FLORID

User Manual
Version 3.0 (FloXML)

Wolfgang May
may@informatik.uni-freiburg.de

Institut für Informatik, Universität Freiburg
Germany

October 2000

This manual is based on the former versions by Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Christian Schlepphorst, and Heinz Uphoff.

Contents

Preface

With FLORID (F-LOgic Reasoning In Databases) an implementation of a programming system based on the concepts of F-Logic is presented. Proposed by Kifer, Lausen and Wu [KLW95], F-Logic is designed as a logical language accounting in a clean, declarative fashion for most of the structural aspects of object-oriented data modeling. Contrasting other approaches, e.g., [Law93], GULOG [DT95] and ROL [Liu96], nearly all of the distinctive F-Logic features are realized in FLORID. In particular we emphasize that the system supports data driven schema definition, multiple, non–monotonic inheritance and furthermore path expressions [FLU94] which can also be used for anonymous object creation. The evaluation of programs is based on a set-oriented bottom-up computation, as an extension of the algorithm well known from Datalog [AHV95, CGT90, Ull89]; also a semi-naive evaluation component is provided. With version 2.0, FLORID has been extended with Web access (see tutorial [FHM$^+$00] for details).

The FLORID system was developed at the universities of Mannheim and Freiburg as part of a research project granted by the DFG (Deutsche Forschungsgemeinschaft) under leadership of Georg Lausen.

The aim of this manual is to demonstrate the usage of the FLORID system. We assume that the reader is familiar with the basic notions of F-Logic. For details about the language and data modeling with F-Logic the reader is referred to the FLORID programming tutorial [FHM$^+$00] and the F-Logic report [KLW95].
The structure of this manual is as follows:

The first part describes the installation and the environment of FLORID: First, we give describe the installation of Florid for working from the unix shell or from within emacs. Section 2 gives a short example of how to run FLORID and in the unix shell. Working with FLORID in emacs is described in Section 3.

The second part serves as user manual to FLORID for everyday's use as F-Logic engine: Section 4 is a manual to programming with FLORID from the user's point of view, describing the evaluation of programs and the main system commands for user interaction and controlling evaluation. Section 5 describes how the output of FLORID can be handled. Section 6 illustrates how programs can be debugged.

The third part addresses the more involved users which want to experiment with FLORID, including changing its behavior (nevertheless, it should also be of interest for "simple" users): Section 7 describes command-line options for invoking FLORID as an experimental system. Section 8 describes the system itself, its object-oriented design, and the deeper background of system commands. Also, it is explained how the user can change the configuration of an experimental system.

Recent Changes

Versions 2.1 and 2.2 contain internal modifications (enabling compilation with glibc6/egcs) and enhanced installation and compilation features. Versions 2.3 to 3.0 stepwise implemented the XML functionality (for documentation, see [May00]).

1 Installation

1.1 Installation of the Source Distribution

Not yet available.

1.2 Installation of the Binary Distribution

The binary distribution of FLORID comes as a packed and compressed file
`florid-<version-number>-<operating-system>.tar.gz`.
In the following we refer to this file simply as `florid.tar.gz`. The file is uncompressed by the command

```
gunzip florid.tar.gz
```

The resulting file `florid.tar` has to be unpacked by entering

```
tar -xvf florid.tar
```

Now the directory `florid` is created. It has the following subdirectories:

```
florid/bin/
florid/environment/
florid/sgml/
florid/doc/
florid/examples/
florid/examples/tutorial/
```

The directory `florid/bin` contains the binary, `florid/environment` contains several files defining the FLORID environment:

- `config.flp.in`: the source of the configuration file (see Section 8.4)
- `default.his`: history lines to preload
- `flp.el`: the emacs flp-mode definition file

configure FLORID

First, the FLORID configuration has to be adapted to the local system by changing to the directory `florid/environment/` and calling `./configure` – this generates `config.flp` from `config.flp.in`.

The directory `florid/sgml` contains several definitions needed when using SGML documents. The directory `florid/doc` contains postscript files of the user manual (`manual.ps`) and the FLORID tutorial (`tutorial.ps`). Additional publications related to F-Logic and FLORID are available from `http://www.informatik.uni-freiburg.de/~dbis/Publications/`. In `florid/examples` a number of F-Logic example programs are found. For the examples from the tutorial there is an extra subdirectory, `florid/examples/tutorial`.
Please send enquiries, comments, suggestions, and bug reports to

`florid@informatik.uni.freiburg.de`

1.3 Environment Variables

Shell environment variables are used to set paths leading to FLORID's configuration files which are needed when Florid is called. It is also possible to specify the paths by command line options when calling FLORID (see Sec. 7). The following variables should be set before starting FLORID:

- DEFAULTCFG
- DEFAULTHIS

DEFAULTCFG tells how to find the configuration file and DEFAULTHIS points to the history file to preload. The configuration file is a sequence of system commands that create the objects needed for a working system and then pass control to the user. See Section 8.4 for details.

If one of these variables is not set and the respective command line option is missing, the system will print a warning. In case of a missing DEFAULTCFG, the FLORID system has to be built "by hand".

UNIX Environment Variables

In the following we assume that FLORID's main directory `florid/` is located at `/home/db/`. Then, the environment variables have to be set to

- DEFAULTCFG : `/home/db/florid/environment/config.flp`
- DEFAULTHIS : `/home/db/florid/environment/default.his`

You can set the variables either directly from the shell or automatically in an operating system file; e.g., for the *bash* shell add the commands

```
export DEFAULTCFG="/home/db/florid/environment/config.flp"
export DEFAULTHIS="/home/db/florid/environment/default.his"
```

to `.bashrc`. If you want to check if your settings are working, proceed with Section 2 now for a first example.
The XML functionality [May00] is only available when the environment variables

`SP_ENCODING=XML` and
`SGML_CATALOG_FILES=/home/db/florid/sgml/xml.soc`

are set.

1.4 Settings for Florid in Emacs

Additionally to the shell, emacs provides a very user-friendly interface to FLORID via the emacs flp-mode. The emacs flp-mode is defined in the file `flp.el` which is located in in `florid/environment/`. To make emacs load and use this mode, the local `.emacs` file has to be extended by the following lines:

.emacs: flp mode

```
;;; enter flp mode if a file with the suffix ".flp" is loaded
(setq auto-mode-alist (cons '("\.flp$" . flp-mode) auto-mode-alist))

;;; autoload "flp.el" if the functions flp-mode or run-flp are executed
(autoload 'flp-mode "flp" "" t)
(autoload 'run-flp "flp" "" t)
```

To be sure that emacs actually finds `flp.el`, either the lines

```
(setq load-path
        (cons "/home/db/florid/environment/" load-path))
```

have to be added to the `.emacs` file or `flp.el` has to be put into a directory where emacs looks for files.

.emacs: flp mode

In `flp.el` it has to be specified where to find the `florid` executable. If the `florid/bin/` directory is in the binary search path,

```
(defvar flp-program-name "florid"
      "*Program name for invoking an inferior Flp with ‘run-flp’.")
```

is sufficient. Otherwise, change it to the actual path.

history file

Additionally, `florid/environment/default.his` has to be copied to `~/.florid-history` for the private readline history. After these installation steps, emacs has to be started again to let the changes take effect.

