tuProlog Manual

tuProlog version: 2.5

Last Changes date: August 25, 2012

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 Chapter 8).

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 7.

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* (Section 7.1) 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 (Section 7.2), 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 (Section 7.3) 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—Section 7.4), 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 Chapter 8

[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 [7, 6], 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 [5], 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 `tu`Prolog, such as the core API, the `Agent` application, libraries, GUI, etc.; this is a runnable JAR, that open the `tu`Prolog IDE when double-clicked.
- `tuprolog.jar`, which contains only the core part of `tu`Prolog, namely, what you will need to include in a Java application project to be able to access the `tu`Prolog 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 `tu`Prolog project, `test` the sources for the `tu`Prolog test suite (partly as FIT test, partly as JUnit tests), `ant` some Ant scripts to automate the build of parts of the `tu`Prolog 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 Chapter 8), 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 `tu`Prolog 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 8, 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.
```

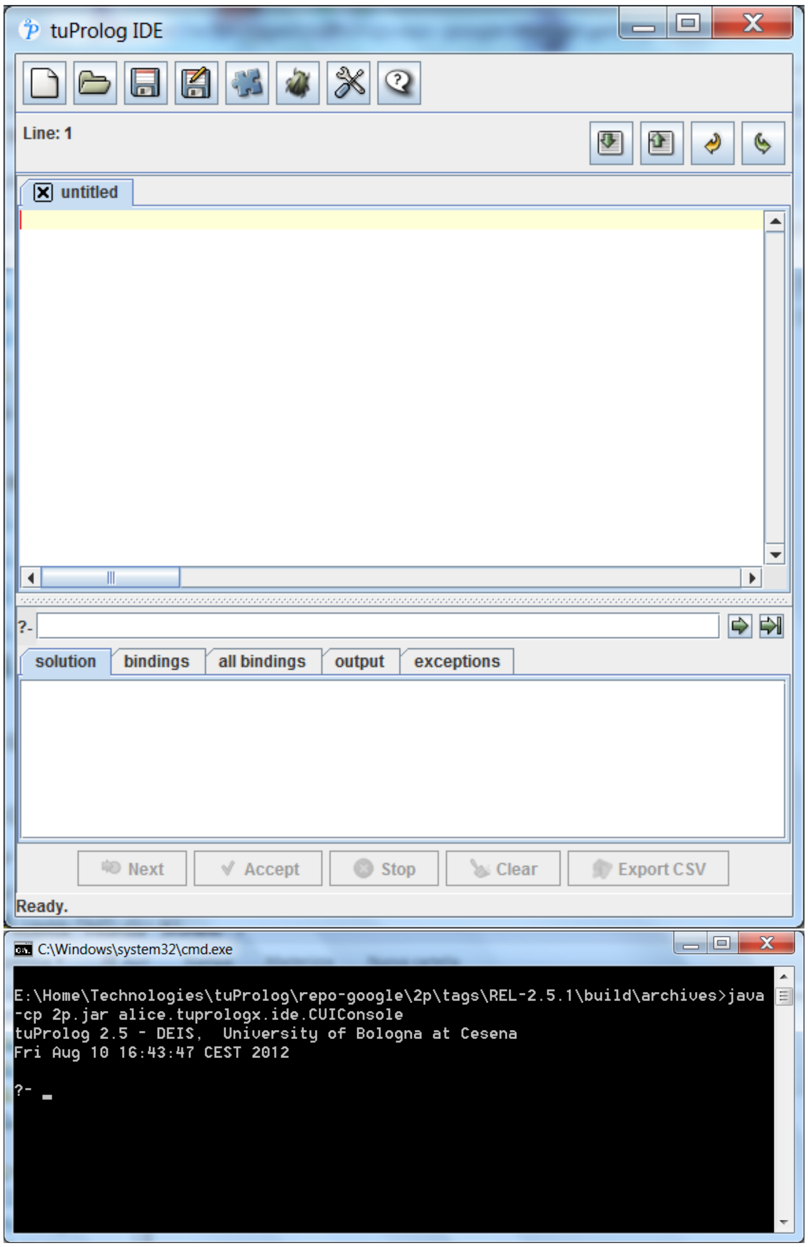

Figure 3.1: The standard tuProlog GUI and CUI.

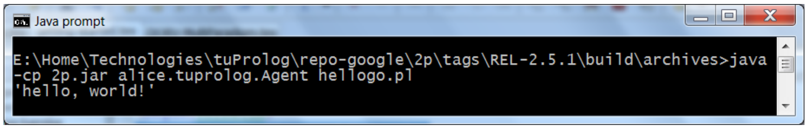

Figure 3.2: The tuProlog Agent tool.

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.

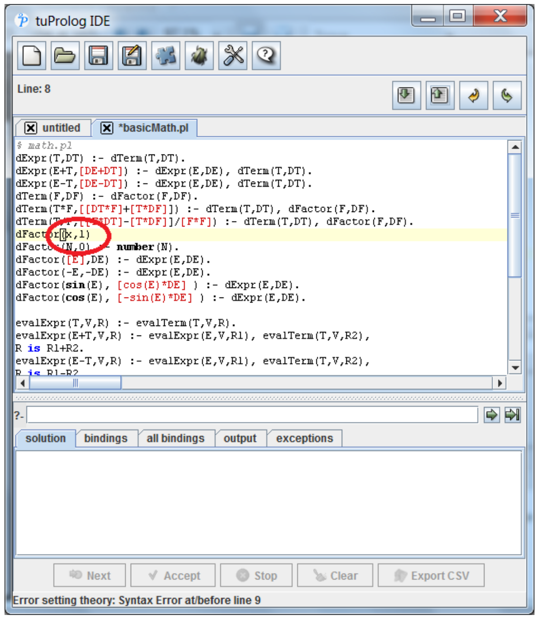

Figure 3.3: Syntax error found when setting a theory

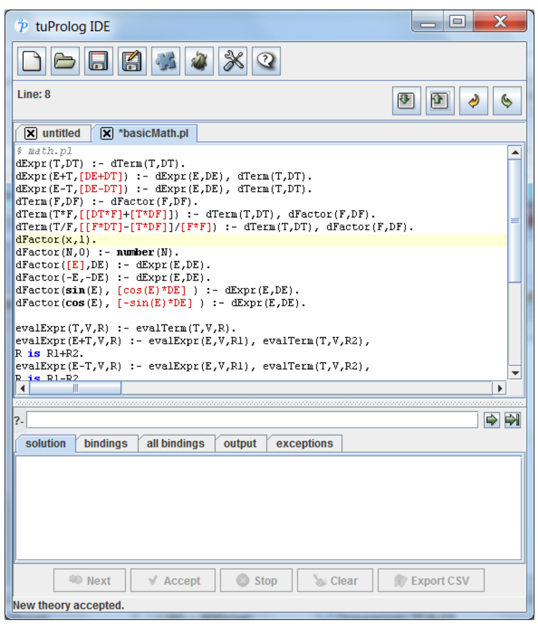

Figure 3.4: Set theory operation succeeded

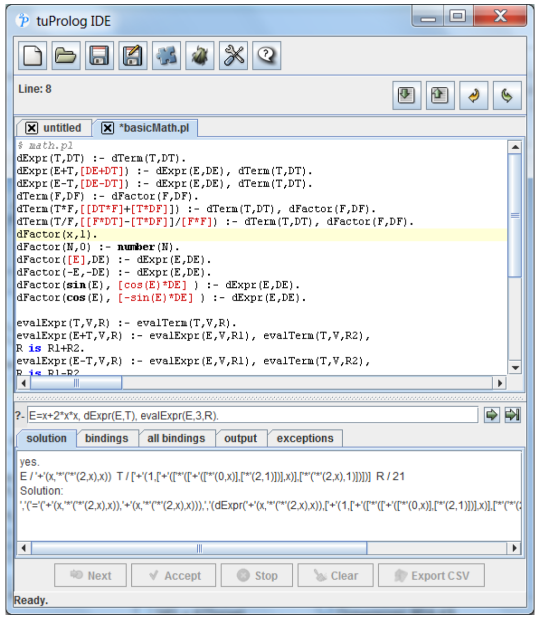

Figure 3.5: The solutions tab showing the query solution.

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:

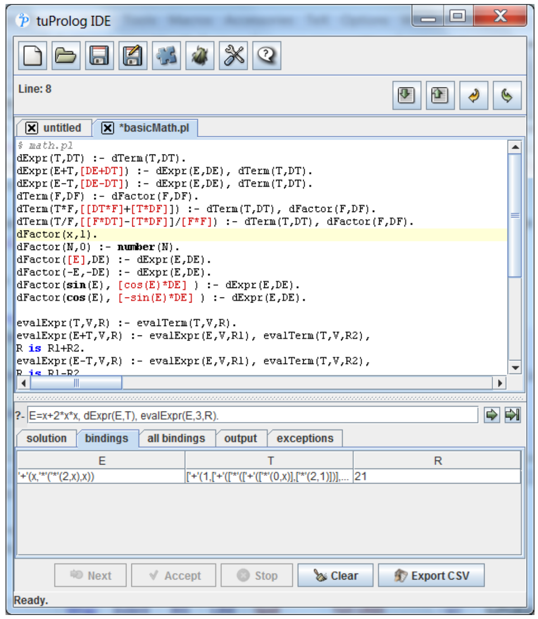

Figure 3.6: The bindings tab showing the bindings of query solution.

- 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 Section 7.1. 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).

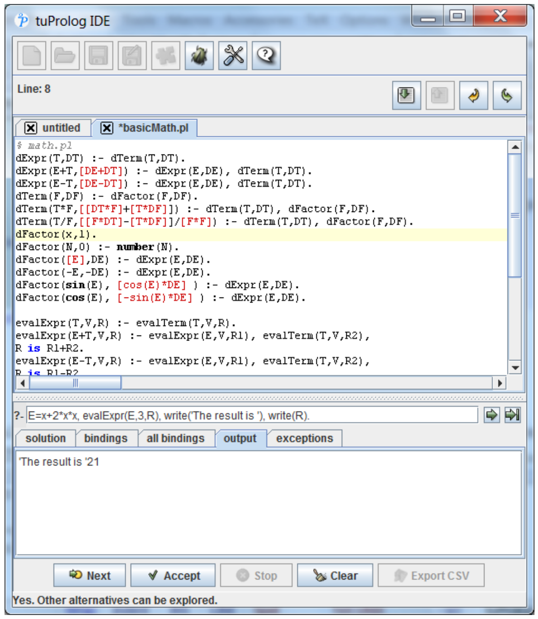

Figure 3.7: The output tab showing the query printing.

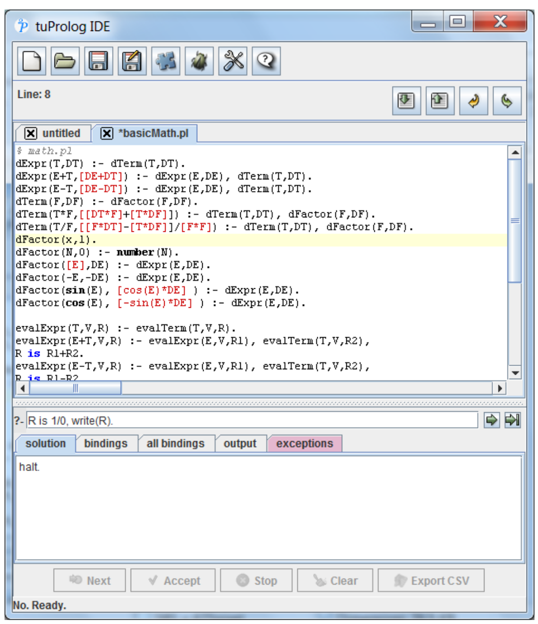

Figure 3.8: The exceptions tab gaining focus and showing raised exceptions.

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

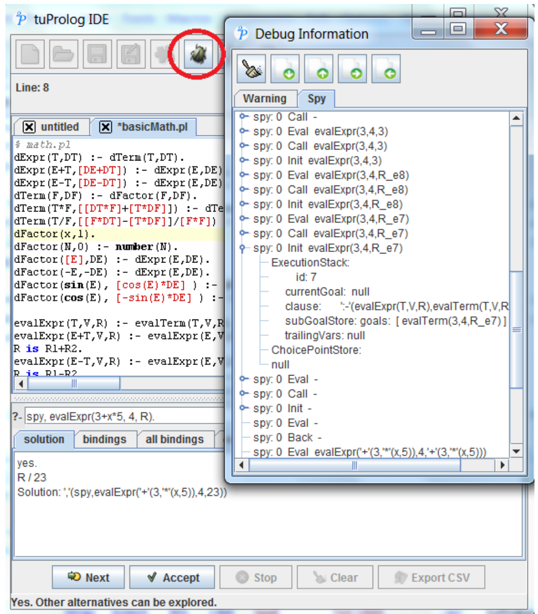

Figure 3.9: Debug Information View after the execution of a goal.

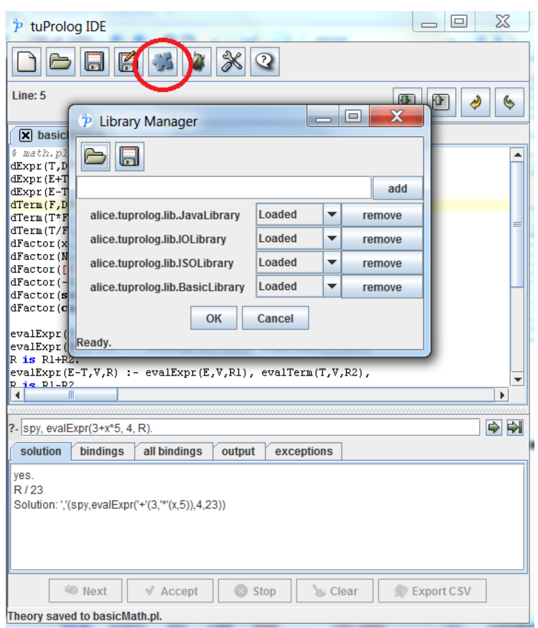

Figure 3.10: The Library Manager window.

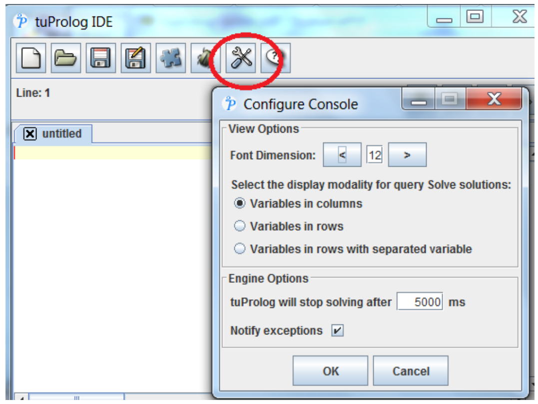

Figure 3.11: The configuration window.

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 Chapters 7, 8), 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;

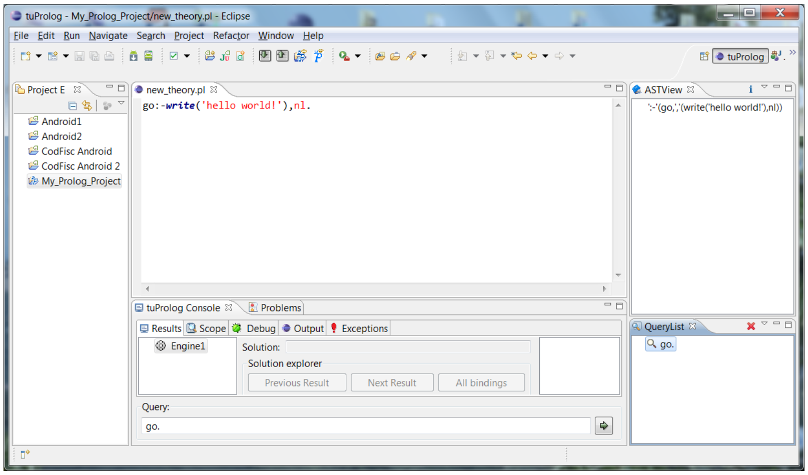

Figure 3.12: The tuProlog plugin GUI for Eclipse.

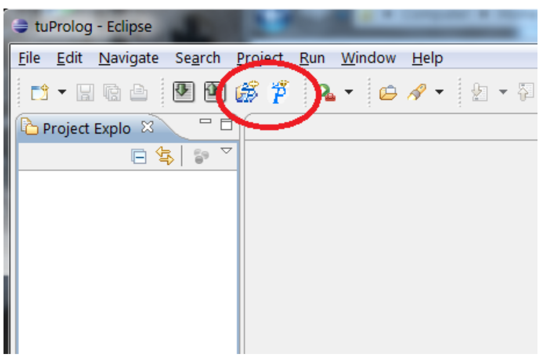

Figure 3.13: The tuProlog toolbar

- or, select `File` > `New` > `Other` > `tuProlog` > `tuProlog Project` from the main menu;
- or, press the *New tuProlog Project* buttons in the `tu`Prolog 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

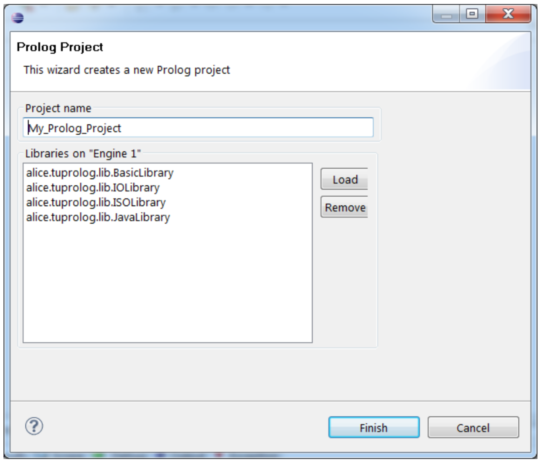

Figure 3.14: new tuProlog project

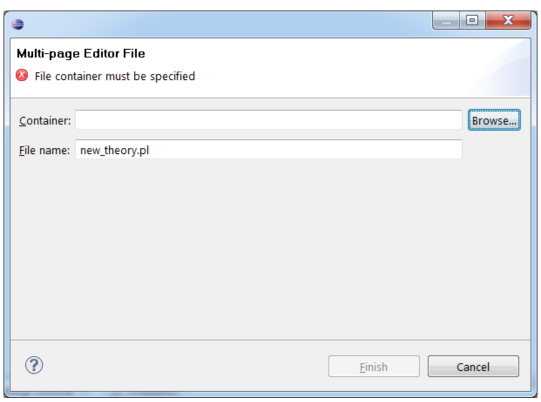

Figure 3.15: new tuProlog file

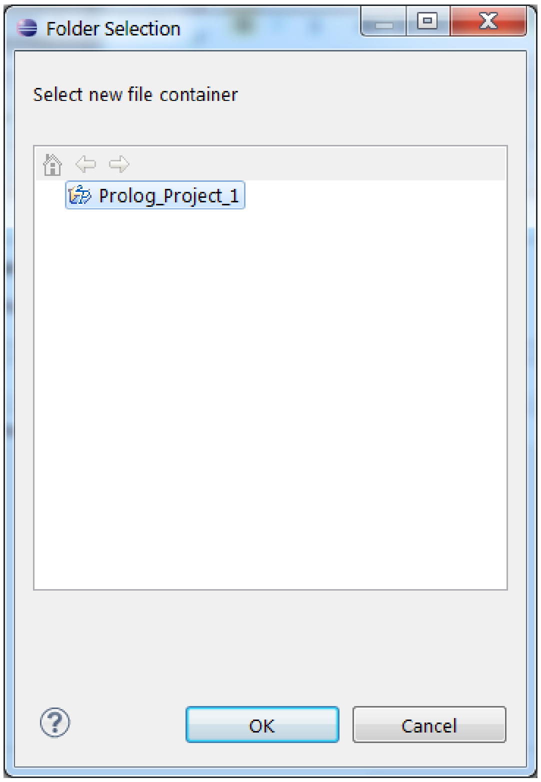

Figure 3.16: new `tuProlog` file > Browse...

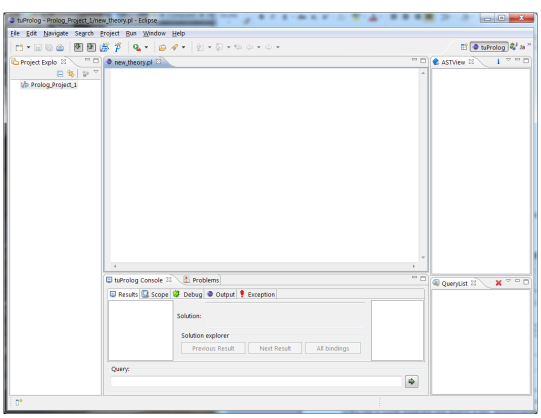

Figure 3.17: the `tuProlog` perspective

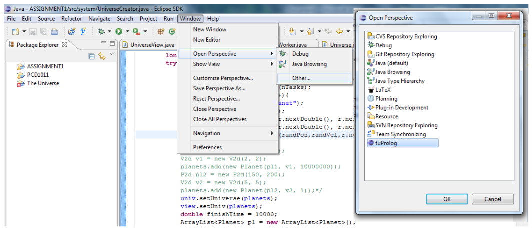

Figure 3.18: opening the tuProlog perspective

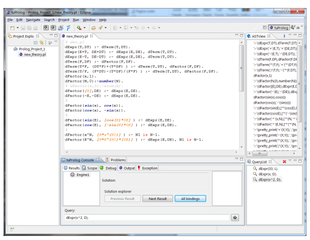

Figure 3.19: executing queries, available views

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

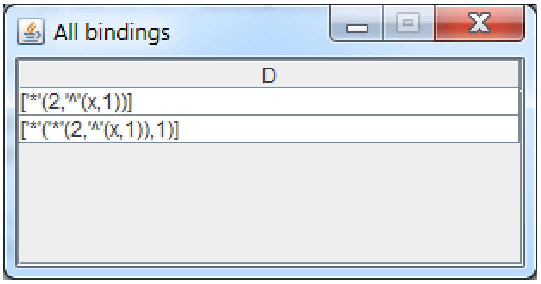

Figure 3.20: all variable bindings

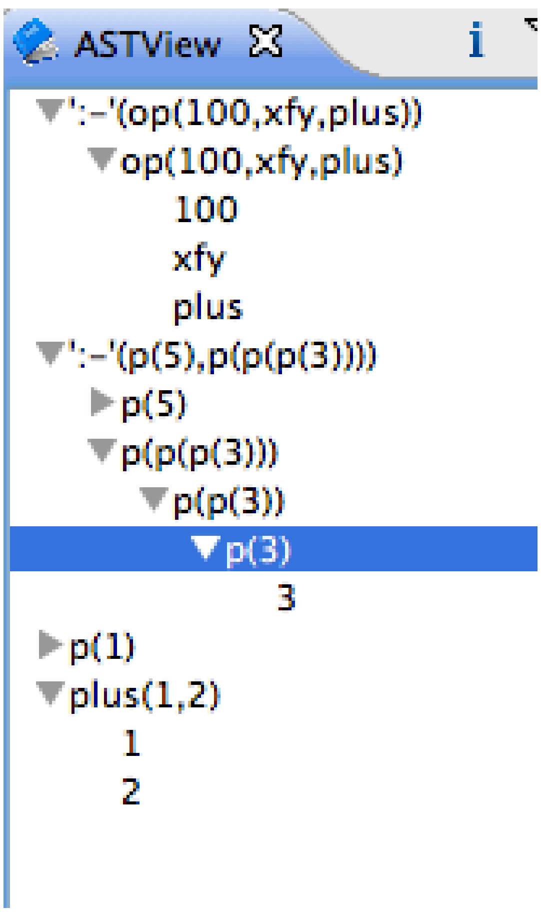

Figure 3.21: AST view (expanded)

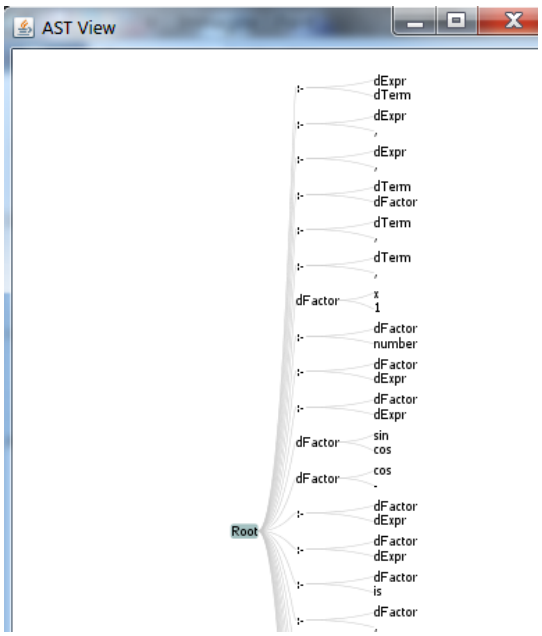

Figure 3.22: AST view (big term, expanded)

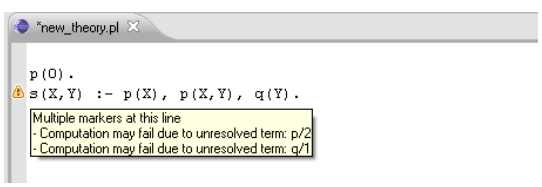

Figure 3.23: warning for undeclared terms

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 [1], 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 8.

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

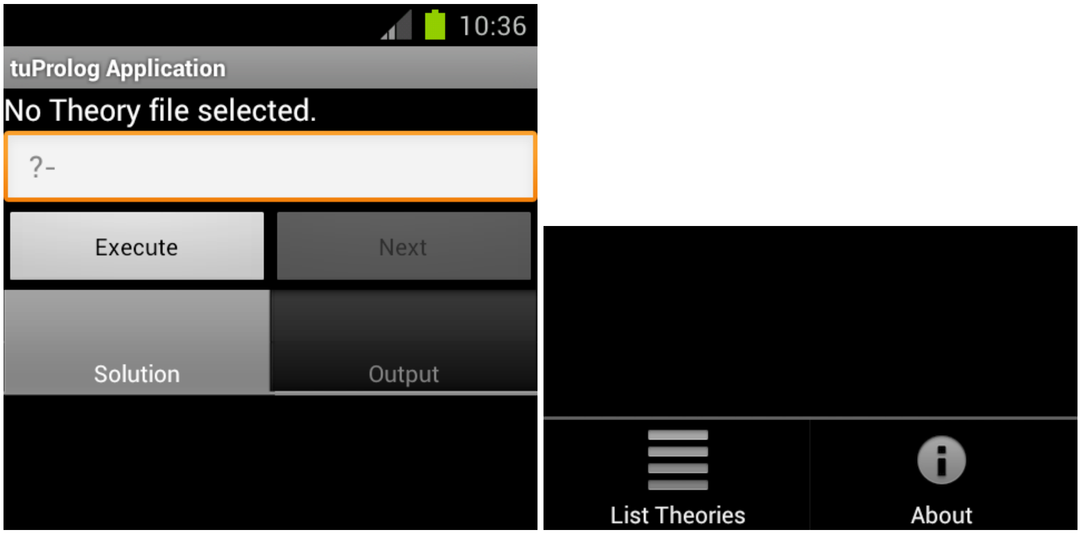

Figure 3.24: Home Activity (left) and its pop-up menu (right).

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)

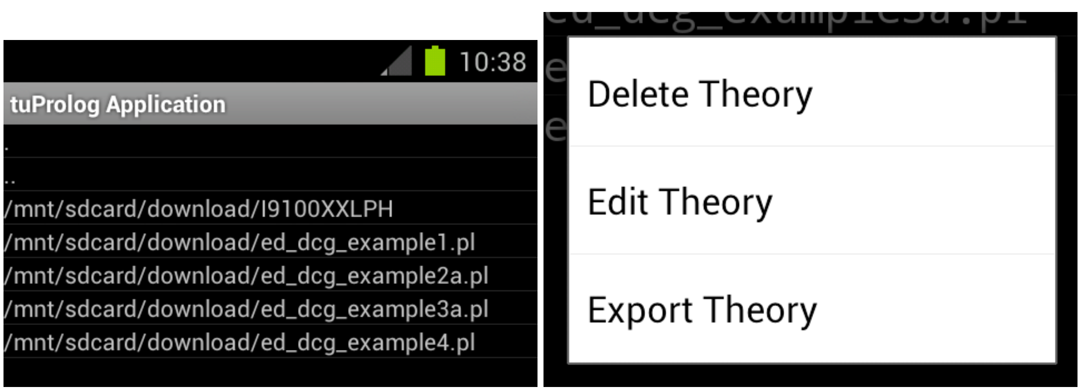

Figure 3.26: Browsing theories (left) and theory operations (right)

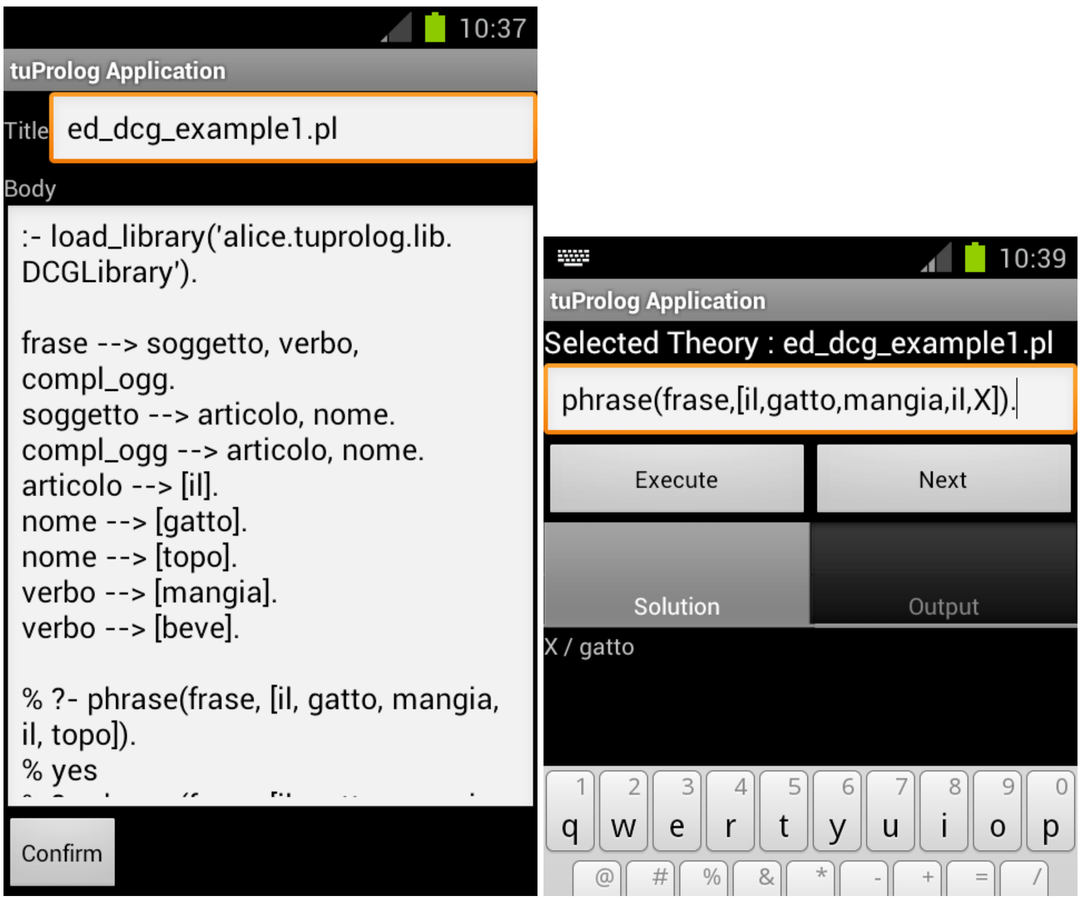

Figure 3.27: Theory editing (left) and query execution (right)

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 organized 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 `tu`Prolog engine, whereas a number of them is provided through libraries.

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

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

`tu`Prolog engines provides four levels of configurability:

Libraries — At the first level, each `tu`Prolog 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 Section 7.3). A library can be loaded by means of the predicate `load_library` (Prolog side), or by means of the method `loadLibrary` of the `tu`Prolog 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 `tu`Prolog 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, `tu`Prolog 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, `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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-ins* 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 [2], 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 `tu`Prolog: features include the creation and access of both existing and new objects, classes, and resources. In the .NET version of `tu`Prolog, this library is replaced[1] by **OOLibrary**, which extends the multi-paradigm programming approach to virtually any language supported by the .NET platform (Chapter 8.)

Other libraries included in the standard `tu`Prolog 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 `tu`Prolog distribution, because of their very specific domain: they can be downloaded from the `tu`Prolog 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 (Chapter 7, or Chapter 8 for its counterpart in .NET). 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 8 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 *occur check* [8] 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 [2] 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, `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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 `tu`Prolog 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 tuProlog 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) [3], 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 `tu`Prolog 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.

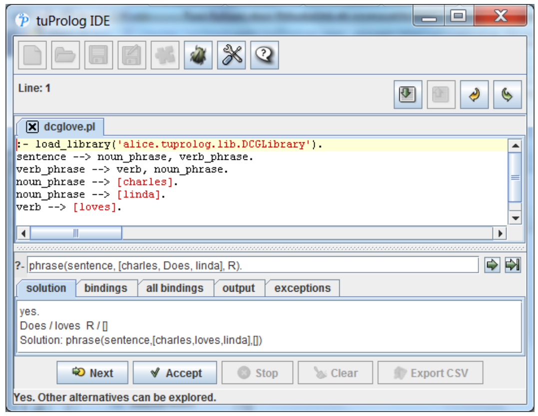

Figure 5.2: The DCG Library example in the tuProlog GUI (note the explicit library loading directive).

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

ss

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 Section 7.1). 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 7 (Section 7.1) 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 `tu`Prolog engine was extended with a new *Exception* state, between the existing *Goal Evaluation* and *Goal Selection* states [**?**].

Then, all the `tu`Prolog 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 `tu`Prolog 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 via Java: *developing new libraries*
- augmenting Java via 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 7.4.1). 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 tuProlog 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`.

To make such a binding permanent, that is, to "keep alive" the binding between a Java object and a Prolog term beyond the current query, so as to exploit it in another, subsequent demonstration, the `register` predicate is provided **(not yet available in tuProlog 2.5.1 – expected in version 2.6)**; this can also be done on the Java side, via the `tu`Prolog Java API. However, this feature should not be abused: generally speaking, when operating from the Prolog side, objects needed by other predicates (within the same demonstration) should be passed over as arguments, coherently with the Prolog philosophy of avoiding any global side effect (except for `assert` predicates).

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[1] 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] 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

(not yet available in tuProlog 2.5.1 – expected in version 2.6)

