$\mathcal{F}$LORA-2: User's Manual

Guizhen Yang Michael Kifer

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

July 13, 2001

Contents

1 Introduction

FLORA-2 is a sophisticated compiler and application development platform that translates a unified language of F-logic [9], HiLog [5], and Transaction Logic [2] into XSB. It takes a program written in the F-logic language with HiLog and Transaction Logic extensions (which must be in a file with extension `.flr`, *e.g.*, `file.flr`) and outputs a regular XSB program (with extension `.P`). This program is then passed to XSB for compilation (which produces `file.O`) and execution, which, however, requires *FLORA-2* runtime support.

FLORA-2 was implemented by Guizhen Yang, but its origins trace back to the FLIP compiler developed by Bertram Ludäescher. The programming language supported by *FLORA-2* is a dialect of F-logic with numerous extensions. Some extensions are borrowed from FLORID, a C++-based F-logic system developed at Freiburg University.[1] In particular, *FLORA-2* fully supports the versatile syntax of FLORID path expressions. Other important extensions are motivated by the need to support a flexible module system (and enable modular software development in F-logic) and in order to support HiLog [5] and Transaction Logic [3, 1, 2], both of which are smoothly integrated with F-logic. Extensions aside, the syntax of *FLORA-2* also differs in some important ways from FLORID, from the original version of F-logic, as described in [9], and from an earlier implementation of *FLORA*. These syntactic changes were needed in order to bring the syntax of *FLORA-2* closer to that of Prolog and make it possible to include typical Prolog programs into *FLORA-2* programs without choking the compiler. Other syntactic deviations from the original F-logic syntax are a direct consequence of the added support for HiLog, which obviates the need for the "@" sign in method invocations (this sign is now used to denote calls to *FLORA-2* modules).

FLORA-2 is part of the official distribution of XSB beginning with version 2.4. It is organized as an XSB package and lives in the directory

```
<xsb-installation-directory>/packages/flora2/
```

FLORA-2 is fully integrated into the XSB system, including its module system. In particular, *FLORA-2* modules can invoke predicates defined in other XSB modules, and regular XSB modules can query the objects defined in *FLORA-2* modules. At present, XSB is the only platform where *FLORA-2* can run, because it heavily relies on tabling and the well-founded semantics for negation, both of which are available only in XSB.

Due to certain problems with XSB, *FLORA-2* runs best when XSB is configured with SLG-WAM and local scheduling. This problem will be fixed in a future release of XSB. Under the default XSB configuration, certain F-logic programs might run 30 times slower than under the suggested configuration. Some *FLORA-2* programs might also give errors when run under the default configuration. To configure XSB for optimal *FLORA-2* performance, build XSB using the following *two* commands:

```
configure --with-local-scheduling --enable-slg-wam --config-tag=
makexsb
```

[1] See `http://www.informatik.uni-freiburg.de/~dbis/florid/` for more details.

If you are using Unix or Cygwin — you are a lucky person, because $\mathcal{F}$*LORA-2* is built together with XSB. If, for some reason it fails to build, follow these steps:

```
cd <xsb-installation-directory>/packages/flora2/
make clean
make
```

If you are using Windows — do not despair. Building $\mathcal{F}$*LORA-2* is easy, although it takes time. You need to have Microsoft `nmake` or compatible installed.

```
cd <xsb-installation-directory>\packages\flora2
nmake /f NMakefile.mak clean
nmake /f NMakefile.mak
```

The fastest and easiest way to get a feel of the system is to start $\mathcal{F}$*LORA-2* shell and begin to enter queries interactively. To this end, you must first invoke XSB and then load the `flora2` package:

```
foo>  xsb
...  XSB loading messages omitted ...
| ?- [flora2].
[flora2 loaded]
| ?-
```

At this point, it is possible to use a limited number of $\mathcal{F}$*LORA-2* commands, but to run queries you must enter the $\mathcal{F}$*LORA-2* command loop:

```
| ?- flora_shell.
...  FLORA messages omitted ...
flora2 ?-
```

At this point, $\mathcal{F}$*LORA-2* takes over and F-logic syntax becomes the norm. To get back to the XSB command loop, type `Control-D` or

```
| ?- flEnd.
```

If you are using $\mathcal{F}$*LORA-2* shell frequently, it pays to define an alias, say,

```
alias flora2='xsb -e "[flora2], flora_shell."'
```

$\mathcal{F}$*LORA-2* can then be invoked directly from the shell prompt by typing

```
foo>  flora2
```

It is even possible to tell it to execute commands on start-up. For instance,

```
foo>  flora2 -e "flHelp."
```

will cause the system to execute the help command right after the start. Then the usual *FLORA-2* shell prompt is displayed.

FLORA-2 comes with a number of demo programs that live in

```
<xsb-installation-directory>/packages/flora2/demos/
```

The demos can be run by issuing the command "`flDemo(demo-filename).`" at the *FLORA-2* prompt, *e.g.*,

```
flora2 ?- flDemo(flogic_basics).
```

There is no need to change to the demo directory, as `flDemo` knows where to find the demos.

2 *FLORA-2* Shell Commands

The most common shell command you might need to execute is loading and compiling a program:

```
flora2 ?- [programfile].
```

or

```
flora2 ?- flLoad programfile.
```

Here `program-file` can contain a *FLORA-2* program or an XSB program. If `program-file.flr` exists, it is assumed to be a *FLORA-2* program. The system will compile the program, if necessary, and then load it. The compilation process is two-stage: first, the program is compiled into a Prolog program (one or more files with extensions `.P` and `.fdb`) and then into an executable byte-code, which has the extension `.O`.

If there is no `program-file.flr` file, the file is assumed to contain a Prolog program and the system will look for the file named `program-file.P`. This file then is compiled into `program-file.O` and loaded. Note that in this case the program is loaded into a *Prolog module* of *FLORA-2* and, therefore, calls to the predicates defined in that program must use the appropriate module attribution — see Section 5.1 for the details about the module system in *FLORA-2*.

By default, all *FLORA-2* programs are loaded into the module `main`, but you can also load them into other modules using the following syntax:

```
flora2 ?- [file>>modulename].
```

Understanding *FLORA-2* modules is very important in order to be able to take full advantage of the system; we will discuss the module system of *FLORA-2* in Section 5.1. Once the program is loaded, you can pose queries and invoke methods for the objects defined in the program.

There is an important special of the `flLoad` and `[...]` command when the file name is `user`. In that case, instead of looking for the program file `user.flr`, *FLORA-2* starts reading user input. At this point, the user can start typing in program clauses, which the system saves in a temporary file. When the user is done and types `Control-D` (end of file), the file is compiled and loaded. It is also possible to load such a program into a designated module, rather than the default one, using one of the following commands:

```
flora2 ?- [file>>module].
flora2 ?- flLoad file>>module.
```

When the user types in a query to the shell, the query is evaluated and the results are returned. A result is a tuple of values for each variable mentioned in the query, except for the *anonymous variables* represented as "_" and named *don't care variables*, which are preceded with the underscore, *e.g.*, _abc.

By default, *FLORA-2* prints out all answers. If only one is desired, type in the following command: `flOne`. You can revert back to the all-answers mode by typing `flAll`.

FLORA-2 shell includes many more commands beyond those mentioned above. These commands are listed below. However, at this point the purpose of some of these commands might seem a bit cryptic, so it is a good idea to come back here after you become more familiar with the various concepts underlying the system.

In the following command list, the suffixes `.flr` `.P`, `.O` are optional. If the file suffix is specified explicitly, the system uses the file with the given name without any modification. The `.flr` suffix denotes a *FLORA-2* program, the `.P` suffix indicates that it is an XSB program, and `.O` means that it is a bytecode file, which can be executed by XSB. If no suffix is given, the system assumes it is dealing with a *FLORA-2* program and adds the suffix `.flr`. If the file with such a name does not exist, it assumes that the file contains an XSB program and tries the suffix `.P`. Otherwise, it tries `.O` in the hope that an executable XSB bytecode exists. If none of these tries are successful, an error is reported.

- `flHelp`: Show the help info.
- `flCompile file`: Compile FILE.flr for the default module `main`.
- `flCompile file>>module`: Compile FILE.flr for the module `module`.
- `flLoad file>>module`: Load `file.flr` into the module `module`. If you specify `file.P` or `file.O` then will load these files.
- `flLoad file`: Load `file.flr` into the default module `main`. If you specify `file.P` or `file.O` then will load these files.
- `[file.{P|O|flr} >> module,...]`: Load the files in the specified list into the module `module`.
- `flDemo(demofilename)`: Consult a demo from *FLORA-2* demos directory.

- `equality {none|basic|flogic}`: Set the level of support for the equality predicate `:=:` in the shell module `main`. `none` means that `:=:` is treated as a regular predicate; `basic` means that only standard first-order equality is supported (*i.e.*, the usual congruence rules); `flogic` means that F-logic style equality is supported (*i.e.*, congruence plus the axiom for scalar methods).
- `abolish_all_tables`: Flush all tabled data. This is sometimes needed when XSB's tabling gets in the way. We describe tabling (as it pertains to *FLORA-2*) in Section 14.
- `firstorder Functor/Arity`: Define Functor/Arity as non-HiLog in shell mode.
- `arguments Functor({oid|form}, ...)`: Define the predicate meta signature in shell mode.
- `op(Precedence,Associativity,Operator)`: Define an operator in shell mode.
- `flReset({firstorder|arguments|op})`: Clear all dynamic `firstorder/arguments/op` definitions in the *FLORA-2* shell.
- `flAll`: Show all solutions (default).
- `flOne`: Show solutions one by one.
- `flMaxerr(all|N)`: Set/show the maximum number of errors *FLORA-2* reports.
- `flTrace/flNoTrace`: Turn on/off *FLORA-2* trace.
- `flChatter/flNoChatter`: Turn on/off the display of the number of solutions at the end of query evaluation.
- `flEnd`: Say Ciao to *FLORA-2*, stay in XSB.
- `flHalt`: Quit both *FLORA-2* and XSB.

All commands with a FILE argument passed to them use the XSB `library_directory` predicate to search for the file, except that the command `flDemo(FILE)` first looks for `FILE` in the *FLORA-2* demo directory. The search path typically includes the standard system's directories used by XSB followed by the current directory.

All XSB commands can be executed from *FLORA-2* shell, if the corresponding XSB library has already been loaded.

After a syntax, parsing, or compilation error, *FLORA-2* shell will discard tokens read from the current input stream until the end of file or a rule delimiter (".") is encountered. If *FLORA-2* shell seems to be hanging after the message

```
++FLORA Warning: discarding tokens (rule delimiter '.' or EOF expected)
```

hit the `Enter` key once, type ".", and then `Enter` again. This should reset the current input buffer and you should see the *FLORA-2* command prompt:

```
flora2 ?-
```

3 F-logic and $\mathcal{F}$LORA-2 by Example

In the future, this section will contain a number of small introductory examples illustrating the use of F-logic and *$\mathcal{F}$LORA-2*. Meanwhile, the reader is referred to the excellent tutorial written by the members of the FLORID project.[2] Since *$\mathcal{F}$LORA-2* and FLORID share much of the same syntax, most examples in that tutorial can be made into valid *$\mathcal{F}$LORA-2* programs by changing the separator ";" used in F-molecules into "," and by eliminating the "@" sign in method invocations.

4 Basic $\mathcal{F}$LORA-2 Syntax

In this section we describe the basic syntactic structures used to build *$\mathcal{F}$LORA-2* programs. Subsequent sections describe the various advanced features that are needed to build practical applications.

4.1 F-logic Vocabulary

- *Symbols*: The F-logic alphabet of *object constructors* consists of the sets $\mathcal{F}$(function symbols), $\mathcal{P}$(predicate symbols including =), and $\mathcal{V}$(variables). Variables begin with a capitalized letter or an underscore, followed by zero or more letters and/or digits and/or underscores (e.g., `X`, `Name`, `_`, `_v_5`). All other symbols, including the constants (which are 0-ary object constructors), are symbols that start with a lowercase letter (e.g., `a`, `john`). Constants can also be any string of symbols enclosed in single quotes (e.g., `'AB@*c'`). In addition to the usual first-order connectives and symbols, there is a number of special symbols:], [, }, {, ",", ";", #, _#, $\rightarrow$, $\twoheadrightarrow$, $\Rightarrow$, $\Rrightarrow$, `:`, `::`. Later we will explain other symbols introduced by the inheritance mechanism.

- *Variables*: The variable "_" is called *anonymous* variable. It is used whenever a *unique* new variable is needed. In particular, two different occurrences of "_" in the same clause are treated as *different* variables. Named variables that start with an underscore, e.g., `_foo`, are called *don't care* variables. Unlike anonymous variables, two different occurrences of such a variable in the same clause refer to the *same* variable. Nevertheless, don't variables have special status when it comes to error checking and returning answers. The practice of logic programming shows that a singleton occurrence of a variable in a clause is often a mistake due to misspelling. Therefore, *$\mathcal{F}$LORA-2* issues a warning when it finds that some variable is mentioned only once in a clause. If such an occurrence is truly intended, it must be replaced by an anonymous variable or a don't care variable to avoid the warning message from *$\mathcal{F}$LORA-2*. Also, bindings for anonymous and don't care variables are not returned as answers.

- *Id-Terms/Oids*: First-order terms over $\mathcal{F}$ and $\mathcal{V}$ are called *Id-terms*, and are used to name objects, methods, and classes. Ground Id-terms (i.e., terms with no variables) correspond to *logical object identifiers* (*oid*s), also called object *names*. Numbers (including integers and floats) can also be used as Id-terms, but such use might be confusing and is not recommended.

[2] See `http://www.informatik.uni-freiburg.de/~dbis/florid/` for more details.

- *Atomic formulas*: Let O, M, R_i, X_i, C, D, T be Id-terms. In addition to the usual first-order atomic formulas, like $p(X_1, \ldots, X_n)$, there are the following basic types of formulas:

  1. O[M→V]
  2. O[M→→{$V_1, \ldots, V_n$}]
  3. C[M⇒T]
  4. C[M⇒⇒{$T_1, \ldots, T_m$}]

  In all of the above cases, O, C, M, V_i, and T_i are HiLog terms, *i.e.*, expressions of the form, a, f(X), X(s, Y), X(f, Y)(X, g(k)), etc., where X and Y are variables and lowercase letters f, s, etc., are constants.

  Expressions (1) and (2) above are *data atoms*, which specify that a *method expression* M applied to an object O yields the result object V in case (1), or a set of objects, V_1, ..., V_n, in case (2). Thus, in (1), M is said to be a *single-valued* (or *scalar*) method expression, i.e., there is at most one V such that O[M→V] holds. In contrast, in (2), M is *multi-valued* (also known as *set-valued*), so the result contains several objects, which *includes* V_1, V_2, ..., V_n. Note that we emphasized "includes" rather than "equals", because other facts and rules in the program can specify additional objects that must be considered part of the method result.

  When $n = 1$ in set-valued data atoms, the curly braces can be omitted. For instance, O[M→→V_1]. In fact, the single expression (2) is equivalent to a the following set of expressions, where the result set is split into singletons:

  O[M→→V_1]
  O[M→→V_2]
  ...
  O[M→→V_n]

  When M is a constant, *e.g.*, abc, then we say that it is an *attribute*; for example, `john[name→'John']`. When M has the form `f(X,Y,Z)` then we refer to it as a method, f, with arguments X, Y, and Z; for example, `john[salary(1998)→50000]`. However, as we saw earlier, method expressions can be much more general than these two possibilities, as they can be arbitrary HiLog terms.

  Expressions (3) and (4) above denote *signature atoms.* They specify that the method expression, M, when applied to objects that belong to class C, must yield objects that belong to class T. In (3), M is declared as single-valued, while in (4) it is set-valued. In a set-valued signature, the intention is that the method expression M must return objects that belong *simultaneously* to the classes T_1, ..., T_m. As with data atoms, a single set-valued signature expression of the form (4) is equivalent to the set of signature expressions

  O[M⇒⇒T_1]
  O[M⇒⇒T_2]
  ...
  O[M⇒⇒T_m]

and the curly braces { and } can be omitted when only one class appears on the right of $\Rightarrow\!\!\!\!\!\!\Rightarrow$.

Note that it is allowed for the same method to have both a single-valued signature and a set-valued one. The single-valued signature controls the data atoms that use $\rightarrow$, and the set-valued signatures control data atoms that use $\twoheadrightarrow$.

Objects are grouped into classes using *ISA-atoms*:

5. `O:C`
6. `C::D`

The expression (5) states that `O` is an *instance* of class `C`, while (6) states that `C` is a *subclass* of D.

- *F-molecules* provide a convenient way to shortcut specifications related to the same object. For instance, the conjunction of the atoms `john:person`, `john[age→31]`, `john[children↠{bob,mary}]`, and `john[children↠john]` is equivalent to the following single F-molecule:

  ```
  john:person[age→31, children↠{bob,mary,john}]
  ```

  Note the use of the "," that separates the expression for the `age` attribute from the expression for the `children` attribute. This is a departure from the original F-logic syntax in [9], which uses ";" to separate such expressions.

- *Rules* are, as usual, the constructs of the form *head* : −*body*, where *head* is an F-molecule and *body* is a conjunction of F-molecules or negated F-molecules. (Negation is specified using \+ or `tnot` — the difference will be explained later.) Each rule must be terminated with a ".".

  Conjunction is specified as in Prolog, using the "," symbol. Like in Prolog, $\mathcal{F}$*LORA-2* also allows disjunction in the rule body, which is denoted using ";". As usual in logic languages, a single rule of the form

  $$\textit{head} \ :- \ \texttt{john}[\texttt{age}{\rightarrow}\texttt{31}],\ (\texttt{john}[\texttt{children}{\twoheadrightarrow}\{\texttt{bob},\texttt{mary}\}]\ ;\ \texttt{john}[\texttt{children}{\twoheadrightarrow}\texttt{john}]). \quad (1)$$

  is equivalent to the following pair of rules:

  ```
  head :- john[age→31], john[children↠{bob,mary}].
  head :- john[age→31], john[children↠john].
  ```

  Disjunction is also allowed inside F-molecules. For instance, the rule (1) can be equivalently rewritten as:

  ```
  head :- john[age→31, (children↠{bob,mary} ; children↠john)].
  ```

  Note that conjunction "," binds stronger than disjunction ";", so the parentheses in the above example are essential.