2 Florid in the UNIX Shell

Florid is started from the shell by simply typing `florid` at the unix prompt. When Florid is started in the unix shell, command line options can be given (see Section 7); e.g., `florid -v` yields the version number. To give an impression of the Florid system and its usage we present a short example session in the unix shell. More example sessions are included in appendix C.
Let the file `first_example.flp` contain the following program:

```
   eagle::bird[fly *-> yes;
               brood *-> eggs].
   bob : eagle.
```

(which defines that *eagles* are *birds*; birds fly and lay eggs. Additionally, *bob* is an eagle). We want to evaluate the program `first_example.flp` and query the database which is defined by this program database (for the program's semantics, see [FHM+00]).

```
naxos:~/florid/bin> florid                  invoke FLORID from the shell
This is Florid
Type 'sys.help.' for further information.
?- sys.consult@("first_example.flp").       system command to load file
?- sys.eval.                                start evaluation of current program
?- X[fly->yes].                             logical query to the model computed
Answer to query : ?- X[fly -> yes].         answer set is printed
X/bob
1 output(s) printed
?- sys.end.                                 quit FLORID
Bye
naxos:~/flogic/bin>
```

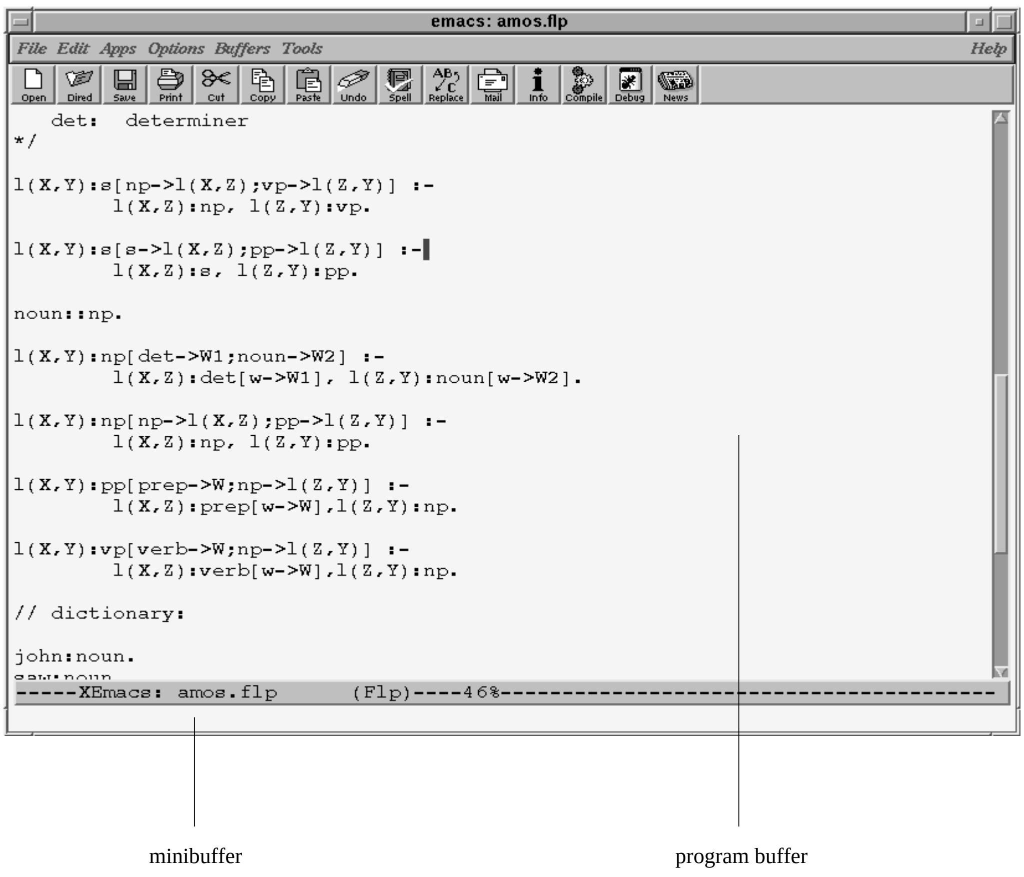
In the above example, Florid entered the interactive mode after having processed the command line. Now the prompt "`?-`" is displayed, indicating that queries can be entered. The queries starting with "`?- sys.`" are system commands which allow user interaction with the system (see Section 4). Here, the program `first_example.flp` is loaded by `?- sys.consult@(‘‘first\_exa`
Then, it is evaluated by `?- sys.eval.` Other queries refer to the evaluated model; in this

case, the answer set is printed (stating that *bob* is the only object which flies). See Section 8.1 for a detailed description of the interactive mode. To leave FLORID, enter "`?- sys.end.`".

If FLORID is started with the option `-q`, it quits after execution of the command line instead of entering the interactive mode. This is helpful if the user wants to call florid in batch mode, e.g., from a shell script: `florid -q test.flp`.

3 Running Florid in Emacs

Running the system from the emacs not only offers high-level editing facilities but also integrates FLORID into an environment where all kinds of tools (mailreader, newsreader, several compilers, TeX etc.) are used in a uniform way. In order to use FLORID from emacs, the configuration steps in Section 1.4 must have been executed. When a file ending with `.flp` is loaded into emacs, emacs automatically enters the F-Logic mode (Figure 1). For editing, the flp-mode provides syntax highlighting facility to make F-Logic programs, using the font-lock package shipped with XEmacs 19.13 or higher (it is recommended to set Syntax Highlighting:Colors and Auto-Fontify in the XEmacs Options menu). If font-lock is not available, syntax highlighting does not work.


emacs: amos.flp
File Edit Apps Options Buffers Tools Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
minibuffer
program buffer

```
   det:  determiner
*/

l(X,Y):s[np->l(X,Z);vp->l(Z,Y)] :-
        l(X,Z):np, l(Z,Y):vp.

l(X,Y):s[s->l(X,Z);pp->l(Z,Y)] :-
        l(X,Z):s, l(Z,Y):pp.

noun::np.

l(X,Y):np[det->W1;noun->W2] :-
        l(X,Z):det[w->W1], l(Z,Y):noun[w->W2].

l(X,Y):np[np->l(X,Z);pp->l(Z,Y)] :-
        l(X,Z):np, l(Z,Y):pp.

l(X,Y):pp[prep->W;np->l(Z,Y)] :-
        l(X,Z):prep[w->W],l(Z,Y):np.

l(X,Y):vp[verb->W;np->l(Z,Y)] :-
        l(X,Z):verb[w->W],l(Z,Y):np.

// dictionary:

john:noun.
saw:noun
-----XEmacs: amos.flp      (Flp)----46%---------------------------------------
```

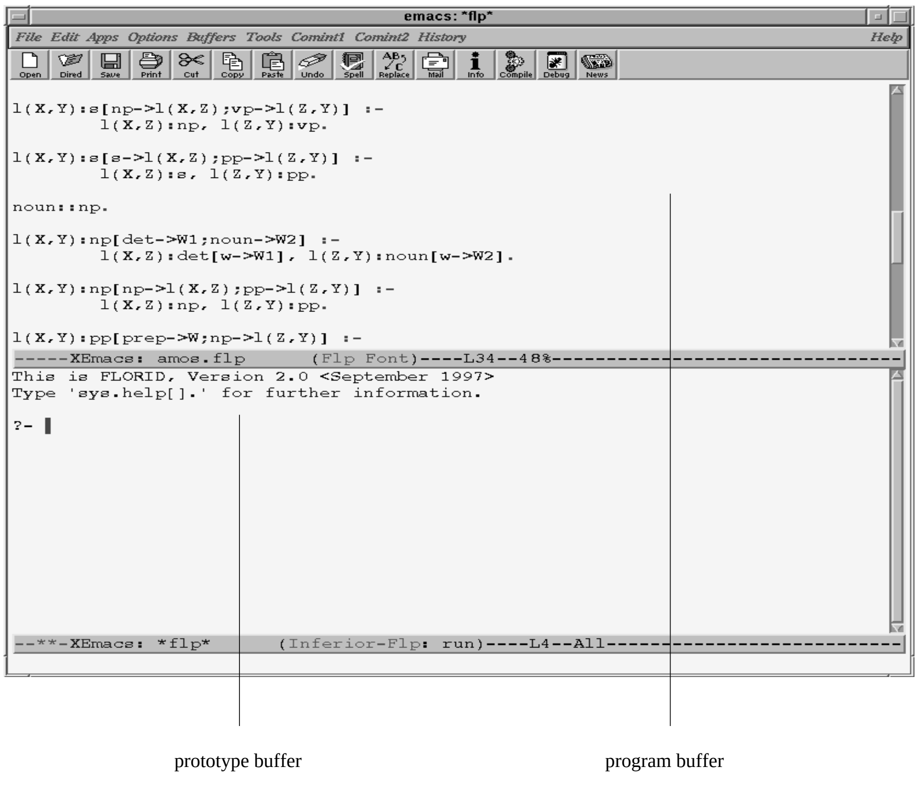
Figure 1: XEmacs 19.13

Additionally to editing capabilities, the flp-mode defines several key codes for interaction between the editor and FLORID (see Table 1).