As explained above, the standard lifetime of the binding between Java objects and Prolog atoms is that of the current demonstration, in coherence with the lifetime of Prolog variable bindings. However, some multi-paradigm applications may require that a Java object is maintained alive and retrieved later without passing it along as an argument throughout the program: this is what the `register` predicate is for. Its syntax is as follows:

`register(`*`ObjectRef`*`)`

where *`ObjectRef`* is already bound to some Java object. The effect is to make such binding survive the current demonstration, until the dual `unregister` predicate is possibly called:

`unregister(`*`ObjectRef`*`)`

The requirement that *`ObjectRef`* is already bound to some Java object inherently excludes pre-existing, non-public Java objects from being registered, since the only ways to establish a binding between a Prolog atom and a Java object from the Prolog side are the `java_object` predicate, which creates a new instance, and the property selection operator (`.`,`<-get(`*`Name`*`)`), which accesses public properties. Public Java objects, including the static ones like `System.out`, can instead be registered by retrieving a reference to their binding first—as in the example shown in Table 7.2 above, where a reference to `System.out` is retrieved to call the `println` method.

Binding registration can be performed also on the Java side, as detailed in Section 7.2.4.

7.1.3 Predicates

The following predicate description details all the JavaLibrary predicates: a summary overview is also reported in Table 7.4. Throughout this Section, only the exceptions specifically related to the JavaLibrary predicates' behaviors are listed: other exceptions might obviously occur, based on the exceptions possibly raised by the invoked method, which can not be foreseen in any way.

Object creation, class compilation and method invocation

- `java_object/3`
  `java_object(ClassName, ArgList, ObjectRef)` instantiates a new instance of class `ClassName` (full class name on the local file system) and initializes it via the Java constructor corresponding to the arguments in `ArgList`; the newly created Java object is bound to the Prolog term `ObjectRef`, which can be any ground term (except for numbers) or a Prolog variable (in which case it is bound to an automatically-generated ground term). By default, such a binding is *not* undone on backtracking, unless the `java_object_backtrackable` flag is set to `true` (see Section 7.1.2).

  Template: `java_object(+full_class_name, +list, ?object_ref)`

  Exception: `java.lang.ClassNotFoundException(Cause, Message, StackTrace)` if `ClassName` does not identify a valid class name on the

Functionality	Predicate(s)	Description
Object creation	java_object(+*ClassName*, +*ArgList*, ?*ObjRef*) Examples: java_object('java.awt.Point', [2,3], P) java_object('java.lang.Integer', [303], n)	Creates a Java object of class *ClassName* by calling the constructor which matches the arguments specified in *ArgList*. If the predicates succeeds, *ObjRef* is bound to a term representing the object. *ObjRef* can be either a variable or a ground term.
Array creation	java_object(+*ClassName*[], [+*Len*], ?*ObjRef*) Example: java_object('java.awt.Point'[], [100], V)	Specialises the java_object/3 predicate for array creation.
Method invocation	*TargetRef* <- *MethodName* *TargetRef* <- *MethodName*(+*Arg0*,+*Arg1*,...) *TargetRef* <- *MethodName* returns *Res* *TargetRef* <- *MethodName*(+*Arg0*,+*Arg1*,...) returns *Res* Example 1: java_object('java.awt.Point', [2,3], P), P <- getX returns X Example 2: Intclass = class('java.lang.Integer') Intclass <- parseInt('200') returns N	Invokes the method *MethodName* on the object associated to the *TargetRef* term, possibly passing arguments *Arg0*, *Arg1*, etc., and possibly binding the return argument to the *Res* term. To invoke static (class) methods, the compound term class(*ClassName*) should be used as *TargetRef*.
Field access	*TargetRef* . *FieldName* <- set(+*Arg*) *TargetRef* . *FieldName* <- get(+*Arg*)	Accesses the public field *FieldName* of object *TargetRef* to set/get its value to the value (or object reference) denoted by *Arg*. For static fields, the compound term class(*ClassName*) should be used as *TargetRef*.
Array access and management	java_array_set(+*ArrayRef*, +*Pos*, +*Content*) java_array_get(+*ArrayRef*, +*Pos*, ?*Content*) java_array_length(+*ArrayRef*, ?*Length*) Example: java_object('java.awt.Point'[], [100], V), java_object('java.awt.Point', [1,2], Point), java_array_set(A, 0, Point)	Accesses position *Pos* of the array bound to the *ArrayRef* term to set/get the content of that position to the value (or object reference) associated to the *Content* term. The third predicate retrieves the array length and binds it to the *Length* term.
Cast	*Arg* as *TypeName*	Forces argument *Arg* to be considered of type *TypeName*.
Dynamic class compilation	java_class(+*Source*, +*ClassName*, +*PathList*, ?*ClassRef*)	Dynamically compiles the source text *Source* to define the new class named *ClassName*. *PathList* denotes the class path to be used for compilation. The compiled class, available as a Class instance, is associated to the *ClassRef* term.

Table 7.4: Summary of JavaLibrary predicates.

local file system.

Exception: `java.lang.NoSuchMethodException(Cause, Message, StackTrace)` if the specified constructor could not be found.

Exception: `java.lang.reflect.InvocationTargetException(Cause, Message, StackTrace)` if the constructor arguments are invalid—for instance, because they are not ground.

Exception: `java.lang.Exception(Cause, Message, StackTrace)` if `ObjectRef` is already bound to another Java object.

- `java_object_bt/3`
  everything as above, but the binding *is* undone on backtracking.

- `destroy_object/1`
  `destroy_object(ObjectRef)` removes the binding possibly established between `ObjectRef` and an underlying Java object.

  Template: `destroy_object(@object_ref)`

- `java_class/4`
  `java_class(SourceText, ClassName, PathList, ObjectRef)` creates the new Java class `ClassName` from the provided ***`SourceText`***, compiles it dynamically using to the classes in the provided `PathList`, and binds the result – a suitable instance of the meta-class `Class` – with the Prolog term `ObjectRef`.

  Template: `java_class(@source, @classname, @path, ?obj_ref)`

  Exception: `java.lang.ClassNotFoundException(Cause, Message, StackTrace)` if `ClassName` does not identify a valid class name in the package hierarchy on the local file system.

  Exception: `java.lang.IOException(Cause, Message, StackTrace)` if ***`SourceText`*** contains errors that prevent the class from being compiled.

  Exception: `java.lang.Exception(Cause, Message, StackTrace)` if `ObjectRef` is already bound to another Java object.

- `java_call/3`
  `java_call(ObjectRef, Method, ResultRef)` is the basic method implementing the infix `<-/2` and (`<-`,`returns`)`/3` operators below. It is true iff `ObjId` is a ground term bound to a Java object, and this provides a method `Method` (that is, whose name is the functor name and

whose arguments are the arguments of the compound term `Method`); `ResultRef` is a Prolog term bound to the returned value.

If needed, the Prolog anonymous variable _ can be used as an argument for the Java `null` value.

Template: `java_call(@obj_id, @method_signature, ?ResultRef)`

Exception: `java.lang.NoSuchMethodException(Cause, Message, StackTrace)` if the specified method could not be found in the target object/class, or the method arguments are invalid.

- `<-/2`
  `ObjectRef <- Method` calls the Java method represented by the `Method` compound term on the target Java object `ObjectRef`. The same argument specifications of `java_call` above apply.

  Template: `'<-'(@obj_id, @method_signature)`

  Exception: as above

- `(<-,returns)/3`
  `ObjectRef <- Method returns ResultRef` calls the Java method represented by the `Method` compound term on the target Java object `ObjectRef`, retrieving the method result into `ResultRef`. The same argument specifications of `java_call` above apply.

  Formally, this operator is defined as the binary `returns/2` predicate, whose first argument has the form of the above `<-/2` predicate (see Table 7.5 below for these operators' priorities).

  Template: `returns('<-'(@obj_id, @method_signature), ?@obj_id)`

  Exception: as above

Array management

- the `java_array_set_*/3` family
  This family of predicates is composed of one main predicate handling arrays of objects, and a set of helper predicates handling arrays of primitive Java types.

  `java_array_set(ArrayRef, Index, ObjectRef)` is the main predicate, setting the `Index`th cell of the array `ArrayRef` to `ObjectRef` (i.e., `ArrayRef[Index]=ObjectRef`). So, `ArrayRef` is a ground term referencing a Java array object, `Index` is a valid 0-based index for that array, and `ObjectRef` is a ground term bound to a Java object (of an assignment-compatible type according to the Java type rules) to be inserted into the array at the given position. As above, the Prolog anonymous variable can be used as `ObjectRef` to denote the Java `null` value.

Arrays of primitive Java types are handled analogously by the following set of helper predicates:

```
java_array_set_int(ArrayRef, Index, Integer)
java_array_set_short(ArrayRef, Index, ShortInteger)
java_array_set_long(ArrayRef, Index, LongInteger)
java_array_set_float(ArrayRef, Index, Float)
java_array_set_double(ArrayRef, Index, Double)
java_array_set_char(ArrayRef, Index, Char)
java_array_set_byte(ArrayRef, Index, Byte)
java_array_set_boolean(ArrayRef, Index, Boolean)
```

Template: `java_array_set(@obj_id, @nonneg_integer, +obj_id)`

Exception: `java.lang.IllegalArgumentException(Cause, Message, StackTrace)` if the `ArrayRef` does not refer to a valid array object, or `Index` is incorrect, or `ObjectRef` is not type-assignable to the array.

- the `java_array_get_*/3` family
  This family of predicates is composed of one main predicate handling arrays of objects, and a set of helper predicates handling arrays of primitive Java types.

  `java_array_get(ArrayRef, Index, ObjectRef)` is the main predicate, getting the `Index`th cell of the array `ArrayRef` into `ObjectRef` (i.e., `ObjectRef=ArrayRef[Index]`). So, `ArrayRef` is a ground term referencing a Java array object, `Index` is a valid 0-based index for that array, and `ObjectRef` is a ground term unified with the reference to the Java object (of an assignment-compatible type according to the Java type rules) at the given array position. Again, the Prolog anonymous variable can be used as `ObjectRef` to denote the Java `null` value.

  Arrays of primitive Java types are handled analogously by the following set of helper predicates:

```
java_array_get_int(ArrayRef, Index, Integer)
java_array_get_short(ArrayRef, Index, ShortInteger)
java_array_get_long(ArrayRef, Index, LongInteger)
java_array_get_float(ArrayRef, Index, Float)
java_array_get_double(ArrayRef, Index, Double)
```

`java_array_get_char(ArrayRef, Index, Char)`
`java_array_get_byte(ArrayRef, Index, Byte)`
`java_array_get_boolean(ArrayRef, Index, Boolean)`

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

Exception: `java.lang.IllegalArgumentException(Cause, Message, StackTrace)` if the `ArrayRef` does not refer to a valid array object, or `Index` is incorrect.

- `java_array_length/2`
  `java_array_length(ArrayRef, ArrayLength)` is true iff `ArrayLength` is the length of the Java array referenced by the term `ArrayRef`.

  Template: `java_array_length(@term, ?integer)`

Helper predicates

- `java_object_string/2`
  `java_object_string(ObjectRef, String)` is true if `String` is the string representation of the Java object bound to the term `ObjectRef`, according to the semantics of the object's own `toString` method.

  Template: `java_object_string(@obj_id, ?string)`

7.1.4 Functors

No functors are provided by the `JavaLibrary` library.

7.1.5 Operators

The full list of JavaLibrary operators, with their priority and associativity, is reported in Table 7.5.

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

Table 7.5: JavaLibrary operators.

7.1.6 Examples

The following examples illustrate `JavaLibrary`'s ways of use and flexibility.

RMI Connection to a Remote Object

This example shoes how to connect to a remote Java object via RMI.

To allow the reader to try this example with no need of other objects, we connect to the remote Java object `'prolog'`, an RMI server bundled with the tuProlog package that can be spawned by typing:

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

Table 7.6 shows the same code in Java and tuProlog: after the RMI (Prolog) server is retrieved, the remote `solve` method is called to execute a demonstration onto the Prolog server.

A Swing GUI

Please see the example reported in Table 7.3 above.

Database access via JDBC

This example shows how to access a database by connecting tuProlog to the Java JDBC interface. The program is logically divided in two parts, one (Table 7.7) devoted to the computational aspect (calculating the minimum path between two given cities), the other (Table 7.8) fetching the required data from the database 'distances' via JDBC.

The first part is a standard Prolog program, and requires no particular comment; the second deserves some more attention. The first line exploits the `forName` reflection method of the `Class` meta-class to obtain a reference to the (static) JDBC/ODBC driver, thus activating the JDBC bridge behind-the-scenes; then the second line opens the connection to the database via the `DriverManager`'s `getConnection` factory method: the `Connection` object is the return argument of the `init_dbase/4` predicate. Analogously, `exec_query/3` creates and executes the query statement, returning the matching data in `ResultSet`; in its turn, this is iterated over by `assert_result/3`, which asserts a `distance/3` fact for each returned tuple.

Java class:

```
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);
```

Prolog program:

```
?-  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.
```

Table 7.6: The RMI example in Java and in tuProlog via JavaLibrary.

```
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).
```

Table 7.7: Calculation of the minimum path between two given cities: the required data are fetched from a database via JDBC as shown in Table 7.8.

```
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 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(_).
```

Table 7.8: Accessing JDBC via tuProlog's JavaLibrary.

Dynamic compilation

As anticipated above, tuProlog supports the dynamic compilation of Java classes via the `java_class` predicate. This predicate compiles the source file passed as a string, and represents the newly-created class as a suitable instance of the Java `Class` meta-class, referenced by a Prolog term. In its turn, this can be used to create instances via the `newInstance` method, to retrieve constructors via the `getConstructor` method, to analyze class methods and fields, and for other meta-services.

Table 7.9 shows a simple example of this technique: the string `Source`, which contains the source of the public class `Counter`, is passed to the `java_class/4` predicate, specifying the `'Counter'` atom as the new class full name. The path list is empty, since the given class is autonomous from the compilation viewpoint. If no errors are detected, the source text is compiled and a reference to the new class is bound to the `counterClass` atom; in its turn, this is exploited to create an actual Counter object, bound to the `myCounter` term, via the `newInstance` factory method.

Now, the instance can be used as any other Java object: so, its value is set, incremented, retrieved and printed as usual.

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

Table 7.9: Dynamic compilation of a Java source code.

Table 7.10 shows a more complex example, where the source text of un unknown class is first retrieved via FTP, and then dynamically compiled and used to instantiate new objects.

7.1.7 Handling Java Exceptions

The handling of Prolog exceptions, according to the ISO standard, was already presented in Chapter 6. `tu`Prolog's peculiar support for multi-paradigm programming via Javalibrary, however, opens one extra challenge: the handling of the Java exceptions possibly raised during the execution of Java methods on objects accessed from the Prolog world.

At a first sight, one might think of re-mapping Java exceptions and constructs onto the Prolog one, but this would be an unsatisfactory approach for three main reasons:

- the semantics of the Java mechanism should not be mixed with the Prolog one, and vice-versa;
- the Java construct admits also a `finally` clause which has no counterpart in ISO Prolog;
- the Java catching mechanisms operates hierarchically, while the ISO Prolog `catch/3` predicate operates via pattern matching and unification, allowing for multiple granularities.

Accordingly, Java exceptions are handled in `tu`Prolog via two further, *ad hoc* predicates: `java_throw/1` and `java_catch/3`.

- `java_throw/1`
  `java_throw(`*`JavaException`*`(`*`Cause`*`, `*`Message`*`, `*`StackTrace`*`))` throws the specified *`JavaException`* (e.g. `'java.io.FileNotFoundException'`):

```
go :-
    get_remote_file('alice/tuprolog/test', 'Counter.java',
                    srvAddr, myName, myPwd, Content),
    java_class(Content, 'Counter', [], CounterClass),
    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),
    Client <- login(FTPUser, FTPPwd),
    Client <- chdir(DirName),
    Client <- get(FileName) returns Content,
    Client <- quit.
```

Table 7.10: Another example of dynamic compilation, where the class source is retrieved via FTP: the user `myName`, whose password is `myPwd`, gets the content of the `Counter.java` file from the server whose IP address is `srvAddr`, dynamically compiles the class and creates a corresponding object. The FTP service is provided here by a shareware Java library, but any other similar library would work.

its three arguments represent the typical properties of any Java exception, that is, the *Cause* of the exception (a string, or `0` if the cause is unknown), the *Message* associated to the error (or, again, `0` if the message is missing), and the *StackTrace* (a list of strings, each representing a stack frame).

- `java_catch/3`
  `java_catch(`*JavaGoal*`, [(`*Catcher1*`, `*Handler1*`),..., (`*CatcherN*`, `*HandlerN*`)], `*Finally*`)` performs a controlled execution of the Java operation *JavaGoal*, like in a Java `try` block: if a (Java) exception is raised, the best-matching *Catcher* is selected, and its *Handler* is executed. The *Finally* predicate expresses the homonomous Java option—actions be executed at the end of the block, independently of the operation result. If unneeded, the conventional placeholder atom (`'0'`) has to be used.

  The predicate behaviour can be informally expressed as follows. When *JavaGoal* is executed, if no exception is raised via `java_throw/1`, the *Finally* goal is executed. Otherwise, if an exception is raised, all the choicepoints generated by *JavaGoal* [2] are cut, and a matching catcher is looked for. If such a matching catcher exists, its handler is executed, maintaining the variable substitutions; otherwise, the resolution tree is back-searched 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`.

  Any side effects possibly generated during the *JavaGoal* execution are *not* undone in case of exception.

  Moreover, it should be clear that `java_catch/3` only protects the execution of *JavaGoal*, *not* of the handler or of the *Finally* predicate. So, even if *JavaGoal* is non-deterministic (like `java_object_bt/3`), and therefore allows for re-execution in backtracking, in case of exception only one handler is executed: then, all the choicepoints generated by *JavaGoal* are removed.

Java exception examples

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

[2] Of course, this is relevant only in the case of a non-deterministic predicate like `java_object_bt/3`

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

This program tries to allocate an instance of `Counter` (the counter name, *MyCounter*, is irrelevant), bind it to the atom `c` and, if everything goes well, print the `'+++'` message.

This is precisely what happens if the class `Counter` is available in the file system at run time, as expected. If, however, it is *not* present, a `ClassNotFoundException` exception is raised, and no side effects occur: so, no object is actually created, and the ***Msg*** is printed on the standard output. Finally, the `'+++'` is printed as well, according to the ***Finally*** clause. Since the ***Msg*** message in this case is the name of the missing class, the global message printed on the console is obviously `Counter+++`.

The following set of mini-examples highlight each an aspect of tuProlog compliance to the ISO standard even when these additional `java_catch` and `java_throw` predicates are considered.

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`.

In the tuProlog GUI, the details of the exception are shown in the *exceptions* tab (Figure 7.1, *bottom*), while the solution and the variable bindings (substitutions) are shown in the respective tabs (Figure 7.1, *top*).

Example 2: *execution must fail if an exception is raised during the execution of a goal and no matching* ***java_catch/3*** *is found.*

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

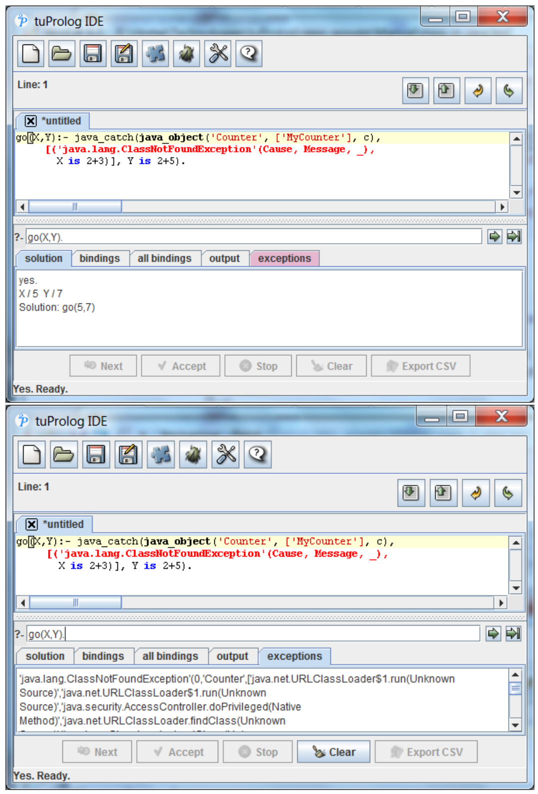

Figure 7.1: Catching the Java exceptions of Example 1 in the `tuProlog` GUI. *Top:* the solutions tab. *Bottom:* details of the exception in the exception tab (see the `Cause` variable bound to `0` and the `Msg` variable bound to `'Counter'`; the other details map onto the anonymous variable _).

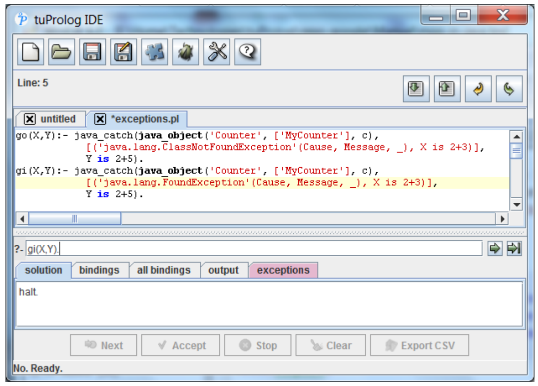

Figure 7.2: A failed exception in the tuProlog GUI: the `No` answer in the status bar and the *halt* message in the Solutions tab.

Answer: `no.`

In the tuProlog GUI, a failed exception not only results into a "No" answer as in other Prolog systems (that answer is shown in the status bar at the bottom of the window: it also causes the *halt* message to appear in the Solutions tab (Figure 7.2).

Example 3: *`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 4: *`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 5: *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 6: *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*

The tuProlog Java API provides a complete support for exploiting Prolog engines from Java: its only requirement is the presence of `tuprolog.jar` (or the more complete `2p.jar`) in the Java project's class path. The API defines a namespace (`alice.tuprolog`) and classes to enable the definition in Java of suitable objects representing Prolog entities (terms, atoms, lists, variables, numbers, etc, but also Prolog engines, libraries and theories), and use them to submit queries and get the results back in Java, thus effectively supporting multi-paradigm, multi-language programming.

7.2.1 A Taxonomy of Prolog types in Java

Prolog types are mapped onto suitable Java classes, organized into the taxonomy shown in Figure 7.3 and summarized in Table 7.1 on page 93.

`Term` is the root abstract class, providing common services such as term unification, term parsing, term copying, etc.; its subclasses distinguish among untyped terms (structures), numbers, and variables.

`Struct` objects are characterized by a functor name (a Java string) and a list of arguments, which are `Term`s themselves and can be individually retrieved via the `getTerm` method.

Atoms are a special case of `Struct` with no arguments; among these, the `true` and `false` atom constants are used to represent the Java boolean values. Atoms are also used to map Java Java `char`s and strings: when converted back to Java, however, atoms are always mapped into Java `String`s.

Prolog lists are another special case of `Struct`, built from either two `Term`s (the list head and tail) or an array of `Term`s; by convention, the default constructor builds the empty list.

The `Number` subtree includes classes for numeric types, and offers methods such as `intValue`, `longValue`, etc. to retrieve the number value as the corresponding primitive Java value. As discussed above, in the Prolog-to-

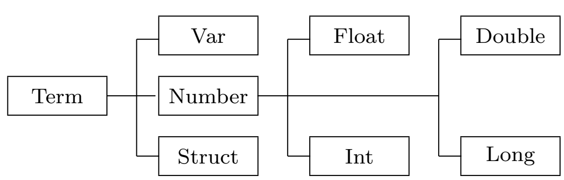

Figure 7.3: Prolog entities as a taxonomy of Java classes.

Java direction, Prolog integers are always mapped onto `Int` instances and Prolog reals onto `Double` instances, while in the Java-to-Prolog direction any of the numeric Java types are accepted (including `short`, `byte` and `float`) for mapping onto Prolog numbers. In particular, Java `int` and `long` values are mapped onto suitable `Int` and `Long` instances of the `tu`Prolog taxonomy, respectively, while `byte` and `short` Java types are mapped into `Int` instances. Please note that to avoid possible name clashes between `tu`Prolog types and Java wrapper classes (e.g. `alice.tuprolog.Long` and `java.lang.Long`), it is often necessary to use the fully qualified class name to denote `tu`Prolog numeric classes.

`Var` represents Prolog variables, built from a Java string representing the variable name: as prescribed by the Prolog rules, the name must start either with a capital letter, or with an underscore. The default constructor builds the anonymous Prolog variable _, mapped onto the Java `null` value.

Table 7.11 on page 119 shows how to manipulate Prolog entities (variables, terms, structures, lists, atoms..) from a Java program: variable creation (lines 1 and 10), list construction (lines 2–4), term construction for `p(a,5)` and `p(X,Y)` (lines 5–6), and term unification (lines 7–14) (the latter requires a Prolog engine as a mediator, to handle execution contexts and inner variables).

It is worth noting that, in general, two different `Var` objects with the same Java name *do not* refer to the same Prolog variable, unless they occur in the same term. So, multiple occurrences of `new Var("Y")` outside the same term refer to two distinct variables, as if they were renamed `Y1` and `Y2`. To refer to the same Prolog variable twice, just use the same Java identifier (see `varY` in lines 1, 3, 12) instead of creating a new variable.

The only exception is the case when the homonymous variables occur in the same term, as in the `q(Y,Y)` term in line 15: then, they will refer to the same variable, *but only after the term has been resolved*. In fact, new terms are always built in an 'unresolved' form, that does not analyze the term variables: the proof is the *`false`* output of line 16. Variables are taken into consideration later, when the term is either explicitly resolved via `resolveTerm` (line 17), or is involved in a `match` or `unify`[3] operation with another term (lines 18-19), as proved by the *`true`* output of line 20.

Further notes about `Terms`

The `Term` class is the home of several general-purpose services, used throughout `tu`Prolog; in particular:

[3] `q(Y,Y)` is unified here with the Prolog anonymous variable, so success is granted.