- *Programs and queries*: A *program* is a set of rules. A *query* is a rule without the head. In $\mathcal{F}$*LORA-2*, such headless rules use `?-` instead of `:-`, *e.g.*,

```
?- john[age->X].
```

The symbol `:-` in headless $\mathcal{F}LORA$-2 expressions is used for various directives, which are plenty and will be introduced in due course.

Example 4.1 (Publications Database) Figure 1 depicts a fragment of a $\mathcal{F}LORA$-2 program that represents a database of scientific publications.

Schema:
conf_p **::** paper.
journal_p **::** paper.
paper[authors$\Rightarrow\!\!\!>$person, title$\Rightarrow$string].
journal_p[in_vol$\Rightarrow$volume].
conf_p[at_conf$\Rightarrow$conf_proc].
journal_vol[of $\Rightarrow$journal, volume$\Rightarrow$integer, number$\Rightarrow$integer, year$\Rightarrow$integer].
journal[name$\Rightarrow$string, publisher$\Rightarrow$string, editors$\Rightarrow\!\!\!>$person].
conf_proc[of_conf$\Rightarrow$conf_series, year$\Rightarrow$integer, editors$\Rightarrow\!\!\!>$person].
conf_series[name$\Rightarrow$string].
publisher[name$\Rightarrow$string].
person[name$\Rightarrow$string, affil(integer)$\Rightarrow$institution].
institution[name$\Rightarrow$string, address$\Rightarrow$string].
Objects:
o_{j1} **:** journal_p[title$\rightarrow$'Records, Relations, Sets, Entities, and Things', authors$\twoheadrightarrow$\{o_{mes}\}, in_vol$\rightarrow$$o_{i11}$].
o_{di} **:** conf_p[title$\rightarrow$'DIAM II and Levels of Abstraction', authors$\twoheadrightarrow$\{o_{mes}, o_{eba}\}, at_conf$\rightarrow$$o_{v76}$].
o_{i11} **:** journal_vol[of$\rightarrow$$o_{is}$, number$\rightarrow$1, volume$\rightarrow$1, year$\rightarrow$1975].
o_{is} **:** journal[name$\rightarrow$'Information Systems', editors$\twoheadrightarrow$\{o_{mj}\}].
o_{v76} **:** conf_proc[of$\rightarrow$vldb, year$\rightarrow$1976, editors$\twoheadrightarrow$\{o_{pcl}, o_{ejn}\}].
o_{vldb} **:** conf_series[name$\rightarrow$'Very Large Databases'].
o_{mes} **:** person[name$\rightarrow$'Michael E. Senko'].
o_{mj} **:** person[name$\rightarrow$'Matthias Jarke', affil(1976)$\rightarrow$$o_{rwt}$].
o_{rwt} **:** institution[name$\rightarrow$'RWTH_Aachen'].

Figure 1: A Publications Object Base and its Schema in $\mathcal{F}LORA$-2

4.2 Symbols, Strings, and Comments

Symbols. $\mathcal{F}LORA$-2 symbols (that are used for the names of constants, predicates, and object constructors) begin with a lowercase letter followed by zero or more letters (`A...Z`, `a...z`), digits (`0...9`), or underscores (`_`), e.g., `student`, `apple_pie`. Symbols can also be *any* sequence of characters enclosed in a pair of single quotes, e.g., `'JOHN SMITH'`,`'default.flr'`. Internally, $\mathcal{F}LORA$-2 symbols are represented as *XSB symbols*,[3] which are used there as names of predicates and function symbols.

[3] Symbols are called "atoms" in XSB, which contravenes the use of this term for atomic formulas in classical logic and F-logic. We avoid the use of the term "atom" in reference to symbols.

Escaped String	ASCII (decimal)	Symbol
\\	92	\
\n	10	NewLine
\N	10	NewLine
\t	9	Tab
\T	9	Tab
\r	13	Return
\R	13	Return
\v	11	Vertical Tab
\V	11	Vertical Tab
\b	8	Backspace
\B	8	Backspace
\f	12	Form Feed
\F	12	Form Feed
\e	27	Escape
\E	27	Escape
\d	127	Delete
\D	127	Delete
\s	32	Whitespace
\S	32	Whitespace

Table 1: Escaped Character Strings and Their Corresponding Symbols

FLORA-2 also recognizes escaped characters inside single quotes (`'`). An escaped character normally begins with a backslash (\). Table 1 lists the special escaped character strings and their corresponding special symbols. An escaped character may also be any ASCII character. Such a character is preceded with a backslash together with a lowercase `x` (or an uppercase `X`) followed by one or two hexadecimal symbols representing its ASCII value. For example, `\xd` is the ASCII character Carriage Return, whereas `\x3A` represents the semicolon. In other cases, a backslash is recognized as itself.

One exception is that inside a quoted symbol, a single quote character is escaped by another single quote, e.g., `'isn''t'`.

Strings (character lists). Like XSB strings, *FLORA-2* strings are enclosed in a pair of double quotes ("). These strings are represented internally as lists of ASCII characters. For instance, `[102,111,111]` is the same as `"foo"`.

Escape characters are recognized inside *FLORA-2* strings similarly to *FLORA-2* symbols. However, inside a string, a single quote character does not need to be escaped. A double quote character, however, needs to be escaped by another double quote, e.g., `"""foo"""`.

Numbers. Normal *FLORA-2* integers are decimals represented by a sequence of digits, e.g., `892`, `12`. *FLORA-2* also recognizes integers in other bases (2 through 36). The base is specified by a decimal integer followed by a single quote (`'`). The digit string immediately follows the single quote. The letters `A`...`Z` or `a`...`z` are used to represent digits greater than 9. Table 2 lists a few example integers.

Integer	Base (decimal)	Value (decimal)
`1023`	10	1023
`2'1111111111`	2	1023
`8'1777`	8	1023
`16'3FF`	16	1023
`32'vv`	32	1023

Table 2: Representation of Integers

Underscore (_) can be put inside any sequence of digits as delimiters. It is used to partition some long numbers. For instance, `2'11_1111_1111` is the same as `2'1111111111`. However, "_" cannot be the first symbol of an integer, since variables can start with an underscore. For example, 1_2_3 represents the number 123 whereas _12_3 represents a variable named _12_3.

Floating numbers normally look like `24.38`. The decimal point must be preceded by an integral part, even if it is 0, e.g., `0.3` must be entered as `0.3`, but not as `.3`. Each floating number may also have an optional exponent. It begins with a lowercase `e` or an uppercase `E` followed by an optional minus sign (`-`) or plus sign (`+`) and an integer. This exponent is recognized as in base 10. For example, `2.43E2 is 243` whereas `2.43e-2 is 0.0243`.

Comments. *FLORA-2* supports three kinds of comments: (1) all characters following `%` until the end of the line; (2) all characters following `//` until the end of the line; (3) all characters inside a pair of `/*` and `*/`. Note that only (3) can span multiple lines.

Comments are recognized like whitespaces by the compiler. Therefore, tokens can also be delimited by comments.

4.3 Operators

As in Prolog, *FLORA-2* allows the user to define operators, to liven up the otherwise boring syntax. There are three kinds of operators: infix, prefix, and postfix. An infix operator appears between its two arguments, while a prefix operator before its single argument and a postfix operator after its single argument. For instance, if `foo` is defined as an infix operator, then `X foo a` will be parsed as `foo(X,a)` and if `bar` is a postfix operator then `X bar` is parsed as `bar(X)`.

Each operator has a *precedence level*, which is a positive integer. Each operator also has a *type*. The possible types for infix operators are: `xfx`, `xfy`, `yfx`; the possible types for prefix operators are: `fx`, `fy`; and the possible types for postfix operators are: `xf`, `yf`. In each of these type expressions, `f` stands for the operator, and `x` and `y` stand for the arguments. The symbol `x` in a type expression means that the precedence level of the corresponding argument should be *strictly less* than that of the operator, while `y` means that the precedence level of the corresponding argument should be *less or equal* than that of the operator.

The precedence level and the type together determine the way the operators are parsed. The general rule is that precedence of a constant or a functor symbol that has not been defined as an operator is zero. Precedence of a Prolog term is the same as the precedence of its main functor. An expression that contains several operators is parsed in such a way that the operator with the highest precedence level becomes the main functor of the parsed term, the operator with the next-highest precedence level becomes the main functor of one of the arguments, and so on. If an expression cannot be parsed according to this rule, a parse error is reported.

It is not our goal to cover the use of operators in any detail, since this information can be found in any book on Prolog. Here we just give an example that illustrates the main points. For example, in *FLORA-2*, `-` has precedence level `800` and type `yfx`, `*` has precedence level `700` and type `yfx`, `->` has precedence level 1100 and type `xfx`. Therefore, `8-2-3*4` is the same as `-(-(8,2),*(3,4))` in prefix notation, and `a -> b -> c` will generate a parsing error.

Any symbol can be defined as an operator. The general syntax is

```
:- op(Precedence,Type,Name).
```

For instance,

```
:- op(800, xfx, foo)
```

As a notational convenience, the argument `Name` can also be a list of operator names of the same type and precedence level, for instance,

```
:- op(800,yfx,[+,-]).
```

It is possible to have more than one operator with the same name provided they have different use (*e.g.*, one infix and the other postfix). However, the *FLORA-2* built-in operators are not allowed to be redefined. In particular, any symbol that is part of F-logic syntax, such as ",', ".", "[", ":", etc., as well as any name that begins with `flora` or `fl` followed by a capital letter should be considered as reserved for internal use.

Although this simple rule is sufficient, in most cases, to keep you out of trouble, you should be aware of the fact that symbols such as "`,`", "`;`", "`+`", "`.`", "`->`", "`::`", and many other parts of *FLORA-2* syntax are operators. Therefore, there is a chance that precedence levels chosen for the user-defined operators conflict with those of *FLORA-2* and, as a result, your program might not parse. If in doubt, check the declarations in the file `flroperator.P` in the *FLORA-2* source code.

4.4 Logical Expressions

In a *FLORA-2* program, any combination of conjunction, disjunction, and negation of literals can appear wherever a logical formula is allowed, e.g., in a rule body.

Conjunction is represented through the infix operator "`,`" and disjunction is made using the infix operator "`;`". Negation is made through the prefix operators "\+" and "`tnot`".[4] When parentheses are omitted, conjunction binds stronger than disjunction and the negation operators

[4] In brief, "\+" represents negation as failure and can be applied only to non-tabled Prolog, *FLORA-2*, or HiLog predicates. "`tnot`", on the other hand, is negation that implements the well-founded semantics. Refer to Section 16 for more information on the difference between negation operators.

bind their arguments stronger than the other logical operators. For example, in *$\mathcal{F}$LORA-2* the following expression: `a, b; c, not d`, is equivalent to the the logical formula: $(\mathtt{a} \wedge \mathtt{b}) \vee (\mathtt{c} \wedge (\neg \mathtt{d}))$.

Logical formulas can also appear inside the specification of an object. For instance, the following F-molecule:

```
o[tnot att1->val1, att2->>val2; meth->res]
```

is equivalent to the following formula:

```
(tnot o[att1->val1], o[att2->>val2]) ; o[meth->res]
```

4.5 Arithmetic Expressions

In *$\mathcal{F}$LORA-2* arithmetic expressions are *not* always evaluated. As in Prolog, the arithmetic operators such as `+`, `-`, `/`, and `*`, are defined as normal binary functors. To evaluate an arithmetic expression, *$\mathcal{F}$LORA-2* provides another operator, `is`. For example, `X is 3+4` will bind `X` to the value 7.

When dealing with arithmetic expressions, the order of literals is important. In particular, all variables appearing in an arithmetic expression must be instantiated at the time of evaluation. Otherwise, a runtime error will occur. For instance,

```
?- X > 1, X is 1+1.
```

will produce an error, while

```
?- X is 1+1, X > 1.
```

will evaluate to true.

As in Prolog, the operands of an arithmetic expression can be any variable or a constant. However, in *$\mathcal{F}$LORA-2*, an operand can also be a *path expression*. For the purpose of this discussion, a path expression of the form p.q should be understood as a shortcut for `p[q->X]`, where `X` is a new variable, and p.q.r is a shortcut for `p[q->X], X[r->Y]`. For set-valued formulas, the notation ".." is used. For instance, `p..q` stands for `p[q->>X]`. More detailed discussion of path expressions appears in Section 6.

Both single-valued and multi-valued path expressions are allowed in arithmetic expressions, and all variables are considered to be existentially quantified. For example, the following query

```
?- john..bonus + mary..bonus > 1000.
```

should be understood as

```
?- john[bonus->>_V1], mary[bonus->>_V2], _V1 + _V2 > 1000.
```

Precedence	Operator	Use	Associativity	Arity
not applicable	()	parentheses	not applicable	not applicable
not applicable	.	decimal point	not applicable	not applicable
300	.	single-valued object reference	left	binary
	..	multi-valued object reference	left	binary
	:	ISA specification	left	binary
	::	subclass specification	left	binary
600	-	minus sign	right	unary
	+	plus sign	right	unary
700	*	multiplication	left	binary
	/	division	left	binary
800	-	subtraction	left	binary
	+	addition	left	binary
1000	=<	less than or equals to	not applicable	binary
	>=	greater than or equals to	not applicable	binary
	=:=	equals to	not applicable	binary
	=\=	unequal to	not applicable	binary
	is	assignment	not applicable	binary

Table 3: Operators in Non-Increasing Precedence Order and Their Associativity and Arity

Note that in first query does not have any variables, so after the evaluation the system would print either yes or no. To achieve the same behavior, we use *don't care variables*, `_V1` and `_V2`. If we used `V1` and `V2` instead, the values of these variables would have been printed out.

FLORA-2 recognizes numbers as oids and, thus, it is perfectly normal to have allows arithmetic expressions inside path expressions such as this: `1.2.(3+4*2).7`. When parentheses are omitted, this might lead to ambiguity. For instance, is the meaning of

```
1.m+2.n.k
```

represented by the arithmetic expression `(1.m)+(2.n.k)`, or by the path expressions `(1.m+2.n).k`, by `(1.m + 2).n.k`, or by `1.(m+2).n.k`? To disambiguate such expressions, we must remember that the operators "." and ".." used in path expressions bind stronger than the arithmetic operators $+$, $-$, etc.

Even more interesting is the following example: `2.3.4`. Does it represent the path expression `(2).(3).(4)`, or `(2.3).4`, or `2.(3.4)` (where in the latter two cases 2.3 and 3.4 are interpreted as decimal numbers)? The answer to this puzzle (according to *FLORA-2* conventions) is `(2.3).4`: when tokenizing, *FLORA-2* first tries to classify tokens into meaningful categories. Thus, when 2.3 is first found, it is identified as a decimal. Thus, the parser receives the expression (2.3).4, which it identifies as a path expression that consists of two components, the oids 2.3 and 4.

Another ambiguous situation arises when the symbols `-` and `+` are used as minus and plus signs, respectively. *FLORA-2* follows the common arithmetic interpretation of such expressions, where the `+/-` signs bind stronger than the infix operators and thus `4--7` and `4-+7` are interpreted as `4-(-7)` and `4-(+7)`, respectively.

Table 3 lists various operators in decreasing precedence order, their associativity, and arity.

When in doubt, use parentheses. Here are some more examples of valid arithmetic expressions:

`o1.m1+o2.m2.m3`	same as `(o1.m1)+(o2.m2)`
`2.(3.4)`	the value of the attribute `3.4` on object `2`
`3 + - - 2`	same as `3+(-(-2))`
`5 * - 6`	same as `5*(-6)`
`5.(-6)`	the value of the attribute `-6` on object `5`

Note that the parentheses in `5.(-6)` are needed, because otherwise "`.-`" would be recognized as a single token. Similarly, the whitespace around "+", "–", and "*" are also needed in these examples to avoid `*-` and `+--` being interpreted as distinct token.

5 Multifile Programs

$\mathcal{F}$*LORA-2* supports many ways in which a program can be modularized. First, an F-logic program can be split into many files with separate namespaces. Each such file can be considered an independent library, and the different libraries can call each other. In particular, the same method name (or a predicate) can be used in different files and the definitions will not clash. Second, a program file can be split of several files, and these files can be included by the preprocessor prior to the compilation. In this case, all files share the same namespace in the sense that the different rules that define the same method name (or a predicate) in different files are assumed to be part of one definition. Third, $\mathcal{F}$*LORA-2* programs can call XSB modules and vice versa. In this way, a large system can be built partly in Prolog and partly in $\mathcal{F}$*LORA-2*.

We discuss each of the aforesaid modularization methods in turn.

5.1 $\mathcal{F}$*LORA-2* Modules

A $\mathcal{F}$*LORA-2 module* is a programming abstraction that allows a large program to be split into separate libraries that can be reused in multiple ways in the same program. Formally, a module is a pair that consists of a *name* and a *contents*. The name is just an alphanumeric symbol (the underscore, _, is also allowed), and the contents consists of the program code that is typically loaded from some file (but it can also be constructed dynamically by inserting facts[5] into another module).

The basic idea behind $\mathcal{F}$*LORA-2* modularization is that reusable code libraries are to be placed in separate files. To use a library, it must be *loaded into a module*. Other parts of the program can then invoke this library's methods by providing the name of the module (and the method/predicate names, of course). There is no need to export anything from a library — any public method or predicate can be called by other parts of the program.[6] In this way, the library loaded into a module becomes that module's content.

Note that there is no a priori association between files and modules. Any file can be loaded into any module and one program file can even be loaded into two different modules at the same

[5] **Unimplemented:** Dynamic insertion of rules will be implemented in the future.
[6] **Unimplemented:** At present, all methods are public, but encapsulation will be implemented in the future.

time. The same module can be reused during the same program run by loading another file into that module. In this case, the old contents is erased and the module gets new contents from the second file.

In $\mathcal{F}$*LORA-2*, modules are completely decoupled from file names. A $\mathcal{F}$*LORA-2* program knows only the module names it needs to call, but not the file names. Specific files can be loaded into modules by another, unrelated bootstrapping program. Moreover, a program can be written in such a way that it calls a method of some module without knowing that module's name. The name of the module can be passed as a parameter or in some other way and the concrete binding of the method to the module will be done at runtime.

This dynamic nature of $\mathcal{F}$*LORA-2* modules stands in sharp contrast to the module system of XSB, which is static and associates modules with files at compile time. Moreover, to call a predicate from another module, that predicate must be imported explicitly and referred to by the same name.