`C-c C-c`	Reset System and consult buffer as file
`C-c C-a`	Consult additional (same as `C-c C-c` but without system reset)
`C-c C-r`	Consult marked region (without reset)
`C-c C-s`	System reset (clear OM and program)
`C-c C-b`	Consult whole buffer as region (without reset)
`C-c C-l`	Display buffer *flp* and jump there
`C-`	Break evaluation or output
`C-c C-i`	Quit FLORID process

Table 1: Special keycodes in the flp-mode

When the key combination `Control-c Control-c` is pressed in the program buffer, a new buffer is created in the lower half of the current buffer. In this buffer FLORID is started and consults and evaluates the program given in the program buffer (see Figure 2). Then, the user can interact in the same way with FLORID as described before for the unix shell. Additionally, it is possible to step through the history by Meta-P (Previous) and Meta-N (Next). The difference between `C-c C-a` and `C-c C-b` is that in the second case, the buffer's contents is not saved before calling FLORID.


emacs: *flp*
File Edit Apps Options Buffers Tools Comint1 Comint2 History Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
prototype buffer
program buffer

```
l(X,Y):s[np->l(X,Z);vp->l(Z,Y)] :-
        l(X,Z):np, l(Z,Y):vp.

l(X,Y):s[s->l(X,Z);pp->l(Z,Y)] :-
        l(X,Z):s, l(Z,Y):pp.

noun::np.

l(X,Y):np[det->W1;noun->W2] :-
        l(X,Z):det[w->W1], l(Z,Y):noun[w->W2].

l(X,Y):np[np->l(X,Z);pp->l(Z,Y)] :-
        l(X,Z):np, l(Z,Y):pp.

l(X,Y):pp[prep->W;np->l(Z,Y)] :-
-----XEmacs: amos.flp      (Flp Font)----L34--48%--------------------------------
This is FLORID, Version 2.0 <September 1997>
Type 'sys.help[].' for further information.

?- 
--**-XEmacs: *flp*       (Inferior-Flp: run)----L4--All----------------------------
```

Figure 2: XEmacs 19.13 in flp mode

If the edited file does not end with `.flp`, the system can be started manually by the command `Meta-x run-flp` (when pressing `Meta-x` the emacs cursor jumps into the minibuffer where the command is entered). Then again a new buffer is created.

4 Programming with Florid

Programs are collections of facts and rules, similar to Prolog programs. In addition to *logical* facts and rules, *system commands* can be used for user interaction with FLORID.
The evaluation of programs is based on two concepts:

- The *ObjectManager (OM)* represents a set of derived facts. The object manager can be emptied by calling "`?- sys.forgetIDB.`"; facts are added by "`?- sys.tp.`", or "`?- sys.eval.`" (see below).
- The *FLogicInterpreter* `interp` holds a *current program.* The `interp` interface object serves as an encapsulation for all logical operations dealing with programs and queries.

  To add new facts and rules to the current program, a program file has to be consulted. Typing "`?- sys.showProgram.`" displays the current program (note that this is not necessarily the complete program file, see Section 4.2) and "`?- sys.forgetProgram.`" discards it.

Note that the current program is independent of the contents of the object manager; the link between these two is the *evaluation* of the current program wrt. the OM which changes the OM.

4.1 Programs and Evaluation

Consulting a program. After entering "`?- sys.consult@("foo.flp").`" (or typing `C-c C-c` in an emacs program buffer holding the program), the following happens: The user interface reads `foo.flp` rulewise using the F-Logic parser.

- Facts und rules are added to the current program held by the FLogicInterpreter `interp`. Note that no evaluation takes place until the system query "`?- sys.eval.`" is encountered.
- Queries are answered immediately:
  - System queries are executed by sending messages to interface objects. Syntactically, system commands are scalar path expressions of the form:

    `?- sys.`*expression*.

    Most system commands which are relevant to the user are directly applied to the user interface object called `sys`; e.g., for "`?- sys.eval.`", the current program is evaluated wrt. the current state of the OM.

    Additionally, *interface objects* (see Section 8.2) can be manipulated and queried by the user via system commands.
  - Other queries are passed to `interp` which answers them according to the current state of the OM.
- In case of a syntax error, the consult process is terminated and the error is reported at the text shell.

When the end of the program is reached, control is given back to the calling level (interactive mode or another consult). The OM then contains a model of the program evaluated so far.

Program evaluation. As already mentioned, FLORID uses a bottom-up evaluation strategy. This algorithm iteratively deduces new facts from already established facts using a forward chaining technique [CGT90]: A program P gives rise to an operator T_P on partial models (for F-Logic defined in [KLW95]). This operator adds all those facts to the model which can be derived from the already existing facts by a single application of a program rule (no recursion). To evaluate recursive rules, it is necessary to iterate this operator. Starting with the empty model (or a given finite object world), a fixpoint $T_P^\infty(\emptyset)$ is reached after a finite[1] number of applications of T_P.

After that, FLORID tries to deduce a new fact by inheritance. If such a fact is found, T_P is applied again until another fixpoint is reached. This process continues until no new information can be inherited. The resulting partial model is a model for the program P. When there are several possible facts inheritable at the same time, FLORID chooses one of them nondeterministically. Thus, the model of P is not unique. For a formal treatment of the semantics of F-Logic and implemented algorithm see [KLW95].

A single application of T_P can be achieved with "`?- sys.tp.`", although, the user will in general use "`?- sys.eval.`" for complete evaluation (which means iteration of deductive fixpoints and inheritance).

Note that the object manager is not cleared before applying T_P, that is, the evaluation does not start with the empty set, by default. This is an important feature for dividing large programs into smaller parts or for user-stratification (using "`sys.strat.doIt`", see Section 4.2).

Sometimes the user may want to cancel a running evaluation without terminating the whole system. This can be done by pressing `Control-\`. This key sends a QUIT signal to the system which causes a break flag to be set. In order to return a somewhat consistent object world, the current T_P-round has to be finished, so that it possibly takes some time before the system actually halts. After canceling, the partial model evaluated up to this point can be examined by querying. Calling "`?- sys.eval.`" again will continue the evaluation process where it was halted.

The pretty printer checks the QUIT signal, too, so that printing huge answer sets can be stopped.

Semi-naive Evaluation: FLORID includes an evaluation component providing a semi-naive evaluation mode. The evaluation mode can be set by a system command:

```
?- sys.theEval.mode@("seminaive").
?- sys.theEval.mode@("naive").
```

Naive evaluation is the default setting. Evaluating in semi-naive mode is promising for recursive programs with many TP rounds to make up for the overhead due to program analysis, rewriting and delta predicate maintainance.

Semi-naive evaluation will probably be slow for programs that derive new hierarchy facts or equate objects often because this makes dependancy analysis very hard. See [Sch97] for details about semi-naive evaluation for F-Logic. To see the rewritten program, set the debug mode `"program"` (see Section 6.1).

[1]Due to the existence of function symbols and object creating path expressions, some programs need countably many iterations to reach a fixpoint. Such programs will not terminate in FLORID. See the tutorial for examples.

Everyday's System Commands.

`?- sys.consult@("foo.flp").`	read the program file *foo.flp* (see Sec. 4.1)
`?- sys.load@("foo.flp").`	load a program file containing only facts directly into the OM
`?- sys.eval.`	evaluate the current program, i.e. calculate a model
`?- sys.strat.doIt.`	evaluate a stratum (see Sec. 4.2)
`?- sys.echo@("...").`	print argument string
`?- sys.break.doIt.`	stop program execution and get into interactive mode
`?- sys.return.`	continue program (entered in interactive mode after `sys.break.doIt.`)
`?- sys.end.`	exit FLORID

System commands are also used for output formatting and redirection, see Sec. 5.

