Linköping University

May 14, 2010

JessTab Manual

Integration of Protégé and Jess

Henrik Eriksson

Linköping University

her@ida.liu.se

Note: Read the installation instructions before attempting to install JessTab.

Introduction

JessTab is a bridge between Protégé-2000 and Jess. It provides a Jess console window in a Protégé tab and a set of extensions to Jess to allow mapping of Protégé knowledge bases to Jess facts and manipulation of Protégé knowledge bases. You can use JessTab to create Jess programs that take advantage of Protégé knowledge bases. For example, Jess rule patterns can match on Protégé instances.

The Jess interface to Protégé is modeled after the Jess-to-Java coupling. You can use these interfaces together; that is, it is possible to implement a system in Jess, Java, and Protégé. Most Protégé functions in Jess are distinct from the corresponding Java functions to avoid name collisions. Furthermore, many of the JessTab functions for object manipulation work the same way as their CLIPS counterparts. CLIPS programmers will recognize functions such as make-instance and class-subclasses.

Uses of JessTab

The Protégé – Jess integration allows you to develop knowledge bases graphically in Protégé and run problem solvers in Jess that use Protégé knowledge bases to perform their task. Furthermore, Jess can manipulate Protégé knowledge bases (e.g., instantiating classes and changing slot values). Jess rules can pattern match on instances in Protégé knowledge bases at knowledge-acquisition time and perform actions, such as modifying the knowledge base.

It is possible to use Jess as the performance element for reusable problem-solving methods that take advantage of Protégé knowledge bases. Jess code can implement the mappings required to integrate domain and method ontologies. In summary, Jess acts as the performance counterpart for Protégé. Meanwhile, Protégé can act as the graphical knowledge-base counterpart for Jess.

JessTab turns Protégé into a limited development environment for Jess. JessTab provides editors for Jess constructs, such as rules and functions. You can use these editors to browse and redefine Jess definitions. In addition, you can instruct JessTab to store Jess definitions with the Protégé knowledge base. Alternatively, you can store your Jess program as a separate file, which can be loaded as a startup file when the knowledge base is loaded.

Typical use is to first develop a small version of the knowledge base in Protégé and than develop the Jess code that operates on it. Once the Jess code performs adequately you can extend the knowledge base.

The remainder of this document requires basic knowledge of Jess and Protégé.

Installation

Assuming you have a working installation of Protégé, follow these steps:

1. Put the downloaded `JessTabXY.jar` file the in the Protégé `plugins` subdirectory (where `XY` is the JessTab version). The `plugins` directory is located under the Protégé installation. (e.g., `../Protege/plugins/).`

2. In addition to the JessTab plug-in, you need a Jess engine. You have to obtain Jess directly from `http://www.jessrules.com/`. (Jess is not part of the JessTab package because the Jess license agreement does not allow redistribution.)

3. Compile Jess and make it available in the plug-in subdirectory (i.e., as a `jess` subdirectory or a `jess.jar` file).

4. In addition, Jess needs the file `scriptlib.clp` in a directory called `jess` under the Protégé installation (e.g., `../Protege/jess/scriptlib.clp`). The file `scriptlib.clp` is part of the Jess distribution.

5. Verify that the JessTab installation works by starting Protégé and opening JessTab. In Protégé, select Project -> Configure… and enable JessTab. You should now see JessTab among the other tabs and be able to interact with the Jess console.

JessTab 1.1 requires Jess version 6.1 (or later). Furthermore, JessTab requires Protégé version 1.9 or 2.0 or later. (Because Protégé 2.0 introduced a change to the tab API that requires recompilation, the correct JessTab11.jar file must be used with each of the Protégé 1.9 and 2.0 versions.)

Installation Troubleshooting

1. Some web browsers do not download files in binary mode by default. If you get the error message "`Exception: SystemUtilities.loadJarManifest (../JessTab.jar) failed. java.util.zip.ZipException: invalid CEN header (bad signature)`" when starting Protégé `JessTab.jar` might be corrupt. Try to open `JessTab.jar` with a zip program (e.g., WinZip). If you cannot open the file it is corrupt. Download the file again using a different web browser (if all else fails send e-mail to her@ida.liu.se).