```
import alice.tuprolog.*;
...
Var varX = new Var("X"), varY = new Var("Y");
Struct atomP = new Struct("p");
Struct list = new Struct(atomP, varY);       // should be [p|Y]
System.out.println(list);               // prints the list [p|Y]
Struct fact = new Struct("p", new Struct("a"), new Int(5));
Struct goal = new Struct("p", varX, new Var("Z"));
Prolog engine = new Prolog();
boolean res = goal.unify(engine,fact);     // should be X/a, Y/5
System.out.println(goal);     // prints the unified term p(a,5)
System.out.println(varX);     // prints the variable binding X/a
Var varW = new Var("W");
res = varW.unify(engine,varY);                 // should be Z=Y
System.out.println(varY);  // prints just Y, since it is unbound
System.out.println(varW);  // prints the variable binding W / Y
Struct st = Struct("q", new Var("Y"), new Var("Y")); // unresolved
System.out.println(st.getArg(0)==st.getArg(1));  // prints false
st.resolveTerm();                    // now the term is resolved
  alternatively:  res = st.match(new Struct());
  alternatively:  res = st.unify(engine, new Struct());
System.out.println(st.getArg(0)==st.getArg(1));  // prints true
```

Table 7.11: Manipulating Prolog entities from Java.

- the static `parse` and `createTerm` methods provides a quick way to get a term from its string representation;

- the `match` and `unify` methods respectively check for term matching (but performing no actual unification) and unify the given term with the provided one; as anticipated above, the latter requires a `Prolog` argument, to be used as a mediator during (nested) unification; Instead, the matching test is performed outside any demonstration context.

- the `equals` method compares terms with the same semantics of the method `isEqual`, which follows the Prolog comparison semantics.

- the `getTerm` method returns the referred term, following variable bindings—that is, if the target term is a bound variable, the term bound to the variable (not the variable itself) is returned.

7.2.2 Prolog engines, theories and libraries

The tuProlog engine is made accessible in Java via the `Prolog` class: so, adding intelligence to a Java program is as easy as creating `Prolog` instance(s), configure it (them) as needed, and perform the desired queries. Query results are expressed as an instance of the `SolveInfo` helper class. Table 7.12 reports the public interface of these classes.

A Prolog engine is built by one of `Prolog` constructors: the default constructor builds a default engine, with the default set of tuProlog libraries loaded, and no user theory. In most cases, this is all you need to bring the power of Logic programming to Java. However, libraries can be loaded and unloaded dynamically at any time after the engine creation, via the `loadLibrary` and `unloadLibrary` methods: their argument is the name of the library. If the library is invalid, an exception is raised. A reference to a loaded library can be obtained via the `getLibrary` method, which returns a reference to the abstract `Library` class. Such a reference can be used to operate on the library, as discussed below.

The user theory can either be set from scratch via the `setTheory` method, which overwrites any previous theory, or be built incrementally, adding new clauses to the existing theory via the `addTheory` method: both take a `Theory` as their argument. This theory can be built in several ways—from an input stream, from a string, or from a clause list (represented as a `Struct` object). The current theory can be retrieved via the `getTheory` method.

Goal resolution is handled via three methods: `solve`, `solveNext`, and `hasOpenAlternatives`. `solve` and `solveNext` take as their argument a `Struct` representing the goal, and return a `SolveInfo` which encapsulates the result information (success or failure, solution, variable bindings, etc). An overloaded version of `solve` takes a string argument representing the text of the goal, embedding its parsing. Both `solve` and `solveNext` raise the proper exceptions when needed.

Further notes about `Prolog` engines

The `Prolog` class is the home of tuProlog engines, so some further information is opportune about its behavior in particular contexts:

- 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 engines (`solve/1`, `initialization/1`, `set_prolog_flag/1`, `load_library/1`, `consult/1`), format and syntax of read-terms (`op/3`, `char_conversion/2`).

```
public class Prolog implements Serializable {
  ...
  public void setTheory(Theory t) throws InvalidTheoryException {...}
  public void addTheory(Theory t) throws InvalidTheoryException {...}
  public Theory getTheory() {...}
  public Library loadLibrary(String name)
                                throws InvalidLibraryException {...}
  public void unloadLibrary(String name)
                                throws InvalidLibraryException {...}
  public Library getLibrary(String name) {...}
  public SolveInfo solve(Term goal) {...}
  public SolveInfo solve(String goalAsString)
                             throws MalformedGoalException {...}
  public boolean hasOpenAlternatives() {...}
  public SolveInfo solveNext() throws NoMoreSolutionException {...}
  public boolean isHalted() {...}
}
```

```
public class SolveInfo implements Serializable {
  public boolean isSuccess() {...}
  public Substitution getSubstitution()
                             throws NoSolutionException {...}
  public Term getTerm() throws UnknownVarException {...}
  public Term getSolution() throws NoSolutionException {...}
}
```

Table 7.12: Classes for interacting with tuProlog engines.

- engines also support 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.

- engines are thread-safe.

- engines have no (static) dependencies with each other, can be created independently on the same Java virtual machine, are very lightweight, and can be serialized. This is true also for engines with the standard libraries pre-loaded: obviously, if other libraries are loaded, these must be serializable, too, for the engine to remain serializable.

7.2.3 Examples

For the sake of concreteness, some examples of use of the `tu`Prolog Java API are now discussed.

Appending lists

In this first example (see Table 7.13, *top*), a `tu`Prolog engine is asked to solve a trivial list append goal, provided in textual form.[4] The program must be compiled and executed normally, taking care of including the `tu`Prolog JAR in the classpath:

```
javac -cp tuprolog.jar;.  Example1.java
java  -cp tuprolog.jar;.  Example1
```

The string `append([1],[2,3],[1,2,3])` should be displayed.

Table 7.13, *bottom* shows a variant where all the solutions are displayed, with their variable bindings. The output should be as follows:

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

A console-based Prolog interpreter

As a final example, Table 7.14 shows a console-based Prolog interpreter: first a `tu`Prolog engine is created and initialized with a theory built from

[4]The `append/3` predicate is included in BasicLibrary, which is part of the engine default configuration.

Basic version:

```
import alice.tuprolog.*;

public class Example1 {
    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());
    }
}
```

Variant:

```
import alice.tuprolog.*;

public class Example2 {
    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;
            }
        }
    }
}
```

Table 7.13: The list appending example.

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

public class ConsoleInterpreter {
 public static void main (String args[]) throws Exception {
  Prolog engine=new Prolog();
  if (args.length>0)
      engine.setTheory(new Theory(new FileInputStream(args[0])));
  BufferedReader stdin =
      new BufferedReader(new InputStreamReader(System.in));
  while (true) {     // interpreter main loop
   String goal;
   do { System.out.print("?- "); goal=stdin.readLine();
   } while (goal.equals(""));
   try {
    SolveInfo info = engine.solve(goal);
    if (engine.isHalted()) break;
    else if (!info.isSuccess()) System.out.println("no.");
    else if (!engine.hasOpenAlternatives()) {
      System.out.println(info);
    } else { // main case
      System.out.println(info + " ?");
      String answer = stdin.readLine();
      while (answer.equals(";") && engine.hasOpenAlternatives()) {
        info = engine.solveNext();
        if (!info.isSuccess()) { System.out.println("no."); break; }
        else {
               System.out.println(info + " ?");
               answer = stdin.readLine();
        } // endif
      } // endwhile
      if (answer.equals(";") && !engine.hasOpenAlternatives())
          System.out.println("no.");
    } // end main case
   } catch (MalformedGoalException ex) {
          System.err.println("syntax error.");
   } // end try
  } // end main loop
  if (args.length>1) {
   Theory curTh = engine.getTheory(); // save current theory to file
   new FileOutputStream(args[1]).write(curTh.toString().getBytes());
  }
 }
}
```


Table 7.14: A simple console-based Prolog interpreter.

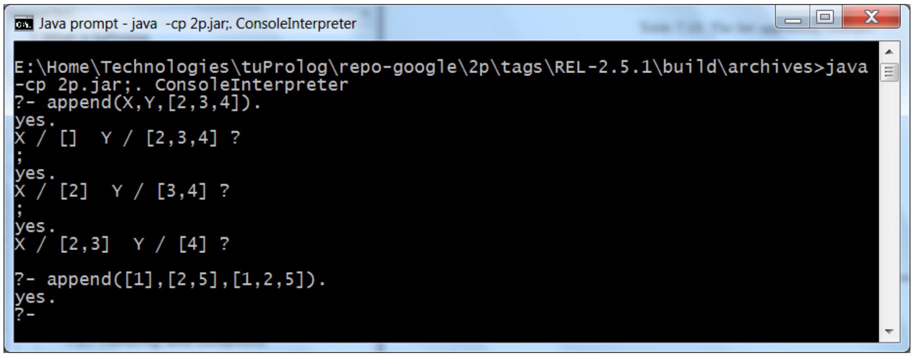

Figure 7.4: A sample session with the Console-based Interpreter.

a text file (whose name is taken from the command line), then a classic read/solve loop is started. For each goal read from the standard input, the `solve` method is invoked: if multiple solutions exist, the `solveNext` makes it possible to explore the open alternatives. The loop ends when the `halt` predicate is typed in: the current theory is then saved to file (if any has been specified). Figure 7.4 shows a sample session with this interpreter.

7.2.4 Registering object bindings

The `register` function, already discussed in Section 7.1.2 on page 99 for what concerns the Prolog side, is also available on the Java side, where its 'global' effect is more natural and coherent with the imperative paradigm than it is on the Prolog side. Its purpose is to permanently associate an existing Java object *obj* to a Prolog identifier *ObjectRef*, as follows:

```
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, let us suppose that we want to permanently bind the Prolog atom `stdout` to the Java (static) object `System.out`, so that Java-based printing can be done from the Prolog side without having to retrieve and re-bind the `out` object every time, as we did in Table 7.2 on page 98 (reported again below for convenience):

```
class('java.lang.System') . out <- get(Out),
Out <- println(...),
```

To bind `System.out` permanently to `stdout` (within the scope of the tuProlog engine `engine`), we can register it as follows:

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

An explicit downcast to `JavaLibrary` is needed to convert the returned reference type `Library`, since `register` is defined in `JavaLibrary` only. Now, a Prolog theory loaded into this `engine` can contain a phrase like:

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

which uses `stdout` directly as a target for the `println` method.

A small yet complete sample program is shown in Table 7.15, where the theory loaded into the `engine` prints the standard greetings message.