Appendix A lists all system commands, and Appendix C contains a number of example sessions that demonstrate the use of system commands more explicitly. A number of frequently used system commands are loaded to the readline history.

Comments. FLORID program files may contain comments. There are three different comment formats available in FLORID: As in C and C++, text between "`/*`" and "`*/`" is ignored. In this case, the text may extend over more than one line. When a "`//`" (C++-Style) or a "`%`" (logic programming, LaTeX) is found, the rest of the line is considered as a comment. These formats may be mixed deliberately.

Nested Programs. File consulting can be nested without restriction. The interface variables may serve as a kind of parameters here. Consider a file *execute.flp* containing the following lines:

```
?- sys.consult@(filename).
?- sys.eval.
?- sys.echo@(filename).
?- sys.echo@("The answer set is:").
?- result(X). // Querying the model
?- sys.forgetProgram.
```

In another file, you can define *filename* and call *execute*:

```
?- sys[filename->"myfile.flp"].
?- sys.consult@("execute.flp").
?- sys[filename->"yourfile.flp"].
?- sys.consult@("execute.flp").
```

4.2 Blocks and Stratification

Sequences of queries (in practice, this concerns mainly system commands) can be combined in a *block*: A block is an interface object providing the method `doIt`. This executes the queries contained in the block. Of course, the same effect could be achieved by consulting a file with the commands or queries. But for short command series often recurring it is more convenient to use a block as a shorthand without having to consult external files.

Syntactically, a block declaration is a multivalued method definition with host object *sys*. Between the curly braces stands a *sequence* of goals, i.e., queries (without "`?-`"). Note that in contrast to the F-Logic semantics of multi-valued methods, the goals form a list here, not a set, i.e., the order is relevant. Note that you can use logical queries (besides system commands) in blocks, too, as far as they contain only one goal.

Stratification. The interplay between the OM and the *current program* allows a user-stratification of programs: a set of rules which is a part of a larger program, can be executed by the two system queries

```
... some rules ...
?- sys.eval.
?- sys.forgetProgram.
... some more rules ...
?- sys.eval.
```

First, the first set of rules is evaluated by "`?- sys.eval.`", computing an OM. Then, the method *forgetProgram* clears the current program, the contents of the OM remain unchanged and the second set of rules is evaluated wrt. this OM.

Thus, an important example is the definition of a block *strat* which makes stratification of programs more readable. (This is done in config.flp by default, cf. Sec. 8.4.)

```
?- sys[strat->>{sys.eval,
                sys.forgetProgram}].
```

Thus, calling

```
?- sys.strat.doIt.
```

executes "`?- sys.eval.`" and "`?- sys.forgetProgram.`" sequentially: to separate the strata of a program, simply put the command between them.

Besides the method `doIt`, a block has the method `display`. It lists the contents of the block as a list of queries (adding the query prompt "`?-`"). In the example above calling the method "`?- sys.strat.display.`" will print

```
Block("
?- sys.eval.
?- sys.forgetProgram.
")
```

5 Output

5.1 Output Formatting

There are several possibilities of printing answer sets, depending on the desired representation style and whether maximum speed or readability is preferred. The display modes are customized by the following methods of the *pretty printer* (represented by the interface object *prn*) providing the methods

`?- sys.prn.style@("<argument>").` and
`?- sys.prn.mode@("<argument>").`

style@(String): The parameter determines whether answers should be printed as fully instantiated molecules or in prolog style as variable bindings to objects:

"instance": Display answers as fully instantiated molecules. E.g., for our introductory example,

```
?- X[fly->yes].
Answer to query : ?- X[fly -> yes].
bob[fly->yes].
```

"bound": Display answers in prolog-like style with variables and objects bound to them:

```
?- X[fly->yes].
Answer to query : ?- X[fly -> yes].
X/bob
```

"html": Same as "bound", but writes answers to an html file and sends this file to a netscape browser.

The html style is in particularly useful when dealing with Web queries (see also [FHM+00]). In the html file, the resulting urls are tagged as links, so you can access them directly from the browser. To clear the html file, type "`?- sys.prn.clear.`".

mode@(String): Here the lexical representation of internal names can be customized. This matters if more than one such representation exists, for example when using path expressions or equating objects. The parameter may be:

"poor": Display internal names.

"const": Constants will be printed with their lexical representation and other objects (path expression and complex IdTerms) as internal names.

"first": Shows the first lexical representation found.

"best": For each object, the best (that is: shortest) existing lexical representation is printed. Of course, all possible representations have to be found and compared, so this is the slowest output mode.

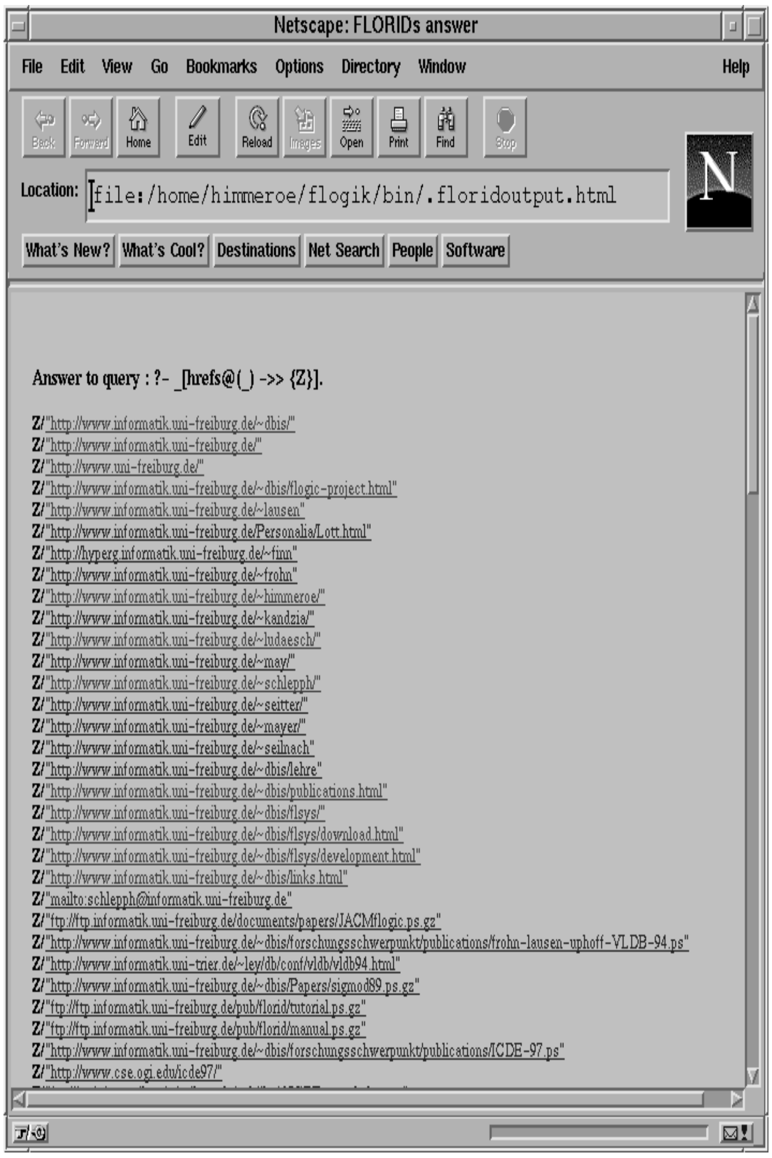
Example:

```
peter[father->mary.husband].

?- sys.eval.
?- sys.prn.mode@("poor").
?- X[father->Y].
   Answer to query : ?- X[father -> Y].
   X/o5   Y/o9

?- sys.prn.mode@("const").
?- X[father->Y].
   Answer to query : ?- X[father -> Y].
   X/peter   Y/o9

?- sys.prn.mode@("first").
```


