a single solution or for all solutions, respectively (see Figure 3.6); here, too, proper control buttons are provided to clear the bindings pane and export the tabular data in a convenient CSV format;
- the *output pane* shows the output performed by the program via `write` and other console I/O predicates (Figure 3.7). Please note that output performed by Java methods – that is, methods invoked on Java objects via `JavaLibrary` – are *not* captured and displayed in this view: for further information on this topic, refer to Chapter **??**. Again, control buttons are provided to clear the output pane.
- the *exceptions* pane shows the exceptions raised during the query demonstration: if exceptions are triggered, it gains focus automatically and is color-highlighted for the user convenience (Figure 3.8).

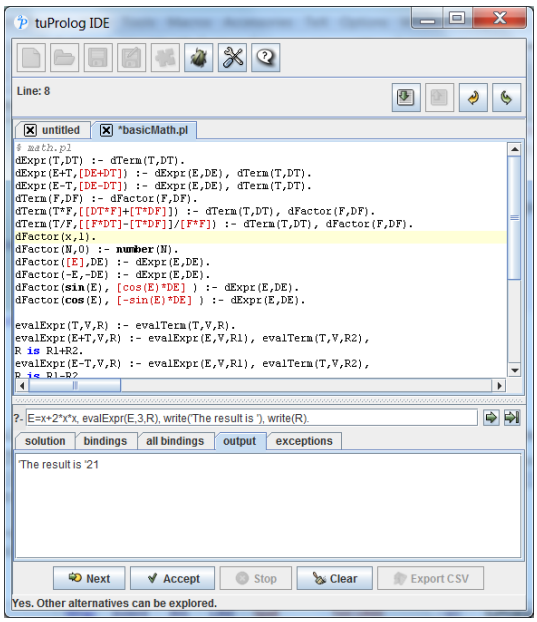

Figure 3.7: The output tab showing the query printing.

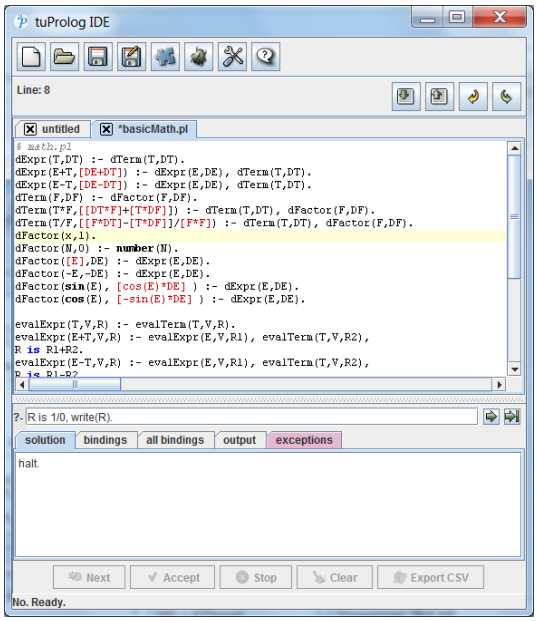

Figure 3.8: The exceptions tab gaining focus and showing raised exceptions.

Query and answers are stored in chronological order, and can be explored by means of **Up** and **Down** arrow keys from the query input textfield.

The Stop button makes it possible to stop the engine if a computation takes too long or a bug in the theory is causing an infinite loop. (Note: like most Prolog systems, tuProlog does not normally perform *occur check*, for performance reasons; this check can be enabled via the proper built-in predicate—see Section 5.1 for details.)

3.1.3 Debugging support

Debug support in tuProlog is actually limited compared to other professional Prolog systems: however, *warnings* and *spy information* are available.

To this end, the View Debug Information button opens the Debug window which lists *i)* all the warnings, produced by events such as the attempt of redefining a library predicate, and *ii)* the step-by-step spy information of the engine computation during a goal demonstration.

Warnings are always active, while spy notification has to be explicitly enabled (and disabled) via the built-in `spy/0` (`nospy/0`) predicate. Figure 3.9 shows an example of spy information for a goal: by default, information is presented in a collapsed form, but single nodes (or all the nodes) can be expanded using the toolbar buttons, to access more detailed information.

3.1.4 Dynamic library management

As anticipated above, tuProlog engines are dynamically extensible via *libraries*: each library can provide its own set of new built-in predicates and functors, as well as a related theory. By default, the standard set of libraries is loaded into any newly-created engine, but the library set of each engine can be easily modified via the *Library Manager*, which is displayed by pressing the Open Library Manager button in the toolbar (Figure 3.10).

This dialog displays the list of the currently loaded libraries—by default, `BasicLibrary`, `IOLibrary`, `ISOLibrary`, `JavaLibrary`. Other libraries can be added by providing the fully qualified name of the library class in the textfield, and pressing the Add button: the added library will be displayed with an initial *Unloaded* status. Please note that any further class needed by a library must be in the system classpath, or the library will not be added to the manager/loaded into the engine.

The library manager takes into account the effects of the `load_library/1` and `unload_library/1` predicates/directives, too: so, for instance, after a

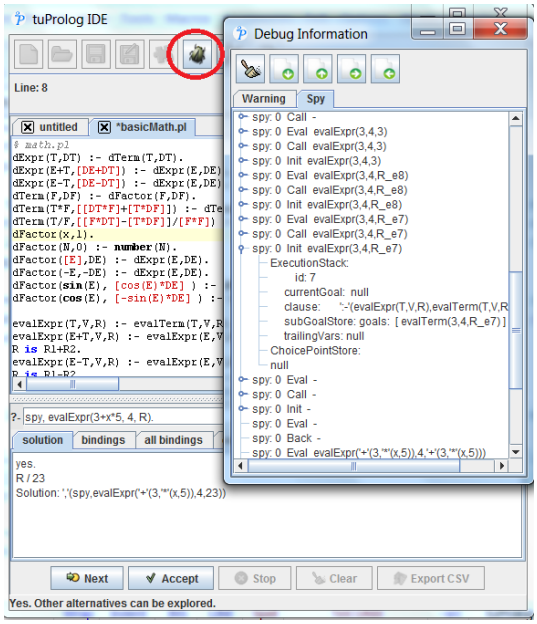

Figure 3.9: Debug Information View after the execution of a goal.

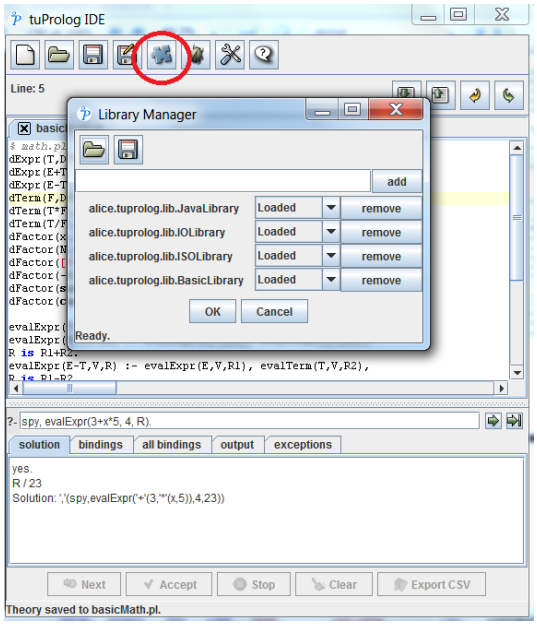

Figure 3.10: The Library Manager window.

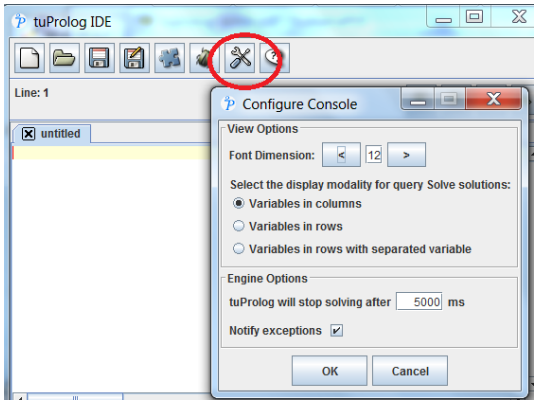

Figure 3.11: The configuration window.

goal such as `load_library('TestLibrary'), test(X).`, a new entry for `TestLibrary` would be displayed.

If the addition of a library to the manager or its loading into the engine fails (for instance, due to an invalid class name, or a class not extending the `alice.tuprolog.Library` class, etc.), an error message will be displayed in the status bar.

Finally, the config button opens the configuration dialog (Figure 3.11), which provides access to a set of options and tunings.

3.2 tuProlog for the Java Developer

As anticipated above, the Java developer can include tuProlog in any of his projects, exploiting the tuProlog API to access the Prolog engine(s) from his Java program.. The easiest way to do so it to exploit the Java plugin available for the Eclipse IDE, which adds a specific *tuprolog perspective* specifically suited for the needs of the Java/Prolog user (Figure 3.12).

This perspective is mainly designed to support the development of multi-language, multi-paradigm applications (see Chapter **??**), but can also be used as a standard Prolog console, writing (or loading) the Prolog theory in the editor and writing the query in the proper textfield—although the direct use of the tuProlog GUI is probably faster for this purpose.

To use tuProlog in Eclipse, one first needs to create a new tuProlog project, and add a new theory file (`*.pl`) to the project. To this end:

- either select `New` > `Project` from the Package Explorer's context menu, then select the `tuProlog` item;

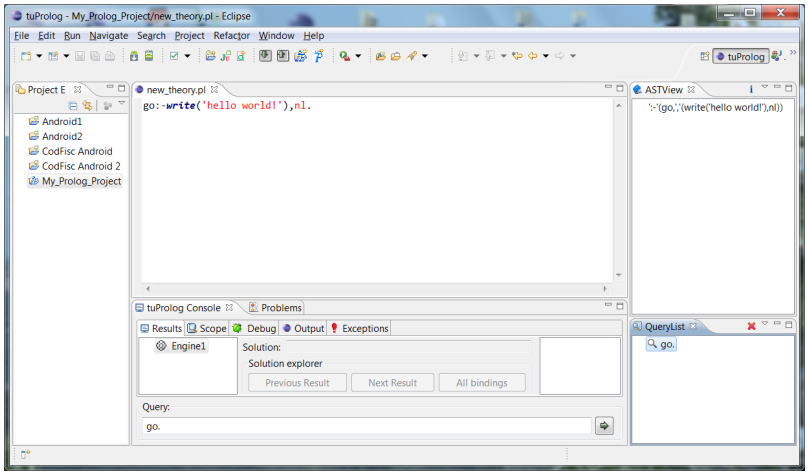

Figure 3.12: The tuProlog plugin GUI for Eclipse.

