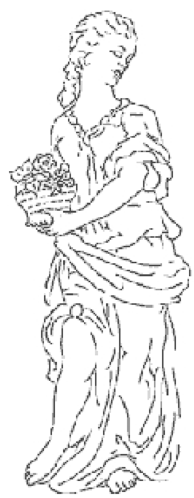
A Guide to $\mathcal{F}$LORA-*2* Packages



Version 1.2.1

(Monstera deliciosa)

September 8, 2017

Contents

Chapter 1

JAVA-to-$\mathcal{F}$LORA-*2* Interfaces

by Aditi Pandit and Michael Kifer

This chapter documents the API for accessing $\mathcal{F}$LORA-*2* from Java programs. The API has two versions: a *low-level API* (used most commonly), which enables Java programs to send arbitrary queries to $\mathcal{F}$LORA-*2* and get results, and an *experimental high-level API*, which is more limited, but is easier to use. The high-level API establishes a correspondence between Java classes and $\mathcal{F}$LORA-*2* classes, which enables manipulation of $\mathcal{F}$LORA-*2* classes by executing appropriate methods on the corresponding Java classes. Both interfaces rely on the Java-XSB interface, called *Interprolog* [1], developed by `Interprolog.com`.

The API assumes that a Java program is started first and then it invokes XSB/$\mathcal{F}$LORA-*2* as a subprocess. The XSB/$\mathcal{F}$LORA-*2* side is passive: it only responds to the queries sent by the Java side. Queries can be anything that is accepted at the $\mathcal{F}$LORA-*2* shell prompt: queries, insert/delete commands, control switches, etc., are all fine. One thing to remember is that the backslash is used in Java as an escape symbol and in $\mathcal{F}$LORA-*2* as a prefix of the builtin operators and commands. Therefore, each backslash must be escaped with another backslash. That is, instead of a query like "`p(?X) \and q(?X).`" the API requires "`p(?X) \\and q(?X).`".

1.1 The Low-level Interface

The low-level API enables Java programs to send arbitrary queries to $\mathcal{F}$LORA-*2* and get results. It is assumed that the following environment variables are set:

`JAVA_BIN`: This variable points to the folder containing the javac and java executable programs. This variable is set in the `windowsVariables.bat` and `unixVariables.sh` scripts in the `java` subfolder of the $\mathcal{F}$LORA-*2* distribution.

`PROLOGDIR`: This variable points to the folder containing the XSB executable. This variable is set in the `flora_settings.bat` and `flora_settings.sh` scripts in the `java` folder.

`FLORADIR`: This variable must point to the folder containing the $\mathcal{F}$LORA-*2* installation. It is set by the `flora_settings.bat` and `flora_settings.sh` files in the `java` subfolder and this is where users should look in order to get the correct values for their systems. Both of the above files are generated automatically by the system installation scripts.

In order to be able to access $\mathcal{F}$LORA-2, the Java program must first establish a session for a running instance of $\mathcal{F}$LORA-2. Multiple sessions can be active at the same time. The knowledge bases in the different running instances are completely independent. Sessions are instances of the class `javaAPI.src.FloraSession`. This class provides methods for opening/closing sessions and loading $\mathcal{F}$LORA-2 knowledge bases (which are also used in the high-level interface). In addition, a session provides methods for executing arbitrary $\mathcal{F}$LORA-2 queries. The following is the complete list of the methods that are available in that class.

- `public FloraSession()`

  This method creates a connection to an instance of $\mathcal{F}$LORA-2.

- `close()`

  This method must be called to terminate a $\mathcal{F}$LORA-2 session. Note that this does not terminate the Java program that initiated the session: to exit the Java program that talks to $\mathcal{F}$LORA-2, one needs to execute the statement

  ```
  System.exit();
  ```

  Note that just returning from the `main` method is not enough.

- `public Iterator<FloraObject> ExecuteQuery(String command)`

  This method executes the $\mathcal{F}$LORA-2 command given by the parameter `command`. It is used to execute $\mathcal{F}$LORA-2 queries that do not require variable bindings to be returned back to Java or queries that have only a single variable to be returned. Each binding is represented as an instance of the class `javaAPI.src.FloraObject`. The examples below illustrate how to process the results returned by this method.

- `public Iterator<HashMap<String,FloraObject>> ExecuteQuery(String query,Vector vars)`

  This method executes the $\mathcal{F}$LORA-2 query given by the first argument. The Vector `vars` (of strings) specifies the names of all the variables in the query for which bindings need to be returned. These variables are added to the vector using the method `add` before calling `ExecuteQuery`. For instance, `vars.add("?X")`.

  This version of `ExecuteQuery` returns an iterator over all bindings returned by the $\mathcal{F}$LORA-2 query. Each binding is represented by a `HashMap<String,FloraObject>` object which can be used to obtain the value of each variable in the query (using the `get()` method). The value of each variable returned is an instance of `javaAPI.src.FloraObject`.

  See the examples below for how to handle the results returned by this method.

- `void loadFile(String fileName,String moduleName)`

  This method loads the $\mathcal{F}$LORA-2 program, specified by the parameter `fileName` into the $\mathcal{F}$LORA-2 module specified in `moduleName`.

- `void compileFile(String fileName,String moduleName)`

  This method compiles (but does not load) the $\mathcal{F}$LORA-2 program, specified by the parameter `fileName` for the $\mathcal{F}$LORA-2 module specified in `moduleName`.

- `void addFile(String fileName,String moduleName)`

  This method adds the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` to an existing $\mathcal{F}$LORA-*2* module specified in `moduleName`.

- `void compileaddFile(String fileName,String moduleName)`

  This method compiles the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` for addition to the $\mathcal{F}$LORA-*2* module specified in `moduleName`.

The code snippet below illustrates the low-level API.

Step 1: Writing a $\mathcal{F}$LORA-*2* program. Let us assume that we have a file, called `flogic_basics.flr`, which contains the following information:

```
person :: object.
dangerous_hobby :: object.
john:employee.
employee::person.

bob:person.
tim:person.
betty:employee.

person[|age=>integer,
       kids=>person,
       salary(year)=>value,
       hobbies=>hobby,
       believes_in=>something,
       instances => person
|].

mary:employee[
    age->29,
    kids -> {tim,leo,betty},
    salary(1998) -> a_lot
].

tim[hobbies -> {stamps, snowboard}].
betty[hobbies->{fishing,diving}].

snowboard:dangerous_hobby.
diving:dangerous_hobby.

?_X[self-> ?_X].

person[|believes_in -> {something, something_else}|].
```

Step 2: Writing a JAVA application to interface with $\mathcal{F}$LORA-2. The following code loads a $\mathcal{F}$LORA-2 program from a file and then passes queries to the knowledge base.