Netscape: FLORIDs answer
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location:
What's New? What's Cool? Destinations Net Search People Software

```
file:/home/himmeroe/flogik/bin/.floridoutput.html
```

Answer to query : ?- _[hrefs@(_) ->> {Z}].

```
Z/"http://www.informatik.uni-freiburg.de/~dbis/"
Z/"http://www.informatik.uni-freiburg.de/"
Z/"http://www.uni-freiburg.de/"
Z/"http://www.informatik.uni-freiburg.de/~dbis/flogic-project.html"
Z/"http://www.informatik.uni-freiburg.de/~lausen"
Z/"http://www.informatik.uni-freiburg.de/Personalia/Lott.html"
Z/"http://hyperg.informatik.uni-freiburg.de/~finn"
Z/"http://www.informatik.uni-freiburg.de/~frohn"
Z/"http://www.informatik.uni-freiburg.de/~himmeroe/"
Z/"http://www.informatik.uni-freiburg.de/~kandzia/"
Z/"http://www.informatik.uni-freiburg.de/~ludaesch/"
Z/"http://www.informatik.uni-freiburg.de/~may/"
Z/"http://www.informatik.uni-freiburg.de/~schlepph/"
Z/"http://www.informatik.uni-freiburg.de/~seitter/"
Z/"http://www.informatik.uni-freiburg.de/~mayer/"
Z/"http://www.informatik.uni-freiburg.de/~seilnach"
Z/"http://www.informatik.uni-freiburg.de/~dbis/lehre"
Z/"http://www.informatik.uni-freiburg.de/~dbis/publications.html"
Z/"http://www.informatik.uni-freiburg.de/~dbis/flsys/"
Z/"http://www.informatik.uni-freiburg.de/~dbis/flsys/download.html"
Z/"http://www.informatik.uni-freiburg.de/~dbis/flsys/development.html"
Z/"http://www.informatik.uni-freiburg.de/~dbis/links.html"
Z/"mailto:schlepph@informatik.uni-freiburg.de"
Z/"ftp://ftp.informatik.uni-freiburg.de/documents/papers/JACMflogic.ps.gz"
Z/"http://www.informatik.uni-freiburg.de/~dbis/forschungsschwerpunkt/publications/frohn-lausen-uphoff-VLDB-94.ps"
Z/"http://www.informatik.uni-trier.de/~ley/db/conf/vldb/vldb94.html"
Z/"http://www.informatik.uni-freiburg.de/~dbis/Papers/sigmod89.ps.gz"
Z/"ftp://ftp.informatik.uni-freiburg.de/pub/florid/tutorial.ps.gz"
Z/"ftp://ftp.informatik.uni-freiburg.de/pub/florid/manual.ps.gz"
Z/"http://www.informatik.uni-freiburg.de/~dbis/forschungsschwerpunkt/publications/ICDE-97.ps"
Z/"http://www.cse.ogi.edu/icde97/"
```

Figure 3: Answer display in html style

```
?- X[father->Y].
   Answer to query : ?- X[father -> Y].
   X/peter   Y/peter.father

?- sys.prn.mode@("best").
?- X[father->Y].
   Answer to query : ?- X[father -> Y].
   X/peter   Y/mary.husband

?- sys.prn.style@("instance").
?- X[father->Y].
```

```
Answer to query : ?- X[father -> Y].
peter[father -> mary.husband]
```

5.2 Output Handling

All output from FLORID is communicated via *output channels*. Output channels wrap standard C++ output streams and can be redirected centrally. There are six predefined channels:

`answerChannel`:	for query results. Main output of pretty printer.
`errorChannel`:	for error messages and warnings.
`statisticChannel`:	for runtime statistics generated by the statisticHandler.
`helpChannel`:	for output of help methods.
`debugChannel`:	for debug information if some debug mode is enabled.
`systemChannel`:	for all other output generated by the system. Default output for all display and print methods.

To redirect the output sent to a channel, the output stream wrapped by an output channel can be changed with *setStream*, e.g., "`?- sys.errorChannel.setStream@(cout).`". Initially, `answerChannel, helpChannel,` and `statisticChannel` write to `cout`, the other channels to `cerr`. The command *resetStream* will restore this setting for a channel. E.g., all output can be sent to the file *myfile.flp* in the current directory:

```
?- sys[output->sys.open@("myfile.flp")].
?- sys.answerChannel.setStream@(output)[].

 -- produce some output --

?- sys.answerChannel.resetStream[].
?- sys.remove@("output").
```

By removing the interface object `output`, the file is automatically closed. This also happens when leaving FLORID.

To suppress printing, it is also possible to direct unwanted output to the stream `devnul` available at the user interface, e.g., "`?- sys.systemChannel.setStream@(devnul).`".

5.3 OM-Browsing

The complete OM can be output *framewise* – i.e., every stored object has a *frame* which contains all its properties – by "`?- sys.theOM.dump`". The OM dumping is written to a file by `?- sys.theOM.saveAs@(String)`. For additional output formatting, see [May00]. The output can then be again consulted by another F-Logic program.
For more detailed OM browsing for debugging see Section 6.1.

6 Debugging

Several components of the system provide a debug mode which can independently turned on or off. All debug output is sent to the debug channel so it can be easily redirected (see Sec. 5.2).

6.1 Debugging the Evaluation

Of course, the most important thing to debug is evaluation of programs. The basic debug mode of the evaluation component `theEval` is enabled with the command

```
?- sys.theEval.debugOn.
```

Then, T_P applications and reached fixpoints are reported. Every time a rule is matched, a dot is printed. So the progress of complex programs or programs dealing with large models can be followed.

Additionally, parameters can be added to the *debugOn* command, telling the evaluation component what to monitor:

"program": The program to evaluate is printed at the beginning. This is especially useful when dealing with nested consults, collecting rules and facts. Also, in the case of semi-naive evaluation, the outcome of the program transformation may be of interest.

"dump": After each T_P round, the contents of the object manager is printed. This makes sense only in the case of very small models. Otherwise, the frames to be watched can be specified (see below).

"insert": If this parameter is given, all inserts to the object model are reported. The respective rule is displayed together with the set of variable bindings.

"delta": This is a special mode for semi-naive evaluation. After each T_P round, the contents of all delta predicates needed for the actual program are printed.

"trigger": In this mode, all facts derived by firing inheritance triggers are reported when they are inserted into the object model.

"*frame*": Any other string is interpreted as the name of an object frame to monitor. The frame is printed after each T_P round.

"*frame*",*n*: If an integer follows after a string, the string is considered as the name of a predicate frame with arity n. The frame is printed after each T_P round.

These parameters may be combined deliberately, e.g.,

```
?- sys.theEval.debugOn@("program","delta","sales","expenditures").
```

Debugging is turned off by the command

```
?- sys.theEval.debugOff.
```

Additionally it is possible to apply the T_P operator by hand with the command

```
?- sys.tp.
```

and to examine the OM state afterwards by querying or browsing. Subsequent calls of the methods `tp` and `eval` will continue the evaluation with the current state of the OM.
The contents of the OM (i.e., all object and predicate frames) can be displayed through

```
?- sys.theOM.dump.
```

This command is useful if only a small number of frames are stored in the OM. For larger models, it is preferable to directly inspect the frames of interest. An object frame can be printed with the command

```
?- sys.theOM.showFrame@("arg").
```

where `arg` is the lexical representation of the object. If `showFrame` has two parameters, the string is considered as the name of a *predicate frame* with arity given as second parameter,

```
?- sys.theOM.showFrame@("ancestor",2).
```

prints the contents of the predicate frame corresponding to the binary predicate `ancestor`. Any Web action (see tutorial [FHM+00]), can be monitored with

```
?- sys.theOMAccess.debugOn.
```

6.2 Tracing the System Input

If the parser reports errors or problems with consulting files and system commands occur, it may be helpful to see what the parser actually reads. After setting the debug mode of the user interface `sys` by the command

```
?- sys.debugOn.
```

every line of input to the parser is echoed to the debug channel. Note that you can also enter this mode by starting florid with the `-d` option. The command

```
?- sys.debugOff.
```

turns the system debug mode off.