As a pragmatic measure, $\mathcal{F}$*LORA-2* defines *three kinds of modules* rather than just one. The kind described above is actually just one of the three: the *user module*. As explained, these modules are decoupled from the actual code, and so they can contain different code at different times. The next kind is a *Prolog module*. This is an abstraction in $\mathcal{F}$*LORA-2*, which is used to call XSB predicates. Prolog modules are static and are assumed to be closely associated with their code. We describe these modules in Section 5.5. (Do not confuse $\mathcal{F}$*LORA-2* Prolog modules — an abstraction used in the language of $\mathcal{F}$*LORA-2*— with XSB modules, which is an abstraction used in XSB.) The third type of modules are the $\mathcal{F}$*LORA-2* `system modules`. These modules are preloaded with $\mathcal{F}$*LORA-2* programs that provide useful methods and predicates (*e.g.*, I/O) and, thus, are also static. These modules are described in Section 5.7 and 20. The abstraction of system modules is a convenience provided by $\mathcal{F}$*LORA-2*, which enables user programs to perform common actions using standard names of predicates and methods implemented in those modules. The syntactic conventions for calling each of these types of modules are similar, but distinct.

5.2 Calling Methods and Predicates Defined in User Modules

If *literal* is an F-molecule or a predicate defined in another user module, it can be called using the following syntax:

literal @ *module*

The name of the module can be any alphanumeric symbol.[7] For instance, `foo(a) @ foomod` tests whether `foo(a)` is true in the user module named `foomod`, and `mary[children->>X]@genealogy` queries the information on `mary`'s children available in the module `genealogy`. More interestingly, the module specifier can be a variable that gets bound to a module name at run time. For instance,

```
..., Agent=zagat, ..., newyork[dinner(italian) ->>X]@Agent.
```

A call to a literal with an unbound module specification or one that is not bound to a symbol will result in a runtime error.

[7] In fact, any symbol is allowed. However, it cannot contain the quote symbol, "'".

When calling the literals defined in the same module, the @*module* notation is not needed, of course. (In fact, since a program does not know where it will be loaded, using the @-notation to call a literal in the same module is hard. However, it is possible with the help of the special predicate `flThisModule/1`, described later, and is left as an exercise.)

The following rules apply when calling a literal defined in another module:

1. Literal reference cannot appear in a rule head or be specified as a fact. For example, the following program will generate a parsing error

   ```
   john[father->smith] @ foomod.
   foo(X) @ foomod :- goo(X).
   ```

   because defining a literal that belongs to another module does not make sense.

2. Module specification is distributive over logical connectives, including the conjunction operator, "`,`", the disjunction, "`;`", and the negation operators, "\+" and "`tnot`". For example, the formula below:

   ```
   (john[father->smith], tnot smith[spouse->mary]) @ foomod
   ```

   is equivalent to the following formula:

   ```
   john[father->smith] @ foomod, tnot (smith[spouse->mary] @ foomod)
   ```

3. Module specifications can be nested. The one closest to a literal takes effect. For example,

   ```
   (foo(a), goo(b) @ goomod, hoo(c)) @ foomod
   ```

   is equivalent to

   ```
   foo(a) @ foomod, goo(b) @ goomod, hoo(c) @ foomod
   ```

4. The module specification propagates to any F-molecule appearing in the argument of a predicate for which the module is specified. For example,

   ```
   foo(a.b[c->d]) @ foomod
   ```

   is equivalent to

   ```
   a[b->X] @ foomod, X[c->d] @ foomod, foo(X) @ foomod
   ```

5. Module specifications do not affect function terms that are not predicates or method names, unless such a specification is explicitly attached to such a term. For instance, in

   ```
   ?- foo(goo(a)) @ foomod.
   ```

   `goo/1` refers to the same functor both in module `foomod` and in the calling module. However, if the module is attached explicitly, as in

   ```
   ?- foo(goo(a) @ goomod) @ foomod.
   ```

   then `foo/1` is assumed to be a meta-predicate that receives the predicate `goo/1` as a parameter.

5.3 Finding the Current Module Name

Since a *FLORA-2* program can be loaded into any module, the program does not have a priori knowledge of the module it will be executing in. However, the program can determine its module at runtime using the special predicate `flThisModule/1`. This predicate accepts a variable argument and binds it to the name of the *FLORA-2* module into which the program was loaded.

5.4 Loading Files into User Modules

FLORA-2 provides commands for compiling and loading program files into specified user modules. The command

```
?- flCompile(file>>module).
```

generates the byte code for the program to be loaded into the user module named `module`. The name of the byte code for the program in *file*.flr, which can later be loaded into the specified module. In practice this means that the compiler generates files named *file_module.P* and *file_module.O* with symbols appropriately renamed to avoid clashes.

If no module is specified, the command

```
?- flCompile(file).
```

compiles *file*.flr for the default module `main`.

To load a file, the following command can be used:

```
[myprog].
```

This loads the program in the file `myprog.flr` into the default user module `main`. If `myprog.flr` is newer than the compiled code, the source file is recompiled.

An optional module name can be used to load the program into a specified module:

```
[myprog >> foomod]
```

This loads the *FLORA-2* program `myprog.flr` into the user module named `foomod`, compiling it if necessary.

Like XSB, *FLORA-2* also the user to compile and load several program files at the same time: If the file was not compiled before (or if the program file is newer), the program is compiled before being loaded. For instance, the following command:

```
[myprog1, myprog2]
```

will loads both `myprog1` and `myprog2` into the default module `main`. However, loading several programs into the same module is not very useful: the code of the last program will wipe out the code of the previous ones. This is a general rule in *FLORA-2* (and XSB). Thus, loading multiple files is always used in conjunction with the module targets:

```
['myprog1.flr', myprog2 >> foomod].
```

which loads `myprog1.flr` into the module `main` and `myprog2.flr` into the module `foomod`.

Note that the `[...]` command can also load and compile XSB programs. The overall algorithm is as follows. If the file suffix is specified explicitly, the corresponding file is assumed to be a *FLORA-2* file, an XSB file, or a byte code depending on the suffix: `.flr`, `.P`, or `.O`. If the suffix is not given explicitly, the compiler first checks if *file*`.flr` exists. If so, the file assumed to be a *FLORA-2* program and is compiled as such. If *file*`.flr` is not found, but *file*`.P` or *file*`.O` is, the file is passed to XSB for compilation.

Sometimes it is useful to know which user modules are loaded or if a particular user module is loaded (say, because your might want to load it, if not). To find out which modules are loaded at the present time. Use the predicate `flLoadedModule/1` for that. For instance, the first query, below, succeeds if the module `foo` is loaded. The second query succeeds and binds `L` to the list of all user modules that are loaded at the present time.

```
?- flLoadedModule(foo).
?- findall(X,flLoadedModule(X),L).
```

5.5 Calling XSB from *FLORA-2*

XSB predicates can be called from *FLORA-2* through the *FLORA-2* module system *FLORA-2* models XSB programs as collections of static *Prolog modules*, *i.e.*, from *FLORA-2*'s point of view, XSB modules are always available and do not need to be loaded explicitly because the association between XSB programs and modules is fixed. The syntax to call XSB predicates is

?- *predicate*`@prolog(`*module*`)`

For instance, since the predicate `member/2` is defined in the XSB module `basics`, we can call it as follows:

```
?- member(abc,[cde,abc,pqr])@prolog(basics).
```

To use this mechanism, you must know which XSB module the particular predicate is defined in. Some predicates are defined by programs that do not belong to any module. When such an XSB program is loaded, the corresponding predicates become available in the special XSB module called `usermod` and *FLORA-2* can call such predicates as follows:

```
?- foo(X)@prolog(usermod).
```

Note that variables are not allowed in the module specifications of XSB predicates, *i.e.*,

```
?- M=usermod, foo(X)@prolog(M).
```

will cause a compilation error.

Some XSB predicates are considered "well-known" and, even though they are defined in various obscure XSB modules, the user can just use those predicates without remembering the corresponding XSB module names. These predicates (that are listed in the XSB manual) can be called from *FLORA-2* with particular ease:

```
?- writeln('Hello')@prolog()
```

i.e., we can simply omit the XSB module name (but parentheses must be preserved).

5.6 Calling *FLORA-2* from XSB

Since XSB does not understand object-oriented syntax, it can call only predicates defined in *FLORA-2* programs. These predicates can be HiLog or first-order predicates (See Section 8). To call predicates defined in *FLORA-2* programs, they must be imported by the XSB program. To do this, the XSB query

?- bootstrap_flora.

must be executed first. To call *FLORA-2* predicates from the XSB shell, the following `flImport` queries must be executed in the shell:

```
?- flImport {hilog|firstorder} flora-predicate/arity as xsb-name(_,_,...,_)
                              from filename >> flora-module-name
?- flImport {hilog|firstorder} flora-predicate/arity as xsb-name(_,_,...,_)
                              from flora-module-name
```

To call *FLORA-2* from within an XSB program, say `test.P`, these `flImport` queries must appear near the top of the program (prior to any call to *FLORA-2* predicates). Also, the query `bootstrap_flora` must be executed *before compiling and loading* `test.P` — otherwise, the program will not compile.

The first syntax for `flImport` above is used to both import the predicate and also load the program file defining it into a given *FLORA-2* user module. The second syntax is used when the flora program is already loaded into a module and we only need to import the corresponding predicate.

In both cases, the option `hilog` is used to import a HiLog predicate and `firstorder` is used to import first-order predicates. The imported predicate must be given a name by which the imported predicate will be known in XSB. (This name can be the same as the name used in *FLORA-2*.) It is important, however, that the XSB name be specified as shown, *i.e.*, as a predicate skeleton with the same number of arguments as in the first-order predicate. For instance, `foo(_,_,_)` will do, but `foo/3` will not.

Once the predicate is imported, it can be used under its XSB name as a regular predicate.

XSB programs can also load and compile *FLORA-2* programs using the following queries (again, bootstrap_ `flora` must be executed in advance):

```
:- import flLoad/1, flCompile/1 from flora2.
?- flLoad flora-file >> flora-module.
?- flLoad flora-file
?- flCompile flora-file >> flora-module.
?- flCompile flora-file
```

The first query loads the file *flora-file* into the given user module and compiles it, if necessary. The second query loads the program into the default module `main`. The last two queries compile the file for loading into the module *flora-module* and `main`, respectively, but do not load it.

Finally, an XSB program can check if a certain $\mathcal{F}$*LORA-2* user module has been loaded using the following call:

```
:- import flLoadedModule/1 from flora2.
?- flLoadedModule(flora-module-name).
```

5.7 $\mathcal{F}$*LORA-2* System Modules

$\mathcal{F}$*LORA-2* provides a special set of modules that are *preloaded* with useful utilities, such as prettyprinting or I/O. These modules have special syntax, `@flora(modname)`, and cannot be loaded by the user. For this reason, these modules are called $\mathcal{F}$*LORA-2 system modules*. For instance, to prettyprint all the attributes and methods of an object, the following method, defined in the system module `pp`, can be used:

```
?- obj[#pp_self(foo)]@flora(pp).
```

Here, the method `#pp_self` is applied to the object `obj`. Since `obj` can have completely different methods in different user modules, we have to tell the prettyprinting method in which module to look. For more details on the existing $\mathcal{F}$*LORA-2* system modules, see Section 20.

5.8 Including Files into $\mathcal{F}$*LORA-2* Programs

The last and the simplest way to construct multi-file $\mathcal{F}$*LORA-2* programs is by using the `#include` preprocessing directive. For instance if file `foo.flr` contains the following instructions:

```
#include file1
#include file2
#include file3
```

the effect is the same as if the above three files were concatenated together and stored in `foo.flr`.

5.9 More on Variables as Module Specifications

Earlier we mentioned that a user module specification can be a variable, *e.g.*, `a[m->b]@X`, which ranges over module names. This variable must be bound to a concrete module name before the call is made. However, this binding cannot be `prolog()`, `prolog(module)`, or `flora(module)`.

Dynamic module bindings can be used to implement *adaptive methods*, which are used in many types of applications, *e.g.*, agent programming. Consider the following example:

`Module foo`	`Module moo`
`something :- ...`	`......`
`something_else :- flThisModule(M),`	`......`
`        a[someservice(M,Arg)->Res]@moo`	`......`
`......`	`a[someservice(Module,Arg)-> Res] :-`
`......`	`        something@Module, ...`
`......`	`......`

Here the method `someservice` in user module `moo` performs different operation depending on who is calling it, because `something` can be defined differently for different callers. When `something_else` is called, it invokes the method `someservice` on object `a` in module `moo`. The current module name (`foo`) is passes as a parameter. When `someservice` is executed in module `moo` it calls the predicate `something` in module `foo`.

This schema is, for instance, how the pretty printing system module of $\mathcal{F}$*LORA-2* works. A pretty-printing method is called on an object in some user module, and to do its job the pretty-printing method needs to query the object *in the context of the calling module* to find the methods the object has.

It is also possible to view adaptive methods as a declarative counterpart of the callback functions in C/C++, which allows the callee to behave differently for different clients.

6 Path Expressions

6.1 Path Expressions in the Rule Body

In addition to the basic F-logic syntax, the $\mathcal{F}$*LORA-2* system also supports *path expressions* to simplify object navigation along single-valued and multi-valued method applications, and to avoid explicit join conditions [7]. The basic idea is to allow the following *path expressions* wherever Id-terms are allowed:

7. `O.M`
8. `O..M`

The path expression in (7) is *single-valued*; it refers to the unique object $\mathtt{R_0}$ for which $\mathtt{O[M \rightarrow R_0]}$ holds; (8) is a *multi-valued* path expression; it refers to each R_i for which $\mathtt{O[M \twoheadrightarrow \{R_i\}]}$ holds. The symbols `O` and `M` stand for an Id-term or path a expression. Moreover, `M` can be a method that takes arguments, in which case $\mathtt{O.M(P_1, \ldots, P_k)}$ and $\mathtt{O..M(P_1, \ldots, P_k)}$ are a valid path expressions.

In order to disambiguate the syntax and to specify the desired order of method applications, parentheses can be used. By default, path expressions associate to the left, so `a.b.c` is equivalent to (a.b).c, which specifies the object o such that a[b→x] ∧ x[c→o] holds (note that x = a.b). In contrast, a.(b.c) is the object o1 such that $b[c \rightarrow x1] \wedge a[x1 \rightarrow o1]$ holds (note that in this case, x1 = b.c). In general, o and $o1$ can be different objects. Note also that in (a.b).c, b is a method name, whereas in a.(b.c) it is used as an object name and `b.c` as a method. Observe that function symbols can also be applied to path expressions, since path expressions, like Id-terms, represent objects. Thus, `f(a.b)` is a valid expression.

As path expressions and F-molecules can be arbitrarily nested, this leads to a concise and flexible specification language for object properties, as illustrated in the following example.

Example 6.1 (Path Expressions) Consider again the schema given in Figure 1. If n is the name of a person, the following path expression is a query that returns all editors of conferences in which n had a paper:

```
?- P:conf_p[authors→→{_[name→n]}].at_conf..editors
```

Likewise, the answer to the query

```
?- P:conf_p[authors→→{_[name→n]}].at_conf[editors→→{E}].
```

is the set of all pairs (P,E) such that P is (the logical oid of) a paper written by n, and E is the corresponding proceedings editor. If we also want to see the affiliations of the above editors, we only need to modify our query slightly:

```
?- P:conf_p[authors→→{_[name→n]}].at_conf[year→Y]..editors[affil(Y)→A].
```

Thus, $\mathcal{F}LORA$-*2* path expressions support navigation along the method application dimension using the operators "." and "..". In addition, intermediate objects through which such navigation takes place can be selected by specifying the properties of such objects inside square brackets.[8]

To access intermediate objects that arise implicitly in the middle of a path expression, one can define the method self as

```
X[self→X].
```

and then simply write ...`[self→O]`... anywhere in a complex path expression. This would bind the Id of the current object to the variable O.

Example 6.2 (Path Expressions with self) To illustrate convenience afforded by the use of the `self` attribute in path expressions, consider the second query in Example 6.1. If, in addition, we want to obtain the names of the conferences where the respective papers were published, that query can be reformulated as follows:

```
X[self→X].
?- P:conf_p[authors→→_[name→n]].at_conf[self→C, year→Y]..editors[affil(Y)→A].
```

[8] A similar feature is used in other languages, e.g., XSQL [8].

6.2 Path Expressions in the Rule Head

Only single-valued path expressions are allowed in a rule head. Set-valued path expressions are not allowed because the semantics is not always clear in such cases.

The following is an example of a path expression in rule head. For any person X, this rule defines the grandchildren for X's mother.

```
X.mother[grandson->>Y] :- X:person[son->>Y].
```

Here X's mother is treated as an unknown object. Should this object become known in the future, complications may arise. For instance, suppose that later on the following facts were added or derived:

```
john[mother->mary].
john[son->>david].
```

Because John's mother is now identified as `mary`, it follows that `mary` and `john.mother` represent the same things, since the attribute `mother` is scalar. To deal with single-valued path expressions in rule heads, *FLORA-2 skolemizes* `john.mother` and adds the requisite equalities to record that the Skolem term that corresponds to `john.mother` equals `mary`. All this is done by the *FLORA-2* compiler transparently to the user: if a path expression in rule head is detected, *FLORA-2* replaces this expression with a Skolem function. The problem here is that maintenance of such equalities is costly (sometimes causing a slowdown by the factor of 2–10 times). Since the user often knows that equations of the above kind will never be derived, *FLORA-2* leaves it to the user to tell the compiler whether equality maintenance is needed. Equality maintenance is discussed in Section 9.

7 Truth Values, Object Values, and Predicate Meta Signatures

Id-terms, F-logic atoms, and path expressions can all be used as objects. This is obvious for Id-terms and the object interpretation of path expressions of the form (7) and (8) on page 22 was discussed earlier. The F-logic atoms (1) through (6) on pages 7 through 8 are typically viewed as formulas and, thus, they are assumed to have a truth value only. However, there also is a natural way to give them object interpretation. For example, `o:c[m->r]` has object value `o` and some truth value. However, unlike the object value, the truth value depends on the database (on whether `o` belongs to class `c` in the database and whether the value of the attribute `m` is, indeed, `r`.