2. If you get the error message "`java.lang.NoClassDefFoundError: jess/Userpackage`" when starting Protégé there is a problem with your Jess installation. Make sure the compiled `*.class` files for Jess are in the `plugins/jess` directory (or in a jar file in the `plugins` directory). For instance, you should have `Protege/plugins/jess/Userfunctions.class` and so on.

3. If you are upgrading JessTab, make sure that you do not have more than one version of `JessTab.jar` on the Protégé `CLASSPATH`. Use `(jesstab-version-string)` to check which version you are running.

General Troubleshooting

If the JessTab console window hangs it sometimes helps to type ctrl-C *once* in the Protégé console window (i.e., the MS-DOS window). Note that ctrl-C normally kills the Protégé application.

Protégé Functions in Jess

A set of Jess functions defines the interface to Protégé. There is a difference between functions that *map* Protégé classes and instances to Jess and functions that *manipulate* Protégé knowledge bases. The former translates Protégé instances to Jess templates and facts. The latter calls the Protégé API to make direct changes to the Protégé knowledge base (e.g., by instantiating classes and changing slot values). Also, there are functions for querying the content of the Protégé knowledge base (e.g., properties of classes, slots, and instances).

Functions that map Protégé classes and instances to Jess

(mapclass <class-name> [nonreactive | reactive])

Maps a Protégé class to the Jess system. This function tells Jess to prepare to match on properties of a Protégé class. It generates the appropriate deftemplate to represent the Protégé class, and asserts facts for all the instances of the class. The deftemplate and facts are named *object*, and the facts hold information about the slot values as well as the class name and a pointer to the Protégé instance in question. The mapclass function maps recursively the subclasses of the Protégé class.

(mapinstance <instance-name>|<instance-address> [nonreactive | reactive])

Maps a specific Protégé instance to a Jess fact. This function tells Jess to match on properties of a specific Protégé instance. The option *nonreactive* maps the instance to the fact once (i.e., by making a copy). There is no update of the fact if the slot values in the Protégé knowledge base change. The option *reactive*, which is the default, propagates changes to the instance in the Protégé knowledge base to the corresponding Jess fact.

(unmapinstance <protege-instance-name>|*)

Unmap Protégé instances in Jess. This function removes the previous mappings. The wildcard (*) can be used to remove all instance mappings.

Functions that manage Protégé classes and instances

(defclass <name> [<comment>] (is-a <superclass>+) [role] <slot>*)

Define a new Protégé class. Overloads the standard defclass function in Jess mapping Java classes to facts. (The Java-to-Jess mapping still works.) Defclass can be viewed as “syntactical sugar” for instantiating the standard metaclass and creating and adding standard slots to it. The *role* specification can be defined as `(role abstract)` or `(role concrete)`. A *slot* definition starts with the slot form (i.e., slot, single-slot, or multislot) and continues with the slot name and the facets. For example: `(slot name (type string)) (multislot degrees (type symbol) (allowed-values BSc MSc MBA PhD MD))`

The slot facets supported are `allowed-values, allowed-classes, cardinality, comment, default, range,` and `type`. The supported slot types are `any, boolean, class, float, instance, integer, string,` and `symbol`.

(make-instance [<instance-name>] of <class-name> <slot-override>* [map])

Create a Protégé instance from Jess. The first argument, which is optional, is the Protégé and Jess name for the instance. If no name is given, Protégé will create a unique generic name for it. The <class-name> argument is the name of the class to instantiate. The slot overrides are lists of slot name and value pairs that specify the initial slot values. The keyword map specifies that the instance should also be mapped to a Jess fact. (It is *not* mapped by default unless the class is mapped.) Returns the new instance.

(initialize-instance <instance-name> <slot-override>*)

Reinitialize a Protégé instance from Jess. The initialize-instance function provides the ability to reinitialize an existing instance with class defaults and new slot-overrides. The return value of initialize-instance is the instance on success or the symbol FALSE on failure.

(modify-instance <instance-name> <slot-override>*)

Modifies a Protégé instance from Jess. This function allows instance slot updates to be performed in blocks without requiring a series of slot assertions. This function returns the symbol TRUE if successful, otherwise the symbol FALSE is returned.

(duplicate-instance <instance-name> [to <instance-name>] <slot-override>*)

Duplicates a Protégé instance from Jess. This function allows instance duplication and slot updates to be performed in blocks without requiring a series of slot assertions. This function returns the new duplicated instance if successful, otherwise the symbol FALSE is returned.