6.3 Tracing of Object Equating

During evaluation of a program the application of the scalarity constraint, a cyclic class hierarchy, or explicit use of equality in a rule head can force objects to be equated. If this is unintended it may lead to an unexpected behavior of the program. In the equating trace mode set by

```
?- sys.theOM.eqTraceOn.
```

any equating of objects is reported;

```
?- sys.theOM.eqTraceOff.
```

leaves this mode again.
Example:

```
peter[father->mary.husband].
peter[father->john].
```

Evaluation with *eqTraceOn* yields:

```
Equality: mary.husband (o7) made equal to john (o8)
```

The OIDs of the equated objects are shown in parentheses. Equating two objects may cause other equatings (e.g., in the case of scalar methods). With path expressions, such equatings may seem superfluous because the objects already have identical *names*. Listing the OIDs reveals that those equatings are indeed necessary.

```
michael.father.
mike.father.
michael = mike.
```

Evaluation with *eqTraceOn* yields:

```
Equality: michael (o5) made equal to mike (o8)
Equality: mike.father (o7) made equal to mike.father (o9)
```

7 Invoking Florid: Command Line Options

When FLORID is started in the unix shell by entering `florid`, command line options can be given. A list of possible options is printed with `florid -h`. If `-h` or an undefined option is given, FLORID terminates after printing an appropriate message.

```
naxos:~/florid/bin> florid -h
This is Florid
Type 'sys.help[].' for further information.

Florid - Options :
 -h                 : display all options
 -c <fileName>      : use non-default configuration file
 -his <fileName>    : use non-default history file
 -d                 : start in trace mode
 -e <Integer>       : set output level for errors
 <list<fileName>>   : start and consult files (separated by blank)
 -q                 : quit after executing (no interactive mode)
 -v                 : prints version number
```

The option `-c` denotes a configuration file to be used instead of the standard configuration file specified in the environment variable `DEFAULTCFG` (cf. Section 8.4). Similarly, the option `-his` loads the given file into the readline-history instead of the default in `DEFAULTHIS`. For operating FLORID with non-standard configuration files regularly, it is more convenient to change the environment variables (cf. Appendix 1.3).

In addition to the options, a list of filenames of F-Logic-programs can be given. FLORID consults them in a left-to-right order (see Section 4.1 for details about consulting). In the example from Section 2 one could reduce typing by invoking `florid first_example.flp`. Then the `consult` command is automatically executed.

With option `-d` (debug), all rules read will be echoed to the console. This is useful for tracing configuration errors and locating errors in program files.

Option `-e` (errormode) followed by a number customizes the system error messages. Possible values are listed in Table 2. The option `-q` causes FLORID to quit after execution of the command line instead of entering the interactive mode. If option -q is not given, FLORID enters the interactive mode after having processed the command line.

Number	Meaning
0	No kind of system message is displayed
1	Only messages but no warnings and errors are displayed
2	Warnings are displayed
4	Errors are displayed
127	Any kind of system message is displayed

Table 2: Parameters for Option -e

8 The Structure of the Florid System

The FLORID system is implemented in C++ [Str92]. Due to the object-oriented programming style in C++, it seemed reasonable to transfer this modeling style to the user interface, too ([Pfe95]). This resulted in a flexibly configurable system. Important internal classes and objects of the system are directly accessible from the user interface and are combined from there. Thus the structure of the FLORID system is reflected by the user interface (Fig. 4).

At the bottom is the so-called *Object Manager* which handles the data storage. The *User Interface* on top is the front-end and communicates with the user, processes queries, and sends the appropriate messages to other components of the system. Additionally, it manages a list of *interface objects* (see Section 8.2) which the user can access by sending messages to them (cf. system commands, Appendix A). All modules shown in Figure 4 are interface objects, their names are given in parentheses.

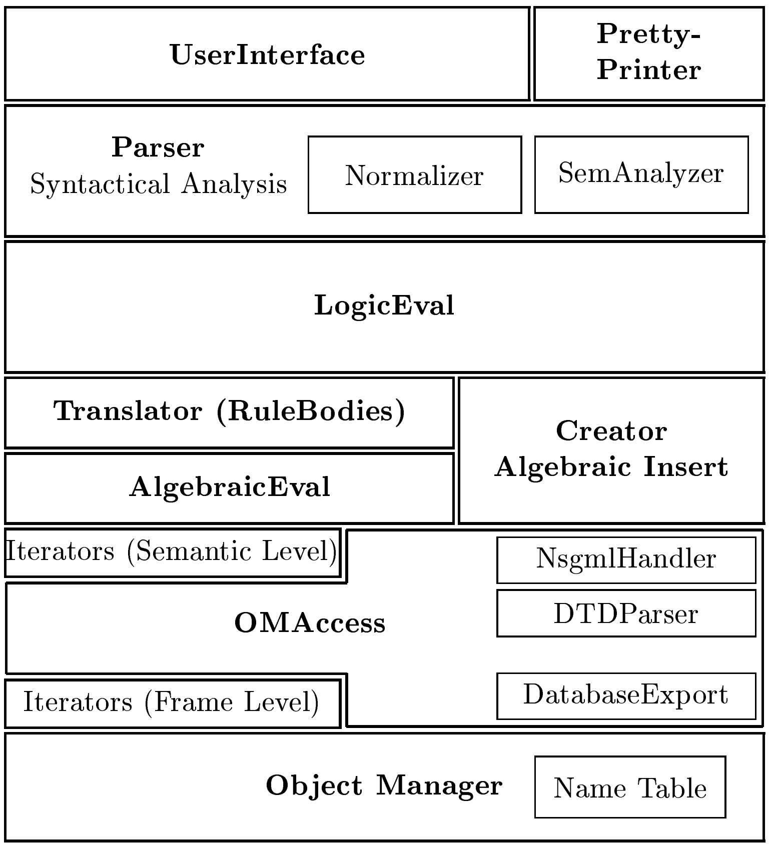


Figure 4: Structure of the FLORID system

The system is generated in the configuration file (see Section 8.4). If users are interested in changing the FLORID system itself, e.g., for investigating evaluation strategies, they are provided with very flexible and declarative methods to configure their system.

8.1 The User Interface

The user interface `sys` has two main functions. Firstly, it manages the system components and their interaction, and secondly, it communicates with the user. It encapsulates the FLORID system providing a single and uniform way of access: queries in F-Logic syntax. That means that all user input, whether he wants to load a database, evaluate a program, examine the calculated object model, or customize the system, has strictly to follow F-Logic syntax. System commands can be distinguished from data queries, facts, or rules as they are queries to `sys`, a virtual host object representing the user interface. Of course, regarding their semantics, such queries differ considerably from logical queries.

8.2 Interface Objects

The components of the FLORID system in Figure 4, as well as some other objects (e.g., integers, strings, or blocks, cf. Section 4.2), are available at the user interface. Since FLORID is an object-oriented system, all operations on system components are carried out by invoking methods on them. In F-Logic syntax, this gives rise to path expressions of the form *object.method* or *object.method@(attribute,...,attribute)*. The interface objects can be accessed as (virtual) methods of `sys`[2]. The path expression `sys.`*objname* yields the object associated with the method *objname*. Furthermore, `sys.`*objname.method* would invoke *method* on this object.