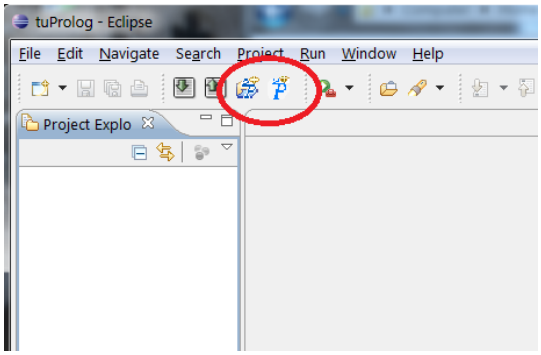

Figure 3.13: The tuProlog toolbar

- or, select `File` > `New` > `Other` > `tuProlog` > `tuProlog Project` from the main menu;
- or, press the *New tuProlog Project* buttons in the tuProlog toolbar (Figure 3.13.

In any case, a dialog appears (Figure 3.14) which prompts for the project name (default: `My_Prolog_Project`) and the desired Prolog libraries (the default set is proposed).

Pressing the *New tuProlog File* button, a dialog appears which asks for the theory name (default: `new_theory.pl`) and the file container, i.e. the tuProlog project where the new file has to be added (Figure 3.15); this is

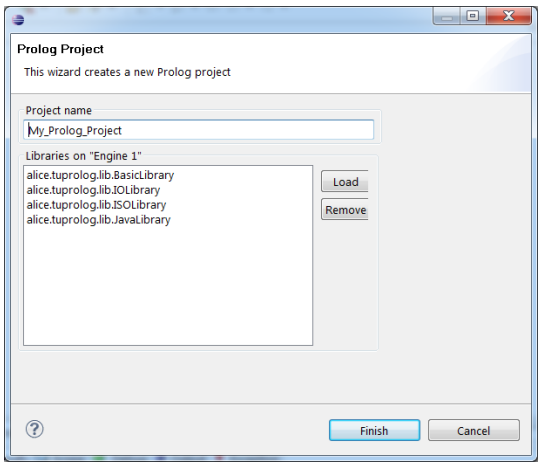

Figure 3.14: new tuProlog project

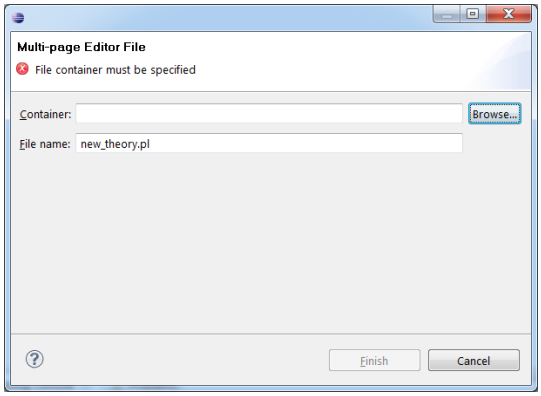

Figure 3.15: new tuProlog file

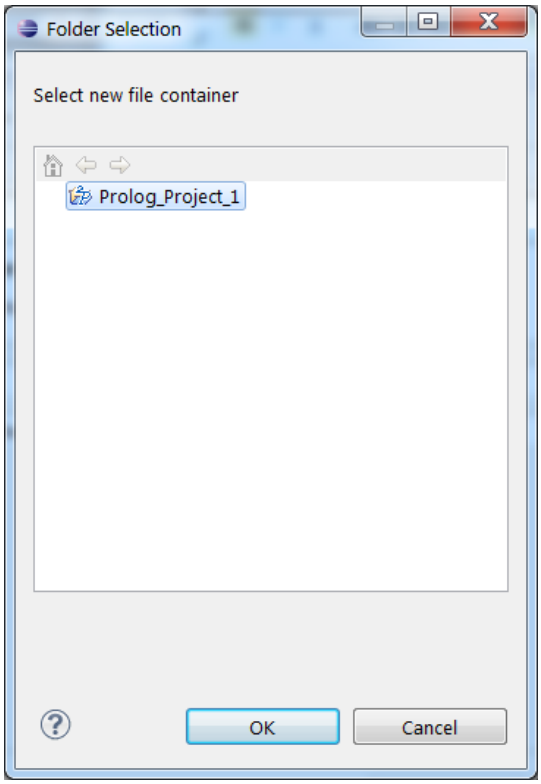

Figure 3.16: new tuProlog file > Browse...

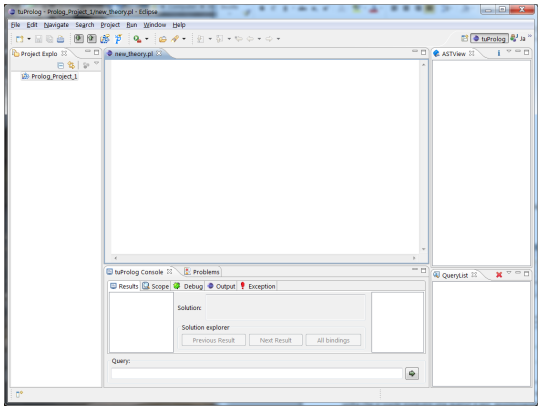

Figure 3.17: the tuProlog perspective

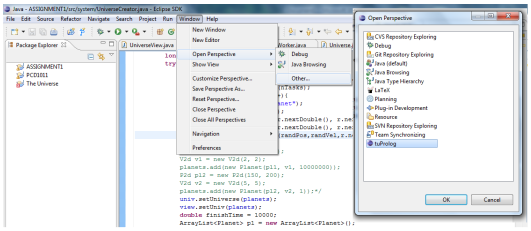

Figure 3.18: opening the tuProlog perspective

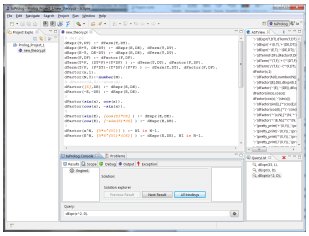

Figure 3.19: executing queries, available views

a mandatory argument. Pressing the *Browse..* button, a new dialog proposes the current tuProlog projects (Figure 3.16); again, the same result can be achieved via menu selection (`File` > `New` > `Other` > `tuProlog` > `tuProlog Theory`). After confirming, the tuProlog perspective automatically opens (Figure 3.17). Again, the same result can be achieved via the `Window` > `Open Perspective` menu (Figure 3.18).

Once the theory has been written (or loaded), the theory file must be saved, either clicking the save icon in the toolbar, or choosing the `File` > `Save` option, or hitting CTRL+S on the keyboard; this is mandatory before issuing any query. The query can be written in the bottom console, and is executed either by pressing the Enter key, or by clicking the *Solve* button.

The query results are shown in different views (Figure 3.19):

- the tuProlog Console view reports the query results: the variable bindings are also available pressing the *All bindings* button (Figure 3.20).
- the Output view shows the program output messages;
- the QueryList view on the left side reports the list of all he executed queries, which can then be re-selected and re-executed in a click;
- the AST view shows the (dynamic) set of current clauses: pressing the `i` icon, a graphical view of the Abstract Syntax Tree produced by the

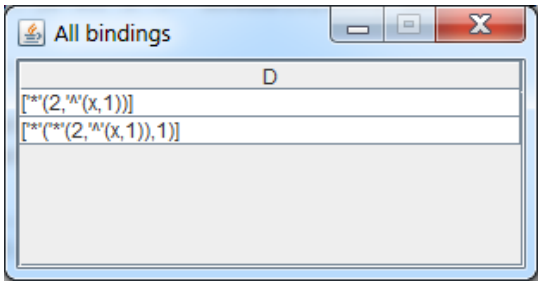

Figure 3.20: all variable bindings

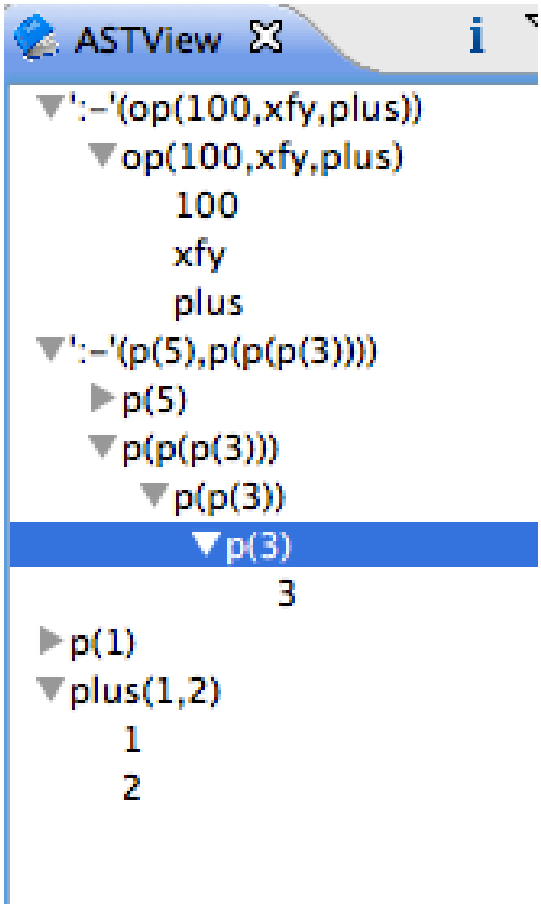

Figure 3.21: AST view (expanded)

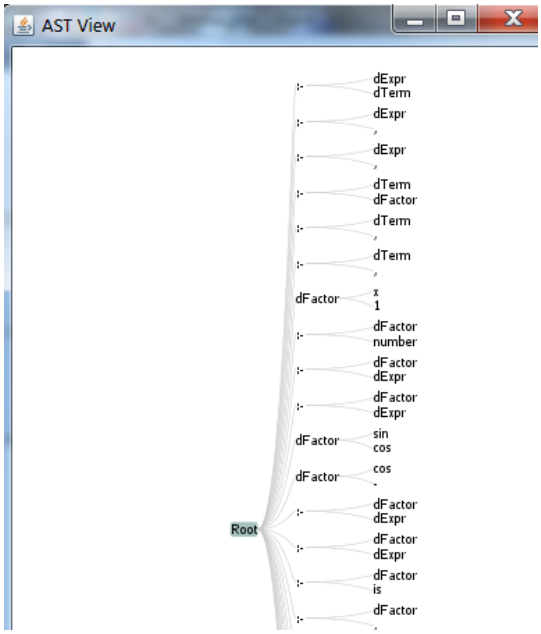

Figure 3.22: AST view (big term, expanded)

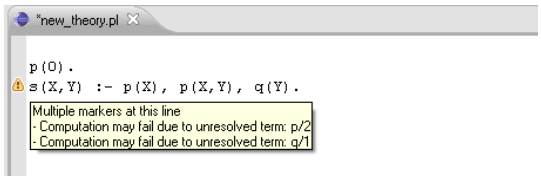

Figure 3.23: warning for undeclared terms

Prolog parser is shown (Figures 3.21 and 3.22).