Although previously we discussed only the object interpretation for path expressions, it is easy to see that they have truth values as well, because a path expression corresponds to a conjunction of F-logic atoms. Consequently, all F-molecules of the form (1) through (8) have dual reading: As logical formulas (*the deductive perspective*), and as expressions that represent one or more objects (*the object-oriented perspective*). Given an intended model, $\mathcal{I}$, of an F-logic program an expression has:

- An *object value*, which yields the Id(s) of the object(s) that are reachable in $\mathcal{I}$ by the corresponding expression, and

- A *truth value*, like any other literal or molecule of the language.

An important property that relates the above interpretations is: a molecule, r, evaluates to *false* if $\mathcal{I}$ has no object corresponding to r.

Consider the following path expression and an equivalent, decomposed expression:

$$\mathtt{a..b[c}{\rightarrow\!\!\!\rightarrow}\{\mathtt{d.e}\}] \quad \Leftrightarrow \quad \mathtt{a[b}{\rightarrow\!\!\!\rightarrow}\mathtt{X_{ab}}] \wedge \mathtt{d[e}{\rightarrow}\mathtt{X_{de}}] \wedge \mathtt{X_{ab}[c}{\rightarrow\!\!\!\rightarrow}\mathtt{X_{de}}]. \tag{2}$$

Such decomposition is used to determine the truth value of arbitrarily complex path expressions in the *body* of a rule. Let *obj*(`path`) denote the Ids of all objects represented by the path expression. Then, for (2) above, we have:

$$obj(\mathtt{a..b}) = \{\mathtt{x_{ab}} \mid \mathcal{I} \models \mathtt{a[b}{\rightarrow\!\!\!\rightarrow}\mathtt{x_{ab}}]\} \qquad \text{and} \qquad obj(\mathtt{d.e}) = \{\mathtt{x_{de}} \mid \mathcal{I} \models \mathtt{d[e}{\rightarrow}\mathtt{x_{de}}]\}$$

where $\mathcal{I} \models \varphi$ means that φ holds in $\mathcal{I}$. Observe two formulas can be equivalent, but their object values might be different. For instance, $\mathtt{d[e}{\rightarrow}\mathtt{f]}$ is equivalent to `d.e` as a formula. However, *obj*(d.e) is f, while *obj*($\mathtt{d[e}{\rightarrow}\mathtt{f]}$) is d.

In general, for an F-logic database $\mathcal{I}$, the object values of ground path expressions are given by the following mapping, *obj*, from ground molecules to sets of ground oids (t, o, c, d, m can be oids or path expressions):

$$\begin{array}{rcl}
obj(t) & := & \{t \mid \mathcal{I} \models t[]\}, \text{ for a ground Id-term } t \\
obj(o[\ldots]) & := & \{o1 \mid o1 \in obj(o), \mathcal{I} \models o1[\ldots]\} \\
obj(o:c) & := & \{o1 \mid o1 \in obj(o), \mathcal{I} \models o1:c\} \\
obj(c::d) & := & \{c1 \mid c1 \in obj(c), \mathcal{I} \models c1::d\} \\
obj(o.m) & := & \{r1 \mid r1 \in obj(r), \mathcal{I} \models o[m{\rightarrow}r]\} \\
obj(o..m) & := & \{r1 \mid r1 \in obj(r), \mathcal{I} \models o[m{\rightarrow\!\!\!\rightarrow}\{r\}]\}
\end{array}$$

Observe that if `t[]` does not occur in $\mathcal{I}$, then *obj*(t) is $\emptyset$. Conversely, a ground molecule r is called *active* if *obj*(r) is not empty. A molecule, r, can be single-valued or multi-valued:

- r is called *multi-valued* if
  - it has the form $o..m$, or
  - it has one of the forms $\underline{o}[\ldots]$, $\underline{o}:c$, $\underline{c}::d$, or $\underline{o}.\underline{m}$, and any of the underlined subexpressions is multi-valued;
- in all other cases, r is *single-valued*.

Dual representation and meta-predicates. Since path expressions can appear wherever Id-terms are allowed, the question arises whether a path expression is intended to indicate a truth value or an object value. For instance, we may want to call the XSB aggregate predicate `findall/3` to retrieve the oids of the managers who like to play tennis:

```
findall(P,P:manager[hobbies->>tennis],L)
```

If all arguments are treated as objects, then this expression would mean

```
P:manager[hobbies->>tennis], findall(P,P,L)
```

and a meaningless result — the list of all instances of P that evaluate to true — will be returned. Since, due to the first F-molecule, these instances of P must be the oids of the tennis-playing managers, such as john, the list L would contain those oids that happen to be true as 0-ary facts. (For example, john would end up on such a list if the fact john is true as a 0-ary predicate).

The upshot of this problem is that the interpretation of F-logic expressions as objects is not always suitable. In our example, we need to indicate to the compiler that the middle argument of findall/3 ought to be translated into XSB as follows:

```
findall(P,F(P),L)
```

where F(P) is an formula into which P:manager[hobbies->>tennis] is translated by the *FLORA-2* compiler.[9] Under this translation, findall/3 will find all instances of P that make F(P) true and return a list of such instances in the list L. This feat is accomplished using the arguments compiler instruction.

By default, when a truth-valued expression, E, such as an F-logic molecule or a path expression, appears as an argument to a predicate or function symbol, p, the *FLORA-2* compiler will pass the object value of the expression to p. In order to tell the compiler that the entire formula obtained by translating E into Prolog is to be passed to p, we need to define the *meta signature* of p using the compiler directive arguments. For instance, the following directive will make findall/3 work as intended:

```
:- arguments findall(oid,bform,oid).
```

This signature declaration tells *FLORA-2* that the first and the third arguments to findall/3 should be translated as oids, while the second argument of findall/3 should be treated as a *syntactic object* that represents formulas of the kind that might occur in the body of a rule.

8 HiLog and Related Issues

HiLog [5] is the default syntax that *FLORA-2* uses to represent functor terms (including object Ids) and predicates. In HiLog, complex terms can appear wherever a function symbol is allowed. For example, group(X)(Y,Z) is a HiLog term where the functor is no longer a symbol but rather a complex term group(X). Variables in HiLog can range over terms, predicate and function symbols, and even over atomic formulas. For instance,

$$?-p(X), X(p). \tag{3}$$

[9] Something like isa(P,manager),mvd(P,hobbies,tennis).

and

$$?-p(X), X(p), X. \tag{4}$$

are perfectly legal queries. If `p(a(b))`, `a(b)(p)`, and `a(b)` are all true in the database, then $X = a$ is one of the answers to the query.

Although HiLog has a higher order syntax, its semantics is first order [5]. Any HiLog term can be consistently translated into a Prolog term. For instance, `group(X)(Y,Z)` can be represented by the Prolog term `apply(apply(group,X),Y,Z)`. The translation scheme is pretty straightforward and described in [5].

In *FLORA-2* any Id-term, including function symbols and predicate symbols, are considered to be HiLog terms and therefore are subject to translation. That is, even a normal Prolog term will by default be represented using the HiLog translation, e.g., `foo(a)` will be represented as `apply(foo,a)`. This guarantees that HiLog unification will work correctly at runtime. For instance, `foo(a)` will unify with `F(a)` and bind the variable `F` to `foo`.

There is one important difference between HiLog, as described in [5], and its implementation in *FLORA-2*. In HiLog, functor terms that appear as arguments to predicates and the atomic formulas (*i.e.*, predicates that are applied to some arguments) belong to the same domain. In contrast, in *FLORA-2* they are in different domains.[10] For instance, (4) will cause an error because `X` is bound to the *term* `a(b)`, which is different from the atomic formula `a(b)`, and thus has no truth value. In a future release, we will provide a predicate `call/1`, which converts HiLog terms in the domain of arguments into HiLog terms in the domain of atomic formulas, so (4) could be rewritten into `p(X), X(p), call(X).`[11]

Like in classical logic, `foo` and `foo()` are different terms. However, in programming, it is convenient to identify these terms when they are treated as predicates. Prologs often disallow the use of the `foo()` syntax altogether. The same distinction holds in HiLog: `foo`, `foo()` and `foo()()` are all different. In terms of the HiLog to Prolog translation, this means that `foo` is different from `apply(foo)` is different from `apply(apply(foo))`. However, just like in Prolog, we treat `p` as syntactic sugar for `p()` when both occur as predicates. Thus, the following queries are the same:

```
?- p.
?- p().
```

In the following program,

```
p.
q().
?- p(), X().
?- q, X().
?- r = r().
```

[10] This is allowed in ***sorted HiLog*** [4].

[11] Of course, `call/1` exists in Prolog as well. However, unlike HiLog and sorted HiLog, this predicate does not have logical semantics.

the first two queries will succeed (with X bound to p or q), but the last one will fail. Identification of p with p() does not extend to p()(), which is distinct from both p and p() not only as a term but also as a formula. Thus, in the following program, all queries fail:

```
p.
q().
?- p()().
?- q()().
?- p = p()().
?- q() = q()().
```

8.1 Meta-programming

F-logic together with HiLog is powerful stuff. In particular, it lends itself naturally to meta-programming. For instance, it is easy to examine the methods and types defined for the various classes. Here are some simple examples:

```
// all unary scalar methods defined for John
?- john[M(_) -> _].

// all unary scalar methods that apply to John,
// for which a signature was declared
?- john[M(_) => _].

// all method signatures that apply to John,
// which are either declared explicitly or inherited
?- john[M => _].

// all method invocations defined for John
?- john[M -> _].
```

However, a number of meta-programming primitives are still needed since they cannot be directly expressed in F-logic. Many such features are provided by the underlying XSB system and $\mathcal{F}$*LORA-2* simply takes advantage of them:

```
?- functor(X,f,3).
X = f(_h455,_h456,_h457)
Yes.

?- compound(f(X)).
X = _h472
Yes.

?- X =.. [f,a,b].
X = f(a,b)
```

```
Yes.
```

Note that these primitives are used for Prolog terms only and are described in the XSB manual.

In *FLORA-2*, variables can be bound to both formulas and terms. For instance, in `X = p(a)`, `p(a)` is viewed as a term and `X` is bound to it. Likewise, in `X = a[m->v]`, the F-molecule is evaluated to its object value and then unified with `X`. To bind variables to formulas instead, *FLORA-2* provides a `meta-unification` operator, $\sim$. This operator treats its arguments as formulas and unifies them as such. For instance, `X` $\sim$ `a[m->v,k->V]` binds `X` to the F-molecule `a[m->v,k->V]` and `a[m->v,k->V]`   `X[M->v,k->p]` unifies the two molecules by binding `X` to `a`, `M` to `m`, and `V` to `p`.

Unimplemented: Meta-programming support for HiLog terms and F-molecules is incomplete. For instance, there is no operator similar to `=..`, like in Prolog (or, rather, it does not do the right thing in *FLORA-2*). Additional meta-programming support will be provided through future enhancements.

8.2 Passing Arguments between *FLORA-2* and XSB

The native HiLog support in *FLORA-2* causes some tension when crossing the border from one system to another. The reason is that *FLORA-2* terms and XSB terms have different internal representation. Even though XSB supports HiLog (according to the manual, anyway), this support is incomplete and is not integrated well into the system — most notably into the XSB module system. As a result, XSB does not recognize terms passed to it from *FLORA-2* as HiLog terms and, thus, many useful primitives will not work correctly. (Try `?- writeln(foo(abc))@prolog()` and see what happens.)

To cope with the problem, *FLORA-2* provides a primitive, `flP2H(Plg,Hlg)`, which does the translation. If the first argument, `Plg`, is bound, the primitive binds the second argument to the Hilog representation of the term. If `Plg` is already bound to a Hilog term, then `Hlg` is bound to the same term without conversion. Similarly, if `Hlg` is bound to a HiLog term, then `Plg` gets bound to the Prolog representation of that term. If `Hlg` is bound to a non-HiLog term, then `Plg` gets bound to the same term without conversion. In all these cases, the call to flP2H/2 succeeds. If `both` arguments are bound, then the call succeeds if and only if

- `Plg` is a Prolog term and `Hlg` is its HiLog representation.
- Both `Plg` and `Hlg` are identical HiLog terms.

Note that if both `Plg` and `Hlg` are bound to the same *Prolog term* then the predicate *fails*. Thus, if you type the following queries into the *FLORA-2* shell, they both succeed:

```
?- flP2H(X,f(a)), flP2H(X,f(a)).
?- flP2H(f(a),f(a)).
```

Not all arguments passed back and forth to XSB need conversion. For instance, `sort/2`, `ground/1`, `compound/1`, and many others do not need conversion because they work the same for Prolog and HiLog representations. On the other hand, most I/O predicates require conversion. In a future release, *FLORA-2* will provide libraries of useful predicates and methods that do appropriate conversion without the user having to do this explicitly. One such library, `io`, is described in Section 20.

8.3 First-Order Predicates and HiLog Predicates

FLORA-2 is an object-relational language in the sense that it supports both the object-oriented syntax of F-logic and the predicate-based (*i.e.*, relational) syntax of Prolog. Incorporation of predicates is conceptually simple, but is somewhat burdened by pragmatic considerations, namely, integration with HiLog. As a result, *FLORA-2* has two types of predicates, *first-order predicates* and *HiLog predicates*.

Recall that the main reason for using HiLog as the native syntax and semantics in *FLORA-2* is meta-programming. However, this is also one of the reasons for adding the first-order predicates. To explain, consider the following query:

```
?- X(a,b).
```

which returns all names of binary predicates that contain the tuple `a,b`. However, it is sometimes useful to exclude certain predicates from the domain of the meta-variables, such as `X` above. For instance, if `write/2` is an I/O predicates, should the above query return the binding `X=write`? If so, if `a` happens to be the name of a file and `b` a string, should this string be written out to that file as a side effect of the evaluation of this query?

To avoid these semantic problems, *FLORA-2* supports the `firstorder` declaration, which defines certain symbols as first-order predicates. For instance,

```
:- firstorder foo/1, moo/3.
```

Predicates defined in this way are outside of the range of HiLog variables. If you want all predicates to be shielded from meta programming, you can use the following compiler directive:

```
:- firstorderall.
```

First-order predicates are module-aware. This means that the predicates defined in different user, system, or Prolog modules are treated as unrelated entities.

There is one other reason why we might sometimes choose first-order predicates over HiLog predicates — predicate tabling. This issue is discussed in Section 14.

8.4 Expunging of First-Order Predicates

To expunge a predicate of the form `functor/arity` from the user module that a program is loaded into, the following compiler directive can be used:

```
:- expunge functor/arity.
```

expunge can also be called like a normal predicate, either from the shell or from within a rule body, to expunge predicates in one specific user module. The syntax is shown below:

```
:- expunge functor/arity, ..., functor/arity in modulename.
```

where in is a special keyword, and modulename stands for the name of the user module from which to expunge the given list of predicates.

Note that when expunge functor/arity is used as a normal predicate (not as a compiler directive), it will expunge the predicate for the current program module.

9 Equality Maintenance

Unlike regular Prolog, *FLORA-2* terms can become equal, which is a side effect of the F-logic semantics for scalar methods. For instance, consider the following F-logic facts:

mary[spouse→john].
mary[spouse→joe].
john[son→→frank].

Since spouse is a scalar attribute, it can have at most one value for any given object. This implies that the oids john and joe must refer to the same object. Therefore, whatever is true about john should be also true about joe, and vice versa. Thus we should be able to derive that joe[son→→frank].

What is illustrated above is just a very simple scenario of equality maintenance. Another scenario arises due to the path expressions in the rule heads (Section 6.2). Furthermore, users can define equality *explicitly* in the source program using the predicate :=:, e.g.,

```
john:=:batman.
```

Once oids are made equal, this fact may need to be propagate to all F-logic structures, including the subclass hierarchy, the ISA hierarchy, etc., depending on the application domain. Although equality is a powerful feature, its maintenance can slow the program down quite significantly. In order to be able to eat the cake and have it at the same time, *FLORA-2* allows the user to control how equality is handles. by providing the following three compiler directives:

```
:- equality none.  (default)
:- equality basic.
:- equality flogic.
```

These directives affect the particular user modules in which they are included. Thus, equality can be treated differently in different modules. This allows to compartmentalize the problem and, if used judiciously, can lead to significant gain in performance.

The first directive, `equality none`, does not maintain equality and `:=:` is considered just like any other user-defined predicate without any special semantics. The directive `equality basic` guarantees that `:=:` obeys the usual congruence rules for equality, i.e., transitivity, reflexivity, and symmetry, and substitution, but not the special F-logic equality rule for scalar methods. Finally, `equality flogic` means that the full equality theory for F-logic is in force. In particular, programs with path expressions in the rule heads might require this level of equality to work correctly (if there is a possibility that the Skolem terms introduced to represent the path expressions might be equal to some real objects).

Note that the level of equality maintenance can also be changed at run time by executing a goal such as

```
?- (equality basic).
```

Furthermore, $\mathcal{F}$*LORA-2* allows one user module to set, at run time, the level of equality maintenance in another user module:

```
equality {none|basic|flogic} in modulename
```

This might be useful for *dynamic* modules, *i.e.*, modules that are not associated with any files and whose content is generated completely dynamically. (See Section 15.)

One final advice regarding equality. In many cases, programmers tend to use equality as an aliasing technique for long messages, numbers, etc. In this case, we recommend to use the preprocessor commands, which achieve the same result without loss of performance. For instance,

```
#define YAHOO  'http://yahoo.com'

?- YAHOO[fetch \fd X].
```

Assuming that `fetch` is a method that applies to strings that represent WWW sites and that fetches the corresponding Web pages, the above program will fetch the page at the Yahoo site, because $\mathcal{F}$*LORA-2* compiler will replace YAHOO with the corresponding string that represents a URL.

10 Inheritance

F-logic supports two types of inheritance: *structural* and *behavioral.* Structural inheritance applies to signatures only. For instance, if `student::person` and a program defines the signature `person[name⇒string]` then the query `?- student[name⇒X]` succeeds with `X=string`.

Behavioral inheritance is much more complicated. The problem is that it is *non-monotonic.* That is, the addition of new facts might obviate previously established inferences.