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

    public static void main(String[] args) {
        // create a new session for a running instance of the engine
        FloraSession session = new FloraSession();
        System.out.println("Engine session started");

        // Assume that Java was called with -DINPUT_FILE=the-file-name
        String fileName = System.getProperty("INPUT_FILE");
        if(fileName == null || fileName.trim().length() == 0) {
            System.out.println("Invalid path to example file!");
            System.exit(0);

        }
        // load the program into module basic_mod
        session.loadFile(fileName,"basic_mod");

        /* Running queries from flogic_basics.flr */

        /* Query for persons */
        String command = "?X:person@basic_mod.";
        System.out.println("Query:"+command);
        Iterator<FloraObject> personObjs = session.ExecuteQuery(command);

        /* Printing out the person names and information about their kids */
        while (personObjs.hasNext()) {
            FloraObject personObj = personObjs.next();
            System.out.println("Person name:"+personObj);


        command = "person[instances -> ?X]@basic_mod.";
        System.out.println("Query:"+command);
        personObjs = session.ExecuteQuery(command);

        /* Prining out the person names  */
        while (personObjs.hasNext()) {
            Object personObj = personObjs.next();
            System.out.println("Person Id: "+personObj);
        }
```

```
        /* Example of ExecuteQuery with two arguments */
        Vector<String> vars = new Vector<String>();
        vars.add("?X");
        vars.add("?Y");

        Iterator<HashMap<String,FloraObject>> allmatches =
            session.ExecuteQuery("?X[believes_in -> ?Y]@basic_mod.",vars);
        System.out.println("Query:?X[believes_in -> ?Y]@basic_mod.");
        while(allmatches.hasNext()) {
            HashMap<String,FloraObject> firstmatch = allmatches.next();
            Object Xobj = firstmatch.get("?X");
            Object Yobj = firstmatch.get("?Y");
            System.out.println(Xobj+" believes in: "+?Yobj);
        }
        // quit the system
        session.close();
        System.exit(0);
    }
}
```

For the information on how to invoke the above Java class in the context of the Java-$\mathcal{F}$LORA-2 API, please see Section 1.3.

1.2 The High-Level Interface (experimental)

The high-level API operates by creating proxy Java classes for $\mathcal{F}$LORA-2 classes selected by the user. This enables the Java program to operate on $\mathcal{F}$LORA-2 classes by executing appropriate methods on the corresponding proxy Java classes. The use of the high-level API involves a number of steps, as described below.

Readers who intend to use only the low-level Java-$\mathcal{F}$LORA-2 interface can skip this section.

Note: This interface will not work for $\mathcal{F}$LORA-2 programs that use *non-alphanumeric* names for methods and predicates. For instance, if a program involves statements like `foo['bar$#123'->456]` then the interface might generate syntactically incorrect Java proxy classes and errors will be issued during the compilation.

Stage 1: Writing a $\mathcal{F}$LORA-2 file. We assume the same `flogic_basics.flr` file as in the previous example.

Stage 2: Generating Java classes that serve as proxies for $\mathcal{F}$LORA-2 classes. The $\mathcal{F}$LORA-2 side of the Java-to-$\mathcal{F}$LORA-2 high level API provides a predicate to generate Java proxy classes for each F-logic class which have a signature declaration in the $\mathcal{F}$LORA-2 knowledge base. A proxy class gets defined so that it would have methods to manipulate the attributes and methods of the corresponding F-logic class for which signature declarations are available. If an F-logic class has a declared value-returning attribute `foobar` then the proxy class will have the following methods. Each method name has the form *action*$S_1S_2S_3$`_foobar`, where *action* is either `get`, `set`, or `delete`. The specifier S_1 indicates the type of the method — `V` for value-returning, `B` for Boolean, and `P` for procedural. The specifier S_2 tells whether the operation applies to the signature of the method (`S`), e.g., `person[foobar=>string]`, or to the actual data (`D`), for example, `john[foobar->3]`. Finally, the specifier S_3 tells if the operation applies to the inheritable variant of the method (`I`) or its non-inheritable variant (`N`).

1. ```
   public Iterator<FloraObject> getVDI_foobar()
   public Iterator<FloraObject> getVDN_foobar()
   public Iterator<FloraObject> getVSI_foobar()
   public Iterator<FloraObject> getVSN_foobar()
   ```
   The above methods query the knowledge base and get all answers for the attribute `foobar`. They return iterators through which these answers can be processed one-by-one. Each object returned by the iterator is of type `FloraObject`. The `getVDN` form queries non-inheritable data methods and `getVDI` the inheritable ones. The `getVSI` and `getVSN` forms query the signatures of the attribute `foobar`.

2. ```
   public boolean setVDI_foobar(Vector value)
   public boolean setVDN_foobar(Vector value)
   public boolean setVSI_foobar(Vector value)
   public boolean setVSN_foobar(Vector value)
   ```
   These methods add values to the set of values returned by the attribute `foobar`. The values must be placed in the vector parameter passed these methods. Again, `setVDN` adds data for non-inheritable methods and `setVDI` is used for inheritable methods. `setVSI` and `setVSN` add types to signatures.

3. ```
   public boolean setVDI_foobar(Object value)
   public boolean setVDN_foobar(Object value)
   public boolean setVSI_foobar(Object value)
   public boolean setVSN_foobar(Object value)
   ```
   These methods provide a simplified interface when only one value needs to be added. It works like the earlier set_* methods, except that only one value given as an argument is added.

4. ```
   public boolean deleteVDI_foobar(Vector value)
   public boolean deleteVDN_foobar(Vector value)
   public boolean deleteVSI_foobar(Vector value)
   public boolean deleteVSN_foobar(Vector value)
   ```
   Delete a set of values of the attribute `foobar`. The set is specified in the vector argument.

5. ```
   public boolean deleteVDI_foobar(Object value)
   public boolean deleteVDN_foobar(Object value)
   public boolean deleteVSI_foobar(Object value)
   public boolean deleteVSN_foobar(Object value)
   ```
   A simplified interface for the case when only one value needs to be deleted.