It is worth highlighting that multiple tuProlog engines can be handled simultaneously: each engine can be selectively loaded with each own set of libraries and theories, and can be separately queried. Moreover, in case of undeclared terms, a direct warning is issued in the plugin editor (Figure 3.23).

3.3 tuProlog for the .NET Developer

Since tuProlog.NET is the result of an automatic conversion of the Java bytecode via IKVM[**?**], everything in the Prolog user experience is identical

whether the .NET or the Java GUI is used (see Section 3.1 above).

The .NET developer, however, can exploit tuProlog in a .NET project, accessing its API from a program written in potentially any language available in the .NET platform. Since no plugin is available for the de-facto standard tool used by most .NET programmers (i.e., Microsoft Visual Studio), there is no immediate way to see tuProlog at work from within Visual Studio; however, the tuProlog libraries can be easily added as external references for exploiting the available APIs, as one would do with any other library or third-party software.

For specific information about multi-paradigm programming in the context of the .NET platform, please refer to Chapter **??**.

3.4 tuProlog for the Android User

Since tuProlog is written in Java, the Java-Android developer wishing to include tuProlog in an Android project can proceed very similarly to the Java developer, adding `tuprolog.jar` to the project libraries—though no plugin is available for this platform.

The Prolog-Android user, instead, can take advantage of the tuProlog app, which shares the same core and libraries as the standard Java version, the only difference being the redesigned GUI–with special regard to the interaction with the file system.

Upon the application loading, the splash screen appears, immediately followed in a few seconds by the *Home Activity* (Figure 3.24, left). At the top, the name of the selected theory is reported (none at the beginning); below is the query textfield. Four buttons enable the user to execute a query, ask for the next solution (when applicable), show the current solution and view the output console. The menu button triggers the pop-up shown in Figure 3.24 (right), whose main feature is *List Theories*.

Indeed, in tuProlog for Android theories are not loaded directly in the Prolog engine from the file system, as in the standard Java version: rather, following Android recommendations, a *theory database* mediator is provided, so as to separate the loading of a theory from its validity check—the latter being performed only when the theory is actually selected for being loaded into the engine. In this way, invalid theories (possibly incomplete, work-in-progress theories) can seamlessly be stored in the theory database, independently of their invalid nature.

So, theories of interest must be first loaded into the theory database (Figure 3.25, left): then, the theory to be actually loaded will be selected

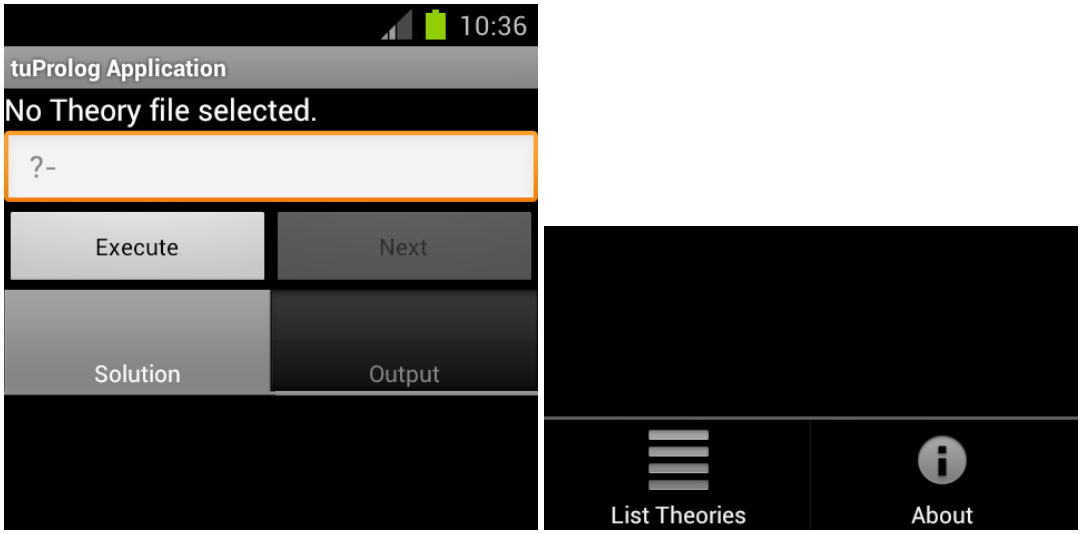

Figure 3.24: Home Activity (left) and its pop-up menu (right).

from such theories. More precisely, to add a theory to the database, the menu option *Import Theory to Database* is provided (Figure 3.25, right): a new activity opens that lets you browser the device's file system (Figure 3.26, left). Only the files that can be actually selected for addition to the theory database are shown: after a theory is successfully imported, the activity remembers the path for the next time, so as to make it faster to import multiple files.

Theories in the database can be deleted, edited and exported in a (long-)click, using the proper the context menu item (Figure 3.26, right). The export path can be changed via the *Edit Export Path* in the activity menu.

Editing (Figure 3.27, left) applies both to existing (loaded) files and to brand new theories: to create a new theory, just click on *New Theory* option in the context menu. After editing, to make your changes permanent, the modified theory must be saved to the theory database by clicking the *Confirm* button: alternatively, the back button discards changes.

When a valid theory is loaded, a query can be written in the input field (Figure 3.27, right): an auto-complete mechanism is available which exploits the previous queries to speed up the typing process. Pressing *Execute*, the query solution is shown in the *Solution* tab, along with variable bindings; any output performed by the application is available in the *Output* tab. If multiple solutions exist, the *Next* button is enabled and can be exploited to browse them—the corresponding output being shown in the *Output* tab.

Figure 3.25: Theory database (left) and context menu (right)

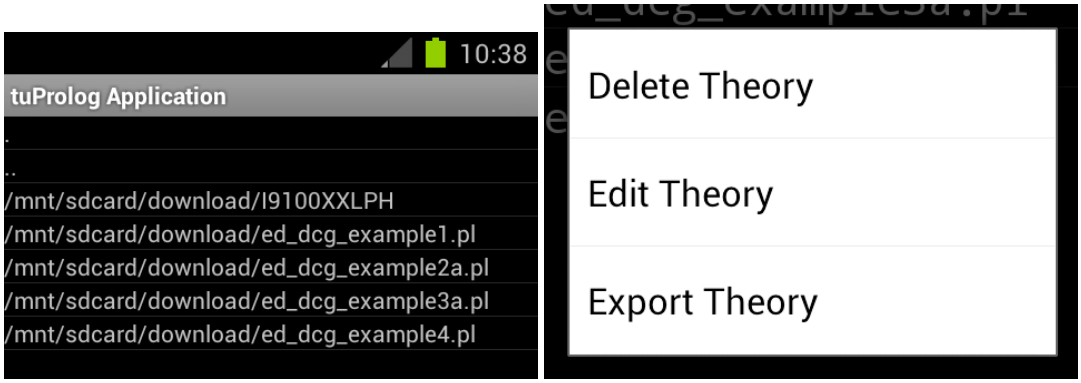

Figure 3.26: Browsing theories (left) and theory operations (right)

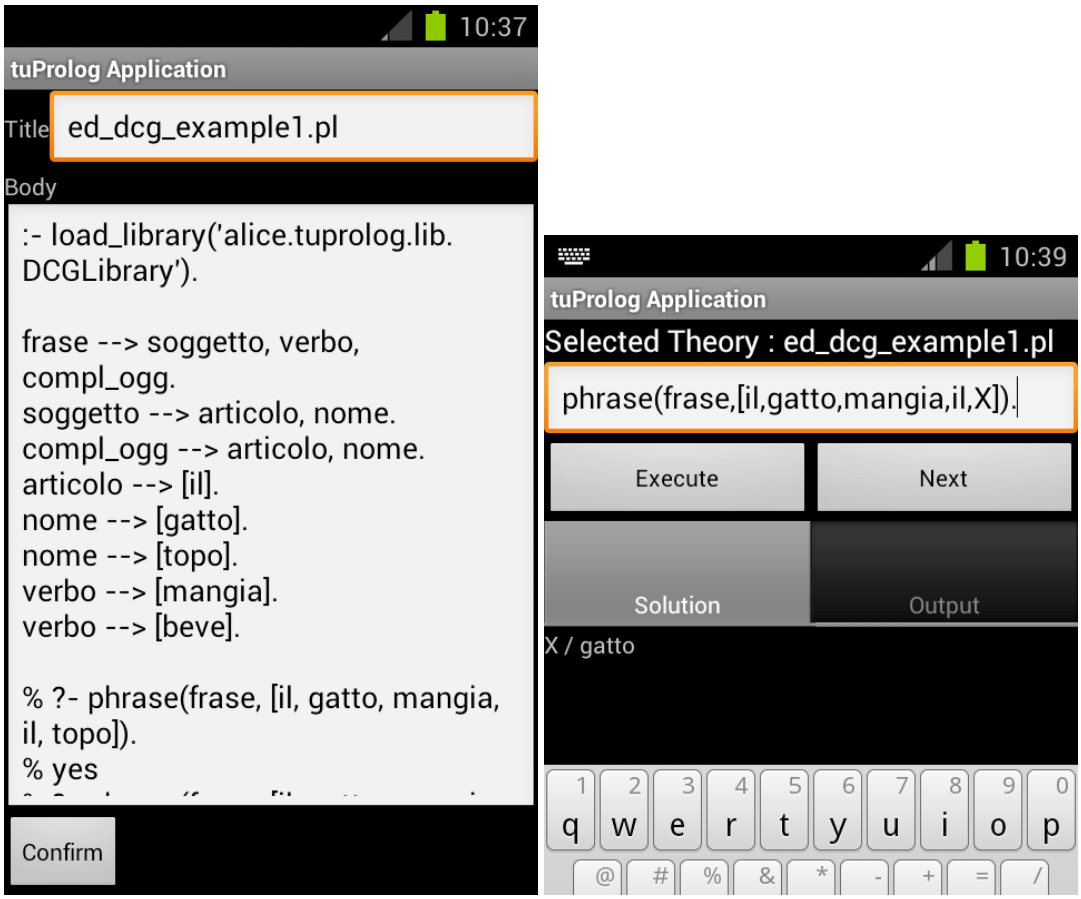

Figure 3.27: Theory editing (left) and query execution (right)

Chapter 4

tuProlog Basics

This chapter overviews the basic elements and structure of the tuProlog engine, the tuProlog syntax, the programming support, and the built-in predicates. Additional predicates, provided by libraries, are presented in the next Chapter.