(definstances <definstances-name> [comment] <instance-template>*)

Create several Protégé instances from Jess. The equivalent of a make-instance function call is made for every instance specified in definstances constructs.

(unmake-instance <instance-expression>+)

This function deletes the specified instances. The argument can be one or more instance-names or instance-addresses. This function returns the symbol TRUE if all instances were successfully deleted, otherwise it returns the symbol FALSE.

(slot-get <instance> <slot-name>)

Gets a value from an instance slot in the Protégé knowledge base. The Protégé instance could be a pointer to the instance (e.g., the result of a call to make-instance) or a Protégé instance name. The slot is the name of the instance slot. Returns the Protégé slot value.

(slot-set <instance> <slot-name> <new-value>)

Sets the value of an instance slot in the Protégé knowledge base. The Protégé instance could be a pointer to the instance (e.g., the result of a call to make-instance) or a Protégé instance name. The slot is the name of the instance slot. The new-value argument is mapped to the corresponding Protégé data type. No return value.

(slot-replace$ <instance-expression> <mv-slot-name> <range-begin> <range-end> <expression>+)

Replacing a multifield slot value of a Protégé instance from Jess. Allows the replacement of a range of fields in a multifield slot value with one or more new values. The range indices must be from 1..n, where n is the number of fields in the multifield slot s original value and n > 0.

(slot-insert$ <instance-expression> <mv-slot-name> <index> <expression>+)

Inserting a multifield value in a slot value of a Protégé instance. Allows the insertion of one or more new values in a multifield slot value before a specified field index. The index must greater than or equal to 1. A value of 1 inserts the new value(s) at the beginning of the slot s value. Any value greater than the length of the slot s value appends the new values to the end of the slot s value.

(slot-delete$ <instance-expression> <mv-slot-name> <range-begin> <range-end>)

Deleting multifield slot-value elements of a Protégé instance from Jess. Allows the deletion of a range of fields in a multifield slot value. The range indices must be from 1..n, where n is the number of fields in the multifield slot s original value and n > 0.

(slot-facets <class-name> <slot-name>)

Get the facets for a slot. This function returns a multifield listing the facet values for the specified slot (the slot can be inherited or explicitly defined for the class). See the Clips manual for detail about the meaning of the facet values.

(slot-types <class-name> <slot-name>)

Get the types for a slot. This function groups the names of the primitive types allowed for a slot into a multifield variable.

(slot-cardinality <class-name> <slot-name>)

Get the cardinality of a slot. This function groups the minimum and maximum cardinality allowed for a multifield slot into a multifield variable. A maximum cardinality of infinity is indicated by the symbol +oo (the plus character followed by two lowercase o s not zeroes).

(slot-range <class-name> <slot-name>)

Get the numeric range of a slot. This function groups the minimum and maximum numeric ranges allowed a slot into a multifield variable. A minimum value of infinity is indicated by the symbol -oo (the minus character followed by two lowercase o s not zeroes). A maximum value of infinity is indicated by the symbol +oo (the plus character followed by two lowercase o s not zeroes). The symbol FALSE is returned for slots in which numeric values are not allowed.

(slot-allowed-values <class-name> <slot-name>)

Get the allowed values for a slot (of type symbol). This function groups the allowed values for a slot into a multifield variable. If no allowed-values facets were specified for the slot, then the symbol FALSE is returned.

(slot-allowed-classes <class-name> <slot-name>)

Get the allowed classes for a slot (of type instance). This function groups the allowed classes for a slot into a multifield variable. If no classes were specified for the slot, then the symbol FALSE is returned.

(slot-allowed-parents <class-name> <slot-name>)

Get the allowed parents for a slot (of type cls). This function groups the allowed classes for a slot into a multifield variable. If no classes were specified for the slot, then the symbol FALSE is returned.

(slot-documentation <class-name> <slot-name>)

Get the documentation for a slot.

(slot-sources <class-name> <slot-name>)

Get the sources for a slot. This function groups the names of the classes which provide facets for a slot (i.e., have slot override) of a class into a multifield variable.