6. ```
   public boolean deleteVDI_foobar()
   public boolean deleteVDN_foobar()
   ```

public boolean deleteVSI_foobar()
public boolean deleteVSN_foobar()
Delete all values for the attribute foobar.

For F-logic methods with arguments, the high-level API provides Java methods as above, but they take more arguments to accommodate the parameters that F-logic methods take. Let us assume that the F-logic method is called foobar2 and it takes parameters arg1 and arg2. As before the getVDI_*, setVDI_*, etc., forms of the Java methods are for dealing with inheritable $\mathcal{F}$LORA-*2* methods and the getVDN_*, setVDN_*, etc., forms are for dealing with non-inheritable $\mathcal{F}$LORA-*2* methods.

1. public Iterator<FloraObject> getVDI_foobar2(Object arg1, Object arg2)
   public Iterator<FloraObject> getVDN_foobar2(Object arg1, Object arg2)
   Obtain all values for the F-logic method invocation foobar2(arg1,arg2).

2. public boolean setVDI_foobar2(Object arg1, Object arg2, Vector value)
   public boolean setVDN_foobar2(Object arg1, Object arg2, Vector value)
   Add a set of methods specified in the parameter value for the method invocation foobar2(arg1,arg2).

3. public boolean setVDI_foobar2(Object arg1, Object arg2, Object value)
   public boolean setVDN_foobar2(Object arg1, Object arg2, Object value)
   A simplified interface when only one value is to be added.

4. public boolean deleteVDI_foobar2(Object arg1, Object arg2, Vector value)
   public boolean deleteVDN_foobar2(Object arg1, Object arg2, Vector value)
   Delete a set of values from foobar2(arg1,arg2). The set is given by the vector parameter value.

5. public boolean deleteVDI_foobar2(Object arg1, Object arg2, Object value)
   public boolean deleteVDN_foobar2(Object arg1, Object arg2, Object value)
   A simplified interface for deleting a single value.

6. public boolean deleteVDI_foobar2(Object arg1, Object arg2)
   public boolean deleteVDN_foobar2(Object arg1, Object arg2)
   Delete all values for the method invocation foobar2(arg1,arg2).

For Boolean and procedural methods, the generated methods are similar except that there is only one version for the set and delete methods. In addition, Boolean inheritable methods use the getBDI_*, setBDI_*, etc., form, while non-inheritable methods use the getBDN_*, etc., form. Procedural methods use the getPDI_*, getPDN_*, etc., forms. For instance,

1. public boolean getBDI_foobar3()
   public boolean getBDN_foobar3()
   public boolean getPDI_foobar3()
   public boolean getPDN_foobar3()

2. public boolean setBDI_foobar3()
   public boolean setBDN_foobar3()

```
public boolean setPDI_foobar3()
public boolean setPDN_foobar3()
```

3. ```
   public boolean deleteBDI_foobar3()
   public boolean deleteBDN_foobar3()
   public boolean deletePDI_foobar3()
   public boolean deletePDN_foobar3()
   ```

In addition, the methods to query the ISA hierarchy are available:

- `public Iterator<FloraObject> getDirectInstances()`
- `public Iterator<FloraObject> getInstances()`
- `public Iterator<FloraObject> getDirectSubClasses()`
- `public Iterator<FloraObject> getSubClasses()`
- `public Iterator<FloraObject> getSuperClasses()`
- `public Iterator<FloraObject> getDirectSuperClasses()`

These methods apply to the java proxy object that corresponds to the F-logic class person.

All these methods are generated automatically by executing the following $\mathcal{F}$LORA-2 query (defined in the `javaAPI` package). All arguments in the query must be bound:

```
// write(?Class,?Module,?ProxyClassFileName).
?- write(foo,example,'myproject/foo.java').
```

The first argument specifies the class for which to generate the methods, the file name tells where to put the Java file for the proxy object, and the model argument tells which $\mathcal{F}$LORA-2 model to load this program to. The result of this execution will be the file `foo.java` which should be included with your java program (the program that is going to interface with $\mathcal{F}$LORA-2). Note that because of the Java conventions, the file name must have the same name as the class name. It is important to remember, however, that proxy methods will be generated only for those F-logic methods that have been declared using signatures.

Let us now come back to our program `flogic_basics.flr` for which we want to use the high-level API. Suppose we want to query the person class. To generate the proxy declarations for that class, we create the file `person.java` for the module `basic_mod` as follows.

```
?- load{'examples/flogic_basics'>>basic_mod}.
?- load{javaAPI}.
?- write(person,basic_mod,'examples/person.java')@\prolog
```

The `write` method will create the file `person.java` shown below. The methods defined in `person.java` are the class constructors for `person`, the methods to query the ISA hierarchy, and the "get", "set" and "delete" methods for each method and attribute declared in the $\mathcal{F}$LORA-2 class `person`. The parameters for the "get", "set" and "delete" Java methods
```

are the same as for the corresponding $\mathcal{F}$LORA-2 methods. The first constructor for class `person` takes a low-level object of class `javaAPI.src.FloraObject` as a parameter. The second parameter is the $\mathcal{F}$LORA-2 module for which the proxy object is to be created. The second `person`-constructor takes F-logic object Id instead of a low-level `FloraObject`. It also takes the module name, as before, but, in addition, it takes a session for a running $\mathcal{F}$LORA-2 instance. The session parameter was not needed for the first `person`-constructor because `FloraObject` is already attached to a concrete session.

It can be seen from the form of the proxy object constructors that proxy objects are attached to specific $\mathcal{F}$LORA-2 modules, which may seem to go against the general philosophy that F-logic objects do not belong to any module — only their methods do. On closer examination, however, attaching high-level proxy Java objects to modules makes perfect sense. Indeed, a proxy object encapsulates operations for manipulating F-logic attributes and methods, which belong to concrete $\mathcal{F}$LORA-2 modules, so the proxy object needs to know which module it operates upon.

person.java file

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class person {

    public FloraObject sourceFloraObject;

    // proxy objects' constructors
    public person(FloraObject sourceFloraObject, String moduleName) { ... }
    public person(String floraOID,String moduleName, FloraSession session) { ... }

    // ISA hierarchy queries
    public Iterator<FloraObject> getDirectInstances() { ... }
    public Iterator<FloraObject> getInstances() { ... }
    public Iterator<FloraObject> getDirectSubClasses() { ... }
    public Iterator<FloraObject> getSubClasses() { ... }
    public Iterator<FloraObject> getDirectSuperClasses() { ... }
    public Iterator<FloraObject> getSuperClasses() { ... }

    // Java methods for manipulating methods
    public boolean setVDI_age(Object value) { ... }
    public boolean setVDN_age(Object value) { ... }
    public Iterator<FloraObject> getVDI_age(){ ... }
    public Iterator<FloraObject> getVDN_age(){ ... }
    public boolean deleteVDI_age(Object value) { ... }
    public boolean deleteVDN_age(Object value) { ... }
    public boolean deleteVDI_age() { ... }
    public boolean deleteVDN_age() { ... }
    public boolean setVDI_salary(Object year,Object value) { ... }
```

```
    public boolean setVDN_salary(Object year,Object value) { ... }
    public Iterator<FloraObject> getVDI_salary(Object year) { ... }
    public Iterator<FloraObject> getVDN_salary(Object year) { ... }
    public boolean deleteVDI_salary(Object year,Object value) { ... }
    public boolean deleteVDN_salary(Object year,Object value) { ... }
    public boolean deleteVDI_salary(Object year) { ... }
    public boolean deleteVDN_salary(Object year) { ... }
    public boolean setVDI_hobbies(Vector value) { ... }
    public boolean setVDN_hobbies(Vector value) { ... }
    public Iterator<FloraObject> getVDI_hobbies(){ ... }
    public Iterator<FloraObject> getVDN_hobbies(){ ... }
    public boolean deleteVDI_hobbies(Vector value) { ... }
    public boolean deleteVDN_hobbies(Vector value) { ... }
    public boolean deleteVDI_hobbies(){ ... }
    public boolean deleteVDN_hobbies(){ ... }
    public boolean setVDI_instances(Vector value) { ... }
    public boolean setVDN_instances(Vector value) { ... }
    public Iterator<FloraObject> getVDI_instances(){ ... }
    public Iterator<FloraObject> getVDN_instances(){ ... }
    public boolean deleteVDI_instances(Vector value) { ... }
    public boolean deleteVDN_instances(Vector value) { ... }
    public boolean deleteVDI_instances(){ ... }
    public boolean deleteVDN_instances(){ ... }
    public boolean setVDI_kids(Vector value) { ... }
    public boolean setVDN_kids(Vector value) { ... }
    public Iterator<FloraObject> getVDI_kids(){ ... }
    public Iterator<FloraObject> getVDN_kids(){ ... }
    public boolean deleteVDI_kids(Vector value) { ... }
    public boolean deleteVDN_kids(Vector value) { ... }
    public boolean deleteVDI_kids(){ ... }
    public boolean deleteVDN_kids(){ ... }
    public boolean setVDI_believes_in(Vector value) { ... }
    public boolean setVDN_believes_in(Vector value) { ... }
    public Iterator<FloraObject> getVDI_believes_in(){ ... }
    public Iterator<FloraObject> getVDN_believes_in(){ ... }
    public boolean deleteVDI_believes_in(Vector value) { ... }
    public boolean deleteVDN_believes_in(Vector value) { ... }
    public boolean deleteVDI_believes_in(){ ... }
    public boolean deleteVDN_believes_in(){ ... }
}
```