4.1 Predicate categories

In tuProlog, predicates are organised into three different categories:

built-in predicates — Built-in predicates are so-called because they are defined at the tuProlog core level. They constitute a small but essential set of predicates, that any tuProlog engine can count on. Any modification possibly made to the engine before or during execution will never affect the number and properties of these 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 run time, the set of the library predicates of a tuProlog engine is not fixed, and can change from engine to engine, as well as at different times for the same engine. It is worth noting that library predicates cannot be individually retracted: to remove an undesired library predicate from the engine, the whole library containing that predicate needs to be unloaded.

Library predicates can be overridden by theory predicates, that is, predicates defined in the user theory.

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, as well as at different times for the same engine.

It is worth highlighting that, though they may seem similar, library and theory predicates are not the same, and are handled differently by the tuProlog engine. The difference between the two categories is both *conceptual* and *structural*.

Conceptually speaking, theory predicates should be used to axiomatically represent domain knowledge at the time the proof is performed, while library predicates should be used to represent what is required (procedural knowledge, utility predicates) in order to actually and effectively perform proofs in the domain of interest. So, from this viewpoint, library predicates are devoted to represent more "stable" knowledge than theory predicates. Correspondingly, library and theory predicates are represented differently at run-time, and are handled differently by the engine—in particular, with respect to the observation level for monitoring and debugging purposes. In particular, library predicates are usually step over during debugging, coherently with their more stable (and expectedly well-tested) nature, while theory predicates are step into in a detailed way during the controlled execution. This is also why all the tools in the tuProlog GUI show in a separate way the theory predicates, on the one hand, and the loaded libraries and predicates, on the other.

4.2 Syntax

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

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.

[1] Some ISO directives, however, are not supported.

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.

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.

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 `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.

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 tuProlog 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)`.

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

4.3 Engine configurability

tuProlog engines provides four levels of configurability:

Libraries — At the first level, each 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 **??**). 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/.NET side).

Directives — At the second level, directives 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 — At the third level, tuProlog supports the dynamic definition of flags to describe relevant aspects of libraries, 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.

Theories — The fourth level of configurability is given by theories: a theory is 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.

4.4 Exception support

As of version 2.2, tuProlog supports exceptions according to the ISO Prolog standard (ISO/IEC 13211-1) published in 1995. Details about the exception handling mechanism are provided in Chapter 6: this short overview is functional to the understanding of the built-in predicate specification presented in the next Section.

According to the ISO specification, an *error* is a particular circumstance that interrupts the execution of a Prolog program: when a Prolog engine

[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.

encounters an error, it raises an *exception*, which is supposed to transfer the execution flow to a suitable exception handler, exiting atomically from any number of nested execution contexts.

4.4.1 Error classification

When an exception is raised, the relevant error information is also transferred by instantiating a suitable *error term*.

The ISO Prolog standard prescribes that such a term follows the pattern `error(`*`Error_term, Implementation_defined_term`*`)` where *`Error_term`* is constrained by the standard to a pre-defined set of values (the error categories), and *`Implementation_defined_term`* is an optional term providing implementation-specific details. Ten error categories are defined:

1. `instantiation_error`: when the argument of a predicate or one of its components is an unbound variable, which should have been instantiated. Example: `X is Y+1` when `Y` is not instantiated at the time `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 is bound to the wrong type of data. *`ValidType`* represents the expected data type (one of `atom`, `atomic`, `byte`, `callable`, `character`, `evaluable`, `in_byte`, `in_character`, `integer`, `list`, `number`, `predicate_indicator`, `variable`), and *`Culprit`* is the actual (wrong) type found. Example: a predicate expecting months to be represented as integers in the range 1–12 called with an argument like `march` instead of `3`.

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`, `not_empty_list`, `not_less_than_zero`, `close_option`, `io_mode`, `operator_priority`, `operator_specifier`, `flag_value`, `prolog_flag`, `read_option`, `write_option`, `source_sink`, `stream`, `stream_option`, `stream_or_alias`, `stream_position`, `stream_property`. Example: a predicate expecting months as above, called with an out-of-range argument like `13`.

4. `existence_error(`*`ObjectType, ObjectName`*`)`: when the referenced object does not exist. *`ObjectType`* is the type of the unexisting object (one of `procedure`, `source_sink`, or `stream`), and *`ObjectName`* is the missing object's name. Example: trying to access an unexisting file like `usr/goofy` leads to an `existence_error(stream, 'usr/goofy')`.

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

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

7. `evaluation_error(`*`Error`*`)`: when the evaluation of a function produces an out-of-range value (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 goal. `Resource` can be any term useful to describe the situation. Examples: maximum number of opened files reached, no further available memory, etc.

9. `syntax_error(`*`Message`*`)`: when data read from an external source have an incorrect format or cannot be processed for some reason. *`Message`* can be any term useful to describe the situation.

10. `system_error`: any other unexpected error not falling into the previous categories.

4.5 Built-in predicates

This section contains a comprehensive list of the built-in predicates, that is the predicated defined directly in the tuProlog core, both for efficiency reasons and because they directly affect the resolution process.

Following an established convention, 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.

4.5.1 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 true goal. It is not opaque to cut.

  Template: `'$call'(+callable_term)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `G` is not a callable goal. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `G` (here, `callable`), while `Culprit` is the actual data type found.

- `halt/0`
  `halt` terminates a Prolog demonstration, exiting the Prolog thread and returning to the parent system. In any of the tuProlog user interfaces – the GUI, the character-based console, the Android app, the Eclipse plugin – the effect is to terminate the whole application (including Eclipse itself).

- `halt/1`
  `halt(X)` terminates a Prolog demonstration, exiting the Prolog thread and returning the provided int value to the parent system. In any of the tuProlog user interfaces – the GUI, the character-based console, the Android app, the Eclipse plugin – the effect is to terminate the whole application (including Eclipse itself).

Template: `halt(+int)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `X` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `X` is not an integer number. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `X` (here, `integer`), while `Culprit` is the actual data type found.

4.5.2 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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Y` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Y` is not a valid expression. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), `ValidType` is the data type expected for `G` (here, `evaluable`), while `Culprit` is the actual data type found.

  Exception: `error(evaluation_error (Error), evaluation_error(Goal, ArgNo, Error))` when an error occurs during the evaluation of `Y`. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), and `Error` is the error occurred (e.g. `zero_division` in case of a division by zero).

- `'='/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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Struct` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Struct` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1`), `ValidType` is the data type expected for `G` (here, `struct`), while `Culprit` is the actual data type found.

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

  Template: `'$fromlist'(-struct, @list)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `List` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `2`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `List` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), `ValidType` is the data type expected for `G` (here, `list`), while `Culprit` is the actual data type found.

- `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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `List` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `2`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `List` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), `ValidType` is the data type expected for `G` (here, `list`), while `Culprit` is the actual data type found.

4.5.3 Knowledge base management

- `'$find'/2`
  `'$find'(Clause, Clauses)` is true if `Clause` is a clause and `Clauses` is a list: as a side effect, all the database clauses matching `Clause` are appended to the `Clauses` list.

  Template: `'$find'(@clause, @list)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Clause` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Clauses` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the data type expected for `Clauses` (i.e. `list`), while `Culprit` is the actual data type found.

- `abolish/1`
  `abolish(Predicate)` completely wipes out the dynamic predicate matching `Predicate`.

  Template: `abolish(@term)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Predicate` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Predicate` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `Predicate`, while `Culprit` is the actual data type found.

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

  Template: `asserta(@clause)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Clause` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Clause` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `Clause`, while `Culprit` is the actual data type found.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Clause` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Clause` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `Clause`, while `Culprit` is the actual data type found.

- `'$retract'/1`
  `'$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. Please do not confuse this built-in predicate with the `retract/1` predicate of *BasicLibrary*.

Template: `'$retract'(@clause)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Clause` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Clause` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `Clause`, while `Culprit` is the actual data type found.

4.5.4 Operator and flag 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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if either `Flag` or `Value` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Flag` is not a structure or `Value` is not ground. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`), `ValidType` is the data type expected for `Flag` or `Value` (`struct` or `ground`, respectively), while `Culprit` is the actual wrong term (either `Flag` or `Value`).

Exception: `error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit))` if `Flag` is undefined in the engine or `Value` is not admissible for `Flag`. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`), `ValidDomain` is the data type expected for `Flag` or `Value` (`prolog_flag` or `flag_value`, respectively), while `Culprit` is the actual wrong term (either `Flag` or `Value`).

Exception: `error(permission_error(Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if `Flag` is unmodifiable. `Goal` is the goal where the problem occurred, `Operation` is the operation that caused the problem (`modify`), `ObjectType` is the data type of the flag (i.e. `flag`), `Culprit` is the actual wrong term (clearly, `Flag`), and `Message` adds possible extra info (by convention, the atom `0` is used when no extra info exists).

- `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. It is not re-executable.

  Template: `get_prolog_flag(+flag, ?term)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `Flag` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `Flag` is not a structure.

Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (clearly, 1), ValidType is the data type expected for G (here, struct), while Culprit is the actual data type found.

Exception: error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)) if Flag is undefined in the engine. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (clearly, 1), ValidDomain is the domain expected for G (here, prolog_flag), while Culprit is the actual wrong term found.

4.5.5 Library 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)

  Exception: error(instantiation_error, instantiation_error(Goal, ArgNo)) when LibraryName is a variable. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (obviously, 1).

  Exception: error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit)) when LibraryName is not an atom. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (obviously, 1), ValidType is the data type expected for LibraryName, while Culprit is the actual data type found.

  Exception: error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message)) when the library LibraryName does not exist. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (obviously, 1), ObjectType is the data type expected for the missing object (here, class), while Culprit is the actual data type found and Message provides extra info about the occurred error.

- 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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` when `LibraryName` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` when `LibraryName` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ValidType` is the data type expected for `LibraryName`, while `Culprit` is the actual data type found.

Exception: `error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message))` when the library `LibraryName` does not exist. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`), `ObjectType` is the data type expected for the missing object (here, `class`), while `Culprit` is the actual data type found and `Message` provides extra info about the occurred error.

4.5.6 Directives

Directives are basically queries immediately executed at the theory load time. Unlike other Prolog systems, tuProlog does not allow directives to be composed—that is, each directive must contain only one query: multiple directives require multiple queries. The standard directives are as follows:

- `:- 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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if any of `Priority`, `Specifier` or `Operator` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (one of `1`, `2`, `3`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Priority` is not an integer number, or `Specifier` is not an atom, or `Operator` is not an atom or a list of atoms. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (one of `1`, `2` or `3`), `ValidType` is the data type expected for the `Culprit`, and `Culprit` is the actual cause of the problem.