(facet-get <class-name> <slot-name> <facet-name>)

Get the value of a facet for a slot. The supported facets are: `:NAME, :DOCUMENTATION, :SLOT-DEFAULTS, :SLOT-MAXIMUM-CARDINALITY, :SLOT-MINIMUM-CARDINALITY, :SLOT-NUMERIC-MAXIMUM, :SLOT-NUMERIC-MINIMUM, :SLOT-VALUE-TYPE, :SLOT-VALUES, :SLOT-INVERSE, :SLOT-CONSTRAINTS`

(facet-set <class-name> <slot-name> <facet-name> <new-value>)

Set the value of a facet for a Protégé slot. The supported facets are: `:NAME, :DOCUMENTATION, :SLOT-DEFAULTS, :SLOT-MAXIMUM-CARDINALITY, :SLOT-MINIMUM-CARDINALITY, :SLOT-NUMERIC-MAXIMUM, :SLOT-NUMERIC-MINIMUM, :SLOT-VALUE-TYPE, :SLOT-VALUES, :SLOT-INVERSE, :SLOT-CONSTRAINTS`

Predicates and functions that return information about Protégé classes and instances

(class <object>)

Get the class of an object. This function returns the Protégé class of a Protégé object. If the <object> parameter is not an instance or instance name string or atom, the function returns the Jess type. This function returns the class as an atom.

(class-existp <class>)

Determine if a Protégé class exists. This function returns the symbol TRUE if the specified class is defined, FALSE otherwise. The argument is the class name to test.

(class-abstractp <class>)

Determine if a Protégé class is abstract. This function returns the symbol TRUE if the specified class is abstract, i.e. the class cannot have direct instances, FALSE otherwise.

(class-reactivep <class>)

Determine if a Protégé class is reactive. This function returns the symbol TRUE if the specified class is reactive (i.e., instances of the class can match object patterns), FALSE otherwise. In other words, the function checks if the class is mapped.

(superclassp <class1-name> <class2-name>)

Determine if a Protégé class is the superclass of another class. This function returns the symbol TRUE if the first class is a superclass of the second class, FALSE otherwise.

(subclassp <class1-name> <class2-name>)

Determine if a Protégé class is the subclass of another class. This function returns the symbol TRUE if the first class is a subclass of the second class, FALSE otherwise.

(class-superclasses <class-name> [inherit])

Get the superclasses of a Protégé class. This function groups the names of the direct superclasses of a class into a multifield variable. If the optional argument *inherit* is given, indirect superclasses are also included. The returned indirect superclasses are not ordered and the order may change between calls to this function.

(class-subclasses <class-name> [inherit])

Get the subclasses of a Protégé class. This function groups the names of the direct subclasses of a class into a multifield variable. If the optional argument inherit is given, indirect subclasses are also included. The returned indirect subclasses are not ordered and the order may change between calls to this function.

(slot-boundp <instance>|<instance-name> <slot-name>)

Determine if a slot is bound. The first argument is the instance. The second argument is the slot. This function returns TRUE if the slot is unbound, otherwise FALSE. For multivalued slots, the function returns TRUE is the value list is empty (because Protégé does not explicitly keep track of unbound multivalue slots).

(slot-makunbound <instance>|<instance-name> <slot-name>)

Makes a slot unbound. The first argument is the instance. The second argument is the slot. This function has no return value.

(add-direct-subclass <superclass> <subclass>)

Adds a subclass to a class. The first argument is the superclass. The second argument is the subclass. This function has no return value.

(remove-direct-subclass <superclass> <subclass>)

Removes a subclass from a class. The first argument is the superclass. The second argument is the subclass. This function has no return value.

(class-direct-subclasses <class>)

Return a list of subclasses of a class.

(class-direct-superclasses <class>)

Return a list of superclasses of a class.

(get-defclass-list)

The function get-defclass-list returns a multifield value (list) containing the names of all defclass constructs.

(class-slots <class-name> [inherit])

Get the slots of a Protégé class. This function groups the names of the explicitly defined slots of a class into a multifield variable. If the optional argument *inherit* is given, inherited slots are also included.

(instancep <object>)

Determine if an object is a Protégé instance. The argument is the object to test.

(instance-existp <object>)

Determine if a Protégé instance exist. The argument is the object to test.

(instance-name <instance-address>)

Returns the name of a Protégé instance. The argument is the instance name or address.