Stage 3: Writing Java applications that use the high-level API. The following program (`flogicbasicsExample.java`) shows several queries that use the high-level interface. The class `person.java` is generated at the previous stage. The methods of the high-level interface operate on Java objects that are proxies for $\mathcal{F}$LORA-*2* objects. These Java objects are members of the class `javaAPI.src.FloraObject`. Therefore, before one can use the high-level methods one need to first retrieve the appropriate proxy objects on which to operate. This is done by sending an appropriate query through the method `ExecuteQuery`—the same method that was used in the low-level interface. Alternatively, `person`-objects could be constructed using the 3-argument proxy constructor, which takes F-logic oids.

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

    public static void main(String[] args) {
        /* Initializing the session */
        FloraSession session = new FloraSession();
        System.out.println("Flora session started");

        String fileName = "examples/flogic_basics"; // must be a valid path
        /* Loading the flora file */
        session.loadFile(fileName,"basic_mod");

        // Retrieving instances of the class person through low-level API
        String command = "?X:person@basic_mod.";
        System.out.println("Query:"+command);
        Iterator<FloraObject> personObjs = session.ExecuteQuery(command);

        /* Print out person names and information about their kids */
        person currPerson = null;
        while (personObjs.hasNext()) {
            FloraObject personObj = personObjs.next();
            // Elevate personObj to the higher-level person-object
            currPerson =new person(personObj,"basic_mod");

            /* Set that person's age to 50 */
            currPerson.setVDN_age("50");

            /* Get this person's kids */
            Iterator<FloraObject> kidsItr = currPerson.getVDN_kids();
            while (kidsItr.hasNext()) {
                FloraObject kidObj = kidsItr.next();
                System.out.println("Person: " + personObj + " has kid: " +kidObj);

                person kidPerson = null;
                // Elevate kidObj to kidPerson
                kidPerson = new person(kidObj,"basic_mod");

                /* Get kidPerson's hobbies */
                Iterator<FloraObject> hobbiesItr = kidPerson.getVDN_hobbies();
```

```
                while(hobbiesItr.hasNext()) {
                    FloraObject hobbyObj = hobbiesItr.next();
                    System.out.println("Kid:"+kidObj + " has hobby:" +hobbyObj);
                }
            }
        }

        FloraObject age;
        // create a person-object directly by supplying its F-logic OID
        // father(mary)
        currPerson = new person("father(mary)", "example", session);
        Iterator<FloraObject> maryfatherItr = currPerson.getVDN_age();
        age = maryfatherItr.next();
        System.out.println("Mary's father is " + age + " years old");

        // create a proxy object for the F-logic class person itself
        person personClass = new person("person", "example", session);
        // query its instances through the high-level interface
        Iterator<FloraObject> instanceIter = personClass.getInstances();
        System.out.println("Person instances using high-level API:");
        while (instanceIter.hasNext())
            System.out.println("    " + instanceIter.next());

        session.close();
        System.exit();
    }
}
```

1.3 Executing Java Application Programs with $\mathcal{F}$LORA-*2*

To run Java programs that interface with $\mathcal{F}$LORA-*2*, follow the following guidelines.

- Place the files `flogicsbasicsExample.java` (the program you have written) and `person.java` (the automatically generated file) in the same directory and compile them using the `javac` command. Add the jar-files containing the API code and `interprolog.jar` to the Java classpath:
  - *FLORADIR*`/java/flora2java.jar`
  - *FLORADIR*`/java/interprolog.jar`

  FLORADIR here should be replaced with the value of the variable `FLORADIR`, which is set by the scripts `flora_settings.sh` (Linux/Mac) or `flora_settings.bat` (Windows), as mentioned in Section 1.1 on page 1.

- Generally, Java programs that call $\mathcal{F}$LORA-*2* should be invoked using the following command. For Unix-like systems (Linux, Mac, etc.), change `%`*VAR*`%` to `$`*VAR*:

```
%JAVA_BIN%\java -DPROLOGDIR=%PROLOGDIR%
                -DFLORADIR=%FLORADIR%
                -Djava.library.path=%PROLOGDIR%
                -classpath %MYCLASSPATH% flogicbasicsExample
```

The above command uses several shell variables, which are explained below. Instead of using the variables, one can substitute their values directly.

`JAVA_BIN`: This variable should point to the directory containing the `java` and `javac` executables of the JDK.

`PROLOGDIR`: This variable should be set to the directory containing the XSB executable.

`FLORADIR`: This variable should be set to the directory containing the $\mathcal{F}$LORA-*2* system.

`MYCLASSPATH`: This variable should include the jar files containing the API code, i.e., `.../java/flora2java.jar` and file `.../java/interprolog.jar`, plus the above `flogicbasicsExample` class. For instance, it can be set to `%CLASSPATH%;`*FLORADIR*`/java/flora2java.jar;`*FLORADIR*`/java/interprolog.jar;` `flogicbasicsExample`. For Linux and Mac, use ':' instead of ';' as a separator. As before, *FLORADIR* should be replaced with a proper value, as explained above.

- Some Java applications may employ additional shell variables. For instance, the program that uses the low-level API in Section 1.1 (in Step 2) has the line