```
import alice.tuprolog.*;
import alice.tuprolog.lib.*;

public class StdoutExample {
  public static void main(String[] args) throws Exception {
  Prolog engine = new Prolog();
  Library lib = engine.getLibrary("alice.tuprolog.lib.JavaLibrary");
  ((JavaLibrary)lib).register(new Struct("stdout"), System.out);
  engine.setTheory(new Theory(
      ":-solve(go). \n go:- stdout <- println('hello!')."));
  }
}
```

Table 7.15: A program registering `stdout` for `System.out`. As an alternative to `getLibrary`, `loadLibrary` could have been used—if the library is already loaded, its behavior is identical to `getLibrary`'s. Also, the fully qualified class name `"alice.tuprolog.lib.JavaLibrary"` is needed in `getLibrary` only because `JavaLibrary` does *not* define a short library name (see Section 7.3.4 for details): otherwise, the shorter name could have been used.

7.2.5 Capturing the Prolog output in Java

If a tuProlog engine is used in a Java application, the output performed by Prolog `write` predicates (more generally, of any predicate writing on the Prolog console) is not available in Java: printed messages are not captured, nor are they retrievable by any of the tuProlog Java API methods. The only way to 'capture' somehow the output of the Prolog engine is to write it to a file or store it in a Prolog term—just two variants of the same inconvenience.

Yet, this feature can be added in a non-intrusive way, thanks to tuProlog's extensible architecture, by simply overriding the `onOutput` method used internally by the engine to handle the write requests.[5] All is needed is to redefine this method so as to capture the output message and store it conveniently—for instance, into a suitable `String` of the Java application (here, `finalResult`), as follows:

```
engine.addOutputListener(new OutputListener() {
  @Override
  public void onOutput(OutputEvent e) {
    finalResult += e.getMsg();
  }
});
```

This elegant approach does not modify the tuProlog code in any way: it just adds listener to an existing event, extending the service non-intrusively. A full example of this technique is reported in Table 7.16 on page 128, together with the corresponding build process and execution.

7.3 Augmenting Prolog via Java: developing new libraries

So far, the two first dimensions of tuProlog's support to multi-paradigm, multi-language programming have been explored, that enable a language (and the corresponding paradigm) to be used from the other. The two further dimensions concerns *augmenting* the language instead—that is, exploiting a language (and a paradigm) to increase the other.

In this section the focus is on augmenting Prolog from Java, exploiting the latter[6] to increase the first by developing new tuProlog libraries; the

[5] This approach was originally suggested by Josh Guzman in the tuProlog users' forum.
[6] Other languages may be used indirectly, via JNI (JavaNative Interface

```
import alice.tuprolog.*;
import alice.tuprolog.lib.*;
import alice.tuprolog.event.*;

public class OnOutputExample {
  static String finalResult = "";
  public static void main(String[] args) throws Exception {
    Prolog engine = new Prolog();
    engine.addOutputListener(new OutputListener() {
        @Override
        public void onOutput(OutputEvent e) {
          finalResult += e.getMsg();
        }
    });
    Term goal = Term.createTerm("write(’Hello world!’)");
    SolveInfo res = engine.solve(goal);
    res = engine.solve("write(’Hello everybody!’), nl.");
    System.out.println("OUTPUT: " + finalResult);
  }
}
```

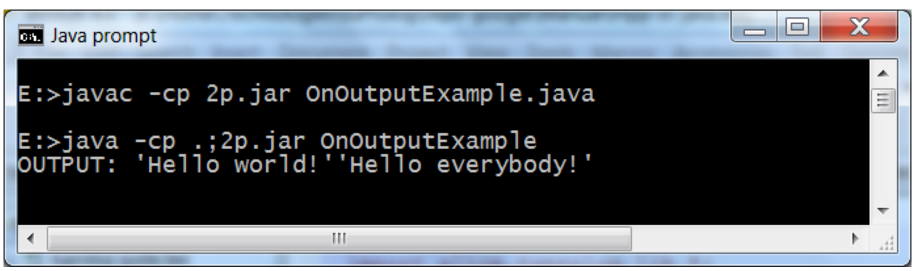

Table 7.16: Capturing the Prolog output from Java: a complete example.

next Section (7.4) will focus on the opposite direction, exploiting Prolog to augment Java via the so-called *P@J* framework.

Moreover, although tuProlog libraries are expressed in Java, they are not required to be fully implemented in this language. In fact, Java-only libraries are the simplest case, but hybrid Java + Prolog libraries are also possible, where a Prolog theory is embedded into a Java string so that the two parts cooperate to define the overall library behavior. This opens further interesting perspectives, that will be discussed below.

7.3.1 Syntactic conventions

Each library must extend the base abstract class `alice.tuprolog.Library` and define new *predicates* and/or *evaluable functors* and/or *directives* in the form of methods, following a simple signature convention.
Predicates must adhere to the signature:

```
public boolean <pred name>_<N>(
         <? extends Term> arg1, ..., <? extends Term> argN)
```

while evaluable functors must follow the form:

```
public Term <eval funct name>_<N>(
         <? extends Term> arg1, ..., <? extends Term> argN)
```

and directives must be provided with the signature:

```
public void <dir name>_<N>(
         <? extends Term> arg1, ..., <? extends Term> argN)
```

where *arg1*, ... *argN* are `Terms`[7] that represent the actual arguments passed to the predicate (functor, directive).

Table 7.17 shows a library defining an evaluable functor (`sum/2`) and two predicates (`println/1`, `invert/2`). 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 is returned, representing success or failure (`true` = success in this case). Analogous considerations hold for `invert/2`, whose input argument is first type-checked to handle variables appropriately (the related bound term must be retrieved), then the input term is scanned to build the output string, which is finally unified with the output variable.

A test Java program, which loads this library and tests its predicates, is shown in Table 7.18. The program creates the Prolog engine, loads `TestLibrary` (checking that it was actually loaded), defines a theory containing the Prolog test code and sets it into the engine: then, the three test goals are solved in sequence. The printed output is reported in the bottom

[7] Please refer to Table 7.3 on page 117 for the full Term taxonomy.

```
import alice.tuprolog.*;
public class TestLibrary extends Library {

  // functor sum(A,B)
  public Term sum_2(Number arg0, Number arg1){
    float n0 = arg0.floatValue();
    float n1 = arg1.floatValue();
    return new Float(n0+n1);
  }

  // predicate println(Message)
  public boolean println_1(Term arg){
    System.out.println(arg);
    return true;
  }

  // predicate invert(StringIn,StringOut)
  public boolean invert_2(Term in, Var out){
    String s1 = null, s2 = "";
    if (in instanceof Var) s1 = in.getTerm().toString();
    else s1 = in.toString();
    for(int i=0; i<s1.length(); i++){
      char ch = s1.charAt(i);
      if (ch=='\'') continue;
      if (Character.isUpperCase(ch))
        s2 += Character.toLowerCase(ch);
      else
        s2 += Character.toUpperCase(ch);
    }
    return out.unify(getEngine(),new Struct(s2));
  }
```

Table 7.17: Definition of a tuProlog library in Java.

part of the Table. The *Name / Value* format is the tuProlog's default for variables, and is *Name* is composed of the Prolog variable name (N, S, etc.) and of a unique internal identifier. As expected, N is bound to 11, S to abcd, the X and Z pair to ab/'AB', bc/'BC' and uk/'UK', respectively.

When developing libraries, two naming issues may arise:

1. the name of the predicate, functor or directive should contain a symbol that cannot legally appear in a Java method's name;

2. a predicate and a directive with the same Prolog signature should be defined, but Java would not be able to distinguish method signatures differing for the return type only.

To overcome these issues, a *synonym map* must be set up, that maps the desired Prolog names onto legal Java method names, bypassing the standard naming convention. This map must have the form of an array of String arrays, and be returned by the ad hoc getSynonymMap method (abstract in the base Library class). For instance, an evaluable functor +, which cannot appear in a Java method name, could be implemented by a defining a Java method with any name (say, add) and then map it onto the Prolog name by adding the array {"+", "add", "functor"} to the synonym map.

Libraries can also inherit from each other: a library can well extend a user library instead of the base Library, as in the case of the HybridLibrary discussed in the next Section.

7.3.2 Hybrid Java+Prolog libraries

Since Java does not support non-determinism, a Java-only library is inherently deterministic: however, non-determinism can be achieved via hybrid Java + Prolog libraries, adding a Prolog layer on top of the Java layer.

To this end, a library can include a new piece of Prolog theory, embedded into the getTheory method. This method returns a string[8] (empty by default) containing the desired Prolog theory, and is automatically called when the library is loaded, so as to add the theory to the engine's configuration.

Table 7.19 shows a hybrid library where the theory expressed by getTheory adds to TestLibrary the non-deterministic predicate myprint/1, whose (potentially infinite) solutions alternately print the argument in upper and lowercase.

[8] In principle, only the external representation of this theory is constrained to the String form, the internal implementation being up to the developer; yet, using a Java String for wrapping the Prolog code guarantees self-containment while loading libraries through remote mechanisms such as RMI, and therefore constitutes the suggested form.

```
import alice.tuprolog.*;
import alice.tuprolog.lib.*;

public class TestLibraryMain {
  public static void main(String[] args) throws Exception {
    Prolog engine = new Prolog();
    Library lib1 = engine.loadLibrary("TestLibrary");
    System.out.println(
        "Lib1 " + (lib1==null ?  "NOT " :  " ") + "LOADED");
    Theory testTheory = new Theory(
        "test1 :- N is sum(5,6), println(N).\n" +
        "test2 :- invert('ABCD',S), println(S).\n" +
        "test3 :- name(X), println(X)," +
                 "invert(X,Z), println(Z), fail.\n" +
        "name(ab).\n name(bc).\n name(uk).\n");
    engine.setTheory(testTheory);
    SolveInfo res = engine.solve("test1.");
    res = engine.solve("test2.");
    res = engine.solve("test3.");
  }
}
```

OUTPUT PRINTED:

```
Lib1 LOADED
N_e2 / 11.0
S_e2 / abcd
X_e11 / ab
Z_e12 / 'AB'
X_e13 / bc
Z_e14 / 'BC'
X_e15 / uk
Z_e17 / 'UK'
```

Table 7.18: A test program for the library defined in Table 7.17 *(top)* and the corresponding output *(bottom)*.

```
public class HybridLibrary extends TestLibrary {
 public String getTheory(){
  return "myprint(X) :- println(X).\n" +
         "myprint(X) :- invert(X,Y), myprint(Y).\n";
 }
}
```

```
import alice.tuprolog.*;
import alice.tuprolog.lib.*;

public class HybridLibraryMain {
  public static void main(String[] args) throws Exception {
    Prolog engine = new Prolog();
    Library lib2 = engine.loadLibrary("HybridLibrary");
    SolveInfo res = engine.solve("myprint(henry).");
int count=0;     while (engine.hasOpenAlternatives() &&
count < 5){
        count++;
        res = engine.solveNext();
    }
```

OUTPUT PRINTED:

```
Lib2 LOADED
X_e1 / henry
X_e5 / yrneh
X_e9 / henry
X_e11 / yrneh
X_e13 / henry
X_e15 / yrneh
```

Table 7.19: A hybrid (mixed) Java + Prolog library *(top)* and the corresponding test program *(bottom)*.

7.3.3 Library loading issues

As shown in the above examples, a library can be loaded (and unloaded) dynamically into a running engine via Java, by means of the `loadLibrary` (`unloadLibrary`) methods; but it can also be loaded (unloaded) from Prolog, via the `load_library/2` (`unload_library/2`) predicate.

As of version 2.5.1, the two approaches are not completely identical, at least if `2p.jar` is used to start the `tu`Prolog GUI (or the console-based CUI) from which the loading is performed. In fact, library loading from Prolog suffers from some limitations due to the Java reflection mechanism used internally by `tu`Prolog to locate the library,[9] which does not search outside the JAR. So, only libraries packed into the same `tu`Prolog JAR can be found and loaded from Prolog via the `load_library/2` predicate. To bypass the problem, currently the only solution is to unpack the JAR archive: **in the near future, the implementation will be improved using the URLClassLoader which can load classes from external JARs, so as to remove the limitation.**

This is why the examples above (`TestLibrary` and `HybridLibrary`) have been loaded from the Java side, via two main programs (`TestLibraryMain` and `HybridLibraryMain`).

7.3.4 Library Name

The concept of library name is introduced in `tu`Prolog to separate the physical class name of a library from its logical name, both for clarity – the library name can be shorter and more meaningful – and to support multiple versions of the same library, enabling the dynamic upgrade of a library implementation.

By default, the library name is identical to the class name: however, a library can specify a different name by overriding the `getName` method. Obviously, the full class name is always needed when loading the library, while the library name is used by `getLibrary` (and similar predicates) to return references to already-loaded libraries.

As an example, in Table 7.20 the `NewStringLibrary` class provides an alternate implementation of `StringLibrary`: this is why it `getName` is redefined so as to return `StringLibrary` as the `NewStringLibrary` library name.

[9] The culprit is the `Class.forName` method, used instead of `loadClass` to maintain the compatibility with the .NET platform. This is why the problem was not present in older (<2.2) `tu`Prolog versions.