(instance-address <instance-name>)

Returns the address of a Protégé instance. The argument is the instance name or address.

(instance-addressp <instance-address>)

Determine if an address is a Protégé instance. The argument is the object to test.

(instance-namep <name>)

Determine if an object is a Protégé instance name (i.e., an atom). The argument is the object to test.

(slot-existp <class-name> <slot-name> [inherit])

Check for slot existence in a Protégé class. This function returns the symbol TRUE if the specified slot is present class, FALSE otherwise. If the inherit keyword is specified then the slot may be inherited, otherwise it must be directly defined in the specified class.

(slot-default-value <class-name> <slot-name>)

Get the default value of a slot. This function returns the default value associated with a slot. If a slot has a dynamic default, the expression will be evaluated when this function is called. The symbol FALSE is returned if an error occurs.

Instance query functions

The instance query functions operate on sets of instances in the Protégé knowledge base. It is possible to use them to find instances with certain properties and to iterate over instances. See the CLIPS Reference Manual (Volume I – Basic Programming Guide, Section 9.7) for an explanation of these functions.

(find-all-instances <instance-set-template> <query>)

(find-instance <instance-set-template> <query>)

(any-instancep <instance-set-template> <query>)

(do-for-instance <instance-set-template> <query> <action>*)

(do-for-all-instances <instance-set-template> <query> <action>*)

(delayed-do-for-all-instances <instance-set-template> <query> <action>*)

Functions that control storage of Jess definitions in Protégé knowledge bases

It is possible to store Jess code in Protégé knowledge bases. You can instruct Jess to save definitions with the knowledge base by marking the definitions for save. You can then reinstate the saved definitions when the knowledge base is reloaded. The function `set-kb-save` controls the save settings. An alternative to setting the save property with the `set-kb-save` function is to use the checkbox for “Save definition in knowledge base” in the JessTab definition subtabs.

The saved code is not immediately reloaded into the Jess engine when the Protégé project is loaded. If the knowledge base contains Jess code, JessTab will open a message panel where you can choose to load the code into the Jess engine. An alternate is to load the code later with the function `load-kb-definitions`. JessTab does not support knowledge-base save and reload of the Jess constructs `defadvice`, `defclass` (for Java classes), and `definstance`.

Note that Jess definitions are *not* saved with the knowledge base by default. You must mark new definitions for save explicitly.

Sometimes it is useful to load a startup file with Jess code when the Protégé knowledge base is loaded. The instance of the JessTab system class `:JESS-ENGINE` has slots for specification of a startup file and of a startup expression. Use the Instances tab to set the values for the "Jess engine" instance. The startup file loads before any Jess code stored in the knowledge base loads. However, JessTab evaluates the startup expression after any Jess code in the knowledge base is loaded.

(set-kb-save <definition-type>|* <definition-name>|* TRUE|FALSE)

The function `set-kb-save` sets the current save setting for Jess definitions. The value TRUE specifies that the Jess definition will be saved together with the Protégé knowledge base. The value FALSE specifies that the definition will *not* be saved (the default behaviour). The supported definitions types are deffunction, deffacts, defglobal, deftemplate, defrule, and defquery. It is possible to use wildcards (`*`) for the definition type and for the definition name. For example, the expression `(set-kb-save * * TRUE)` specifies save for all current definitions. (Note that new definitions will have the FALSE setting by default.)

(get-kb-save <definition-type> <definition-name>)

The function `get-kb-save` returns the current save setting for Jess definitions. The return value TRUE means that the Jess definition will be saved together with the Protégé knowledge base. The value FALSE means that the definition will *not* be saved. The supported definitions types are deffunction, deffacts, defglobal, deftemplate, defrule, and defquery.

(load-kb-definitions)

The function `load-kb-definitions` loads Jess definitions represented as instances in the Protégé knowledge base into the Jess engine. (Note that Jess definitions in a Protégé knowledge base are not defined immediately when you load the Protégé project [.pprj file]. The function `load-kb-definitions` sends them to Jess.)

Miscellaneous Functions

(load-project <file-path>)

Load a new Protégé project.

(include-project <file-path>)

Include a new Protégé project.

(save-project <file-path>)

Save the current Protégé project.

(jesstab-version-number)

Returns the version number of the JessTab implementation.