Exception: `error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit))` if the type of `Priority` and `Specifier` is correct, but their values are not admissible for the operator priority or associativity, respectively. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`), `ValidDomain` is the data type expected for `Culprit`, and `Culprit` is the actual wrong term found.

- `:- 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`
  The directive version of the `load_library/1` predicate documented in Subsection 4.5.5. However, here errors in the library name do not raise exceptions—rather, the directive simply fails, yielding no effect at all.

- `:- include/1`
  `include(Filename)` immediately loads the theory contained in the file specified by `Filename`. Again, errors in the file name do not raise exceptions: the directive simply fails, yielding no effect at all.

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, which includes only the small set of *built-in*s introduced in the previous Chapter. Any other piece of functionality, in the form of predicates, functors, flags and operators, is delivered by *libraries*, which can be loaded and unloaded to/from the engine at any time: each library can provide a set of predicates, functors and a related theory, which can be used to define new flags and operators.

The dynamic loading of libraries can be exploited, for instance, to bound the availability of some functionalities to a specific use context, as in the following example:

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

The tuProlog distribution include several standard libraries, some of which are loaded by default into any engine–although it is always possible both to create an engine with no pre-loaded libraries, and to create an engine with different (possibly user-defined or third party) pre-loaded libraries.

The fundamental libraries, loaded by default, are the following:

BasicLibrary (class `alice.tuprolog.lib.BasicLibrary`) — provides the most common Prolog predicates, functors, and operators. In order to separate computation and interaction aspects, no I/O predicates are included.

ISOLibrary (class `alice.tuprolog.lib.ISOLibrary`) — provides predicates and functors that are part of the built-in section in the ISO standard [1], and are not provided as built-ins or by BasicLibrary.

IOLibrary (class `alice.tuprolog.lib.IOLibrary`) — provides the classic Prolog I/O predicates, except for the ISO-I/O ones.

JavaLibrary (class `alice.tuprolog.lib.JavaLibrary`) — provides predicates and functors to support multi-paradigm programming between Prolog and Java, enabling a complete yet easy access to the object-oriented world of Java from tuProlog: features include the creation and access of both existing and new objects, classes, and resources. In the .NET version of tuProlog, this library is replaced[1] by **OOLibrary**, which extends the multi-paradigm programming approach to virtually any language supported by the .NET platform. (More on this in Chapter **??**.)

Other libraries included in the standard tuProlog distribution, but not loaded by default, are the following:

ISOIOLibrary (class `alice.tuprolog.lib.ISOIOLibrary`) — extends the above IOLibrary by adding ISO-compliant I/O predicates.

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.

Further libraries exist that are *not* included in the standard tuProlog distribution, because of their very specific domain: they can be downloaded from the tuProlog site, along with their documentation. Among these, for instance, `RDFLibrary` (class `alice.tuprolog.lib.RDFLibrary`) provides predicates and functors to handle RDF documents, etc.

The next Sections present the predicates, functors, operators and flag of each library, as well as the dependencies from other libraries, *except for JavaLibrary*, which is discussed in detail in the context of multi-paradigm programming (Chapters **??**, **??**, **??**, **??** and **??**). Throughout this chapter, `string` means a single-quoted or double-quoted string, as detailed in Chapter 4, while `expr` means an evaluable expression—that is, a term that can be interpreted as a value by some library functors.

[1] Actually, integrated: please see Chapter **??** for details.

5.1 BasicLibrary

5.1.1 Predicates

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 this 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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `N` or `Term` are variables. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1` or `2`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `N` is not an integer number or `Term` is not a compound term. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here,

1 or 2), ValidType is the expected data type (integer or compound, respectively), Culprit is the wrong term found (either N or Term).

Exception: error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)) if N is an int value less than 1. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (clearly, 1), ValidDomain is the expected domain (greater_than_zero, respectively), Culprit is the wrong term found (obviously, N).

- 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(Text1, Text2, TextDest) is true iff TextDest is the text resulting by appending the text Text2 to Text1.

  Template: text_concat(@string, @string, -string)

  Exception: error(instantiation_error, instantiation_error(Goal, ArgNo)) if Text1 or Text2 are variables. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (here, 1 or 2).

  Exception: error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit)) if Text1 or Text2 are not atoms. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused the problem (here, 1 or 2), ValidType is the expected data type (e.g. atom), Culprit is the wrong term found (either Text1 or Text2).

- 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)

  Exception: error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit)) if Atom is a variable and Number is not a number, or, viceversa, if Atom is not an atom. Goal is the goal where the problem occurred, ArgNo indicates the argument that caused

the problem (here, `1` or `2`), `ValidType` is the expected data type for the wrong argument (e.g. either `number` or `atom`), `Culprit` is the wrong term found (either `Number` or `Atom`).

Exception: `error(domain_error(ValidType, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit))` if `Atom` is an atom that does not represent a number. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), `ValidDomain` is the expected domain for the wrong argument (`num_atom`), `Culprit` is the wrong term found (obviously `Atom`).

Occur 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* [6] 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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `G` is not a callable goal (for instance, it is a number). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the expected data type (`callable`), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `G` is not a callable goal (for instance, it is a number). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the expected data type (`callable`), `Culprit` is the wrong term found.

- `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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `G` is not a callable goal (for instance, it is a number). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the expected data type (`callable`), `Culprit` is the wrong term found.

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)`

  Exception: the same as the built-in predicate `$call/1`; the exception results to be raised by the auxiliary predicate `call_guard(G)`.

- `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 Retrieval, Creation and Destruction

Every Prolog engine lets programmers modify its logic database during execution by adding or deleting specific clauses. The ISO standard [1] 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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Head` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

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

  Template: `assert(@term)`

  Exception: the same as the built-in predicate `assertz/1`.

- `retract/1`
  `retract(Clause)` removes from the knowledge base a dynamic clause

that matches `Clause` (which must be at least partially instantiated). Multiple solutions are given upon backtracking.

Template: `retract(@term)`

Exception: the same as the built-in predicate `$retract/1`; the exception is raised by the auxiliary predicate `retract_guard(Clause)`.

- `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)`

  Exception: the same as the built-in predicate `$retract/1`; the exception is raised by the auxiliary predicate `retract_guard(Clause)`.

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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `TheoryText` is a variable. `Goal` is the goal where

the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `TheoryText` is not an atom (i.e. a string). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

Exception: `error(syntax_error(Message), syntax_error(Goal, Line, Position, Message))` if `TheoryText` is not a valid theory. `Goal` is the goal where the problem occurred, `Message` describes the error occurred, `Line` and `Position` report the error line and position inside the theory, respectively; if the engine is unable to provide either of them, the corresponding value is set to `-1`.

- `add_theory/1`
  `add_theory(TheoryText)` is true iff `TheoryText` is the text representation of a valid tuProlog theory, with the side effect of appending it to the current theory of the engine.

  Template: `add_theory(@string)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `TheoryText` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `TheoryText` is not an atom (i.e. a string). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

  Exception: `error(syntax_error(Message), syntax_error(Goal, Line, Position, Message))` if `TheoryText` is not a valid theory. `Goal` is the goal where the problem occurred, `Message` describes the error occurred, `Line` and `Position` report the error line and position inside the theory, respectively; if the engine is unable to provide either of them, the corresponding value is set to `-1`.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `TheoryText` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `TheoryText` is not an atom (i.e. a string). `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if either `TheoryText` or `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `TheoryText` is not an atom (i.e. a string) or `G` is not a structure. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1` or `2`), `ValidType` is the expected data type (`atom` or `struct`), `Culprit` is the wrong term found.

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 spy event notification.

  Template: `spy`

- `nospy/0`
  `nospy` is true and disables spy event notification.

  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 `List`

  Template: `member(?term, +list)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `List` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `2`), `ValidType` is the expected data type (`list`), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `List` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`list`), `Culprit` is the wrong term found.

- `delete/3`
  `delete(Element, ListSource, ListDest)` is true iff `ListDest` list can be obtained by removing `Element` from the list `ListSource`.

  Template: `delete(@term, +list, -list)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `ListSource` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the expected data type (`list`), `Culprit` is the wrong term found.

- `element/3`
  `element(Pos, List, Element)` is true iff `Element` is the `Pos`-th element of `List` (element numbering starts from 1).

  Template: `element(@integer, +list, -term)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `List` is not a list. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `2`), `ValidType` is the expected data type (`list`), `Culprit` is the wrong term found.

- `quicksort/3`
  `quicksort(List, ComparisonPredicate, SortedList)` is true iff `SortedList` contains the same elements as `List`, but sorted according to the criterion defined by `ComparisonPredicate`.

  Template: `element(@list, @pred, -list)`

5.1.2 Functors

The following functors for expression evaluation (with the usual semantics) are provided:

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

5.1.3 Operators

The full list of BasicLibrary operators, with their priority and associativity, is reported in Table 5.1.

Expression comparison operators (`=:=` (equal), `=\=` (different), `>` (greater), `<` (smaller), `>=` (greater or equal), `<=` (smaller or equal)) can raise the following exceptions:

- *Exception:* `error(instantiation_error, instantiation_error(Goal, ArgNo))` if any of the arguments is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`).
- *Exception:* `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if any of the two arguments is not an evaluable expression. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`), `ValidType` is the expected data type (`evaluable`), `Culprit` is the wrong term found.
- *Exception:* `error(evaluation_error(Error), evaluation_error(Goal, ArgNo, Error))` if an error occurs during the evaluation of any of the two arguments. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1` or `2`), `Error` is the error occurred (e.g. `zero_division` in case of a division by zero).

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

Table 5.1: BasicLibrary operators.

5.2 ISOLibrary

Library Dependencies: BasicLibrary.

This library contains all the predicates and functors of the Prolog ISO standard and that are not provided directly at the tuProlog core or at the BasicLibrary levels.

5.2.1 Predicates

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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Atom` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Atom` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

- `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)`

- `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)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Atom` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Atom` is a variable and `List` is not a list, or, conversely, `List` is a variable and `Atom` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (either `1` or `2`), `ValidType` is the expected data type (`atom` or `list`, respectively), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Code` is a variable and `Char` is not a character (that is, an atom of length 1), or, conversely, `Char` is a variable and `Code` is not an integer. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (either `1` or `2`), `ValidType` is the expected data type (`character` or `integer`, respectively), `Culprit` is the wrong term found.