```
String fileName = System.getProperty("INPUT_FILE");
```

which means that it expects the shell variable `INPUT_FILE` to be set. In this particular case, it expects that variable to have the address of the `flogic_basics.flr` $\mathcal{F}$LORA-*2* file, which it then loads. Therefore, the java command shown above would also need this parameter:

```
-DINPUT_FILE=%INPUT_FILE%
```

In general, one such additional parameter is needed for each property that the Java application queries using the `getProperty()` method.

1.4 Summary of the Variables Used by the Interface

The Java-$\mathcal{F}$LORA-*2* interface needs the following shell variables to be set:

- `JAVA_HOME` - this is normally set when you install Java. If not, set this variables manually.
- The following variables can be set by executing the scripts `flora_settings.bat` (Windows) or `flora_settings.sh` (Linux/Mac) located in `flora2/java/`:
  - `FLORADIR` — the path to the $\mathcal{F}$LORA-*2* installation directory.
  - `PROLOGDIR` — the path to the folder containing XSB executable.

If you need to set the above variables in some other way, look inside the above scripts to get the exact values these variables should be set to.

- The following variable is set by the scripts `unixVariables.sh` or `windowsVariables.bat`:

  – `JAVA_BIN` — the directory where Java executables are (java, javac).

  If you need to set this variable without running the aforesaid script, you need to know the correct value for that variable. The simplest way is to execute the script and then check the value of environment variable `JAVA_BIN`.

1.5 Building the Prepackaged Examples

Sample applications of the Java-$\mathcal{F}$LORA-2 interface are found in the `java/API/examples` folder. To build the code for the interface, use the scripts `build.bat` or `build.sh` (or `build.bat` on Windows) in the `java/API` folder. To build the the examples, use the scripts `buildExample.sh` or `buildExample.bat` in the `java/API/examples` folder, whichever applies. For instance, to build the `flogicbasicsExample` example, use these commands on Linux, Mac, and other Unix-like systems:

```
cd examples
buildExample.sh flogicbasicsExample
```

On Windows, use this:

```
cd examples
buildExample.bat flogicbasicsExample
```

To run the demos, use the scripts `runExample.sh` or `runExample.bat` in the `java/API/examples` folder. For instance, to run the `flogicbasicsExample`, use this command on Linux, Mac, and the like:

```
runExample.sh flogicbasicsExample
```

On Windows, use this:

```
runExample.bat flogicbasicsExample
```

Chapter 2

Persistent Modules

by Vishal Chowdhary

This chapter describes a $\mathcal{F}$LORA-2 package that enables persistent modules. A *persistent module* (abbr., PM) is like any other $\mathcal{F}$LORA-2 module except that it is associated with a database. Any insertion or deletion of base facts in such a module results in a corresponding operation on the associated database. This data persists across $\mathcal{F}$LORA-2 sessions, so the data that was present in such a module is restored when the system restarts and the module is reloaded.

2.1 PM Interface

A module becomes persistent by executing a statement that associates the module with an ODBC data source described by a DSN. To start using the module persistence feature, first load the following package into some module. For instance:

```
?- [persistentmodules>>pm].
```

The following API is available. Note that if you load `persistentmodules` into some other module, say `foo`, then `foo` should be used instead of `pm` in the examples below.

- `?- ?Module[attach(?DSN,?DB,?User,?Password)]@pm.`
  This action associates the data source described by an ODBC `DSN` with the module. If `?DB` is a variable then the database is taken from the DSN. If `?DB` is bound to an atomic string, then that particular database is used. Not all DBMSs support the operation of replacing the DSN's database at run time. For instance, MS Access or PostgresSQL do not. In this case, `?DB` must stay unbound or else an error will be issued. For other DBMS, such as MySQL, SQL Server, and Oracle, `?DB` can be bound.

  The `?User` and `?Password` must be bound to the user name and the password to be used to connect to the database.

  The database specified by the DSN must already exist and must be created by a previous call to the method `attachNew` described below. Otherwise, the operation is aborted.

The database used in the `attach` statement must not be accessed directly—only through the persistent modules interface. The above statement will create the necessary tables in the database, if they are not already present.

Note that the same database can be associated with several different modules. The package will not mix up the facts that belong to different modules.

- `?- ?Module[attachNew(?DSN,?DB,?User,?Password)]@pm.`
  Like `attach`, but a new database is created as specified by `?DSN`. If the same database already exists, an exception of the form `FLORA_DB_EXCEPTION(?ErrorMsg)` is thrown. (In a program, include `flora_exceptions.flh` to define `FLORA_DB_EXCEPTION`; in the shell, use the symbol `'_$flora_db_error'`.) This method creates all the necessary tables, if they are not already present.

  Note that this command works only with database systems that understand the SQL command `CREATE DATABASE`. For instance, MS Access does not support this command and will cause an error.

- `?- ?Module[detach]@pm.`
  Detaches the module from its database. The module is no longer persistent in the sense that subsequent changes are not reflected in any database. However, the earlier data is not lost. It stays in the database and the module can be reattached to that database.

- `?- ?Module[loadDB]@pm.`
  On re-associating a module with a database (i.e., when `?Module[attach(?DSN, ?DB,?User,?Password)]@pm` is called in a new $\mathcal{F}$LORA-*2* session), database facts previously associated with the module are loaded back into it. However, since the database may be large, $\mathcal{F}$LORA-*2* does not preload it into the main memory. Instead, facts are loaded on-demand. If it is desired to have all these facts in main memory at once, the user can execute the above command. If no previous association between the module and a database is found, an exception is thrown.

- `?- set_field_type(?Type)@pm.`
  By default, $\mathcal{F}$LORA-*2* creates tables with the VARCHAR field type because this is the only type that is accepted by all major database systems. However, ideally, the CLOB (character large object) type should be used because VARCHAR fields are limited to 4000-7000 characters, which is usually inadequate for most needs. Unfortunately, the different database systems differ in how they support CLOBs, so the above call is provided to let the user specify the field types that would be acceptable to the system(s) at hand. The call should be made right before `attachNew` is used. Examples:

```
?- set_field_type('TEXT DEFAULT NULL')@pm.   // MySQL, PostgreSQL
?- set_field_type('CLOB DEFAULT NULL')@pm.   // Oracle, DB2
```

Once a database is associated with the module, querying and insertion of the data into the module is done as in the case of regular (transient) modules. Therefore PM's provide a transparent and natural access to the database and every query or update may, in principle, involve a database operation. For example, a query like `?- ?D[dept -> ped]@StonyBrook.` may invoke the SQL `SELECT` operation if module `StonyBrook` is associated with a database. Similarly `insert{a[b -> c]@stonyBrook}` and `delete{a[e -> f]@stonyBrook}` will invoke

SQL `INSERT` and `DELETE` commands, respectively. Thus, PM's provide a high-level abstraction over the external database.

Note that if `?Module[loadDB]@pm` has been previously executed, queries to a persistent module will *not* access the database since $\mathcal{F}$LORA-*2* will use its in-memory cache instead. However, insertion and deletion of facts in such a module will still cause database operations.

2.2 Examples

Consider the following scenario sequence of operations.