(jesstab-version-string)

Returns the version string (human readable) of the JessTab implementation.

(get-knowledge-base)

Returns the Protégé knowledge-base object (i.e., the Java object).

(get-tabs)

The function get-tabs returns a list of pointers to the active Protégé tabs. This information can be used to communicate with other tabs, such as your own custom tab. In addition, there is a static Java method `JessTab.getEngine()`, which returns the `Rete` object. You can use it to communicate with Jess from custom tabs and slot widgets.

JessTab and Protégé-OWL

Originally, JessTab was developed to support Protégé-Frames (i.e., JessTab predates the Protégé-OWL implementation). Unfortunately, there are fundamental differences between the frames-oriented ontologies of Protégé-Frames and OWL ontologies that make it difficult to fully support OWL in JessTab. However, JessTab includes limited support for handling OWL ontologies. Most of the functions described in this manual either work correctly under OWL or use a best-effort strategy to support OWL behavior. Furthermore, JessTab adds the following convenience function under Protégé-OWL:

(full-name <resource-name>)

Returns the internal full OWL name of a Protégé individual. The resource-name argument could be a string or symbol. Under OWL, the default name space will be added to the argument and returned as a symbol. This function can be used as a short-hand mechanism to avoid typing the full URI.

Running JessTab without the Protégé User Interface

It is possible to run JessTab without the normal Protégé user interface. There are two different modes of running without the graphical user interface: (1) Jess console mode and (2) Java API mode. In the Jess console model, you start the Java VM with Jess, JessTab, and Protégé on the CLASSPATH. For example

```
java –classpath plugins/jess.jar:plugins/JessTab.jar:protege.jar jess.Main
```

(The above command assumes the current working directory is the directory where Protégé is installed. If you are using other Protégé plug-ins, such as PAL, you should put those `.jar` files on the CLASSPATH too.) The class `jess.Main` creates a Jess engine and enters the Jess read-evaluate-print loop. Once Jess is running load the JessTab functions:

```
jess> (load-package se.liu.ida.JessTab.JessTabFunctions)
```

You can now use Jess with the “object-oriented extension” that Protégé and JessTab provides. For example, you can define classes with *defclass* and instantiate them with *make-instance* just as in JessTab residing inside Protégé.

The Java API mode is an alternative to starting Jess in the console mode. The method is to create a Jess engine from a Java application. Just as in the previous example, Jess, JessTab, and Protégé must be on the Java CLASSPATH. In this approach, you instantiate the `jess.Rete` class and add the JessTab-specific functions to it:

```
      import jess.*;
      import se.liu.ida.JessTab.*;
      Rete engine = new Rete();
      engine.addUserpackage(new JessTabFunctions());
```

After initializing the Jess engine, it is possible to use the `executeCommand()` method to send expressions to it for evaluation. For example, the statement

```
engine.executeCommand(”(load-project example.prj)”);
```

loads the project `example.prj` into Protégé. You can now obtain the current Protégé knowledge base by calling `JessTab.getProtegeKB()` and the Protégé project by calling `JessTab.getProtegeKB().getProject()`. In this manner, your Java program has access to the full Jess and Protégé APIs.

Appendix A: Basic Examples

This section contains illustrative examples of the use of the JessTab extensions.

Define a Protégé class from Jess:

```
(defclass A (is-a :THING)
   (slot x (type string))
)
```

Create an instance of the class:

```
(make-instance foo of A)
```

Set the value of slot x:

```
(slot-set foo x ”bar”)
```

Get the value of slot x:

```
(slot-get foo x)
```

Map the instances of class A to Jess facts:

```
(mapclass A)
```

You can now examine the Jess fact that represents the instance `foo`. Use the `(facts)` function or the Facts subtab in JessTab.

Appendix B: Advanced Examples

Understanding these examples requires basic knowledge of the concept of metaclasses and how the metaclasses are structured in Protégé.