- `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

The full list of ISOLibrary operators, with their priority and associativity, is reported in Table 5.2.

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

Table 5.2: ISOLibrary operators.

5.2.4 Flags

The full list of ISOLibrary flags, with their admissible and default values, is reported in Table 5.3.

Flag Name	Possible Values	Default Value	Modifiable
bounded	true	true	no
max_integer	2147483647	2147483647	no
min_integer	-2147483648	-2147483648	no
integer_rounding_function	down	down	no
char_conversion	off	off	no
debug	off	off	no
max_arity	2147483647	2147483647	no
undefined_predicates	fail	fail	no
double_quotes	atom	atom	no

Table 5.3: ISOLibrary flags. Any tentative to modify unmodifiable flags will result into a `permission_error` exception.

5.3 IOLibrary

Library Dependencies: BasicLibrary.

The IOLibrary defines the classical Prolog I/O predicates; further ISO-compliant I/O predicates are provided by ISOIOLibrary (Section 5.4).

5.3.1 Predicates

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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `StreamName` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `StreamName` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

  Exception: `error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit))` if `StreamName` is not the name of an accessible file. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1`), `ValidDomain` is the expected domain (`stream`), `Culprit` is the wrong term found.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `StreamName` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `StreamName` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

  Exception: `error(domain_error(ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit))` if `StreamName` is not the name of an accessible file. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1`), `ValidDomain` is the expected domain (`stream`), `Culprit` is the wrong term found.

- `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)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Char` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `Char` is not a character, i.e. an atom of length 1. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (`1`), `ValidType` is the expected data type (`char`), `Culprit` is the wrong term found.

Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to write on the output stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `output`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the output stream, and `Message` provides extra info about the occurred error.

- `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)`

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to read from the input stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `input`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the input stream, and `Message` provides extra info about the occurred error.

- `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)`

Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to read from the input stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `input`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the input stream, and `Message` provides extra info about the occurred error.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `NumSpaces` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `NumSpaces` is not an integer number. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`integer`), `Culprit` is the wrong term found.

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to write on the output stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `output`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the output stream, and `Message` provides extra info about the occurred error.

- `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)`

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to read from the input stream.

`Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `input`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the input stream, and `Message` provides extra info about the occurred error.

Exception: `error(syntax_error(Message), syntax_error(Goal, Line, Position, Message))` if a syntax error occurred when reading from the input stream. `Goal` is the goal where the problem occurred, `Message` is the string read from the input that caused the error, while `Line` and `Position` are not applicable in this case and therefore default to `-1`.

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

  Template: `write(@term)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Term` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to write on the output stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `output`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the output stream, and `Message` provides extra info about the occurred error.

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Term` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType,`

`Culprit, Message))` if it was impossible to write on the output stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `output`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the output stream, and `Message` provides extra info about the occurred error.

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

  Template: `nl`

  Exception: `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))` if it was impossible to write on the output stream. `Goal` is the goal where the problem occurred, `Operation` is the operation to be performed (here, `output`), `ObjectType` is the type of the target object (`stream`), `Culprit` is the name of the output stream, and `Message` provides extra info about the occurred error.

Helper Predicates

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `File` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `File` is not an atom. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (here, `1`), `ValidType` is the expected data type (`atom`), `Culprit` is the wrong term found.

  Exception: `error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message))` if `File` does not exist. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (clearly, `1`), `ObjectType` is the type of the missing object (`stream`), `Culprit` is the

wrong term found and `Message` provides an error message (here, most likely `file_not_found`).

- `agent_file/1`
  `agent_file(FileName)` is true iff `FileName` is an accessible file containing a Prolog knowledge base, and as a side effect it spawns a `tu`Prolog agent provided with that knowledge base.

  Template: `agent_file(+string)`

  Exception: the predicate maps onto the above `text_from_file(File, Text)` with `File=FileName`, so the same exceptions are raised.

- `solve_file/2`
  `solve_file(FileName, Goal)` is true iff `FileName` 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)`

  Exception: the predicate maps onto the above `text_from_file(File, Text)` with `File=FileName`, so the same exceptions are raised.

  Moreover, it also raises the following specific exceptions:

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `G` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (in this case, `2`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` if `G` is not a callable goal. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (in this case, `2`), `ValidType` is the expected data type (`callable`), `Culprit` is the wrong term found.

- `consult/1`
  `consult(FileName)` is true iff `FileName` 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)`

  Exception: the predicate maps onto the above `text_from_file(File, Text)` with `File=FileName`, so the same exceptions are raised.

  Moreover, it also raises the following specific exceptions:

Exception: `error(syntax_error(Message), syntax_error(Goal, Line, Position, Message))` the theory in `FileName` is not valid. `Goal` is the goal where the problem occurred, `Message` contains a description of the occurred error, `Line` and `Position` provide the line and position of the error in the theory text.

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)`

5.4 ISOIOLibrary

The ISO specification requires a lot of I/O predicates—many more than tuProlog IOLibrary supports. Table 5.1 summarizes the differences between tuProlog IOLibrary and the ISO specifications.

The main reason for such a large number of differences lays is that the ISO Prolog standard defines very general concepts for I/O handling, aimed at supporting a wide variety of I/O modes and devices. More precisely:

- **Sources** represent the resources from which data are read;
- **Sinks** represent the resources to which data are written.

Sources and sinks can be file, standard input/output stream, or any other resource supported by the underlying system: the only assumption is that each resource is associated to a sequence of bytes of characters.

Stream terms provide a logical view of sources and sinks, and are used to identify a stream in I/O predicates. A stream term is a term respecting the following constraints:

- it is a ground term;

Predicate category	tuProlog I/O Library	Prolog ISO
Stream selection and flow control	seeing/1 telling/1	current_input/1 current_output/1 **set_input/1** **set_output/1** **at_end_of_stream/0** **at_end_of_stream/1** **set_stream_position/2** **stream_property/2** **flush_output/0** **flush_output/1**
Opening a data stream	see/1 tell/1	open/4 open/3
Closing a data stream	seen/0 told/0	close/1 close/2
Character I/O	put/1 nl/0 tab/1 get/1 get0/1 put/1	**get_char/2**, get_char/1 **get_code/2, get_code/1** **peek_char/2, peek_char/1** **peek_code/2, peek_code/1** **put_char/2**, put_char/1 **put_code/2, put_code/1** nl/0, **nl/1**
Reading from a binary stream		**get_byte/2, get_byte/1** **peek_byte/2, peek_byte/1** **put_byte/2, put_byte/1**
Term I/O	read/1	**read_term/2, read_term/3** read/1, **read/2**
Writing terms	write/1	**write_term/3,** **write_term/2** write/1, **write/2** **writeq/1, writeq/2** **write_canonical/1,** **write_canonical/2**

Figure 5.1: Comparison between the I/O predicates provided by IOLibrary and the ISO standard specification. Bold style indicates missing predicates, plain style indicates existing functionalities to be refactored, improved, or be provided with a different signature.

- it is not an atom (this requirement means to distinguish stream terms from stream aliases–see below for details);
- it is not used to identify other streams at the same time.

The ISO standard does not specify whether the stream terms must result from an explicit source/sink opening by the `open/4` predicate, nor whether different sources/sinks must be represented by different stream terms at different, subsequent times: these issues are left to the specific implementation.

Moreover, each stream can be associated to a *stream alias*, that is an atom used to refer to the stream. The association between a stream and its alias is created when the stream is opened, and automatically canceled when the stream is closed. The same stream can be associated to multiple aliases simultaneously.

Two pre-defined streams exist that are always automatically open, the standard input (alias `user_input`) and the standard output (alias `user_output`): such streams must never be closed.

The ISO standard also introduces the concepts of *current input stream* and *current output stream*: initially, they default to the standard input and standard output above, but can be reassigned at any time via the `set_input/1` e and `set_output/1` predicates. However, when such an input/output stream is closed, the current input/output stream must be re-set to their default values (i.e., the standard input/output, respectively).

One further concept is the *stream position*, which defines the point where the next input/output will takes place. The position can be changed via `set_stream_position/2`. The stream position is always supported, even by predicates whose operations do not change the position itself. Syntactically, the stream position is an implementation-dependent ground term.

5.5 DCGLibrary

Library Dependencies: BasicLibrary.

This library provides support for Definite Clause Grammars (DCGs) [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: it must be explicitly loaded by the user, or via a `load_library` directive inside any theory using DCGs.

A DCG rule has the general form `Head --> Body`: to distinguish terminal from nonterminal symbols, a phrase (that is, a sequence of terminal

symbols) must be written as a Prolog list, with the empty sequence written as the empty list `[]`. The body can contain also executable blocks in parentheses, which are interpreted as normal Prolog rules.

Here is a simple example (see also Figure 5.2):

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

To verify whether a phrase is correct according to the given grammar, the `phrase/2` or `phrase/3` predicates are used—the latter form providing an extra argument for the 'remainder' of the input string not recognised as being part of the phrase. Some examples follow:

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

?- phrase(sentence, [Who, loves, linda])
Who/charles
Who/linda

?- phrase(sentence, [charles, loves, linda, but, hates, laura], R)
R/[but, hates, laura]
```

5.5.1 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, 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.

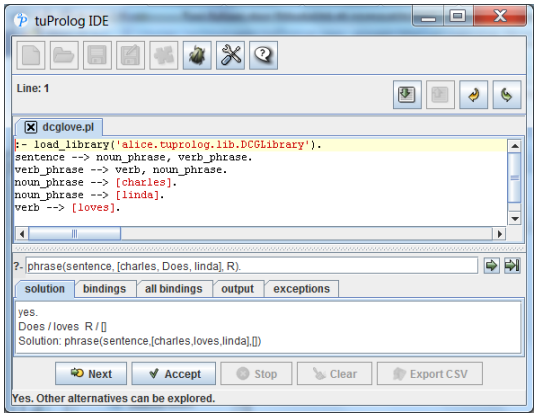

Figure 5.2: The DCG Library example in the tuProlog GUI (note the explicit library loading directive).

Template: `phrase(+term, ?list)`

Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Category` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

- `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)`

  Exception: `error(instantiation_error, instantiation_error(Goal, ArgNo))` if `Category` is a variable. `Goal` is the goal where the problem occurred, `ArgNo` indicates the argument that caused the problem (obviously, `1`).

5.5.2 Operators

The full list of DCGLibrary operators, with their priority and associativity, is reported in Table 5.4.

Operator	Associativity	Priority
`-->`	xfx	1200

Table 5.4: DCGLibrary operators.

Chapter 6

tuProlog Exceptions

6.1 Exceptions in ISO Prolog

Exception handling was first introduced in the ISO Prolog standard (ISO/IEC 13211-1) in 1995.

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 is 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, the `catch(`*`Goal`*`, `*`Catcher`*`, `*`Handler`*`)` predicate enables the controlled execution of a goal, while the `throw(`*`Error`*`)` predicates makes it possible to raise an exception—very much like the `try/catch` construct of many imperative languages.

Semantically, executing the `catch(`*`Goal`*`, `*`Catcher`*`, `*`Handler`*`)` means that *`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 ancestor 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 one is found *`Handler`* is executed, maintaining the substitutions made during the previous unification process. Then, execution continues with the subgoal following `catch/3`. Any side effects possibly occurred during the execution of a goal are *not* undone in case of exceptions—as it normally happens when a predicate fails.

Summing up, `catch/3` succeeds if:

- `call(Goal)` succeeds *(standard behaviour)*;

  –OR–

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

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

As an example, let us consider the following toy program:

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

with the following query:

```
?:- 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 following `catch/3`, i.e. `fail`. So, `write('---')` is never reached, nor is `write('+++')` since all the choicepoints are cut upon exception.