The existing objects can be listed by entering "`?- sys.display.`" (that means, sending the method *display* to the host object `sys`). A new interface object, that is, a new method of `sys`, is defined by a query of the form

```
?- sys[objname -> expression].
```

Note, that this is syntactically a query (as every system command) but semantically it plays the role of a fact[3]. If *objname* is unknown to the user interface (not in the list printed after the `display` command), a new method is added and bound to the value of *expression* (which is an object), otherwise the old value is replaced by the new one.
In the method definition, *expression* may stand for:

- *methname* (i.e., an identifier starting with lower case letter denoting a method)
- *Integer*
- *String*
- *expression.methname*
- *expression.methname@(expression,...,expression)*

Here are some examples for definitions of interface objects:

```
?- sys[loops->3].
?- sys[filename->"config.flp"].
?- sys[newWorld->classOM.new].
?- sys[debugfile->sys.open@("logfile")].
```

An interface object can be deleted by

```
?- sys.remove@("thisobject").
```

There are also a number of *class* objects available at the user interface. They provide a construction kit to build a FLORID system according to individual needs. Class objects have

[2]In conventional systems, this would be done by environment variables.

[3]In conventional systems, this would be an environment variable assignment.

the single method *new* to generate instances. By convention, the names of such objects begin with the prefix *class*.

Using class objects opens up the possibility to configure FLORID in various ways, for example with different evaluation components (e.g., the semi-naive evaluation module). FLORID can even hold several separate systems at the same time. The standard configuration of FLORID is build by the file `config.flp` which is automatically consulted after startup (if `DEFAULTCFG` is set and the `-c` option is not used). See Section 8.4 for details of the system building process. Normally, the class objects are only used in configuration files.

Interface objects can be manipulated and queried by the user. These commands are of the form

```
?- sys.<which_object>.<expr>.
```
, e.g.,
```
?- sys.centralStatistic.show.
```

which is interpreted as follows: The method *centralStatistic* is sent to the object *sys* which will return the respective interface object. This in turn gets the method call *show*, causing the statistics handler to print the runtime statistics table.
All interface objects (except the class objects) have the following basic methods in common:

`display@(outputChannel)`	Display the actual contents or state
`print@(outputChannel)`	Print class or type of the object
`help`	List all methods understood by the object

The parameter specifying the output channel may be omitted (`systemChannel` is used by default).

8.3 Restrictions for System Commands

For system commands, only a subset of the full F-Logic syntax [FHM$^+$00] is allowed. The restrictions are listed below. On violation, the respective error message is reported.

1. No function symbols are allowed, i.e., complex terms like *functor(varname)*. Use methods with parameters instead:
   methname@(param,...).
   ```
   ?- sys.consult("test.flp").
   Function symbols with arguments are not allowed in system commands.
   Maybe a @ is missing.
   ```
2. The called method must be defined for the host object.
   ```
   ?- sys.theEval.foo.
   Unknown method : foo
   ```
   Sending the `help` method to any interface object lists the methods that are defined for this object.
3. Additionally, *sys* treats the interface objects as methods (yielding themselves), so these must exist before calling. Otherwise, *sys* reports an unknown method call.
   ```
   ?- sys.anotherEval.help.
   Unknown method : anotherEval
   ```
4. No variables, i.e., names beginning with upper case letters, are allowed.
   ```
   ?- sys.Eval.
   Variables are not allowed in system commands.
   ```

```
    Use identifiers starting with lower case characters.
```

5. Set-valued path expressions are forbidden.

```
    ?- sys..eval.
    Only simple scalar paths are allowed in system commands.
    Use format ?- sys.object.method@(args).
```

6. System queries have to consist of a single goal.

```
   ?- sys.eval, sys.forgetProgram.
   System query with more than one literal in body.
   Split the query into simple ones.
```

7. If a goal starting with "`sys.`" is not the leftmost goal of a query, it is not recognized as a system command and treated as a query to the model instead.

```
   ?- john[age->X], sys.end.

   Answer to query : ?- john[age -> X], sys.end.

   false
```

8.4 System Configuration in `config.flp`

The system is generated in the configuration file. If users are interested in changing the FLORID system itself, e.g., for investigating evaluation strategies, they are provided with very flexible and declarative methods to configure their system. Writing or modifying configuration files requires a detailed knowledge of internals of the system and should be used with care.

If FLORID is called without the `-c` option it first consults the file pointed to by the environment variable `DEFAULTCFG`. (normally this is the file `config.flp`). This generates the components of a standard FLORID system and links them. The process is echoed if the option `-d` was given.

In addition to building the system, the config file can also be used to configure some interface objects differently than by default. To have semi-naive evaluation instead of the default (naive) evaluation, append "`?- sys.theEval.mode@("seminaive").`" to the config file. It is useful to add all permanent customizations to the config file instead of typing them again and again at the FLORID shell.

config.flp:

```
// Configuration file

?- sys.centralErrorHandler.setOutputFilter@(2). // short error messages
?- sys[theOM->classOM.new].
?- sys[theOMAccess->classWebOMAccess.new@(theOM)].
?- sys[theCreator->classCreator1.new@(theOMAccess)].
?- sys[theAlgebraicEval->classAlgebraicEval.new@(theOMAccess)].
?- sys[theTranslator->classTranslator.new].
?- sys[theMatcher->classMatcher.new@(theTranslator,theCreator,theAlgebraicEval)].
?- sys[theEval->classPrSemiNaive.new@(theMatcher,theOMAccess)].
?- sys[theSemAnalyser->classSemanticAnalyser.new].
?- sys[prn->classPrettyPrinter.new@(theOMAccess)].
?- sys[interp->classFLogicInterpreter.new@(theEval,prn,theSemAnalyser)].
?- sys.switch@(interp,theOM.giveSymTab).
?- sys[theNormalizer->classNormalizer.new@(interp)].
?- sys[strat->>{sys.echo@(""),sys.echo@("%%% BEGIN EVALUATION"),
              sys.eval,sys.forgetProgram,
              sys.echo@("%%% END EVALUATION")}].
?- sys[evaluate->>{sys.strat.doIt}].
?- sys[break->>{sys.strat.doIt,
              sys.echo@(""),sys.echo@("*** START QUERY-SUBSHELL"),
              sys.shell,
              sys.echo@("*** END QUERY-SUBSHELL")}].
?- sys.theOMAccess.setAccess@("all").
```

Explanation:

```
?- sys.centralErrorHandler.setOutputFilter@(2).
```

Display all error messages in short form.

```
?- sys[theOM->classOM.new].
```

Generate an object manager and assign it to the interface variable `theOM`.

?- sys[theOMAccess-¿classWebOMAccess.new@(theOM)].

An Object Manager Accessor (OMA) encapsulates read access to the object manager. Therefore it needs the OM as a parameter.

```
?- sys[theCreator->classCreator1.new@(theOMAccess)].
?- sys[theAlgebraicEval->classAlgebraicEval.new@(theOMAccess)].
?- sys[theTranslator->classTranslator.new].
```

These three objects are needed by the matcher as part of the evaluation process.

```
?- sys[theMatcher->
    classMatcher.new@(theTranslator,theCreator,theAlgebraicEval)].
```

Now the matching component is generated which may get an optimizer, too, as an additional parameter. No such component has been implemented up to now, though.

```
?- sys[theEval->classPrSemiNaive.new@(theMatcher,theOMAccess)].
```

This configuration file uses a semi-naive evaluation component.

```
?- sys[theSemAnalyser->classSemanticAnalyser.new].
```

A component used by the FLSysInterpreter (the front-end user interface) to check queries for safety.

```
?- sys[prn->classPrettyPrinter.new@(theOMAccess)].
```

The pretty printer displays answer sets in readable form. To do this, it needs access to the object manager.

```
?- sys[interp->classFLogicInterpreter.new@(theEval,prn,theSemAnalyser)].
```

The FLogicInterpreter serves as an interface between the user front-end (FLSysInterpreter) and the pure F-Logic part of the system. Here, all input except system commands is processed. The FlogicInterpreter manages the evaluating component, the pretty printer and the SemanticAnalyser.

```
?- sys.switch@(interp,theOM.giveSymTab).
```

For passing actions, the FLSysInterpreter needs to know `interp` and the system's symbol table. They cannot be given as parameters because they are generated later than the FLSys-Interpreter.

```
?- sys[strat ->> ...]. ?- sys[evaluate ->> ...]. ?- sys[break ->> ...].
```

With this command, blocks for user stratification and program interruption are defined.