```
// Create new modules mod, db_mod1, db_mod2.
flora2 ?-  newmodule{mod}, newmodule{db_mod1}, newmodule{db_mod2}.
flora2 ?-  [persistentmodules>>pm].

// insert data into all three modules.
flora2 ?-  insert{q(a)@mod,q(b)@mod,p(a,a)@mod}.
flora2 ?-  insert{p(a,a)@db_mod1, p(a,b)@db_mod1}.
flora2 ?-  insert{q(a)@db_mod2,q(b)@db_mod2,q(c)@db_mod2}.

//  Associate modules db_mod1, db_mod2 with an existing database db
//  The data source is described by the DSN mydb.
flora2 ?-  db_mod1[attach(mydb,db,user,pwd)]@pm.
flora2 ?-  db_mod2[attach(mydb,db,user,pwd)]@pm.

// insert more data into db_mod2 and mod.
flora2 ?-  insert{a(p(a,b,c),d)@db_mod2}.
flora2 ?-  insert{q(a)@mod,q(b)@mod,p(a,a)@mod}.

// shut down the engine
flora2 ?-  \halt.
```

Restart the $\mathcal{F}$LORA-*2* system.

```
// Create the same modules again
flora2 ?-  newmodule{mod}, newmodule{db_mod1}, newmodule{db_mod2}.

// try to query the data in any of these modules.
flora2 ?-  q(?X)@mod.
No.

flora2 ?-  p(?X,?Y)@db_mod1.
No.

//  Attach the earlier database to db_mod1.
flora2 ?-  [persistentmodules>>pm].
```

```
flora2 ?-  db_mod1[attach(mydb,db,user,pwd)]@pm.

// try querying again...

// Module mod is still not associated with any database and nothing was
// inserted there even transiently, we have:
flora2 ?-  q(?X)@mod.
No.

// But the following query retrieves data from the database associated
// with db_mod1.
flora2 ?-  p(?X,?Y)@db_mod1.
?X = a,
?Y = a.

?X = a,
?Y = b.

Yes.

// Since db_mod2 was not re-attached to its database,
// it still has no data, and the query fails.
flora2 ?-  q(?X)@db_mod2.

No.
```

Chapter 3

SGML and XML Import for $\mathcal{F}$LORA-*2*

by Rohan Shirwaikar and Michael Kifer

This chapter documents the $\mathcal{F}$LORA-*2* package that provides XML and XPath parsing capabilities. The main predicates support parsing SGML, XML, and HTML documents, and create $\mathcal{F}$LORA-*2* objects in the user specified module. Other predicates evaluate XPath queries on XML documents and create $\mathcal{F}$LORA-*2* objects in user specified modules. The predicates make use of the **sgml** and **xpath** packages of XSB.

3.1 Introduction

This package supports parsing SGML, XML, and HTML documents, converting them to sets of $\mathcal{F}$LORA-*2* objects stored in user-specified $\mathcal{F}$LORA-*2* modules. The SGML interface provides facilities to parse input in the form of files, URLs and strings (Prolog atoms).

For example, the following XML snippet

```
<greeting id='1'>
<first ssn=111'>
John
</first>
</greeting>
```

will be converted into the following $\mathcal{F}$LORA-*2* objects:

```
obj1[ greeting -> obj2]
obj2[ attribute(id) -> '1']
obj2[ first -> obj3]
obj3[ attribute(ssn) -> '111']
obj3[ \text -> 'John']
```

To load the XML package, just call any of the API calls at the $\mathcal{F}$LORA-2 prompt.

The following calls are provided by the package. They take SGML, XML, HTML, or XHTML documents and create the corresponding $\mathcal{F}$LORA-2 objects as specified in Section 3.3.

`?InDoc[load_sgml(?Module) -> ?Warn]`@**xml**
: Import XML data as $\mathcal{F}$LORA-2 objects.

`?InDoc[load_xml(?Module) -> ?Warn]`@**xml**
: Import SGML data as $\mathcal{F}$LORA-2 objects.

`?InDoc[load_html(?Module) -> ?Warn]`@**xml**
: Import HTML data as $\mathcal{F}$LORA-2 objects.

`?InDoc[load_xhtml(?Module) -> ?Warn]`@**xml**
: Import XHTML as $\mathcal{F}$LORA-2 objects.

The arguments to these predicates have the following meaning:

`?InDoc` is an input SGML, XML, HTML, or XHTML document. It must have one of these forms: `url('`*url*`')`, `file('`*file name*`')` or `string('`*document as a string*`')`. If `?InDoc` is just a plain Prolog atom ($\mathcal{F}$LORA-2 symbol) then `file(?Source)` is assumed. `?Module` is the name of the $\mathcal{F}$LORA-2 module where the objects created by the above calls should be placed; it must be bound. `?Warn` gets bound to a list of warnings, if any are generated, or to an empty list; it is an output variable.

3.2 Import Modes for XML in Ergo

XML can be imported into $\mathcal{F}$LORA-2 in several different ways, which can be specified via the `set_mode(...)@\xml` primitive. These modes control two aspects of the import:

- white space handling, and
- navigation links that may be added to the imported data.

3.2.1 White Space Handling

The XML standard requires that white space (blanks, tabs, newlines, etc.) must be preserved by XML parsers. However, in the applications where $\mathcal{F}$LORA-2 is used, XML typically is viewed as a format for data in which white space is immaterial. For that reason, by default, the $\mathcal{F}$LORA-2's XML parser operates in the *data mode* in which every string is trimmed on both sides to remove the white space. In addition, the empty strings `''` are ignored. This implies that, for example, there will be no `\text` attribute to represent a situation like this:

```
<doc>
    <spaceonly>   </spaceonly>
</doc>
```

and the only data created to represent the above document will be

```
obj1[doc->obj2]@bar
obj2[spaceonly->obj3]@bar
```

(plus some additional navigational data about order, siblings, parents, etc.). This means that, if capturing certain white space is needed, it should be encoded explicitly in some way, e.g.,

```
<spaceonly>___</spaceonly>
```

instead of three spaces.

Alternatively, one can request to change the XML parsing mode to *raw*:

```
   ?- set_mode(raw)@\xml.
```

In this case, the parser will switch to the pedantic way XML parsers are supposed to interpret XML and all white space will be preserved. However, beware what you wish because even for the above tiny example the representation will end up not pretty because every little bit of white space will be there (even the one that comes from line breaks):

```
obj1[doc->obj2]@bar
obj2[\text->obj3]@bar
obj2[\text->obj6]@bar
obj2[spaceonly->obj4]@bar
obj3[\string->'
  ']@bar
obj4[\text->obj5]@bar
obj5[\string->'   ']@bar
obj6[\string->'
']@bar
```

It is more than likely an $\mathcal{F}$LORA-*2* user will not want objects like `obj3` and `obj6`.

Finally, if the *raw* mode is not what is desired, one can always switch back to the data mode:

```
   ?- set_mode(data)@\xml.
```

3.2.2 Requesting Navigation Links

This aspect can be changed via the calls

```
   ?- set_mode(nonavlinks)@\xml.  // the default
   ?- set_mode(navlinks)@\xml.
```

where `nonavlinks` is the default.

The `nonavlinks` method uses a slightly simpler translation from XML to $\mathcal{F}$LORA-*2* objects and no extra navigation links are provided. This mode is used when the imported XML document has known tructure and is viewed simply as set of data to be ingested (e.g., payroll data).

In the `navlinks` mode, the representation is slightly more complex but, most importantly, that imported data includes additional information that provides parent/child/sibling links among XML objects as well as the ordering information, which allows one to reconstruct the original XML document. This mode is used when the structure of the input XML has high variability or may even be arbitrary. This arises, for instance, when one needs to transform arbitrary XML import or to extract certain information from unknown structures. The exact representation of this navigational information is described in subsequent sections.

3.3 Mapping XML to $\mathcal{F}$LORA-*2* Objects

This mapping is based on an XML-to-$\mathcal{F}$LORA-*2* object correspondence developed by Guizhen Yang. It specifies how an XML parser can construct the corresponding F-logic objects after parsing an input XML document. The basic ideas are as follows:

- XML elements, attribute values, and text strings are modeled as objects in F-logic.
- XML elements are reachable from parent objects via F-logic frame attributes of the same name as the XML element name.
- XML element attributes are also modeled as F-logic frame attributes but their name is `attribute(`*XML attribute name*`)`.

This mapping does not address comments or processing instructions—they are simply ignored. However, this mapping does address the issue of mixed text/element content in which plain text and subelements are interspersed. This mapping also assumes that XML entities are resolved by the XML parser.

3.3.1 Invention of Object Ids for XML Elements

According to the XML specification 1.0, an XML element can be identified by an oid that is unique across the document. The import mechanism invents such an oid automatically. Sitting on top of the XML root element, there is an additional root object which just functions as the access point to the entire object hierarchy corresponding to the XML document. The oids of leaf nodes, which have no outgoing arcs and carry plain text only, are just the string values themselves.

For example, the following XML document

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name first="John"
        last="Smith"/>
</person>
```

is represented via the following F-logic objects:

```
obj1[person -> obj2].
obj2[attribute(ssn) -> '111-22-3333', name -> obj3].
obj3[attribute(first) -> John, attribute(last) -> Smith].
```

Here `obj1` is the root object, `obj2` is the object corresponding to the `person` element, and `obj3` is the object that represents the `name` element. The strings `'111-22-3333'`, `John`, and `Smith` are oids that stand for themselves.

3.3.2 Text and Mixed Element Content

The content of an XML element may consist of plain text, or subelements interspersed with plain text as in

```
<greeting>Hi! My name is <first>John</first><last>Smith</last>.</greeting>
```

How text is actually handled in the translation to F-logic depends on the mode of import: `nonavlinks` or `navlinks`. The former is simpler because it discards all the information about the order of the text nodes with respect to subelements and other text nodes.

- **In the `nonavlinks` mode**:
  Each text segment is modeled as a value of the attribute `\text` of the parent element-object of that text segment.[1] Thus, for the above XML fragment, the translation would be

  ```
  obj1[greeting -> obj2].
  obj2[\text -> {'Hi! My name is ', '.'},
       first -> obj3,
       last  -> obj4
  ].
  obj3[\text -> John].
  obj4[\text -> Smith].
  ```

- **In the `navlinks` mode**:
  Here the order of the text and subelement nodes must be preserved and so each text node is modeled as if it were a value of a special attribute `\string` in an empty XML element named \text, e.g.,

  ```
  <\text \string="John"/>
  ```

  As a consequence, a separate F-logic object is created to represent each text segment. (Compare this to the translation in the `nonavlinks` mode, which does not create separate objects for text nodes.) Thus, for the aforesaid `greetings` element the translation will be

[1] Of course, XML does not allow such names for tags and attributes, and this is the whole point: adding such an invented name to the F-logic translation will not clash with other tag names that might be used in the XML documents.

```
obj1[greeting -> obj2].
obj2[\text -> {obj3, obj8},
     first -> obj4,
     last  -> obj6
].
obj3[\string -> 'Hi! My name is '].
obj4[\text -> obj5].
obj5[\string -> John].
obj6[\text -> obj7].
obj7[\string -> Smith].
obj8[\string -> '.'].
```

How exactly the aforesaid order is preserved in the `navlinks` mode is explained later.

3.3.3 Translation of XML Attributes

An XML attribute, *attr*, in an element is translated as an attribute by the name `attribute`(*attr*) attached to the object that corresponds to that element.

XML element attributes of type IDREFS are multivalued, in the sense that their value is a string consisting of one or more oids separated by whitespaces. Therefore, the value of such an attribute is a set. The value of an XML IDREFS attribute is represented as a list.

For example, the following XML segment:

```
<paper id="yk00" references="klw95 ckw91">
  <title>paper title</title>
</paper>
```

will generate the following F-logic atoms, assuming that the `reference` attribute is of type IDREFS:

```
obj1[paper -> obj2]
obj2[title -> obj4]
obj2[attribute(id) -> yk00]
obj2[attribute(references) -> 'klw95 ckw91'
obj4[\text -> obj5]      // here we assume that the navlinks mode was used
obj5[\string -> 'paper title']
```

However: if the document has an associated DTD *and* the attribute `references` were specified there as `IDREFS` as in

```
<!ATTLIST paper references IDREFS #IMPLIED>
```

then that attribute is translated as

```
obj2[attribute(references)->[klw95,ckw91]]
```

i.e., the value becomes a list.

With this, we are done describing the `nonavlinks` mode. The remaining subsections in the current section apply to the `navlinks` mode only.

3.3.4 Ordering

This section applies to the `navlinks` mode only.

XML is order-sensitive and the order in which elements and text appear is significant, in general. The order of the attributes within the same element tag is *not* significant, however.

While the `nonavlinks` mode is sufficient for most data-intensive uses of XML in $\mathcal{F}$LORA-*2*, more complex tasks may require the knowledge of how items are ordered within XML documents. Specifying a total order among the elements and text in an XML document suffices for that purpose, if this order agrees with the local order within each element's content.