F-logic (and $\mathcal{F}$*LORA-2*) distinguishes between attributes and methods that can inherit values from superclasses and those that do not. The syntax that we have seen so far applies to *non-inheritable* attributes only. *Inheritable attributes* are declared using the `*=>` and `*=>>` style arrows and defined using the `*->` and `*->>` style arrows. For instance, the following is a *FLORA-2* program for the classical `Royal Elephant` example:

```
elephant[color*=>color].
royal_elephant::elephant.
clyde:elephant.
elephant[color*->gray].
```

The question is what is the color of `clyde`? `clyde`'s color has not been defined in the above program. However, since `clyde` is an elephant and the default color for elephants is gray, `clyde` must be gray. Thus, we can derive:

```
clyde[color->gray].
```

Observe that when inheritable methods are inherited from a class by its members, the attribute becomes non-inheritable. On the other hand, when such a method is inherited by a subclass from its superclass, then the method is still inheritable, so it can be further inherited by the members of that subclass or by its subclasses. For instance, if we have

```
circus_elephant::elephant.
```

then we can derive

```
circus_elephant[color*->gray].
```

Non-monotonicity of behavioral inheritance becomes apparent when certain new information gets added to the knowledge base. For instance, suppose we learn that

```
royal_elephant[color*->white].
```

Although we have previously established that `clyde` is gray, this new information renders our earlier conclusion invalid. Indeed, Since `clyde` is a royal elephant, he must be white, while being an elephant he must be gray. The conventional wisdom in object-oriented languages, however, is that inheritance from more specific classes must take precedence. Thus, we must withdraw our earlier conclusion that `clyde` is gray and infer that he is white:

```
clyde[color->white].
```

Behavioral inheritance in F-logic is discussed at length in [9]. The above problem of non-monotonicity is just a tip of the iceberg. Much more difficult problems arise when inheritance interacts with the regular deduction. To illustrate, consider the following program:

```
b[m*->>c].
a:b.
a[m->>d] :- a[m->>c].
```

In the beginning, it seems that `a[m->>c]` should be derived by inheritance, and so we can derive `a[m->>d]`. Now, however, we can reason in two different ways:

1. `a[m->>c]` was derived based on the belief that attribute `m` is not defined for the object `a`. However, once inherited, necessarily we must have `a[m->>{c,d}]`. So, the value of attribute `m` is not really the one produced by inheritance. In other words, inheritance of `a[m->>c]` negates the very premise on which the original inheritance was based, so we must undo the operation and the ensuing rule application.

2. We did derive `a[m->>d]` as a result of inheritance, but that's OK — we should not really be looking back and undo previously made inheritance inferences. Thus, the result must be `a[m->>{c,d}]`.

A similar situation (with similarly conflicting conclusions) arises when the class hierarchy is not static. For instance,

```
d[m*->e]
d::b.
b[m*->c].
a:b.
a:d :- a[m->c].
```

If we inherit `a[m →c]` from `b` (which seems to be OK in the beginning, because nothing overrides this inheritance), then we derive `a:d`, *i.e.*, we get the following: `a:d::b`. This means that *now* `d` seems to be negating the reason why `a[m →c]` was inherited in the first place. Again, we can either undo the inheritance or adopt the principle that inheritance is never undone.

A semantics that favors the second interpretation was proposed in [9]. This approach is based on a fixpoint computation of non-monotonic behavioral inheritance. However, this semantics is very hard to implement efficiently, especially using a top-down deductive engine provided by XSB. It is also unsatisfactory in many respects because it is not based on any model-theory. *$\mathcal{F}$LORA-2* uses a different, more cautious semantics for inheritance, which favors the first interpretation above. The basic idea can be summarized using the following rules, which define how class instances inherit from the classes they belong to:

```
// inheritance rules for scalar attributes
:- table defined/2, overwritten/3, conflict/3.

Obj[A->V] <- Obj:Class, Class[A*->V], tnot defined(Obj,A),
             tnot overwritten(Obj,Class,A), tnot conflict(Obj,Class,A).

defined(Obj,A) <- Obj[A->V].

overwritten(Obj,Class,A) <- Obj:Class1, Class1::Class,
                            Class1[A*->W], Class1 \= Class.
```

```
conflict(Obj,Class,A) <- Obj:Super, Super[A*->V],
                         tnot Super::Class, tnot Class::Super.

// inheritance rules for multi-valued attributes
:- table definedSet/2, overwrittenSet/3, conflictSet/3.

Obj[A->>V] <- Obj:Class, Class[A*->>V], tnot definedSet(Obj,A),
              tnot overwrittenSet(Obj,Class,A),
              tnot conflictSet(Obj,Class,A).

definedSet(Obj,A) <- Obj[A->>V].

overwrittenSet(Obj,Class,A) <- Obj:Class1, Class1::Class,
                               Class1[A*->>W], Class1 \= Class

conflictSet(Obj,Class,A) <- Obj:Super, Super[A*->V],
                            tnot Super::Class, tnot Class::Super.
```

Negation here is implemented using the *well-founded semantics* for negation [10, 11] (as indicated by the tnot operator). Similar rules are needed to describe how classes inherit from superclasses.

Under this semantics, clyde will still be white, but in the other two examples a[m->c] is *not* inherited. Details of this semantics and its model theory will be presented in a future paper.

In the examples that we have seen so far, path expressions use only non-inheritable attributes. Clearly, there is no reason to disallow inheritable attributes in such expressions. To distinguish inheritable attributes from non-inheritable ones, $\mathcal{F}$*LORA-2* uses the symbols ! and !! in its path expressions. For instance,

```
clyde!color          means: some X, such that clyde[color*->X]}.
obj!!attr            means: some Y, such that obj[attr*->>Y].
```

11 Aggregates

The syntax for aggregates is similar to that in FLORID. An aggregate has the following form:

agg{X[Gs] | *body*}

where agg represents the aggregate operator, X is called the aggregation variable, Gs is a list of comma-separated grouping variables, and *body* is a logical formula that specifies the query conditions. The grouping variables, Gs, are optional. *body* can be any combination of conjunction, disjunction, and negation of literals.

All the variables appearing in *body* but not in X or Gs are considered to be existentially quantified. Furthermore, the syntax of an aggregate must satisfy the following conditions:

1. All names of variables in both X and Gs must appear in *body*;
2. Gs should not contain X.

Aggregates are evaluated as follows: First, the query condition specified in *body* is evaluated to obtain all the bindings for the template of the form `<X, Gs>`. Then, these tuples are grouped according to each distinct binding for `<Gs>`. Finally, for each group, the aggregate operator is applied to the list of bindings for the aggregate variable X.

The following aggregate operators are supported in *FLORA-2*: `min`, `max`, `count`, `sum`, `avg`, `collectset` and `collectbag`.

The operators `min` and `max` can apply to any list of terms. The order among terms is defined by the XSB operator `@=<`. In contrast, the operators `sum` and `avg` can take numbers only. If the aggregate variable is instantiated to something other than a number, `sum` and `avg` will discard it and generate a runtime warning message.

For each group, the operator `collectbag` collects all the bindings of the aggregation variable into a list. The operator `collectset` works similarly to `collectbag`, except that all the duplicates are removed from the result list.

In general, aggregates can appear wherever a number or a list is allowed. Therefore, aggregates can be nested. The following examples illustrate the use of aggregates (some borrowed from the FLORID manual):

```
?- Z = min{S|john[salary(Year)->S]}.
?- Z = count{Year|john.salary(Year) < max{S|john[salary(Y)->S], Y < Year}}.
?- avg{S[Who]|Who:employee[salary(Year)->S]} > 20000.
```

If an aggregate contains grouping variables that are *not* bound by a preceding subgoal, then this aggregate would backtrack over such grouping variables (In other words, grouping variables are considered to be existentially quantified). For instance, in the last query above, the aggregate will backtrack over the variable `Who`. Thus, if `john`'s and `mary`'s average salary is greater than 20000, this query will backtrack and return both `john` and `mary`.

The following query returns, for each employee, a list of years when this employee had salary less than 60. This illustrates the use of the `collectset` aggregate.

```
?- Z = collectset{Year[Who]|Who[salary(Year)->X], X < 60}.
Z = [1990,1991]
Who = mary

Z = [1990,1991,1997]
Who = john
```

11.1 Aggregation and Set-Valued Methods

Aggregation is often used in conjunction with set-valued methods, and *FLORA-2* provides several shortcuts to facilitate this use. In particular, the operators `->->` and `*->->`, for non-inheritable and inheritable multivalued methods, collects all the values of the given method for a given object in a set. The semantics of these operators is as follows:

```
O[M->->L] :- L=collectset{V|O[M->>V]}

O[M*->->L] :- L=collectset{V|O[M*->>V]}
```

Note that in `O[M->->L]` and `O[M*->->L]` `L` is a list of oids.

Having special meaning for `->->` and `*->->` means that these constructs *cannot* appear in the head of a rule. One other caveat: recursion through aggregation is not supported and can produce incorrect results.

Sets collected in the above manner often need to be compared to other sets. For this, *FLORA-2* provides another pair of primitives: `+>>` and `*+>>` for non-inheritable and inheritable methods, respectively. The atom of the form `o[m+>>s]` is true if the set of all values of the non-inheritable attribute `m` for object `o` *contains* every element in the list `s`.

For instance, the following query tests whether all Mary's children are also John's children:

```
?- mary[children->->L], john[children+>>L].
```

As with `->->` and `*->->`, the use of `+>>` and `*+>>` is limited to rule bodies.

12 Boolean Methods

As a syntactic sugar, *FLORA-2* introduces boolean methods, which can be considered as scalar methods that return some fixed value, e.g., `void`. For example, the following facts:

```
john[is_tall -> void].
john[loves(tennis) -> void].
```

can be simplified as boolean methods as follows:

```
john[is_tall].
john[loves(tennis)].
```

Conceptually, boolean methods are statements about objects whose truth value is the only concern. Boolean methods do not return any value (not even the value `void`). Therefore, boolean methods `cannot` appear in path expressions. For instance, `john.is_vegetarian` is illegal.

Like scalar methods, boolean methods can be inheritable. To make a boolean method inheritable, the `*` sign is prepended to the method name:

```
buddhist[*is_vegetarian].
john:buddhist.
```

The above says that all Buddhists are vegetarian and John (the object with oid `john`) is a Buddhist. Since `is_vegetarian` is inheritable, it follows that John is also a vegetarian, i.e., `john[is_vegetarian]`.

13 Anonymous Oid

For applications where oids are not important, *FLORA-2* provides the compiler directive `_#` to automatically generate a new oid. `_#` can be used wherever an Id-term is allowed. Like the anonymous variable `_`, each occurrence of `_#` represents an anonymous oid. The difference is that such an oid is not only unique in each rule, but in the source program as well.

Of course, uniqueness is achieved through the use of special weird naming schema for such oids. However, as long as the user does not use a similar naming convention (who, on earth, would give names that begin with lots of ``_$''s?), uniqueness is guaranteed.

For example, in the following program:

```
_#[ssn->123, father->_#[name->john, spouse->_#[name->mary]]].
foo[_#(X)->Y] :- foo(X,Y).
```

the compiler will generate unique oids for each occurrence of `_#`. Note that, in the second clause, only one oid is generated and it serves as a method name.

14 *FLORA-2* and Tabling

14.1 Tabling in a Nutshell

Tabling is a technique that enhances top-down query evaluation with a mechanism that remembers the calls made previously in the process. This technique is known to be essentially equivalent to the Magic Sets method for bottom-up evaluation. However, tabling combined with top-down evaluation has the advantage of being able to utilize highly optimized compilation techniques developed for Prolog. The result is a very efficient deductive engine.

XSB lets the user specify which predicates must be tabled. The *FLORA-2* compiler automatically tables the predicates used to represent F-molecules. However, the user is responsible for telling the system which other predicates — first-order and HiLog — must be tabled. *FLORA-2* understands one tabling directive for first-order predicates and another for HiLog. (Section 19 lists all the compiler directives.)

To table a first-order predicate, the following can be used:

```
:- firstorder tc/2.
:- table tc/2.

tc(X,Y) :- edge(X,Y).
tc(X,Y) :- edge(X,Y), tc(Y,Z).
```

which renders `tc/2` as a tabled first-order predicate. (**Unimplemented**: In the future, the tabling directive will imply `firstorder`.)

To table HiLog predicates, *FLORA-2* requires you to table *all* predicates of a given arity: For instance,

```
:- hilogtable 3.
```

tables all HiLog predicates of arity 3.

It is important to keep in mind that XSB does not do reordering of F-logic molecules and predicates during joins. Instead, all joins are performed left-to-right. Thus, program clauses must be written in such a way as to ensure that smaller predicates and classes appear early on in the join. Also, even though XSB tables the results obtained from previous queries, the current tabling engine has several limitations. In particular, when a new query comes in, XSB tries to determine if this query is "similar" to one that already has been answered (or is in the process of being evaluated). Unfortunately, the default notion of similarity used by XSB is fairly weak, and many unnecessary recomputations might result. Recently, a new technique, called ***subsumptive tabling***, has been implemented in XSB. It is known that subsumptive tabling can speed up certain queries by an order of magnitude. A future version of *FLORA-2* might take advantage of this technique.

14.2 Procedural Methods

When XSB (and *FLORA-2*) evaluate a program, all tabled predicates are partially materialized and all the computed tuples are stored in XSB tables. Thus, if you change the set of facts, the existing tables must be discarded in order to allow XSB to recompute the results. This is accomplished by issuing the predicate `abolish_all_tables/0` described in the XSB manual. We discuss updates in Section 15.

Because tabling is not integrated with the update mechanism in XSB, it can have undesirable effect on predicates with non-logical "side effects" (e.g., writing or reading a file) and predicates that change the state of the database. If a tabled predicate has a side effect, the first time the predicate is called the side effect is performed, but the second time the call simply returns with success or failure (depending on the outcome of the first call), because XSB will simply look it up in a table. Thus, if the predicate is intended to perform the side effect each time it is called, it will not operate correctly.

Object-oriented programs often rely on methods that produce side effects or make updates. In *FLORA-2* we call such methods *procedural*. Because by default *FLORA-2* tables everything that looks like an F-molecule, these procedural methods are potentially subject to the aforesaid problem.

To sidestep this program, *FLORA-2* introduces a new syntax to identify procedural methods — by allowing the "`#`" sign in front of a procedural method. For instance, the following rule defines an output method that, for every object, writes out its oid:

```
O[#output] :- write(O)@prolog().
```

Like boolean methods, procedural methods can take arguments, but do not return any values. The only difference is that procedural methods are *not* tabled, while boolean methods are.

14.3 Cuts

No discussion of a logic programming language is complete without a few words about the infamous Prolog cut (!). Although Prolog cut has been (mostly rightfully) excommunicated as far as Database Query Languages are concerned, it is sometimes indispensable when doing "real work", like pretty-printing *$\mathcal{F}$LORA-2* programs or implementing a pattern matching algorithm. To facilitate this kind of tasks, *$\mathcal{F}$LORA-2* lets the programmer use cuts. However, the current implementation of XSB has a limitation that Prolog cuts cannot "cut across tabled predicates." If you get an error message telling something about cutting across the tables — you know that you have cut too much!

The basic rule that can keep you out of trouble is: do not put a cut in the body of a rule *after* any F-molecule or tabled first-order or HiLog predicate. However, it is OK to to put a cut before any F-molecule. It is even OK to have a cut in the body of a rule that *defines* an F-molecule (again, provided that the body has no F-molecule to the left of that cut). If you need to use cuts, plan on using procedural methods or non-tabled predicates.

In a future release, XSB will implement a different tabling schema. While cutting across tables will still be prohibited, it will provide an alternative mechanism that achieves many of the goals a cut is used to achieve.

15 Updates

$\mathcal{F}$LORA-2 provides primitives to update the runtime database. Unlike Prolog, *$\mathcal{F}$LORA-2* does not require the user to define a predicate as dynamic in order to update it. Instead, every predicate (HiLog or first-order) and object has a *base part* and a *derived part*. Updates directly change only the base parts and only indirectly the derived parts.

Note that the base part of a predicate or an object contains *both* the facts that were *inserted explicitly* into the database and the facts that you specified in the program. For instance, in

```
p(a).
a[m->b].
```

the fact `p(a)` will be placed in the base part of the predicate p/1 tt and it can be deleted by the `delete` primitive. Likewise, the fact `a[m->b]` is updatable. If you do not want some facts to be updatable, use the following syntax:

```
p(a) :- true.
a[m->b] :- true.
```

Unimplemented: In the current release, the base part of first-order predicates cannot be queried in the *$\mathcal{F}$LORA-2* shell (but it can be queried in the program file). Likewise, if the `firstorderall` directive is in effect, the base part of the first-order predicates is inaccessible to querying. These limitations will be removed in the future.

FLORA-2 updates can be *non-backtrackable,* as in Prolog, or backtrackable, as in Transaction Logic [2]. We first describe non-backtrackable updates.

15.1 Non-backtrackable Updates

The effects of non-backtrackable updates persist even if a subsequent failure causes the system to backtrack.

FLORA-2 supports the following non-backtrackable update primitives: `insert`, `insertall`, `delete`, `deleteall`, `erase`, `eraseall`. These primitives use special syntax (the curly braces) and are *not* predicates. Thus, it is allowed to have a user-defined predicate such as `insert`.

Insertion. The syntax of an insertion is as follows (note the {,}s!):

insop{*literals* [| *query*]}

where *insop* stands for either `insert` or `insertall`. The *literals* part represents a comma separated list of literals, which can include predicates and F-molecules. The optional part, |*query*, is an additional condition that must be satisfied in order for *literals* to be deleted. The semantics is that *query* is posed first and, if it is satisfied, *literals* is inserted (note that the query may affect the variable binding and thus the particular instance of *literals* that will be inserted). For instance, in

```
flora2 ?- insert{p(a),mary[spouse->smith,children->>frank]}
flora2 ?- insertall{P[spouse->S] | S[spouse->P]}
```

the first statement inserts a particular molecule. In the second case, the query `S[spouse->P]` is posed and one answer (a binding for `P` and `S`) is obtained. If there is no such binding, nothing is inserted and the statement fails. Otherwise, the instance of `P[spouse->S]` is inserted for that binding and the statement succeeds.

The difference between `insert` and `insertall` is that `insert` inserts only one instance of *literals* that satisfies the formula, while `insertall` inserts *all* instances of the literals that satisfy the formula. In other words, *query* is posed first and *all* answers are obtained. Each answer is a tuple of bindings for some (or all) of the variables that occur in *literals.* To illustrate the difference between `insert` and `insertall`, consider the following statements:

```
flora2 ?- p(X,Y), insert{q(X,Y,Z)|r(Y,Z)}.
flora2 ?- p(X,Y), insertall{q(X,Y,Z)|r(Y,Z)}.
```

In the first case, if `p(x,y)` and `r(y,z)` are true, then the fact `q(x,y,z)` is inserted. In the second case, if `p(x,y)` is true, then the update means the following:

For each z such that r(y,z) holds, insert(x,y,z).