```
public class StringLibrary extends Library {
    public boolean to_lower_case_2(Term source, Term dest){
        String st = source.toString().toLowerCase();
        return unify(dest, new Struct(st));
    }
  ...
  // the inherited getName returns "StringLibrary"
  ...
}
```

```
public class NewStringLibrary extends Library {
  public String getName(){ return "StringLibrary"; }
  ...
}
```

Table 7.20: Defining a new library with the same name as another.

7.4 Augmenting Java via Prolog: the P@J framework

The last dimensions of `tu`Prolog's support to multi-paradigm, multi-language programming is still a form of *augmenting* a language (that is, exploiting a language and a paradigm to increase the other)—in this case, augmenting Java from Prolog, exploiting the so-called *P@J* framework [4].

This approach makes it possible to "inline intelligence" into Java code, enabling Prolog to be used for implementing Java (abstract) methods, via Java reflection and suitable annotations. The basic idea is that the methods to be implemented in Prolog are declared `abstract` from the Java syntax viewpoint[10], so that the Java compiler does not expect to find any implementation, while annotating them with the Prolog clauses that provide the actual implementations. On the user side, the factory method `PJ.newInstance` will be used to automatically create a Java implementation of this method, which interacts with the Prolog engine in a totally transparent way.

[10] Of course, the corresponding class must be syntactically qualified `abstract`, too.

The technique relies on advanced features of Java such generic types, wildcards, and type inference, as well as reflection to "put things together"; for this reason, some syntax conventions are required for method signatures:

- the Prolog predicate name must be identical to the Java method name;
- the argument types must be explicitly declared each with the corresponding bounding, and their names must start with `$`;
- the argument position in the Java method signature must reflect their role as input or output arguments in the Prolog predicate: the first are to be put in the argument list, and the latter in the return type.

The last requirement is necessary to bridge between the Prolog predicate syntax, where both input and output arguments are in the argument list (with nothing explicitly qualifying these roles, according to the declarative nature of the language), and the Java method syntax, where the only output argument is not in the argument list, but is "returned from" the method.

7.4.1 Term taxonomy

Here, too, a suitable taxonomy is needed to map the relevant Prolog types (term, atom, number, list, variable, etc) in Java; however, while the domain to represent is the same as above (Section), the requirements due to type inference and strong type checking made it necessary to define one further, *ad hoc* taxonomy as the base of the annotation layer.

The new hierarchy exploits the basic types in Figure 7.3 on page 117 as its building bricks, and builds a new layer on top. The new root is the abstract class `Term<X>`, whose definition exploits a recursive pattern to reify (represent) the type of the actual term content:

```
abstract class Term<X extends Term<?>> {..}
```

Accordingly, the term subclasses are defined as:

```
class Atom extends Term<Atom> {..}
class Int extends Term<Int> {..}
class Double extends Term<Double> {..}
class List<X extends Term<?>> extends Term<List<X>> {..}
...
```

where, clearly, `Term<Int>` is used for a term containing an `Int`, `Term<Double>` for a term containing a `Double`, `Term<List<Int>>` for a term containing a `List<Int>`, etc.

Variables are a notable exception, because they must be able to contain values of the above types: for this reason, `Var<X>` is *not* defined as a subtype of `Term<Var<X>>`, but directly of `Term<X>`.

```
class Var<X extends Term<?>> extends Term<X> {..}
```

As a consequence, both types `X` and `Var<X>` derive from the common ancestor Term¡X¿, which makes it possible to represent method arguments that may be a logical input or output—i.e., that must accept both a value (a term of type `X`) or a variable (a term of type `Var(X)`).

Thanks to this approach, a method definition like the following:

```
boolean length(Term<? extends List<?>> list, Term<Int> size)
```

can be read as follows:

- list is a term containing any list, and size is an integer;
- both arguments can be either input or output.

The term hierarchy is completed by the `Compound` term family, which enables the definition of compound terms of any arity by means of a list-like approach—that is, starting from the empty compound term `Nil` and building bigger compounds with the `Cons` class constructor. However, shortcut classes `Compound1`, `Compound2` and `Compound3` are provided for the user convenience to specify the most common terms of 1, 2 or 3 arguments:

```
public abstract class Compound<X extends Compound<?>>
  extends Term<X> {..}

public class Cons<H extends Term<?>, R extends Compound<?>>
  extends Compound<Cons<H,R>> implements Iterable<Term<?>>{..}

public class Nil extends Compound<Nil> {..}

public class Compound1<X1 extends Term<?>>
  extends Cons<X1,Nil> {..}
```

```
public class Compound2<X1 extends Term<?>, X2 extends Term<?>>
  extends Cons<X1,Cons<X2,Nil>> {..}

public class Compound3<X1 extends Term<?>, X2 extends Term<?>,
                       X3 extends Term<?>>
  extends Cons<X1,Cons<X2,Cons<X3,Nil>>> {..}
```

7.4.2 Examples

As an example, Table 7.21 shows a Java class `Perm` with the `permutation` method implemented in Prolog. The Java method declaration specifies that there is one input argument and one output argument: the first (`$X`) is a `List<Int>` (or a covariant type), the second (`$Y`) is an `Iterable` over a `List<Int>` (or a covariant type). The `Iterable` specification is needed to iterate over all solutions: if only the first solution is needed, `$Y` could have been used instead of `Iterable<$Y>`.

Moreover, since arguments are declared in the order (`$X`), (`$Y`) in the Java method signature, they will be mapped in this order on the Prolog predicate arguments: so (`$X`) will map onto the first argument of `permutation/2`, and (`$Y`) on the second argument.

In the client program, the `Perm` instance `p` is created indirectly via the `PJ.newInstance` factory method, whose argument is the corresponding `Class` meta-class, `Perm.class`. Then, the `p` object can be used normally, like any other Java object: here it computes all the permutations of a given list of integers (built from an array, just to play with types), which is then iterated over by a *for-each* loop that prints every result. The actual type for both `$X` and `$Y`, `List<Int>`, is inferred automatically by the P@J runtime.

Two further examples are shown in Table 7.22 and Table 7.23, respectively. The first operates on lists, and finally generates (and prints) five "lists of anything" of 1,2,3,4,5 arguments; the second computes the path between two given nodes in a graph. In both cases, Prolog is delegated the reasoning part, while Java is exploited as the front-end to the user. Technically, attention is required to distinguish Java lists (i.e., instances of `java.util.List` and its subclasses) from P@J `List`, which handles terms like `Term<X>`; moreover, the example in Table shows the inner structure of compounds.

For completeness, Table Table 7.24 shows a last, more complex example, where the Prolog code specifies a parser for arithmetic expressions.

```
import alice.tuprologx.pj.annotations.*;
import alice.tuprologx.pj.engine.*;
import alice.tuprologx.pj.model.*;
import alice.tuprologx.pj.meta.*;
import java.util.List;
import java.util.ArrayList;
```

A Java class augmented via Prolog

```
abstract class Perm{
    @PrologMethod ( clauses = {
        "permutation([],[])." ,
        "permutation(U,[X|V]):-remove(U,X,Z),permutation(Z,V)." ,
        "remove([X|T],X,T)." ,
        "remove([X|U],E,[X|V]):-remove(U,E,V)."
        }
    )
    public abstract < $X extends List<Int>, $Y extends List<Int> >
                Iterable<$Y> permutation($X list);
}
```

A sample client class

```
public class PJexample {
  public static void main(String[] args) throws Exception {
    java.util.Collection<Integer> v = java.util.Arrays.asList(1,2,3);
    Perm p=PJ.newInstance(Perm.class);
    for (List<Int> list : p.permutation(new List<Int>(v))) {
        System.out.println(list.toJava());
    }
  }
}
```

Output printed:

```
    [1, 2, 3]
    [1, 3, 2]
    [2, 1, 3]
    [2, 3, 1]
    [3, 1, 2]
    [3, 2, 1]
```

Table 7.21: A Java class exploiting Prolog for implementing an abstract method *(top)* and a client using it *(bottom)*. Note that the `Arrays.asList` method exploits the Java shortcut syntax for varargs. To run the example, the `javassist.jar` library, used by the P@J runtime, must be in the class path: `E:>java  -cp .;2p.jar;javassist.jar PJexample`

```
import alice.tuprologx.pj.annotations.*;
import alice.tuprologx.pj.engine.*;
import alice.tuprologx.pj.model.*;
import alice.tuprologx.pj.meta.*;
```

Another Java class augmented via Prolog

```
@PrologClass(
clauses = {"size(X,Y) :- length(X,Y)."}
)
public abstract class PJLength {
    @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int>
    Boolean size($Ls expr, $Ln rest);
    @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int>
    $Ln size($Ls expr);
    @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int>
    $Ls size($Ln expr);
    @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int>
    Iterable<Compound2<$Ls,$Ln>> size();

    public static void main(String[] args) throws Exception {
        PJLength pjl = PJ.newInstance(PJLength.class);
        java.util.List<?> v = java.util.Arrays.asList(12,"ok",false);
        List<?> list = new List<Term<?>>(v);
        Boolean b = pjl.size(list, 3);   // true
        Int i = pjl.size(list);          // length is 3
        List<?> l = pjl.size(3);         // produces [_,_,_]
        int cont = 0;
        for (Term<?> t : pjl.size()) {   // [],[_],...,[_,_,_,_,_]
            System.out.println(t);
            if (cont++ == 5) break;
        }
    }
}
```

Output printed:

```
    Compound:'size'(List[],Int(0))
    Compound:'size'(List[Var(_)],Int(1))
    Compound:'size'(List[Var(_), Var(_)],Int(2))
    Compound:'size'(List[Var(_), Var(_), Var(_)],Int(3))
    Compound:'size'(List[Var(_), Var(_), Var(_), Var(_)],Int(4))
    Compound:'size'(List[Var(_), Var(_), Var(_), Var(_), Var(_)],Int(5))
```

Table 7.22: Another Java class exploiting Prolog for method implementation. The `length/2` predicate used in the `clauses` section on top is part of the standard ISO list management predicates.

```
import alice.tuprologx.pj.annotations.*;
import alice.tuprologx.pj.engine.*;
import alice.tuprologx.pj.model.*;
import alice.tuprologx.pj.meta.*;
```

Another Java class augmented via Prolog

```
@PrologClass (
  clauses={"arc(a,b)." , "arc(a,d)." , "arc(b,e)." , "arc(d,g).",
           "arc(g,h)." , "arc(e,f)." , "arc(f,i)." , "arc(e,h)."}
)
public abstract class PJPath {
  @PrologMethod (
    clauses = { "path(X,X,[X]).",
                "path(X,Y,[X|Q]):-arc(X,Z),path(Z,Y,Q)."}
  )
  public abstract <$X,$Y,$P> Iterable<$P> path($X from, $Y to);

  public static void main(String[] s) throws Exception {
    PJPath pjp = PJ.newInstance(PJPath.class);
    for (Object solution : pjp.path(new Atom("a"), new Var<Atom>("X"))) {
      System.out.println(solution);
    }
  }
}
```

Output printed:

```
List[Atom(a)]
List[Atom(a), Atom(b)]
List[Atom(a), Atom(b), Atom(e)]
List[Atom(a), Atom(b), Atom(e), Atom(f)]
List[Atom(a), Atom(b), Atom(e), Atom(f), Atom(i)]
List[Atom(a), Atom(b), Atom(e), Atom(h)]
List[Atom(a), Atom(d)]
List[Atom(a), Atom(d), Atom(g)]
List[Atom(a), Atom(d), Atom(g), Atom(h)]
```

Table 7.23: Another Java class exploiting Prolog for method implementation.

```
import alice.tuprologx.pj.annotations.*;
import alice.tuprologx.pj.engine.*;
import alice.tuprologx.pj.model.*;
import alice.tuprologx.pj.meta.*;

@PrologClass
public abstract class PJParser {
  @PrologMethod (clauses={"expr(L,R):-term(L,R).",
                          "expr(L,R):-term(L,['+'|R2]), expr(R2,R).",
                          "expr(L,R):-term(L,['-'|R2]), expr(R2,R)."})
  public abstract <$E extends List<?>, $R extends List<?>>
  Boolean expr($E expr, $R rest);

  @PrologMethod (clauses={"term(L,R):-fact(L,R).",
                          "term(L,R):-fact(L,['*'|R2]), term(R2,R).",
                          "term(L,R):-fact(L,['/'|R2]), term(R2,R)."})
  public abstract <$T extends List<?>, $R extends List<?>>
  Boolean term($T term, $R rest);

  @PrologMethod (clauses={"fact(L,R):-num(L,R).",
                          "fact(['(' | E],R):-expr(E,[')'|R])."})
  public abstract <$F extends List<?>, $R extends List<?>>
  Boolean fact($F fact, $R rest);

  @PrologMethod (clauses={"num([L|R],R):-num_atom(_,L)."})
  public abstract <$N extends List<?>, $R extends List<?>>
  Boolean num($N num, $R rest);