Consider the following XML document

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <email>jsmith@abc.com</email>
</person>
```

It can be represented by the tree in Figure 3.1 in which the parenthesized integers show the total order assigned to the F-logic objects.

The ordering information that exists in XML documents is captured in F-logic via a special attribute called `\order`, which tells position within the total ordering for each element and text node. It is for that purpose that text segments are modeled in the `navlinks` mode as element-style objects (each segment having its own oid) and not simply as attributes, as is the case with the simpler `nonavlinks` mode.

3.3.5 Additional Attributes and Methods in the `navlinks` Mode

Since the `navlinks` mode is intended for applications that need to navigate from children to parents, to siblings, and more, the importer adds the following additional attributes and methods to the F-logic objects into which XML elements and text are mapped.

1. `\in_arc`
   For each node, `\in_arc` returns the unordered set of labels of the arcs pointing to this node, i.e., this node's in-arcs. Roughly, `\in_arc` is defined as follows:

   ```
   ?O[\in_arc -> ?InArc] :- ?[?InArc -> ?O].
   ```

```
                    obj0 (0)
                    |
                    | person
                    |
        id /----  obj1 (1)
          /         ^
         /         / \
111-22-3333  name /   \ email
                 /     \
          (2) obj2     obj7 (7)
               / \        \
        first /   \last    \ \text
             /     \        \
      (3) obj3     obj5 (5)  obj8 (8)
            |      |          \
      \text |      | \text     \  \string
            |      |            \
      (4) obj4     obj6 (6)    'jsmith@abc.com'
            |      |
    \string |      | \string
            |      |
           John  Smith
```

Figure 3.1: Total ordering of the F-logic objects arising from XML ordering

Note that for a node representing a text segment, the value of its `\in_arc` attribute is `\text`.

2. `\parent`
   For each node, `\parent` returns the oid of the parent node.

3. `\leftsibling`
   For each node, `\leftsibling` returns the oid of the node appearing immediately before the current node. This attribute is not defined for the nodes without a left sibling.

4. `\rightsibling`
   For each node, `\rightsibling` returns the oid of the node appearing immediately after the current node. This attribute is not defined for the nodes without a right sibling.

5. `\childcount`
   For each element node, `\childcount` returns the number of the immediate children of that element, which includes subelements and text segments.

6. `\childlist`
   For each element node, `\childlist` returns a list of the oids of the immediate children (subelements and text segments) of that element.

7. `\child(N)`

For each node, \child(N) returns the N-th child, where 0 $\leq$ N < \childcount. Note: the first child is the 0-th child.

8. \in_child_arc(N)
   For each node, \in_child_arc(N) returns the in-arcs of the N-th child, where 0 $\leq$ N < \childcount. This attribute is defined as follows:

```
?O[\in_child_arc(?N)->?InArc] :- ?O[\child(?N)->?[\in_arc->?InArc]].
```

3.4 Inspection Predicates

This section applies both to the nonavlinks mode and the navlinks mode.

It is sometimes hard to see which objects have actually been created to represent an XML document or an element. This is especially true in case of navlinks mode, which includes a host of special navigational attributes. The purpose of inspection predicates is to provide a simple way to view the objects, and they also filter the navigational attributes out. Consider the document foo.xml below:

```
<mydoc id='1'><first ssn='111'>John</first></mydoc>
```

Even for such a simple document, the query

```
?- 'foo.xml'[load_xml(bar) -> ?W]@\xml.    // load foo.xml into module bar
?- ?_X[?_Y->?_Z]@bar, ?Z = ${?_X[?_Y->?_Z]}.  // get all facts
```

that asks for all the facts—stored and derived—will yield 56 results in the navlinks mode, which is overwhelming to inspect visually. However, the core facts that describe these objects are only 8, and they can be obtained by asking the query

```
?- bar[show->?P]@\xml.
```

One furthermore might want to see the representation of individual elements (e.g., element named first):

```
?- bar[show(first)->?P]@\xml.
```

and this is much more manageable:

```
?P = ${obj4[\text->obj5]@bar}
?P = ${obj4[attribute(ssn)->'111']@bar}
```

or of elements that have particular attributes (ssh in this example):

```
?- bar[show(attribute(ssn))->?P]@\xml.
```

which yields the same result as above (because the element first has the attribute ssn).

3.5 XPath Support

The XPath support is based on the XSB `xpath` package, which must be configured as explained in the XSB manual. This package, in turn, relies on the XML parser called `libxml2`. It comes with most Linux distributions and is also available for Windows, MacOS, and other Unix-based systems from `http://xmlsoft.org`. Note that both the library itself and the `.h` files of that library must be installed.

Note: XPath support does not currently work under Windows 64 bit (but does under 32 bits) due to the fact that we could not produce a working `libxml2.lib` file (`xmlsoft.org` provides `linxml2.dll` for Windows 64, but not `libxml2.lib`).

The following predicates are provided. They select parts of the input document using the provided XPath expression and create $\mathcal{F}$LORA-*2* objects as specified in Section 3.3. These predicates handle XML, SGML, HTML, and XHTML, respectively.

`?InDoc[xpath_xml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to an XML document and import the result
`?InDoc[xpath_xhtml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to XHTML and import the result

The arguments have the following meaning:

`InDoc` specifies the input document; this parameter has the same format as in Section 3.1. `?XPath` is an XPath expression specified as a Prolog atom. `?Module` is the module where the resulting $\mathcal{F}$LORA-*2* objects should be placed. `?Module` must be bound. `?Warn` gets bound to a list of warnings, if any are generated during the processing—or to an empty list, if none.

`?NamespacePrefList` is a string that has the form of a space separated list of items of the form *`prefix = namespaceURL`*. This allows one to use namespace prefixes in the `?XPath` parameter. For example if the XPath expression is `'/x:html/x:head/x:meta'` where `x` stands for `'http://www.w3.org/1999/xhtml'`, then this prefix would have to be defined in `?NamespacePrefList`:

```
url('http://w3.org')[xpath_xhtml('/x:html/x:head/x:meta',
                                  'x=http://www.w3.org/1999/xhtml',
                                  foomodule)
                          -> ?Warnings]@\xml.
```

3.6 Low-level Predicates

This section describes low-level predicates in the XML package. These predicates parse the input documents into Prolog terms that then must be further traversed recursively in order to get the desired information.

- `parse_structure(?InDoc,?InType,?Warnings,?ParsedDoc)@\xml` — take the document `?InDoc` or type `?InType` (`xml`, `xhtml`, `html`, `sgml`) and parse it as a Prolog term (will not be imported into any module as an object).

- `apply_xpath(?InDoc,?InType,?XPathExp,?NamespacePrefList,?Warnings,?ParsedDoc)@\xml` — like the above, but first applies the XPath expression `?XPathExp` to `?InDoc`. The `?InType` parameter must be bound to `xml` or `xhtml`.

The output, `?ParsedDoc`, is a Prolog term that represents the parse of the input XML document in case of `parse_structure` and the result of application of `?XPathExp` to the input document in case of `apply_xpath`. The format of that parse is described in the *XSB Manual, Volume 2: Interfaces and Packages*, in the chapter on *SGML/XML/HTML Parsers and XPath*.

Bibliography

[1] Miguel Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, pages 714–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.