The primitive `insertall` is also known as a bulk-insert operator.

Note that literals appearing inside the update predicates are treated as facts and should follow the syntactic rules for facts and literals in the rule head. In particular, multi-valued path expressions are not allowed. However, *FLORA-2* permits database updates whose target is a user module other than the current one. Thus, *FLORA-2* module specifications *are* allowed. As before, module specification is distributive through parentheses, so

```
flora2 ?- insert{(mary[children->>frank], john[father->smith]) @ foomod}
```

is equivalent to the following one:

```
flora2 ?- insert{mary[children->>frank] @ foomod, john[father->smith] @ foomod}
```

However, @prolog(...) is not allowed: If you want to update a dynamic XSB predicate, use `assert/retract`:

```
flora2 ?- assert(foo(a,b,c))@prolog().
```

Well, not quite. Recall Section 8.2 on the issues concerning the difference between Prolog representation of terms and the one used in XSB. The problem is that `f(a,b,c)` is a HiLog term that XSB does not understand and will not associate it with the predicate `foo/3` that it might have. To do it right, use explicit conversion:

```
?- flP2H(PrologRepr,foo(a,b,c)) assert(PrologRepr)@prolog().
```

This will insert `foo(a,b,c)` into the catch-all XSB module called `usermod`. If `foo/3` is defined in another XSB module, `bar`, and is imported by `usermod`, then the above statement will not do anything useful due to certain idiosyncrasies in the XSB module system. However, the following will work if `foo/3` is defined as dynamic in the module `bar`:

```
flora2 ?- assert(foo(a,b,c)@prolog(bar))@prolog().
```

We should also mention one important difference between insertion of facts in *FLORA-2* and Prolog. Prolog treats facts as a *bag*, so duplicates are allowed. In contrast, *FLORA-2* treats the database as a *set* of facts with no duplicates. Thus, insertion of a fact that is already in the database has no effect.

Deletion. The syntax of a deletion primitive is as follows:

```
delop{literals [| query]}
```

where *delop* can be `delete`, `deleteall`, `erase`, and `eraseall`. The *literals* part is a comma separated list of F-molecules and predicates. The optional part, | *query*, represents an additional constraint or a restricted quantifier, similarly to the one used in the insertion primitive.

For instance, the following predicate:

```
flora2 ?- deleteall{john[Year(Semester)->>Course] | Year < 2000}
```

will delete John's course selection history before the year 2000.

Note that the semantics of a `delete`{*literal* | *query*} statement is that first the query *literal* $\wedge$ *query* should be asked. If it succeeds, then deletion is performed. For instance, if the database is

```
p(a). p(b). q(a). q(c).
```

then the statement below:

```
flora2 ?- delete{p(X)|q(X)}
```

will succeed with the variable `X` bound to `a` and `p(a)` will be deleted. However, if the database contains only the facts `p(b)` and `q(c)`, then the above predicate will fail and the database will stay unchanged.

FLORA-2 provides four deletion primitives: `delete`, `deleteall`, `erase`, and `eraseall`. The primitive `delete` removes at most one fact from the database. The primitives `deleteall` and `eraseall` are `bulk delete` operations; `erase` is kind of a hybrid: it starts slowly, by deleting one fact, but may go on a joy ride and end up deleting much of your data. These primitives are described below.

1. If there are several bindings or matches for the literals to be deleted, then `delete` will choose only one of them nondeterministically, and delete it. For instance, suppose the database contains the following facts:

   ```
   p(a). p(b). q(a). q(b).
   ```

   then

   ```
   flora2 ?- delete{p(X),q(X)}
   ```

   will succeed with `X` bound to either `a` or `b`, depending on the ordering of facts in the database at runtime.

2. In contrast to the plain `delete` primitive, `deleteall` will try to delete all bindings or matches. Namely, for each binding of variables produced by *query* it deletes the corresponding instance of *literal*. If *query* $\wedge$ *literal* is false, the `deleteall` primitive fails. To illustrate, consider the following:

   ```
   flora2 ?- p(X,Y), deleteall{q(X,Y,Z)|r(Y,Z)}.
   ```

and suppose `p(x,y)` is true. Then the above statement will, for each `z` such that `r(y,z)` is true, delete `q(x,y,z)`.

For another example, suppose the database contains the following facts:

```
p(a). q(b). q(c).
```

and the query is `?- deleteall{p(a),q(X)}`. The effect will be the deletion of `p(a)` and of all the facts in `q`. (If you wanted to delete just one fact in `q`, `delete` should have been used.

Unlike the `delete` predicate, `deleteall` *always* succeeds. However, when it succeeds, `deleteall` will leave all variables unbound.

3. `erase` works like `delete`, but with an object-oriented twist: For each F-logic fact, f, that it deletes, `erase` will traverse the object tree by following f's methods and delete all objects reachable in this way. It is a power-tool that can cause maiming and injury. Safety glasses and protective gear are recommended.

   Note that only the base part of the objects can be erased. If the object has a part that is derived from the facts that still exist, this part will not be erased.

4. `eraseall` is the take-no-prisoners version of `erase`. Just like `deleteall`, it first computes *query* and for each binding of variables it deletes the corresponding instance of *literal*. For each deleted object, it then finds all objects it references through its methods and deletes those. This continues recursively until nothing reachable is left. This primitive always succeeds.

15.2 Backtrackable Updates

The effects of backtrackable updates are undone upon backtracking, i.e., if some post-condition fails and the system backtracks, a previously inserted item will be removed from the database, and a previously deleted item will be put back.

The syntax of backtrackable update primitives is similar to that of non-backtrackable ones and the names are similar, too. The syntax for backtrackable insertion is:

btinsop{*literals* [| *formula*]}

while the syntax of a backtrackable deletion is:

btdelop{*literals* [| *query*]}

where `btinsop` stands for either `btinsert` or `btinsertall`, and `btdelop` stands for either of the following four deletion operations: `btdelete`, `btdeleteall`, `bterase`, and `bteraseall`. The meanings of *literals* and *query* is the same as in Section 15.1.

`btinsert`, `btinsertall`, `btdelete`, `btdeleteall`, `bterase`, and `bteraseall` work similarly to `insert`, `delete`, `deleteall`, `erase`, and `eraseall`, respectively, except that the new operations are backtrackable. Refer to Section 15.1 for details of these operations.

To illustrate the difference between backtrackable and non-backtrackable updates, consider the following execution trace immediately after the $\mathcal{F}$*LORA-2* system starts:

```
flora2 ?- insert{p(a)}, fail.

No.

flora2 ?- p(a).

Yes.

flora2 ?- btinsert{q(a)}, fail.

No.

flora2 ?- q(a).

No.
```

In the above example, when the first `fail` executes, the system backtracks to `insert{p(a)}` and does nothing. Thus the insertion of `p(a)` persists and the following query `p(a)` returns with `Yes`. However, when the second `fail` executes, the system backtracks to `btinsert{q(a)}` and removes `q(a)` that was previously inserted into the database. Thus the next query `q(a)` returns with `No`.

Keep in mind that some things that Prolog programmers routinely do with `assert` and `retract` goes against the grain of the use of backtrackable updates. In particular, `fail`-loops are not going to work (will leave the database unchanged) for obvious reasons. The `while` primitive should be used in such situations.

Unimplemented: This release does not support the `while`-loops.

Unimplemented: There are certain less obvious pitfalls in using backtrackable predicates in *FLORA-2*. For instance, the query

```
flora2 ?- p(X), btinsert{q(X)}
```

will succeed (if `p(X)` is true for some `X`), but will leave the database unchanged! The reason has to do with the fact that, by default, *FLORA-2* returns all answers to a query. In the current release, this involves a `fail`-loop and, obviously, hurts backtrackable updates badly. In a future release, these updates will be integrated with the query answering machinery much better. For now, we recommend using the `flOne` primitive, which turns the all-answer mode off:

```
flora2 ?- flOne.
flora2 ?- p(X), btinsert{q(X)}.
```

(Note: `flOne,p(X),btinsert{q(X)}` will not work: `flOne` must be issued in a separate query.)

Unimplemented: In the current release, arithmetic expressions in the query part of an update must have their variables be bound by the subgoals that precede the update primitive, except that the literal part does not currently bind. For instance,

```
?- delete{X[salary->Y] | Y<20000}.
```

is going to cause a run-time error. This limitation will be removed in a future release.

15.3 Updates and Tabling

We have earlier remarked in Section 14.2 that tabling and database updates do not mix well. The problem is that the results from previous queries are stored in XSB tables, and database updates do not modify XSB tables. Thus, a user might get the following counterintuitive result:

```
flora2 ?- insert{o[m->v]}.

Yes.

flora2 ?- o[m->v].

Yes.

flora2 ?- delete{o[m->v]}, o[m->v].

Yes.
```

The last positive answer is a consequence of the fact that XSB tables remember that the query `o[m->v]` is true. So, when the same query is asked after `delete`, a "stale" answer is returned from the tables. Similarly, tabling might interact poorly with `insert`:

```
flora2 ?- o[m->v].

No.

flora2 ?- insert{o[m->v]}, o[m->v].

No.
```

The reason for this result is, again, that XSB tables remember that `o[m->v]` is false. However, this result becomes stale after the insertion.

In a future release, *FLORA-2* will provide a workaround for these problems (and it is even possible that a future release of XSB will start doing the right thing in these situations). For now, the only remedy is to use a call to `abolish_all_tables`, which will clear all tables. However, at present, the only safe way to do this is by executing `abolish_all_tables` as a *separate* query.

One other temporal solution could be to design your program in such a way that tabled predicates and F-molecules will not depend on the facts that are dynamically inserted or deleted.

Another, related, issue is that a tabled predicate might have an update in its body. In this case, the update will be executed the first time the predicate is evaluated. Subsequent calls will return the predicate truth value from the tables, without invoking the predicate definition. Such a dependency is most likely a mistake, which is caught by the compiler.

Unimplemented: This compiler check has not been implemented yet.

15.4 Updates and Meta-programming

The update operators can take variables in place of literals to be inserted and queries. For instance,

```
flora2 ?- X ~ a[b->c], insert{X}.
```

One use for this facility is when one module, `foo`, provides methods that allow other modules to perform update operations on objects in `foo`. For instance, `foo` can have a rule

```
update(X,Y) :- delete{X}, insert{Y}.
```

Other modules can then issue queries like

```
flora2 ?- john[salary->X]@foo, Y is X+1000,
          update(john[salary->X],john[salary->Y]).
```

16 Negation

FLORA-2 supports two kinds of negation: the usual Prolog's negation as failure [6] and negation based on well-founded semantics [10, 11]. Both types of negation are compiled into clauses that invoke the corresponding operators in XSB.

Negation as failure is specified using the operator \+. Negation based on well-founded semantics is specified using the operator `tnot`. The well-founded negation, `tnot`, is applied to predicates that are tabled or to F-molecules that do not contain procedural methods (which are the only methods that are not tabled).

For non-tabled predicates and procedural methods, \+ is usually recommended (we assume that the user is familiar with the use of negation as failure in Prolog programs). It is also possible to apply `tnot` to these types of queries. The effect is that the system will first generate a new tabled predicate to hold the results of the query and then apply `tnot` to this new predicate. This may lead to incorrect results, if the negated formula is defined in terms of an update primitive (see Section 15 for a discussion of this issue).

For more information on the implementation of the negation operators in XSB we refer the reader to the XSB manual.

17 Control Constructs

For user convenience, *FLORA-2* provides a number of control structures borrowed from other languages. One often used such construct is `if-then-else`. For instance,

```
?- if (foo(a),foo2(b)) then (abc(X),cde(Y)) else (qpr(X),rts(Y)).
```

Here the system first evaluates `foo(a),foo2(b)` and, if true, evaluates `abc(X),cde(Y)`. Otherwise, it evaluates `qpr(X),rts(Y)`. The entire clause succeeds or fails depending on whether the `then`-part or the `else`-part succeeds or fails (whichever applies). The `if-then` version (without the `else`-part) is also provided. In this case, when the `if`-condition succeeds, the `then`-part is evaluated. The entire clause then succeeds or fails depending on the `then`-part. When the `if`-condition is false, the entire clause fails.

Note that `if`, `then`, and `else` are implemented as operators, and they bind stronger than the conjunction ",", the disjunction ";", etc. This is why the parentheses are needed in the above example.

Unimplemented: In the current version, the *FLORA-2* parser does not yet handle all the syntax errors associated with the use of `if-then-else` and nested clauses are not always parsed the way one would expect (say, as in C). Therefore, explicit parentheses are recommended around the entire clause.

18 Type Checking

Although *FLORA-2* allows specification of object types through signatures, type correctness is not checked automatically. So, what are the signatures good for then? One answer is that future versions of *FLORA-2* might support some form of type checking. However, because F-logic can naturally support powerful meta-programming, even the current level of support for signatures is useful. For instance, users can write simple queries to check the types of methods that might look suspicious. Here is one way to construct such a type-checking query:

```
scalar_type_incorrect(O,M,R) :- O[X->R], O:C, C[X=>D], tnot R:D.
?- scalar_type_incorrect(obj,meth,Result).
```

Here, we define what it means to violate type checking using the usual F-logic semantics. The corresponding predicate can then be queried. A "no" answer means that the corresponding attribute *does not* violate the typing rules.

In this way, one can easily construct special purpose type checkers. This feature is particularly important when dealing with *semistructured* data. (Semistructured data has object-like structure but normally does not need to conform to any type; or if it does, the type would normally cover only certain portions of the object structure.)

19 Summary of Compiler Directives

Like XSB compiler, *FLORA-2* compiler can take compiler directives, which begin with `:-` in contrast to `?-` for queries. *FLORA-2* requires that all directives appear at the top of the program prior to the first appearance of a rule or a fact. The following is a list of all the supported compiler directives:

- `expunge` *functor/arity, ..., functor/arity*
  Expunges the listed first-order predicates in the current user module, *i.e.*, deletes the contents as well as the definition of the corresponding predicate. By the current module we mean the user module into which the program that contains this particular `expunge` directive is loaded.

- `expunge` *functor/arity, ..., functor/arity* `in` *module*
  Same as above, except the predicates are expunged in the specified module rather than in the current user module.

- `equality none|basic|flogic`
  Sets the equality maintenance level in the current user module. With `none`, equality is not maintained, and the symbol `:=:` works like an ordinary predicate. With `basic`, the predicate `:=:` is treated as the equality, but only the usual congruence axioms for equality are enforced. With `flogic`, the usual congruence axioms are maintained *plus* the additional equality axiom for single-valued methods in F-logic.

- `equality none|basic|flogic in` *module*
  Same as above, except that equality maintenance is set for the specified user module.

- `firstorder` *functor/arity, ..., functor/arity*
  Declare the listed predicates as first-order predicates (as opposed to HiLog predicates).

- `firstorderall` Makes all predicates into first-order predicates. (The default for predicates is HiLog.)

- `arguments` *functor*(*type*, ..., *type*), where *type* is either `oid` or `bform`
  Specifies the meta signature to the given predicate, which tells how the arguments of the predicate are to be compiled. The `oid` type means that the oid of the corresponding argument is computed and passed to the predicate as an argument; `bform` means that the argument is treated as a *syntactic object* and its entire structure is preserved. For instance, if the argument is a formula (an F-logic molecule, a conjunction or a disjunction of literals), then it is converted to an object that preserves the entire structure of the formula. This type of compilation is useful in meta-predicates, such as `findall/3`, which need to consider the truth value of some of their arguments rather than their oid.

- `index(`*PredicateSpec,IndexSpec*`)`
  Specifies how the given first-order predicate must be indexed. *PredicateSpec* is of the form functor/arity. *IndexSpec* is either `trie` — meaning that trie indexing should be used — or a number. In the last case, indexing is hash-based on the specified argument (which should not exceed the arity, of course). Trie indexing is best if several arguments are instantiated, beginning with the first. Hash indexing is better if the fist argument is uninstantiated, because then you can specify another argument to index on.

  Note that the `index` directive is not very useful for predicates that mostly contain facts, because these are trie-indexed anyway (regardless of what you say). Thus, this instruction is useful only for predicates with partially instantiated arguments that appear in the rule heads.

- `op(`*precedence*`,`*type*`,`*operator*`)`
  Defines *operator* as a *FLORA-2* operator with the given precedence and type. The *type* is the same as in Prolog operators, *i.e.*, `fx`, `xf`, `xfy`, etc.

- `op(`*precedence*`,`*type*`,[`*operator*`, ..., `*operator*`])`
  Same as above, except that this directive defines a list of operators with the same precedence and type.

- `table` *functor/arity*`, ..., `*functor/arity* Requests that the specified first order predicates must be tabled.

  Unimplemented: Currently, the `table` directive also requires an explicit `firstorder` directive, but this restriction will be eliminated in the future.

20 *FLORA-2* System Modules

FLORA-2 provides a number of useful libraries that other programs can use. These libraries are statically preloaded into modules that are accessible through the special `@flora(modname)` syntax, and they are called *system modules*. We describe the functionality of these modules below.

20.1 Pretty Printing

This library provides methods for pretty printing the information about an object or about all objects in a given class. This information can be saved in a file or printed on the screen. This library is preloaded in the system module `pp` and is accessible using the `@flora(pp)` syntax.

To pretty print information about an object, the following calls can be used. The first argument is the user module whose object is to be pretty printed. (Recall that the same object can have completely different sets of properties in different user modules, so the pretty printing methods need to know which set of properties to use.)

- `Class[#pp_class(Module,Outfile)]` — pretty print all objects in `Class` and put the result in `Outfile`.
- `Class[#pp_class(Module)]` — same, but the information is printed to the screen.
- `Obj[#pp_self(Module,Outfile)]` — pretty print `Obj`, send the result to Outfile.
- `Obj[#pp_self(Module)]` — same, but print to the screen.
- `Class[#pp_isa(Module,Outfile)]` — Print the part of the isa hierarchy beneath `Class`.
- `Class[#pp_isa(Module)]` — same, but print to the screen.

The following example illustrates the use of this library:

```
?- flThisModule(M), john[#pp_self(M)]@flora(pp).
```

20.2 Input and Output

This library simplifies access to the most common XSB I/O facilities. This library is preloaded into the system module `io` and can be accessed using the `@flora(io)` syntax.