  public static void main(String[] args) throws Exception {
    PJParser ep = PJ.newInstance(PJParser.class);
    String tokenizer_regexp =
      "(?<!^)(\\b|(?=\\()|(?=\\))|(?=\\-)|(?=\\+)|(?=\\/)|(?=\\*))";
    List<Atom> exp1 = new Atom("12+(3-4)").split(tokenizer_regexp);
    List<Atom> exp2 = new Atom("(12+(3-4))").split(tokenizer_regexp);
    System.out.println(ep.expr(exp1, List.NIL)); // 12+(3*4)   expression ?
    System.out.println(ep.fact(exp1, List.NIL)); // 12+(3*4)   factor ?
    System.out.println(ep.expr(exp2, List.NIL)); // (12+(3*4)) expression ?
    System.out.println(ep.fact(exp2, List.NIL)); // (12+(3*4)) factor ?
  }
}
```

Table 7.24: A parser for arithmetic expressions encoded in Prolog inside an annotated Java program. The output prints `true`, `false`, `true`, `true` in this order, since 12+(3*4) is an expression but not a factor, while (12+(3*4)) is both an expression and a factor.

Chapter 8

Multi-paradigm programming in Prolog and .NET

tuProlog.NET now provides the user with the same features as the Java version, extending and specializing the multi-paradigm, multi-language experience to the plethora of languages available onto the Microsoft .NET platform. In this Chapter, the impact of such change is discussed, both in terms of specific conceptual concepts (namely, *language conventions* to handle multiple languages) and new/specialized libraries and predicates to be used for language interaction.

Since the current status of tuProlog.NET depends *a)* on its past history and *b)* on the IKVM tool [**?**], the two following Sections summarize its evolution from version 2.1 and the basics of IKVM translation, respectively.

While their reading is recommended to everyone, the reader wishing only to exploit tuProlog.NET in its current version can safely bypass them and jump directly yo Section **??**.

8.1 A bit of history

8.1.1 tuProlog 2.1 and CSharpLibrary

tuProlog.NET appeared as a usable tool for the first time in April 2007, with the .NET conversion of tuProlog 2.1; an earlier, experimental version had been made with version 2.0, but was never officially published.

tuProlog.NET 2.1 run on Microsoft .NET 2.0 and on Mono 1.2.5[1], and was a complete rewriting in C# of the original Java code: the executable became a .NET `exe` file, and all the libraries became .NET `dll` assemblies.

The Java-based, key feature to multi-paradigm-programming, *JavaLibrary*, was replaced by a corresponding *CSharpLibrary*, which provided the very same features, except for a few syntactic changes:

- any `java_`*`xxx`* predicate was renamed as `csharp_`*`xxx`*.
- C# objects defined in other namespaces than `System` required that the new namespace be explicitly passed to the predicate creating the object: so, `java_object/3` became `csharp_object/4`: `csharp_object(`*`AssemblyName`*`, `*`ClassName`*`, `*`ArgumentList`*`, `*`ObjRef`*`)` Moreover, the assembly containing the definition of the object type must be in the same folder as the `alice-tuProlog.dll` file.
- an *ad hoc* predicate was added for array creation, instead of using the standard `csharp_object/3-/4` resulting from the direct conversion of JavaLibrary predicates: `csharp_array(`*`AssemblyName`*`, `*`Type`*`, `*`Length`*`, `*`ObjRef`*`)`

An annoying limitation concerned the loading of user-defined libraries (and theories), which had to be in the same folder as the tuProlog (`IDE.exe` or `CUIConsole.exe`) executable.

From the developers' viewpoint, using tuProlog classes in a Visual Studio project required a reference to the `alice-tuProlog.dll` assembly be added to the project, and the `tuProlog` namespace be imported in the usual C# fashion (e.g. `using tuProlog;`).

8.1.2 tuProlog 2.1.3: CSharpLibrary + exceptions

As a further step towards the convergence of the .NET and Java versions, the "tuProlog 3" project –later renamed as 2.1.3 – was started to add the exceptions support, being developed for the Java version, to the .NET version, too. However, this version was never officially released, because of the quasi-simultaneous development of tuProlog 2.2, whose *CLILibrary* could provide a much larger interest from the multi-paradigm, multi-language viewpoint.

[1]The MONO version required a source tuning for the `TheoryManager.find` method.

8.1.3 tuProlog 2.2 and CLILibrary

Version 2.2[2] was a milestone in `tu`Prolog.NET history, as it generalized *CSharpLibrary* to enable multi-language programming with *virtually any language available on the .NET platform*, rather than C# only (unfortunately, it lacked exception support, due to the race between the two quasi-simultaneous projects).

To this end, the concept of *Language Convention* was introduced to encapsulate the language-specific aspects, so that a single library – renamed *CLILibrary* instead of *CSharpLibrary* – could handle any language. Each convention contains the syntax conversion operations and the post-compilation transformations required for a given language. Conventions were developed for C#, J#, VisualBasic.NET, F#, Eiffel.NET and IronPythonStudio.

Following the generalization renaming of *CSharpLibrary* as *CLILibrary*, a few syntactic changes were also made:

- any `csharp_`*`xxx`* predicate of CSharpLibrary was renamed here as `cli_`*`xxx`*; this applies both to predicates derived from the JavaLibrary (of the form `java_`*`xxx`*) and to predicates added by CSharpLibrary, like `csharp_array/4`;

- to create objects bound to a particular *Convention*, the `cli_object/5` predicate was introduced whose first argument specifies the convention to be used:
  ```
  cli_object(Convention, AssemblyName, ClassName,
             ArgumentList, ObjRef)
  ```

- furthermore, for those .NET programming languages whose constructor function is not constrained to coincide with the class name, and therefore require such a name to be explicitly specified on object creation, the `cli_object/6` predicate was introduced:
  ```
  cli_object(Convention, AssemblyName, ClassName,
             ContructorName , ArgumentList, ObjRef)
  ```

- two convention handling predicates, also usable as directives, were introduces to load/unload conventions to/from a Prolog theory:

[2]Unfortunately, version numbering for .NET was incoherent with the Java version at that time: in Java, 2.2 was the version that introduced the exception support, which was absent in 2.2 for .NET because the development `tu`Prolog 2.1.3, where exceptions were being added, occurred quasi-simultaneously, but not in time for the two projects to converge. In addition, this version was never tested on Mono.

`load_convention(`*`Assembly, ConventionName, ConventionAtom`*`)`
`unload_convention(`*`ConventionAtom`*`).`

From the developers' viewpoint, the new aspect is how to define new conventions: this is done by starting a new project (class library), importing the `alice-tuprolog.dll` reference and implement a new class extending `tuProlog.Convention` in the `tuProlog.Conventions` namespace.

The `dll` generated by the compilation must then be moved to the main project compiling folder.

8.2 IKVM Basics

IKVM.NET [1] is basically a .NET implementation of Java (language, infrastructure, tools) enriched with special tools for Java/.NET conversion. Its distribution, which adheres to the *zlib* open source license, includes:

- a .NET implementation of a Java Virtual Machine;
- a Java class library, based on OpenJDK, re-implemented in .NET;
- tools for Java/.NET inter-operability—in particular, the `ikvmc` bytecode translator that converts Java bytecode to Microsoft .NET Common Intermediate Language (CIL).

Both Microsoft .NET 2.0 and Mono platforms 2.0 are supported, both for *x86* and *x64* architectures. If necessary, the source pack is also available.

Debugging is also very well supported: if the Java sources are available, proper information can be generated[3] that enable Microsoft Visual Studio to keep the .NET and and Java sources in sync, following the program execution on the Java source, too, as well as enabling breakpoints, variable inspection, etc.

8.2.1 Dynamic vs. Static modality

IKVM can work in two modalities. In the *dynamic* modality, Java applications are converted in .NET on-the-fly and immediately executed; in the *static* modality, instead, Java applications (or libraries) are translated into a .NET assembly, to be used to develop a .NET native application.

[3]The option must be specified to generate the `pdb` (*Program Debug Database*) file, to be copied to the application folder in Visual Studio.

The dynamic modality is supported by the `ikvm` tool, which is analogous to Java's `java` interpreter[4]: so, a Java application can be executed in .NET as in would be in a Java-enabled machine, just replacing `java` with `ikvm`, in a totally user-transparent way. Quite notably, the class loading mechanisms in this modality behaves exactly as in Java, with the same class path options. The only drawback is performance, which is obviously penalized by the on-the-fly translation.

The static modality is supported by the `ikvmc` tool, which generates a `dll` or `exe` .NET assembly (depending whether the translation concerns a Java library or application, respectively) converting Java types to .NET types. Obviously, this tool has no Java counterpart: its options control the target architecture (*x86* or *x64*), the kind of output (`dll`/`exe`), etc. Unlike the previous case, here the Java class loading mechanisms has some limitations, that are discussed below. One possible drawback is IKVM choice of translating the Java *package* visibility into .NET *internal*'s, making it impossible to access such properties and methods from other assemblies (even though they were accessible in the Java architecture).

8.2.2 Class loading issues

The class loading mechanism is perhaps the major issue when translating Java applications to .NET, because of the very different approach adopted by the two architectures, which makes it difficult to define a general mapping. In fact,

- the Java approach is based on the *class path* concept, which defines the set of paths where classes must be looked for;
- the .NET approach, instead, exploits the current folder, the Global Assembly Cache (GAC) and configuration files for the same purpose.

In order to bridge this gap, IKVM adopts the following intelligent approach:

- each *statically-generated* assembly is associated to its own class loader—either a user-supplied one, or the default one;
- the default class loader looks for classes:
    1. first, in the assembly itself;
    2. then, in all the assemblies *directly referenced* by the former.

[4]Most command line options work identically with both tools.

This approach guarantees that classes are always found *if all dependencies are statically expressed*, i.e. if all the libraries used by an application are statically known, and their references are added in the application project. Problems are to be expected, instead, for dynamically loaded classes, whose references were not included in the project—and whose assemblies, therefore, are not considered by the class loader.

To overcome this issue, four alternatives can be followed:

1. creating a *single assembly*, if size is not a problem and run-time modularity is irrelevant (that is, loading all modules even when just one is actually used is irrelevant);
2. adding a static reference (`-r` option) to the library to be dynamically loaded, when the application is translated to .NET: then, the default .NET loading will locate the library, but the need to specify all its details (including version number) cancels most of the advantage of dynamic loading, since any change in the library to be loaded still requires a rebuild;
3. using the special `ikvm.runtime.AppDomainAssemblyClassLoader` class loader provided by IKVM;
4. writing an ad-hoc class loader, typically extending `URLClassLoader`: this is perhaps the most flexible, but also the user-heaviest, solution.

One further interesting aspect is that the IKVM implementation of Java's `Class.forName` method adopts a more general behavior than Java's default implementation, supporting the dynamic loading of classes also *beyond* the current assembly, provided that their `AssemblyQualifiedName` is specified; otherwise, only the current assembly is checked, as in Java.

So, a Java application that exploited `Class.forName` for dynamic class loading, that could originally load only classes in the application JAR, will be able to load .NET[5] classes beyond the application's own assembly when translated to .NET via IKVM.

8.2.3 The other way: writing .NET applications in Java

Beyond converting Java applications in .NET, IKVM also supports the opposite direction—that is, writing .NET applications *in Java*, as if this were one of .NET-supported languages.

[5]The reason why this feature is limited to .NET classes is, trivially, that only .NET classes possess the `AssemblyQualifiedName` property and the other assembly details (version, culture, public key token).

This feature is provided by the `ikvmstub` tool, which generates a Java JAR archive from a .NET assembly (`dll`/`exe`). As the tool name suggests, the generated JAR is just a stub, containing all the Java classes and interfaces corresponding to the .NET originals, but no actual implementation, since this will be written directly in Java: its purpose is just to satisfy the `javac` compiler's type checking, and enable the code completion feature on the IDE (e.g. Eclipse) used for the Java application development.

In this way, a Java application can be written (in Java—using Eclipse, Netbeans, etc.)) that exploits the .NET types extracted from the .NET original assemblies. This application can be compiled with `javac` as usual, specifying the above stub JAR in the class path (`-cp` option).

Obviously, such an application can *not* be run in Java with the standard `java` interpreter, as the above stub JAR does not contain any actual implementation—nor would that be reasonable, since the goal was to exploit Java to write a .NET application, not a Java one. Instead, the resulting "fake" Java application is to be translated via `ikvmc`, and then executed *in .NET* where the original assemblies provide the "missing" classes.

In this context, .NET concepts are mapped onto suitable Java concepts by `ikvmc` as follows:

- *namespaces* are mapped onto Java packages, pre-pending the `cli.` prefix to prevent name clashes;
- *properties* are mapped onto a pair of Java *`get`*/*`set`* methods;
- *enumerations* **were** mapped onto classes extending `cli.System.Enum`, with static fields with integer values for each possible value of the .NET enumerative type; **still true? or using Java Enums now?**
- *delegates* are mapped onto a Java class and a nested helper `Method` interface: the class derives from `System.MulticasDelegate` and has the same name as the original delegate, while the nested interface always declares an `Invoke` method whose signature matches the delegate: this is method called when an event occurs. For this reason, the class constructor takes as its argument an object implementing the `Method` interface, whose implementation of `Invoke` does the actual job.
- *events* are mapped onto a pair of Java *`add_*`*/*`remove_*`* methods, whose argument is an object of the class representing the delegate;

- *params* is mapped onto an array of `Object`s;
  maybe now using generics?

- *attributes* are mapped onto a Java class with the same name as the .NET attribute, plus a pair of Java ***get / set*** methods for each property defined by the attribute.[6]

8.3 tuProlog.NET now

The management difficulties in keeping coherent two such evolving projects (the Java and the .NET versions) indicated that the approach of a separate development was not sustainable in the perspective. This led to a complete strategic change, resulted into the adoption of the IKVM [1] bytecode translator as a tool to automate the generation of tuProlog.NET *from the same Java bytecode* (other than sources) as the Java version, which could then become the only one to be actively maintained "by hand".

Despite some (minor) performance issues (the IKVM-generated tuProlog version appears 15% slower, in the average, than its Java counterpart), the approach turned out to be winning, enabling the two platforms to converge for all they have in common—namely, everything other than the *CLILibrary* and the .NET-specific issues.

8.3.1 Highlights

tuProlog.NET 2.5 builds on top of the winning idea of version 2.2 (language conventions for multi-language interoperability with Prolog), but goes farther by exploiting the value-added brought by the IKVM approach: the chance to use *even Java* as if it were directly available on the .NET platform. This extra value spreads into several directions:

- .NET objects can be accessed, in addition to Java objects, via *OOLibrary* – the renovated version of JavaLibrary – from tuProlog;

- .NET applications can be developed (instead of Java applications, which obviously require the tuProlog Java version) that exploit tuProlog

[6]The java class also includes a nested Java annotation, called `Annotation`, which defines Java methods homonomous to the .NET attribute properties: any reference to such an annotation in the Java code will be translated into the corresponding .NET attribute when the application is converted to .NET. However, only read properties are supported, even if the original .NET attribute properties were read/write.

as a third-party library, with the only difference that a `dll` assembly is to be referenced by the (Visual Studio) project, instead of a JAR archive;

- the whole P@J framework for implementing Java methods in Prolog remains available, and takes a newer form in the .NET context;
- `tu`Prolog libraries can be written in Java, as well as in other .NET languages, resulting into a `dll` assembly in the end;
- Java can be used together with C#, F#, and other .NET languages in the same .NET application, where `tu`Prolog can possibly play the role of the director (orchestrator, coordinator) in-front-of or behind the scenes.

In the next Sections of this Chapter, these dimensions are discussed and explored, roughly following the same structure as Chapter 7.

8.4 Using .NET from Prolog: OOLibrary

cap 5 Montanari

8.5 Using Prolog from .NET: the API

Since `tu`Prolog.NET is automatically generated from the Java sources via IKVM, the available API is the same presented in Section 7.2.

An example? Montanari appendice!

8.6 Augmenting Prolog via .NET: developing new libraries

New `tu`Prolog.NET libraries can be written in any of the .NET languages, and then compiled normally via Microsoft Visual Studio; alternatively, libraries written in Java can be used, by translating them in .NET via IKVM (if they are not part of the standard `tu`Prolog distribution, of course). In any case, the approach is the basically same presented in Section 7.3.

Two examples, one in .NET and another one in Java+IKVM?

8.7 Augmenting .NET via Prolog: the PJ framework.. revised

Since tuProlog.NET is automatically generated from the Java sources via IKVM, the P@J framework presented in Section 7.4 is also available.

But we are working on a different, more .NET specific, approach...

An example?

8.8 Putting everything together

Pipolo?

Bibliography

[1] The IKVM project. `http://www.ikvm.net`.

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

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

[4] Maurizio Cimadamore and Mirko Viroli. Integrating Java and Prolog through generic methods and type inference. In *Proc. ACM SAC 2008*, pages 198–205, 2008.

[5] 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.

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

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

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