6.1.1 Error classification

This classification was already presented in Section 4.4 above as a hint to predicate and functor readability: however, we report it here too both for completeness and for the reader's convenience.

When an exception is raised, the relevant error information is also transferred by instantiating a suitable *error term*.

The ISO Prolog standard prescribes that such a term follows the pattern `error(`*`Error_term, Implementation_defined_term`*`)` where *`Error_term`* is constrained by the standard to a pre-defined set of values (the error categories), and *`Implementation_defined_term`* is an optional term providing implementation-specific details. Ten error categories are defined:

1. `instantiation_error`: when the argument of a predicate or one of its components is an unbound variable, which should have been instantiated. Example: `X is Y+1` when `Y` is not instantiated at the time `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 is bound to the wrong type of data. *`ValidType`* represents the expected data type (one of `atom`, `atomic`, `byte`, `callable`, `character`, `evaluable`, `in_byte`, `in_character`, `integer`, `list`, `number`, `predicate_indicator`, `variable`), and *`Culprit`* is the actual (wrong) type found. Example: a predicate expecting months to be represented as integers in the range 1–12 called with an argument like `march` instead of `3`.

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`, `not_empty_list`, `not_less_than_zero`, `close_option`, `io_mode`, `operator_priority`, `operator_specifier`, `flag_value`, `prolog_flag`, `read_option`, `write_option`, `source_sink`, `stream`, `stream_option`, `stream_or_alias`, `stream_position`, `stream_property`. Example: a predicate expecting months as above, called with an out-of-range argument like `13`.

4. `existence_error(`*`ObjectType, ObjectName`*`)`: when the referenced object does not exist. *`ObjectType`* is the type of the unexisting object (one of `procedure`, `source_sink`, or `stream`), and *`ObjectName`* is the missing object's name. Example: trying to access an unexisting file like `usr/goofy` leads to an `existence_error(stream, 'usr/goofy')`.

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

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

7. `evaluation_error(`*`Error`*`)`: when the evaluation of a function produces an out-of-range value (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 goal. `Resource` can be any term useful to describe the situation. Examples: maximum number of opened files reached, no further available memory, etc.

9. `syntax_error(`*`Message`*`)`: when data read from an external source have an incorrect format or cannot be processed for some reason. *`Message`* can be any term useful to describe the situation.

10. `system_error`: any other unexpected error not falling into the previous categories.

6.2 Exceptions in tuProlog

tuProlog aims to fully comply to ISO Prolog exceptions. In the following, a set of mini-examples are presented which highlight each one single aspect of tuProlog compliance to the ISO standard.

6.2.1 Examples

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.`

6.2.2 Handling Java Exceptions from tuProlog

One peculiar aspect of tuProlog is the ability to support multi-paradigm programming, mixing object-oriented (mainly, but not exclusively, Java) and Prolog in several ways—in particular, by enabling Java objects to be accessed and exploited from Prolog world via Javalibrary (see Chapter **??**). In this context, the problem arises of properly sensing and handling Java exceptions from the Prolog side.

At a first sight, one might think of re-mapping Java exceptions and constructs onto the Prolog ones, but this approach is unsatisfactory for three 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 exceptions;
- the Java catching mechanisms operates hierarchically, while the `catch/3` predicate operates via pattern matching and unification, allowing for a finer-grain, more flexibly exception filtering.

Accordingly, Java exceptions in tuProlog programs are handled by means of two further, *ad hoc* predicates: `java_throw/1` and `java_catch/3`. Since their behaviour can be fully understood only in the context of JavaLibrary, we forward the reader to Chapter **??** for further information.

6.3 Appendix: Implementation notes

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 .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_error`s or `evaluation_error`s; 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.

Per quel che riguarda il modo in cui è stato implementato il meccanismo di controllo degli errori, bisogna distinguere i predicati espressi in Java da quelli espressi in Prolog.

Nel primo caso le eccezioni (cioè le opportune istanze di `PrologError`) vengono lanciate direttamente dai corrispondenti metodi Java ogniqualvolta si verifica un errore, mentre nel secondo caso sono lanciate da metodi "guardia" (sempre espressi in Java) invocati per controllare i parametri prima dell'esecuzione del predicato Prolog.

Nell'implementazione si è cercato di individuare il maggior numero possibile di condizioni di errore, rispettando però sempre il requisito fondamentale di correttezza: se una chiamata a un predicato non falliva prima dell'introduzione del meccanismo delle eccezioni, non deve fallire neanche ora—ovvero, il lancio di una eccezione deve avvenire soltanto in circostanze in cui il motore tuProlog originario falliva.

La correttezza del comportamento del motore è garantita anche se ci si dimentica di identificare qualche condizione di errore inaspettata: in questo caso infatti il motore non lancia un'eccezione, ma comunque fallisce. Ciò permette ad un utente sia di gestire gli errori che si possono verificare durante l'esecuzione, sia di non gestirli, nel qual caso l'esecuzione fallirà e dunque l'estensione resterà trasparente.

A parte le inevitabili modifiche ai built-in e alle librerie (*BasicLibrary*, *ISOLibrary*, *IOLibrary*, *DCGLibrary*), sono state necessarie le seguenti semplici modifiche al motore:

- alla classe `alice.tuprolog.FlagManager` sono stati aggiunti due metodi per ricavare informazioni su un flag:

  - `boolean isModifiable(String name)`
    che restituisce true se esiste nel motore un flag di nome `name`, e tale flag è modificabile;
  - `boolean isValidValue(String name, Term Value)`
    che restituisce true se esiste nel motore un flag di nome `name`, e `Value` è un valore ammissibile per tale flag.

- il metodo `getEngineManager` della classe `alice.tuprolog.Prolog` è ora pubblico (in precedenza aveva visibilità di package) per permettere alle librerie di ricavare dal motore l'informazione sul goal correntemente in esecuzione e inserirla nell'eccezione lanciata;

- il metodo `evalAsFunctor` della classe `alice.tuprolog.PrimitiveInfo` lancia ora un'istanza di `Throwable` in caso di errore durante la valutazione del goal, mentre prima ritornava `null`, per permettere di discriminare il tipo di errore verificatosi durante la valutazione di un funtore;

- analogamente, il metodo `evalExpression` di `alice.tuprolog.Library` rilancia ora l'istanza di `Throwable` ricevuta dal metodo `evalAsFuntor` di `alice.tuprolog.PrimitiveInfo`.

Chapter 7

Multi-paradigm programming in Prolog and Java

tuProlog supports multi-paradigm (and multi-language) programming between Prolog and Java in a complete, four-dimensional way:

- using Java from Prolog: *JavaLibrary*
- using Prolog from Java: *the Java API*
- augmenting Prolog from Java: *developing new libraries*
- augmenting Java from Prolog: *the P@J framework*

7.1 Using Java from Prolog: *JavaLibrary*

The first MPP dimension offered by tuProlog is the ability to fully access Java resources (objects, classes, methods, etc) in a full-fledged yet straightforward way, completely avoiding the intricacies (object and method predeclarations in some awkward syntax, pre-compilations, etc) that are often found in other Prolog systems. The unique tuProlog approach keeps the two computational models clearly separate, so that neither the Prolog nor the Java semantics is affected by the coexistence of the logical and imperative/object-oriented paradigms in the same program. In this way, any Java package, library, etc. is immediately available to the Prolog world with no effort, according to the motto "one library for all libraries". So, for

instance, Swing classes can be easily exploited to build the graphical support of a Prolog program, and the same holds for JDBC to access databases, for the socket package to provide network access, for RMI to access remote Java objects, and so on.

The two basic bricks of JavaLibrary are:

- the mapping between Java types and suitable Prolog types;
- the set of predicates to perform operations on Java objects.

7.1.1 Type mapping

The general mapping between Prolog types and Java types is summarized in Table 7.1.

Categories	*From Prolog to Java*	*From Java to Prolog*
integers	Prolog integers are mapped onto Java `int` or `long` types, as appropriate	all Java integer types are mapped onto Prolog (integer) numbers
reals	Prolog reals are mapped onto Java `double`	all Java floating-point types are mapped onto Prolog (real) numbers
booleans	N/A	Boolean Java values are mapped onto ad-hoc constants (`true` and `false`)
strings	Prolog atoms are mapped onto Java `String`s	Java `char`s and `String`s are mapped onto Prolog atoms
wildcards	Prolog indifference (the *any* variable (_)) is mapped onto the Java `null` costant	The Java `null` value is mapped onto the Prolog *any* variable (_)

Table 7.1: Prolog/Java type mapping.

Two aspects are worth highlighting:

- although the Prolog language considers a comprehensive `number` type for both integer and real values, the two kinds are considered separately in this table, both for the user's convenience and because `tu`Prolog internally does use different types for this purpose (indeed, the `tu`Prolog internal representation of numbers does distinguish `Number`

into `Int`, `Long`, `Double` and `Float`, based on the value to be stored—details in Section 7.2.2). More precisely, in the Prolog-to-Java direction, only the Java `int`, `long` and `double` types are used as target types for the mapping, while in the opposite Java-to-Prolog direction any of the numeric Java types are accepted (including `short`, `byte` and `float`) for mapping onto Prolog numbers.

- since the Prolog language does not include a specific boolean type, the table reports N/A in the Prolog-to-Java direction; however, the `true` and `false` atoms can be provided to Java methods when appropriate, as Java boolean methods return/accept these atoms when boolean values are involved.

7.1.2 Creating and accessing objects: an overview

JavaLibrary provides many predicates to access, manipulate and interact with Java objects and classes in a complete way. In this section, the fundamental predicates are presented that enable the Prolog user to create and access Java objects—that is, calling methods and getting return values. A detailed description of all the available features is reported in Section 7.1.3.