The purpose of the I/O library is not to replace the standard I/O predicates with *FLORA-2* methods, but rather to relieve the user from the need to do explicit conversion of arguments between the HiLog representation of terms used in *FLORA-2* and the standard Prolog representation used in XSB.[12] However, for uniformity, the `io` library also provides certain methods that do not suffer from the conversion problem.

The library contains two types of I/O operations: *stream-based I/O* and *port-based*. Stream-based I/O is based on the standard Prolog I/O primitives. It uses symbols as file handles. Port-based I/O is specific to XSB. Its file handles are internally represented as numbers. Although stream-based I/O is often easier to use, there are many more port-based primitives that can accomplish various low-level I/O operations. This *FLORA-2* library provides just a few common ones. See the XSB manual, volume 2, for a complete list of these primitives.

The methods and predicates accessible through the `io` library are listed below. Note that some operations are defined as procedural methods and others as predicates. This is because we use the object-oriented representation only where it makes sense — we avoid introducing additional classes and objects that require more typing just for the sake of keeping the syntax object-oriented.

Stream-based I/O.

- `Filename[#see]` — open `Filename` and make it into the current input stream.
- `seeing(Stream)` — binds `Stream` to the current input stream.
- `seen` — closes the current input stream.
- `Filename[#tell]` — opens `Filename` as the current output stream.
- `telling(Stream)` — binds `Stream` to the current output stream.
- `told` — closes the current output stream.
- `write(Obj)` — writes `Obj` to the current output stream.
- Stream[#write(Obj)] — writes `Obj` to the stream `Stream`.
- `writeln(Obj)`, `Stream[#writeln(Obj)]` — same as above, except that the newline character is output after `Obj`.
- `nl` — writes the newline character to the current output stream.
- `read(Result)` — binds `Result` to the next term in the current input stream.
- `Stream[#read(Result)]` — same as above, but use `Stream` as the input stream.

[12] See Section 8 for a discussion of the problems associated with this representation mismatch.

Port-based I/O.

- `Filename[#open(Mode,Port)]` — opens `Filename` with mode `Mode` (which can be `r`, `w`, or `a`) and binds `Port` to the file handle.
- `Port[#close]` — closes the file handle to which `Port` is bound.
- `Port[#read(Result)]` — bind `Result` to the next term in the previously open input `Port`.
- `stdread(Result)` — same, but use the standard input as the port.
- `Port[#write(Result)]` — write `Result` out to the previously open output `Port`.
- `stdwrite(Result)` — same, but use standard output as the port.
- `fmt_write(Format,Term)` — C-style formatted output to the standard output. See the XSB manual, volume 2, for the description of all the options.
- `Port[#fmt_write(Format,O)]` — same, but use `Port` for the output.
- `fmt_write_string(String,Format,Obj` — same as above, but bind `String` to the result. See the XSB manual for the details.
- `fmt_read(Format,Result,Status)` — C-style formatted read from standard input. See the XSB manual.
- `Port[#fmt_read(Format,Result,Status)]` — same, but use `Port` for input.
- `write_canonical(Term)` — write `Term` to standard output in canonic Prolog form.
- `Port[#write_canonical(Term)]` — same, but use `Port` for output.
- `read_canonical(Term)` — read standard input and bind `Term` to the next term in the input. The term *must* be in canonical Prolog form, or else an error will result. This method is much faster than the usual `read` operation, but it is not as versatile, as it assumes that input is in canonical form.
- `Port[#read_canonical(Term)]` — same, but use `Port` for input.
- `readline(Type,String)` — read the standard input and bind `String` to the next line. `Type` is either `atom` or `charlist`. The former means that `String` is to be bound to a Prolog atom and the latter binds it to a list of characters.
- `Port[#readline(Type,String)]` — same, but use `Port` for input.

21 A Note on Programming Style

Programming in *FLORA-2* is similar to programming in Prolog, but is more declarative. For one thing, F-molecules are always tabled, so the programmer does not need to worry about tabling the right predicates. Second, there is no need to worry that a predicate must be declared as dynamic in order to be updatable. Third — and most important — the facts specified in the program are considered to be part of the database. In particular, their order does not matter and duplicates are eliminated automatically.

This has a far-reaching implication on the programming style. In Prolog, it is a common practice to put the catch-all facts at the end of a program block in order to capture subgoals that do not match the rest of the program clauses. For instance,

```
p(f(X)) :- ...
p(g(X)) :- ...
%% If all else fails, simply succeed.
p(_).
```

This will not work in *FLORA-2*, because `p(_)` will be treated as a database fact, which is placed in no particular order with respect to the program. If you want the desired effect, represent the catch-all facts as rules:

```
p(f(X)) :- ...
p(g(X)) :- ...
%% If all else fails, simply succeed. Use the rule notation for p/1.
p(_) :- true.
```

22 Bugs and How to Report Them

The *FLORA-2* system includes a compiler and runtime libraries, but for execution it relies on XSB. Thus, some bugs that you might encounter are the fault of *FLORA-2*, while others are XSB bugs. For instance, a memory violation that occurs during the execution is in all likelihood an internal XSB bug. (*FLORA-2* is like a stress test for XSB — all bugs come to the surface.)

An incorrect result during the execution can be equally blamed on XSB and on *FLORA-2*— it requires a close look at the program. A compiler or a runtime error for a perfectly valid program is probably a bug in the *FLORA-2* system.

Bugs that are the fault of XSB are particularly hard to fix, because *FLORA-2* programs are translated into an unreadable Prolog code. To make things worse, this code might contain calls to *FLORA-2* system libraries.

To simplify bug reporting, *FLORA-2* provides utilities and make the compiled XSB program more readable. The flDump/1 predicate can be used to strip the macros from the code, making it much more palatable. If you issue the following command

```
flora2 ?- flDump(foo).
```

the program `foo.flr` will be compiled without the macros and dump the result in the file `foo_dump.P`. This file is still not very readable, because the program clauses are not properly indented and, to add insult to injury, are represented using the *canonical* Prolog form. For instance, the clause

```
p :- q,r,s.
```

will look as follows:

```
':-'(p,','(q,','(r,s))).
```

Unimplemented: In a future release, a pretty-printing program will be provided to cope with this difficulty.

Reporting XSB bugs. If you find an XSB bug triggered by a *FLORA-2* program, here is a set of guidelines that can simplify the job of the XSB developers and increase the chances that the bug will be fixed speedily:

1. Reduce the size of your *FLORA-2* program as much as possible.
2. Eliminate all calls to the system modules that use the `@flora(lib)` syntax. (Prolog modules that are accessible through the `@prolog(modname)` syntax are OK, but the more you can eliminate the better.)
3. If the program has several user modules, try to put them into one file and use just one module.
4. Use `flDump/1` to strip *FLORA-2* macros from the output of the *FLORA-2* compiler.
5. Use a pretty printing utility to produce a human-readable version of the compiler output. (**Unimplemented**: such a utility will be provided in the future.)
6. See if the resulting program runs under plain XSB system (without the *FLORA-2* shell). If it does not, it means that the program contains calls to *FLORA-2* runtime libraries. Try to eliminate such calls.

   One common library call is used to collect all query answers in a list and then print them out. You can get rid of this library call by finding the predicate `fllibprogramans/2` in the compiled `.P` program and removing it while preserving the subgoal (the first argument) and renaming the variables (as indicated by the second argument). Make sure the resulting program is still syntactically correct!

   If there are other calls to *FLORA-2* runtime libraries, try to delete them, provided that the bug is still reproducible.
7. Try to reduce the size of the resulting program as much as possible. Even eliminating the numerous comment blocks in the dumped `.P` file would be useful.

Finally, remember to include the details of your OS and other relevant information. Some bugs might be architecture-dependent.

Reporting *FLORA-2* bugs. If you believe that the bug is in the *FLORA-2* system rather than in XSB, the algorithm is much more simple:

1. Reduce the size of the program as much as possible by deleting unrelated program clauses and squeezing a multi-module program into just one file.
2. Remove all the calls to system modules, unless such a call that is the essence of the bug.

The current version contains the following known bugs, which are due to the fact that certain features are yet to be implemented:

1. Certain programs (*e.g.*,

   ```
   f[a->b].
   ?- f[].
   ```

   will cause the following XSB error message:

   ++Error: Illegal cut over a tabled predicate ...

   This is due to certain limitations in the implementation of tabled predicates in the XSB system. This problem will be eliminated in a future release of XSB. Meanwhile, as explained in the Introduction, configuring XSB for SLG-WAM and local scheduling will avoid the error.
2. Error messages when *FLORA-2* update predicates contain arithmetic expressions in the query part. This problem will be fixed in the future.
3. Undefined predicates and methods simply fail instead of generating error messages. Generating compile-time error messages in such cases is hard due to the fact that *FLORA-2* programs likely contain meta-programming feature. We will consider this possibility in the future. Generating run-time error messages is much easier and will be incorporated in a future release.
4. If-then-else does not have the right associativity, so explicit parentheses are often required.
5. Multiple types (which represent type intersection in F-logic) are not allowed in set-valued signature expressions: `a[m=>>c,d]` and `a[m*=>>c,d]`. Use `a[m=>>c],a[m=>>d]` for now.
6. Inheritance of procedural methods is not supported: `a[*#p(X)]`.

Appendices

A The *FLORA-2* Debugger

FLORA-2 debugger is implemented as a presentation layer on top of the XSB debugger, so familiarity with the latter is highly recommended (XSB Manual, Part I). Here we sketch only a few basics.

The debugger has two facilities: *tracing* and *spying*. Tracing allows the user to watch the program being executed step by step, and spying allows one to tell *FLORA-2* that it must pose when execution reaches certain predicates or object methods. The user can trace the execution from then on. At present, only the tracing facility has been implemented.

Tracing. To start tracing, you must issue the command `flTrace` at the *FLORA-2* prompt. It is also possible to put the subgoal `flTrace` in the middle of the program. In tat case, tracing will start after this subgoal gets executed. This is useful when you know where exactly you want to start tracing the program. To stop tracing, type `flNoTrace`.

During tracing, the user is normally prompted at the four ports of subgoal execution: `Call` (when a subgoal is first called), `Exit` (when the call exits), `Redo` (when the subgoal is tried with a different binding on backtracking), and `Fail` (when a subgoal fails). At each of the prompts, the user can issue a number of commands. The most common ones are listed below. See the XSB manual for more.

- `carriage return (creep)`: to go to the next step
- `s (skip)`: execute this subgoal non-interactively; prompt again when the call exits (or fails)
- `S (verbose skip)`: like `s`, but also show the trace generated by this execution
- `l (leap)`: stop tracing and execute the remainder of the program

The behavior of the debugger is controlled by the predicate `debug_ctl`. For instance, executing `debug_ctl(profile, on)` at the *FLORA-2* prompt tells XSB to measure the CPU time it takes to execute each call. This is useful for tuning your program for performance. Other useful controls are: `debug_ctl(prompt, off)`, which causes the trace to be generated without user intervention; and `debug_ctl(redirect, foobar)`, which redirects debugger output to the file named `foobar`. The latter feature is usually useful only in conjunction with the aforesaid prompt-off mode. See the XSB manual for additional information on debugger control.

Low-level tracing. *FLORA-2* debugger also supports low-level tracing via the shell command `flTraceLow`. With normal tracing, the debugger converts low-level subgoals to subgoals that are found in the user program and are thus meaningful to the programmer. With low-level tracing, the debugger displays the actual XSB subgoals (of the compiled .P program) that are being executed. This facility is useful for debugging *FLORA-2* runtime libraries.

Note: *FLORA-2* turns on various XSB optimizations. Some (like specialization and unification factoring) can cause certain subgoals to be omitted from the trace or the trace might show subgoals that are not in the original program. In the future, it will be possible to turn these optimizations off for debugging.

B Emacs Support

Editing and debugging *FLORA-2* programs can be greatly simplified with the help of *flora-mode*, a special Emacs editing mode designed specifically for *FLORA-2* programs. Flora-mode provides support for syntactic highlighting, automatic indentation, and the ability to run *FLORA-2* programs right out of the Emacs buffer.

B.1 Installation

To install *flora-mode*, you must perform the following steps. Put the file

```
XSB/packages/flora2/emacs/flora.el
```

found in your XSB distribution on the load path of Emacs or XEmacs (whichever you are using). The best way to work with Emacs is to make a separate directory for Emacs libraries (if you do not have one), and put `flora.el` there. Such a directory can be added to emacs search path by putting the following command in the file `~/.emacs` (or `~/.xemacs`, if you are running one of the newer versions of XEmacs):

```
(setq load-path (cons "your-directory" load-path))
```

It is also a good idea to compile emacs libraries. To compile flora.el, use this:

```
emacs -batch -f batch-byte-compile flora.el
```

This will produce the file `flora.elc` — a compiled byte code. If you are using XEmacs, use `xemacs` instead of `emacs` above — the two emacsen use incompatible byte code, and you cannot use `flora.elc` compiled under one system for editing files under another.

Finally, you must tell X/Emacs how to recognize *FLORA-2* program files, so Emacs will be able to invoke the Flora major mode automatically when you are editing such files:

```
(setq auto-mode-alist (cons '("\\.flr$" . flora-mode) auto-mode-alist))
(autoload 'flora-mode "flora" "Major mode for editing Flora programs." t)
```

To enable syntactic highlighting of Emacs buffers (not just for *FLORA-2* programs), you can do the following:

- In Emacs: select `Help.Options.Global Font Lock` on the menubar. To enable highlingting permanently, put

```
(global-font-lock-mode t)
```

in `~/.emacs`.

- In XEmacs: select `Options.Syntax Highlighting.Automatic` in the menubar. To enable this permanently, put

```
(add-hook 'find-file-hooks 'turn-on-font-lock)
```

in `~/.emacs` or `~/.xemacs` (whichever is used by your XEmacs).

B.2 Functionality

Menubar menu. Once *FLORA-2* editing mode is installed, it provides a number of functions. First, whenever you edit a *FLORA-2* program, you will see the "Flora" menu in the menubar. This menu provides commands for controlling the Flora process (i.e., XSB with the *FLORA-2* shell). You can start and stop this process, type queries to it, and you can tell it to consult regions of the buffer you are editing, the entire buffer, or some other file.

Because Emacs provides automatic file completion and allows you to edit what you typed, performing these functions right out of the buffer takes much less effort than typing the corresponding commands to the *FLORA-2* shell.

Keyboard functions. In addition to the menu, *flora-mode* lets you execute most of the menu commands using the keyboard. Once you get the hang of it, keyboard commands are much faster to invoke:

```
Load file:                 Ctl-c Ctl-f
Load file dynamically:     Ctl-u Ctl-c Ctl-f
Load buffer:               Ctl-c Ctl-b
Load buffer dynamically:   Ctl-u Ctl-c Ctl-b
Load region:               Ctl-c Ctl-r
Load region dynamically:   Ctl-u Ctl-c Ctl-r
```

When you invoke any of the above commands, a *FLORA-2* process is started, unless it is already running. However, if you want to invoke this process explicitly, type

```
ESC x run-flora
```

You can control the *FLORA-2* process using the following commands:

```
Interrupt Flora Process:     Ctl-c Ctl-c
Quit Flora Process:          Ctl-c Ctl-d
Restart Flora Process:       Ctl-c Ctl-s
```

Interrupting *FLORA-2* is equivalent to typing `Ctl-c` at the *FLORA-2* prompt. Quitting the process stops XSB, and restarting the process shuts down the old XSB process and starts a new one with *FLORA-2* shell running.

Indentation. Flora editing mode understands some aspects of the $\mathcal{F}$LORA-2 syntax, which enables it to provide correct indentation of program lines (in many cases). In the future, flora-mode will know more about the syntax, which will let it provide even better support for indentation.

The most common use of $\mathcal{F}$LORA-2 indentation facility is by typing the `TAB`-key. If *flora-mode* manages to understand where the cursor is, it will indent the line accordingly. Another way is to put the following in your emacs startup file (`~/.emacs` or `~/.xemacs`):

```
(setq flora-electric t)
```

In this case, whenever you type the return key, the next line will be indented automatically.

C Inside $\mathcal{F}$LORA-2

C.1 How $\mathcal{F}$LORA-2 Works

As an F-logic-to-XSB compiler, $\mathcal{F}$LORA-2 first parses its source file, compiles it into XSB syntax and then outputs XSB code. For instance the command

```
flora ?- flLoad(myprog).
```

compiles the $\mathcal{F}$LORA-2 program `myprog.flr` and generate the following files: `myprog.P`, `myprog_main.O`, and `myprog.fdb` (if `myprog.flr` contains F-logic facts). By default, `flLoad(myprog)` loads the program into the default user module named `main`. If `myprog.flr` contains F-logic facts, all these facts will be compiled separately into the XSB file `myprog.fdb` that is dynamically loaded at runtime. Next, the file `myprog.P` is generated — take a look at "`myprog_main.P`" to see what has become of your $\mathcal{F}$LORA-2 program! — and passed to the XSB compiler, yielding Prolog byte code `myprog.O`, which is then renamed to `myprog_main.O`. This file is then loaded and executed. If `myprog.flr` contains queries, they are immediately executed by XSB (provided there are no errors).

In the module system of $\mathcal{F}$LORA-2, the same program can be loaded into any user module. The same program can even be loaded into two different modules at the same time, in which case there will be two distinct copies of the same program running at the same time. For each user module, a different byte code is generated (this is why `myprog.O` was renamed into an object file that contains the module name as part of the file name).

The main purpose of the $\mathcal{F}$LORA-2 shell is to allow the evaluation of ad-hoc F-logic queries. For example, after consulting and loading the the file `default.flr` from the demo directory by launching the command `flora2 ?- flDemo(default).`, pose the following query and see what happens.

```
flora2 ?-  X..kids[                      % Whose kids
                    self -> K,           % ... (list them by name)
                    hobbies ->>          % ... have hobbies
                      H:dangerous_hobby  % ... that are dangerous?
           ].
```

$\mathcal{F}$*LORA-2* **compilation.** The basic idea behind the implementation of F-logic by translating it into predicate calculus is described in [9]. It consists of two parts: translation of F-molecules into various kinds of Prolog predicates, and addition of appropriate "closure rules" that implement the object-oriented semantics of the logic.