Print the abstract classes in Protégé:

```
(mapclass :THING)

(defrule print-abstract-classes-1 "Print all abstract classes"
   ?c <- (object )
   (test (class-abstractp ?c))
 =>
   (printout t "The class " (instance-name ?c) " is abstract." crlf))
```

Alternative rule for printing the abstract classes in Protégé:

```
(defrule print-abstract-classes-2 "Print all abstract classes"
   (object (:NAME ?n) (:ROLE Abstract))
 =>
   (printout t "The class " ?n " is abstract." crlf))
```

Print all (standard) slots of type *integer* in the current Protégé project:

```
(defrule print-integer-slots "Print all integer slots"
   (object (is-a :STANDARD-SLOT) (:NAME ?n) (:SLOT-VALUE-TYPE Integer))
 =>
   (printout t "The slot " ?n " is of type Integer" crlf))
```

Change the role to *abstract* for classes that have subclasses, but do not have any instances:

```
(defrule make-classes-abstract "Turn classes abstract"
   ?c <- (object (:NAME ?n) (:ROLE Concrete) (:DIRECT-INSTANCES ))
   (not (object (:NAME ?n) (:DIRECT-SUBCLASSES)))
 =>
   (slot-set ?c :ROLE Abstract))
```

Create and attach new instances to objects in the knowledge base. Create an *address* instance automatically for every *person* instance without an address:

```
(defrule add-objects "Create address instances for persons"
  ?o <- (object (is-a person) (location nil))
  =>
  (slot-set ?o location (make-instance of address)))
```

Appendix C: Using Java Reflection in Jess

Jess has functionality for communication with Java (see Section 2.6 of the Jess manual). The ability to manipulate Java object from Jess makes it possible to control the Protégé Java API from your Jess program (or from the Jess console window). Information about the Protégé Java API is available in Javadoc format and in the source code as part of the Protégé release. (The current documentation in minimal.)

The function `(get-knowledge-base)` returns the current knowledge base. This value is a good starting point for low-level communication with Protégé. For example, you can use `(call (get-knowledge-base) getProject)` to get the current project. Given the project, you can now call the `show` method to open an instance editor:

```
(call (call (get-knowledge-base) getProject) show (instance-address
myInstance))
```

Another useful method is `setEditable`, which controls the possibility to edit classes, slots, and instances in Protégé. For example, you invoke `(call (instance-address myInstance) setEditable FALSE)` to make an instance uneditable in Protégé.

Appendix D: PAL Communication

JessTab provides support functions for invoking the Protégé Axiom Language (PAL) engine. PAL allows Protégé users to define constraints expressions and to check for violations of the constraints. Unlike Jess, however, PAL is not an inference engine in the sense that can make changes to the ontology or the knowledge base. It is primarily a tool for checking the integrity of ontologies.

It is possible to create PAL constraints from Jess by instantiating the `:PAL-CONSTRAINT` class. For example, the expression `(make-instance C1 of :PAL-CONSTRAINT (:PAL-NAME "C1") (:PAL-STATEMENT "(< 2 3)"))` creates a trivial constraint, which is always true. Because the `:PAL-CONSTRAINT` instances are normal Protégé instances, you can manipulate them in the same way as regular instances.

The functions `pal-evaluate-constraints` and `pal-evaluate-constraints*` allows you to check such constraint by evaluating them. The functions detect and return *violations* of constraints. Note that the PAL support functions in JessTab are only available when the PAL plug-in is installed in the Protégé environment (i.e., they are undefined when PAL is not installed).

(pal-evaluate-constraints <constraints-to-evaluate>|* [<max-number-of-counter-examples>])

Invoke the PAL engine and evaluate a set of constraints. The first parameter is the set of constraints to evaluate. The wildcard (*) results in evaluation of all PAL constraints. The second optional parameter is the maximum number of constraint violations that will be returned. Returns a list of PAL constraint instances in violation. If there is no violation, the function returns the empty list ().

(pal-evaluate-constraints* <constraints-to-evaluate>|* [<max-number-of-counter-examples>])

Invoke the PAL engine and evaluate a set of constraints. The first parameter is the set of constraints to evaluate. The wildcard (*) results in evaluation of all PAL constraints. The second optional parameter is the maximum number of constraint violations that will be returned. Returns `ConstraintViolation` object(s) for any constraint violation. The returned objects are Java instances that implement the `edu.stanford.smi.protegex.pal.engine.ConstraintViolation` interface, which is part of the Protégé API. It is possible to obtain detailed information, such as textual descriptions and variable bindings, from these objects by calling their Java methods from Jess (e.g., using the `call` function). If there is no violation, the function returns the empty list ().