For the sake of concreteness, Table 7.2 reports a simple Java class (a counter) and the Prolog program that exploits it via JavaLibrary.

Object creation Java objects are created via the predicate

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

where *`ClassName`* is a Prolog atom bound to the name of the Java class (e.g. `'Counter'`, `'java.io.FileInputStream'`, etc.), *`ArgumentList`* is a Prolog list supplying the required arguments to the class constructor (the empty list matches the default constructor), and *`ObjectRef`* holds the reference to the newly-created object. In the case of arrays, *`ClassName`* ends with `[]`.

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 of the form `'$obj_N'`, where `N` is an 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: this is how `tu`Prolog guarantees that the two computational models are never mixed, and therefore that each semantics is preserved.

The predicate fails if *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 another object in the system.

The lifetime of the binding between the Java object and the Prolog term is the duration of the demonstration: by default, the binding is maintained in case of backtracking, but this behavior can be changed by setting the flag `java_object_backtrackable` flag to `true`.

There is no way to make such a binding permanent from the Prolog side[1]—that is, there is no way to "keep alive" a Java object beyond the current query, for retrieving it in a subsequent demonstration, because this would contradict the Prolog approach of avoiding anything like "global variables". So, the only way to maintain a Java object available to another predicate (within the same demonstration) is to pass it over as an argument. **Perché non prevedere un predicato in JavaLibrary ??**

method calling methods can be invoked on Java objects via the `<-/2` predicate, according to a send-message pattern. The predicate comes in two flavors, with/without return argument:

ObjectRef `<-` *MethodName*`(`*Arguments*`)`

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

where *ObjectRef* is an atom interpreted as a Java object reference as above, and *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; otherwise, *Term* is handled as an object reference, that is, as a Prolog ground term[2] bound to the Java object returned by the method.

Static methods can also be invoked, adopting `class(`*ClassName*`)` as the target *ObjectRef*.

The call fails if *MethodName* does not identify a valid method for the object (for the class, in the case of static methods), or arguments in

[1] There is, however, a way to do so from the Java side, coherently with the side-effect nature of imperative languages: see the `register` primitive in Section 7.2

[2] If it is not ground, it is automatically bound to a term like `$obj_N`.

ArgumentList are invalid because of a wrong signature or because they are not ground.

property selection public object properties can be accessed via the `.` infix operator, in conjunction with the `set` / `get` pseudo-method pair:

ObjectRef`.`*Field* `<-` `set(`*GroundTerm*`)`

ObjectRef`.`*Field* `<-` `get(`*Term*`)`

The first construct sets 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 it is not ground, the infix predicate fails.

Analogously, the second construct retrieves the value of the public field *Field*, handling the returned *Term* as above.

Again, class properties can be accessed using the `class(`*ClassName*`)` form for *ObjectRef*.

It is worth to point out that such `set` / `get` pseudo-methods are *not* methods of some class, but just part of of the property selection operator.

array access Due to the special Java syntax for arrays, ad hoc helper predicates are required to access Java array elements:

`java_array_set(`*ArrayRef*`,` *Index*`,` *Object*`)`

`java_array_set_Basic Type(`*ArrayRef*`,` *Index*`,` *Value*`)`

`java_array_get(`*ArrayRef*`,` *Index*`,` *Object*`)`

`java_array_get_Basic Type(`*ArrayRef*`,` *Index*`,` *Value*`)`

`java_array_length(`*ArrayObject*`,` *Size*`)`

type cast the `as` infix operator is used to explicitly cast method arguments to a given type, typically for exploiting overloading resolution:

ObjectRef `as` *Type*

By writing so, the object represented by *ObjectRef* is considered to belong to type *Type*: the latter can be either a class name, as above, or a primitive Java type such as `int`.

class loading and dynamic compilation The `java_class/4` creates, compiles and loads a new Java class from a source text:

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

where *SourceText* is a string representing the text source of the new Java class, *FullClassName* is the full 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. In this case, *ObjectRef* is a reference to an instance of the meta-class `java.lang.Class` representing the newly-created class.

The predicate fails if *SourceText* leads to compilation errors, or the class cannot be located in the package hierarchy, or *ObjectRef* already identifies another object in the system.

Exceptions thrown by Java methods or constructors can be managed by means of tuProlog's special `java_catch` and `java_throw` predicates, discussed in Section 7.1.7.

Examples

To taste the flavor of `JavaLibrary`, let us consider the example shown in Table 7.2, which reports both a simple Java class (a counter) and the Prolog program that exploits it via JavaLibrary.

First, a `Counter` instance is created (line 1) providing the `MyCounter` name as the constructor argument: the reference to the new object is bound to the Prolog atom `myCounter`.

Then, this reference is used to invoke several methods (lines 2–4) via the `<-/2` and the (`<-`,`returns`)`/3` operators—namely, `setValue(5)` (which is void and therefore returns nothing), `inc` (which takes no arguments and is void, too) and `getValue` (which takes no argument but returns an int value); the returned value (hopefully, 5) is bound to the `X` Prolog variable, which is finally printed via the Prolog `write/1` predicate (line 5). Of course, the predicate succeeds also if `X` is already bound to 5, while fails if it is already bound to anything else.

Then, the class (static) method `getVersion` is called (line 6) and the retrieved version number is printed (line 7).

Now the (public) instance *Name* property is read, and its value printed via the Java `System.out.println` method: to this end, a reference to the `java.lang.System` class is first obtained (line 8), then its `out` (static) field is accessed and its value retrieved and bound to the `Out` Prolog variable (line 9), which is used as the target for the invocation of the `println` method (line 10).

Finally, the `name` property of the `myCounter` object is changed to the new `'MyCounter2'` value (line 11).

Java class:

```
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"; }
}
```

Prolog program:

```
    ?-  java_object('Counter', ['MyCounter'], myCounter),

        myCounter <- setValue(5),
        myCounter <- inc,
        myCounter <- getValue returns Value,
        write(X), nl,

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

        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).
```

Table 7.2: The Java `Counter` class and the Prolog program that exploits it via JavaLibrary.

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

Table 7.3: Creating and using a Swing component from a tuProlog program.

The last part of the example deals with an array of 10 `Counter`s: the array is first created (line 12), and the `myCounter` object is assigned to its first (0th) element (line 13).

The key point is that the only requirement here is the presence of the `Counter.class` file in the proper location in the file system, according to Java naming conventions: no other auxiliary information is needed—no headers, no pre-declarations, pre-compilations, etc.

Table 7.3 shows one further example, where the 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 bound to that variable) and is then used to invoke the `showOpenDialog` and `getSelectedFile` methods: the Prolog anonymous variable _ is used to represent the Java `null` value in `showOpenDialog`. The `File` object returned by the file chooser is finally queries for the corresponding `FileName`, which is returned to the outer predicate caller.

Registering object bindings

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 permanently associate an existing Java object *`obj`* to a Prolog identifier *`ObjectRef`*, according to the syntax:

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

where *`ObjectRef`* is a ground term (otherwise an `InvalidObjectIdException` exception is raised) representing the Java object *`obj`* in the context of `JavaLibrary`'s predicates. The function returns `false` if that object is already registered under a different *`ObjectRef`*.

As an example of use, let us consider the following case:[3]

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

Here, the Java object `System.out` is registered for use in tuProlog 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 tuProlog identifier.

7.1.3 Predicates

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)`

[3] 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.

- `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)`

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 tuProlog 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)`

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)`

7.1.4 Functors

No functors are provided by the `JavaLibrary` library.

7.1.5 Operators

7.1.6 Examples

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

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

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);
```

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).
```

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(_).
```

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 7.1. The `java_class` arguments in the example specify, besides the source text and the binding variable, the

Figure 7.1: 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).
```

full class name (`Counter`), which is necessary to locate the class in the package hierarchy, and possibly a list of class paths required for a successful compilation (if any).

Figure 7.2 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.

7.1.7 Handling Java Exceptions

One peculiar tuProlog aspect concerns the handling of Java exceptions possibly raised during the execution of Java methods on 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 would have been unsatisfactory for three 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.

Figure 7.2: 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.
```

Accordingly, Java exceptions in tuProlog programs are handled by means of two further, *ad hoc* predicates, which are clearly not present in ISO Prolog: `java_throw/1` and `java_catch/3`.

Syntax:

`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 choice-points 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.

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`.

7.2 Using Prolog from Java: *the Java API*

7.2.1 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,[4] the string `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]
```

7.2.2 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 tuProlog data structures, defining the basic common services, such as term comparison, term unification and so

[4] 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`

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.

The general mapping between Prolog types and Java types has been summarized in Table 7.1 above.

Although the Prolog language considers a comprehensive ***`number`*** *type for both integer and real values, the two kinds have been considered separately in this table, both for the user's convenience and because* `tu`*Prolog internally does use different types for this purpose (indeed, the* `tu`*Prolog internal representation of numbers does distinguish* ***`Number`*** *into* ***`Int`****,* ***`Long`****,* ***`Double`*** *and* ***`Float`****, based on the value to be stored—details in Section 7.2.2). More precisely, in the Prolog-to-Java direction, only the Java* ***`int`****,* ***`long`*** *and* ***`double`*** *types are used as target types for the mapping, while in the opposite Java-to-Prolog direction any of the numeric Java types are accepted (including* ***`short`****,* ***`byte`*** *and* ***`float`****) for mapping onto Prolog numbers.*

1. Prolog integers are mapped onto Java `int` or `long` types, as appropriate; in the opposite direction, Java `int` and `long` values are mapped onto Prolog integers (namely, onto tuProlog inner `Int` and `Long` types, respectively);

2. `byte` and `short` Java types are mapped into tuProlog's `Int` instances

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.
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.

7.2.3 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 preloaded standard libraries: in the case that other libraries are loaded, these must be serializable in order to have the engine serializable.

7.2.4 Examples

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)
```

7.3 Augmenting Prolog from Java: *developing 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.

7.3.1 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 7.4 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 implements 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.

Table 7.4: 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).
```

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.

7.3.2 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`).

7.4 Augmenting Java from Prolog: *the P@J framework*

to be written (take from DORV-Multiparadigm)

Chapter 8

Multi-paradigm programming in Prolog and .NET

Bibliography

[1] Information technology – Programming languages – Prolog – Part 1: General core. International Standard ISO/IEC 132111, International Organization for Standardization, 1995.

[2] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2000.

[3] Enrico Denti and Andrea Omicini. LuCe: A tuple-based coordination infrastructure for Prolog and Java agents. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):139–141, March-June 2001.

[4] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[5] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[6] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1994.