Consider, for instance, the following complex F-molecule, which represents some facts about the object `mary`:

```
mary:employee[age->29, kids->>{tim,leo}, salary(1998)->100000].
```

As described in [9], any complex F-molecule can be decomposed into a conjunction of simpler F-logic atomic formulas. These latter atoms can be directly represented using Prolog syntax. For different kinds of F-logic atoms we use different Prolog predicates. For instance, the result of translating the above F-molecule might be:

```
isa(mary,employee).            % mary:employee.
fd(mary,age,29).               % mary[age->29].
mvd(mary,kids,tim).            % mary[kids->>{tim}].
mvd(mary,kids,leo).            % mary[kids->>{leo}].
fd(mary,salary(1998),100000).  % mary[salary(1998)->a_lot].
```

The `fd` predicate is used to encode scalar attributes and methods and `mvd` is used for set methods. The predicates `isa` ans `sub` encode the IS-A and subclass relationships, respectively. Of course, $\mathcal{F}$*LORA-2* has much more: signatures, inheritable and non-inheritable methods, directives, and all kinds of auxiliary predicates needed to improve efficiency. The following diagram shows the main predicates involved and their dependency:

```
 fd <-                            inferred_fd
 fd <- tnot defined_inferred_fd <- inferred_fd <- base_fd
                                  inferred_fd <- derived_fd <- fd
 fd <- tnot conflict_obj_ifd <- ifddef <- fd
                                ifddef <- ifd
 fd <- immediate_isa
 fd <- ifd
ifd <- inferred_ifd
ifd <- immediate_sub
ifd <- tnot defined_inferred_ifd
ifd <- tnot conflict_ifd
```

Here we listed only the predicates that are used to model scalar inheritable (`ifd`) and non-inheritable (`fd`) methods. A similar diagram exists for set methods (which uses `imvd` and `mvd`), and another one exist for method signatures. There is additional machinery for IS-A and subclass relationships, and for equality maintenance.

The closure axioms tie all these predicates together to implement the semantics of F-logic. In particular, they take care of the following features:

- Computing the transitive closure of "`::`" (the subclass relationship). A runtime check warns about cycles in the subclass hierarchy.
- Computing the closure of "`:`" with respect to "`::`", i.e., if $X:C, C::D$ then $X:D$.
- Performing monotonic and non-monotonic inheritance.
- Making sure that scalar methods are, indeed, scalar.

Templates for the files that implement these axioms reside in the subdirectory `closure/` and have the suffix `.fli`. These files are called *trailers* because they are typically included at the end of the compiled program. There are three kinds of trailers: the no-equality trailer (the file `closure/flrtrailer.fli`), which maintains no equality, the basic trailer (`closure/flreqltrailer.fli`), which maintains only the standard equality axioms, and the trailer (`closure/flrscalareql.fli`), which, in addition, maintains the F-logic axioms for scalar methods.

When a $\mathcal{F}$LORA-2 program is compiled, the compiler includes the trailers into the `.P` file. However, there also is a need to be able to load the trailers dynamically. First, this is needed in the system shell, because the shell is not represented by any particular user program and so there is no place where we can include the trailer. Second, the user might enter the executable instruction

```
?- equality {basic|flogic|none}
```

at the shell prompt and user programs can contain these instructions as part of their code. Precompiled, loadable trailers for the default module `main` are stored in the `trailer/` subdirectory. This is done at the system build time. When an equality maintenance instruction is executed for a particular module, the trailer for that module must be compiled dynamically. (The need for this compilation will become clear after reading about the implementation of the module system.) These trailers are stored in the user home directory in the subdirectory `.xsb/flora/`.

The above is a much simplified picture of the inner-workings of $\mathcal{F}$LORA-2. The actual translation into Prolog and the form of the closure rules is very complex. Some of this complexity exists to ensure good performance. Other complications come from the need to provide a module system and integrate it into the XSB system. The module system serves two purposes. First, it promotes modular design for $\mathcal{F}$LORA-2 programs, making it possible to split the code into separate files and import objects defined in other modules. Second, it allows $\mathcal{F}$LORA-2 programs to communicate with XSB by using the predicates defined in XSB programs and letting XSB programs use $\mathcal{F}$LORA-2 objects. Some of these implementation issues are described in [12].

The module system. The module system is implemented by providing separate namespaces for the various predicates used to encode F-logic formulas. First, all predicates have a weird prefix to make clashes with other XSB programs unlikely. The prefix is defined in `includes/header.flh` and currently is `_$_$_flora`. The user, of course, does not need to worry about it, unless she runs $\mathcal{F}$LORA-2 programs in a very unfriendly XSB environment. In this case, the prefix can be made even harder to match.[13]

[13] It is necessary to ensure that the resulting predicate names are symbol strings acceptable to the XSB compiler. Look at the macros `FLORA_WORKSPACE` and `FLORA_WSSTORAGE` to see what is involved.

Apart from the general prefix, each predicate name's prefix contains the module name where this predicate is defined. Since the same F-logic program can be loaded into different modules, the $\mathcal{F}$LORA-2 compiler does not actually know the real names of the predicates it is producing. Instead, it dumps code where each predicate is wrapped with a preprocessor macro. For instance, the predicate `fd` would be dumped as

```
FLORA_WORKSPACE(FLORA_VAR_WORKSPACE,fd)
```

When the program, `myprog.P`, compiled by the $\mathcal{F}$LORA-2 compiler needs to be loaded into a user module, say `main`, the XSB's preprocessor, `gpp`, is called with the macro `FLORA_VAR_WORKSPACE` set to `main`. Gpp replaces all macros with the actual values, includes the necessary files, and then pipes the result to the XSB compiler. The latter produces the object `myprog.O` file where all the predicate names are wrapped with the user module name, as described above. This object file is renamed to `myprog_main.O`. If later we need to compile `myprog.P` for another user module, `foo`, `gpp` is called again, but this time it sets `FLORA_VAR_WORKSPACE` to `foo`. When XSB finally compiles the program into the object file, the file is renames to `myprog_foo.O`.

It is important to keep in mind that only the predicate names are wrapped with the `FLORA_PREFIX` macro and a module name. Predicate arguments are not wrapped and thus, the space of object Ids is shared among modules. However, this is not a problem and, actually, is very convenient: we can easily refer to objects defined in other modules and yet the same object can have completely different sets of properties in each separate module. This does not preclude the possibility of encapsulating objects, because only the methods need to be encapsulated — oids do not carry any meaning by themselves.

To provide encapsulation for HiLog predicates, they are also prepended with the module name. In particular, this implies that HiLog atomic formulas have different representation than HiLog terms: a formula `p(a,f(b))` would be encoded as

```
FLORA_WORKSPACE(FLORA_VAR_WORKSPACE,apply)(p,a,FLORA_PREFIX'apply(f,b))
```

The same term would be encoded differently if it occurs as an argument of a predicate of another functor:

```
FLORA_PREFIX'apply(p,a,FLORA_PREFIX'apply(f,b))
```

Thus, $\mathcal{F}$LORA-2 implements a 2-sorted version of HiLog [4].

The updatable part of the database. All objects and facts that are explicitly inserted by the program are kept in the special *storage trie* associated with the user module where the program is loaded. A tries is a special data structure, which is well-suited for indexing tree-structured objects, like Prolog terms. This workhorse does much of the grudge work in the XSB engine. To manipulate the storage tries, $\mathcal{F}$LORA-2 uses the XSB package called `storage.P`, which is described in the XSB manual. This package was originally created to support $\mathcal{F}$LORA-2, but it has independent uses as well.

All primitives in this package take a Prolog symbol, called a *triehandle*, a Prolog term, and some also return status in the third argument. Here are some of the most relevant predicates:

```
storage_insert_fact(Triehandle,Term,Status)
storage_delete_fact(Triehandle,Term,Status)
storage_insert_fact_bt(Triehandle,Term,Status)
storage_delete_fact_bt(Triehandle,Term,Status)
```

The first two methods insert and delete in a non-backtrackable manner, while the last two are backtrackable.

$\mathcal{F}$LORA-2 associates a separate triehandle (and, thus, a separate trie) with each module. The mechanism is similar to that used for predicate names:

```
FLORA_WSSTORAGE(FLORA_VAR_WORKSPACE)
```

As explained earlier, when XSB compiles the Prolog file generated by the $\mathcal{F}$LORA-2 compiler, the macro `FLORA_VAR_WORKSPACE` gets replaced with the module name and out comes a unique, hard to replicate triehandle.

Unfortunately, putting something in a trie does not mean that XSB will find it there automatically. That is, if you insert `p(a)` in a trie, it does not mean that the query `?- p(a)` will evaluate to true, and this is another major source of complexity that the $\mathcal{F}$LORA-2 compiler has to deal with. To find out if a term exists in a trie, we must use the primitive

```
storage_find_fact(Triehandle,Term)
```

If the term exists in the trie identified by its triehandle, then the predicate succeeds; if the term does not exist, then it fails. The above primitive can be used to query tries in a more general way, with the second variable unbound. In this case, we can backtrack through all the terms that exist in the trie.

Suppose we insert a fact, `a[m->v]`, represented by the formula `fd(a,m,v)`. Since this formula is inserted in the trie and XSB knows nothing about it, we need to connect the trie to XSB through a rule like this:

```
fd(O,M,V) :- storage_find_fact(triehandle,f(O,M,V)).
```

Of course, the name of the triehandle and the predicate names must be generated using the macros, as described above, so that they could be used for any module. In $\mathcal{F}$LORA-2 such rules are called *patch rules*.

Since F-logic uses only about a dozen of predicates to represent F-molecules, we can create such rules statically and let gpp wrap them with the appropriate prefixes on the fly. Even for HiLog we can create such rules statically. After all, XSB limits us to functor arity no larger than 255! The problem arises with first-order predicates, because they need to be collected by the compiler and a separate rule needs to be created for each first-order predicate. The static patch rules are located in `genincludes/flrpatch.fli` (from which `flrpatch.flh` is generated during the installation). The patch rules for the first-order predicates are dumped directly into the `.P` file of the main program generated by the *$\mathcal{F}$LORA-2* compiler.

A similar problem arises when the user enters a `firstorder` directive to the shell. In this case, a patch rule must be generated dynamically and asserted into the system.

One complication is that the file `flrpatch.flh` must be compiled for the right module, so the preprocessor macro `FLORA_VAR_WORKSPACE` can be given value only when we load the patch rules into a specified module. If the directory `genincludes` that holds `flrpatch.flh` is not writable by the user (which is often the case, if XSB is installed in a shared public directory on the server) then it will not be possible to run any *$\mathcal{F}$LORA-2* programs at all. For this reason, a copy of the patch rules is stored in the user's home directory, `.xsb/flora/`, in the file `patch.P`, so patch rules are always compiled in the user's home directory.

C.2 System Architecture

The overall architecture of *$\mathcal{F}$LORA-2* is depicted in Figure 2. The program is first tokenized and then the *composer* combines the disparate tokens into terms. Since, due to the existence of operators, not everything looks like a term in the source program, the composer consults the operator definitions in the file `flroperator.P` to get the directives on how to turn the operator expressions into terms. Next, the parser checks the syntax of the rules and of the various other primitives (*e.g.*, the aggregates, updates, module specifications, etc.). The output of the parser is a *canonic term* list, which represents the entire parsed program. The canonic term is taken up by the intermediate code generator, which generates abstract code. This code is represented in a form that is convenient for manipulation and is not yet Prolog code. The compiler might add additional rules (such as patch rules) and XSB instructions. The compiled program is converted into (almost) Prolog syntax by the coder. As mentioned previously, the code produced by the compiler is full of preprocessor macros, so before passing it to XSB it must be preprocessed by GPP. GPP pipes the result to XSB, which finally produces the byte code program that can run under the control of the XWAM abstract machine, which is part of the XSB system.

The following is a list of the key files of the system.

- `flrshell.P`: The top level XSB module that implements the *$\mathcal{F}$LORA-2* shell — a subsystem for accepting user commands and queries and passing them to the compiler. See Section 2 for a full description of shell commands.
- `flrlexer.P`: The *$\mathcal{F}$LORA-2* tokenizer.
- `flrcomposer.P`: The *$\mathcal{F}$LORA-2* composer, which parses tokens according to the operator grammar and does other magic.
- `flrparser.P`: The *$\mathcal{F}$LORA-2* parser.
- `flrcompiler.P`: The generator of the intermediate code.
- `flrcoder.P`: The *$\mathcal{F}$LORA-2* coder, which generates XSB code.

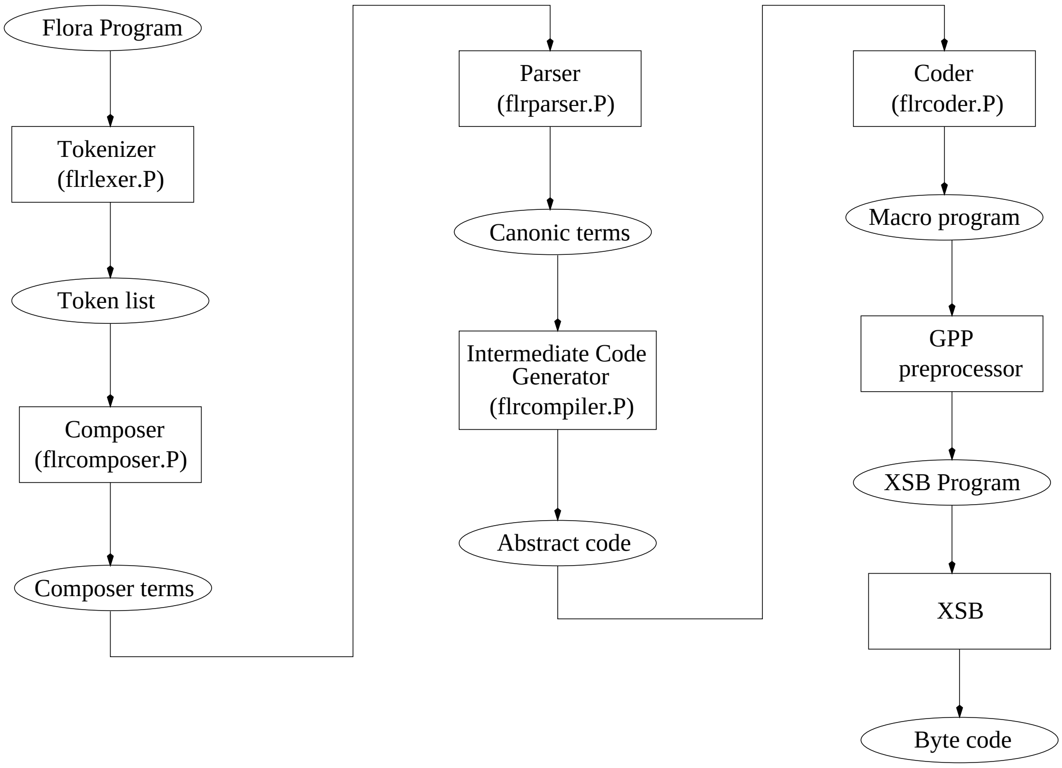

Figure 2: The architecture of the $\mathcal{F}LORA$-2 system.

- `flrutils.P`: Miscellaneous utility predicates for loading programs, checking if files exist, whether they need to be recompiled, etc.

Additional system libraries are located in the `syslib/` subdirectory. These include the various printing utilities, implementation for aggregates, update primitives, and some others. The compiler determines which of these libraries are needed while parsing the program. When a library is needed, the compiler generates an `#include` statement to include an appropriate file in the `syslibinc` directory. For instance, to include support for the `avg` aggregate function, the compiler copies the file `syslibinc/flraggavg_inc.flh` to the output `.P` file. Since `syslibinc/flraggavg_inc.flh` contains the code to load the library `syslib/flraggavg.P`, this library will be loaded together with that output file. The association between the libraries and the files that need to be included to provide the appropriate functionality is implemented in the file `flrlibman.P`, which also implements the utility used to load the libraries.

While `syslib/` directory contains the libraries implemented in Prolog, the `lib/` directory contains libraries implemented in $\mathcal{F}LORA$-2 itself. Apart from that, the two types of libraries differ in functionality. The libraries in `syslib/` implement the primitives that are part of the syntax of the $\mathcal{F}LORA$-2 language itself. In contrast, the libraries in `lib/` are utilities that are part of the system, but not part of the syntax. An example is the pretty-printing library. Methods and predicates defined in the libraries in `lib/` are accessible through the `@flora(lib-name)` system module and (unlike user modules) they are loaded automatically at startup.

There are several subdirectories that hold the various files that contain definitions included at compile time. These will be described in a technical document.

A number of other important directories contain the various included files (many of which include other files). The directory `flrincludes/` contains the all-important `flora_terms.flh` file, which defines all the names used in the system. These names are defined as preprocessor macros, so that it would be easy to change them, if necessary. The directory `genincludes/` currently contains the already mentioned patch rules. The file `flrpatch.fli` is a template, and `flrpatch.flh`, which contains the actual patch rules, is generated from `flrpatch.fli` during the installation.

The directory `includes/` contains the header file, which defines the macros (*e.g.*, `FLORA_WORKSPACE`) that wrap all the names with prefixes to separate the different modules of the user program. The directory `headerinc/` is another place where the template files are located. Each of these files contains just a few `#include` statements, mostly for the files in the `closure/` directory (which, if you recall, contains pieces of the trailer). When the system is installed certain combinations of these files are concatenated and dumped into the `trailer/` directory. Recall that the files in the `trailer/` directory are trailers that implement the closure axioms (there are three trailers: without equality, with standard equality, and with F-logic equality). This double-indirection is needed to simplify the installation procedure and to eliminate code duplication among the various trailers.

The directory `p2h` contains (the only!) C program in the system. It implements conversion of Prolog terms to HiLog and back. Finally, the `pkgs/` directory is empty. Some day it will contain add-on programs, such as Internet access, etc.

References

[1] A.J. Bonner and M. Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[2] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[3] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[4] W. Chen and M. Kifer. Sorted HiLog: Sorts in higher-order logic programming. Number 893 in Lecture Notes in Computer Science, January 1995.

[5] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[6] K. Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.

[7] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *VLDB*, pages 273–284, 1994.

[8] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 393–402, New York, June 1992. ACM.

[9] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, July 1995.

[10] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *ACM Principles of Database Systems*, pages 1–10, New York, 1989. ACM.

[11] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[12] G. Yang and M. Kifer. Implementing an efficient DOOD system using a tabling logic engine. In *First International Conference on Computational Logic, DOOD-2000 Stream*, July 2000.

Index