Acknowledgements. First of all, we want to thank Georg Lausen, the head of our group, who made the FLORID project possible. Furthermore, our thanks go to the former team members Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Bertram Ludäscher, Christian Schlepphorst, Markus Seilnacht, and Heinz Uphoff who developed FLORID up to version 2.0 together with students from the universities at Mannheim and Freiburg.

A Florid System Commands

Most system commands which are relevant to the user are directly applied to the user interface object called `sys`. The following system commands are of interest for everydays users:

`?- sys.consult@("foo.flp").`	read the program file *foo.flp*
`?- sys.load@("foo.flp").`	load a program file containing only facts directly into the OM
`?- sys.tp.`	apply T_P operator of current program once to the current OM
`?- sys.eval.`	evaluate the current program, i.e. calculate a model
`?- sys.strat.doIt.`	evaluate a stratum
`?- sys.echo@("...").`	print argument string
`?- sys.break.doIt.`	stop program execution and get into interactive mode
`?- sys.return.`	continue program (entered in interactive mode after `sys.break.doIt.`)
`?- sys.end.`	exit FLORID

The OM contents can be dumped or saved:

`?- sys.theOM.dump.`	dump OM
`?- sys.theOM.saveAs@(String)`	save OM to file

For additional output formatting, see [May00].

The FLORID output can be formatted and redirected using the following commands:

`?- sys.prn.style@("instance"|"bound"|"html").`	select output style
`?- sys.prn.mode@("poor"|"const"|"first"|"best").`	select output of internal names
`?- sys.open@(<filename>).`	open a FLSysOStream pointing to a file. Usage: `?- sys[output->sys.open@("logfilestream")].` (creates an interface object `output` which is assigned to the stream)
`?- sys.<channel>.setStream@(<stream>).`	redirect `<channel>` to `<stream>` Usage: `?- sys.answerChannel.setStream@(output).`
`?- sys.<channel>.resetStream.`	reset `\flq channel\frq` to default.
`?- sys.remove@(<if-object>).`	remove the interface object; if it is a stream, the corresponding file is closed

The following system commands support debugging:

`?- sys.showProgram.`	show current program (facts and rules)
`?- sys.forgetProgram.`	clear the current program
`?- sys.forgetIDB.`	clear the Object Manager
`?- sys.debugOn.`	enter debug mode for consult
`?- sys.debugOff.`	
`?- sys.eqTraceOn.`	enter debug mode for tracing equatings
`?- sys.eqTraceOff.`	
`?- sys.theOM.showFrame@("arg").`	display object frame
`?- sys.theOM.showFrame@("pred",`n`)./`	display predicate frame of `pred` for arity n

Additional help is also available:

`?- sys.help.`	lists all methods of `sys`
`?- sys.display.`	lists all interface objects

Each of the above interface objects (adressible by `sys.<interface-obj>`) provides the method `sys.<interface-obj>.help` which lists its methods.

B Readline and History Commands

The following tables list the key commands available for the readline and the history packages, they were taken from [FR94]. The shortcut `C-k` is read 'Control-k' and describes the character produced when the Control key is pressed and the `k` key is struck. The text `M-k` is read as 'Meta-k' and describes the character produced when the meta key (if you have one) is pressed, and the `k` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` first and then typing `k`. Either process is known as metafying the `k` key.

`C-_`	Undo the last thing that you did.
`C-a`	Move to the start of the line.
`C-e`	Move to the end of the line.
`M-f`	Move forward a word.
`M-b`	Move backward a word.
`C-l`	Clear the screen, reprinting the current line at the top.
`C-k`	Kill the text from the current cursor position to the end of the line.
`C-u`	Kill backward from the cursor to the beginning of the current line.
`M-d`	Kill from the cursor to the end of the current word, or if between words, to the end of the next word.
`M-DEL`	Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.
`C-y`	Yank the most recently killed text back into the buffer at the cursor.
`TAB`	Attempt to do completion on the text before the cursor.
`M-?`	List the possible completions of the text before the cursor.

Table 3: Important readline commands

`C-p`	Move 'up' through the history list. You can also use the ↑ key.
`C-n`	Move 'down' through the history list. You can also use the ↓ key.
`M-<`	Move to the first line in the history.
`M->`	Move to the end of the input history, i.e., the line you are entering.

Table 4: Important history commands

C Example Sessions

In this section we present and comment some FLORID sessions in order to demonstrate the use of system commands. User input and system output is printed in the `typewriter` style, comments in *italics*.

C.1 Evaluating

Let the file `edge.flp` consist of the following F-Logic program:

```
e1:edge[1->n1; 2->n2].
e2:edge[1->n2; 2->n4].
e3:edge[1->n1; 2->n3].
e4:edge[1->n3; 2->n4].
e5:edge[1->n4; 2->n5].
e6:edge[1->n5; 2->n6].

edge::path.

p(E,P):path[1->X; 2->Z] :- E:edge[1->X; 2->Y],P:path[1->Y; 2->Z].
```

Scenario 1 : Evaluate the program `edge.flp` and query the computed model.

```
naxos:~/florid/bin> florid
This is FLORID
Type 'sys.help.' for further information.
?- sys.consult@("edge.flp").              load edge.flp
?- sys.eval.                              start evaluation of current program
?- p(e1,P):path.                          logical query to the model computed
Answer to query : ?- p(e1,P):path.        answer set is printed
P/e2
P/p(e2,e5)
P/p(e2,p(e5,e6))

3 output(s) printed
?- sys.end.                               quit FLORID
Bye
naxos:~/florid/bin>
```

Scenario 2 : Evaluate `edge.flp`, discard the computed model and evaluate again by single step T_P application to see how the answer set changes. Instead of typing "`?- sys.consult@("edge.flp").`" the filename is given in the command line.

```
naxos:~/florid/bin> florid edge.flp
This is FLORID
Type 'sys.help.' for further information.
?- sys.eval.                  start evaluation
?- sys.forgetIDB.             clear the OM, program remains unchanged
?- sys.tp.                    apply TP operator once
TP Round 0
?- p(e1,P):path.              query the (partial) model yet computed
Answer to query : ?- p(e1,P):path.

false                         answer set is enpty
?- sys.tp.                    apply TP again

TP Round 1
?- p(e1,P):path.

Answer to query : ?- p(e1,P):path.

P/e2

1 output(s) printed
?- sys.tp.
TP Round 2
?- p(e1,P):path.

Answer to query : ?- p(e1,P):path.

P/e2
P/p(e2,e5)

2 output(s) printed
?- sys.tp.
TP Round 3
?- p(e1,P):path.

Answer to query : ?- p(e1,P):path.

P/e2
P/p(e2,e5)
P/p(e2,p(e5,e6))

3 output(s) printed
?- sys.tp.
FP reached in round 4           further application of TP yields no new facts,
Model evaluated                 (a fixpoint is reached),
?-                              no inheritance was possible
```

Scenario 3 : Evaluate `edge.flp`, then consult `anc.flp`. After checking the current program and evaluating it, discard program and object model to evaluate another program (`test.flp`) from scratch.

```
naxos:~/florid/bin> florid edge.flp
This is FLORID
Type 'sys.help.' for further information.
?- sys.eval.                              start evaluation of current program
?- sys.consult@("anc.flp").               load anc.flp
?- sys.showProgram.                       display the current program

e1:edge[1 -> n1; 2 -> n2].
e2:edge[1 -> n2; 2 -> n4].
e3:edge[1 -> n1; 2 -> n3].
e4:edge[1 -> n3; 2 -> n4].
e5:edge[1 -> n4; 2 -> n5].
e6:edge[1 -> n5; 2 -> n6].
edge::path.
p(E,P):path[1 -> X; 2 -> Z] :- E:edge[1 -> X; 2 -> Y], P:path[1 -> Y;
2 -> Z].
X[anc ->> {Y}] :- X[father ->> {Y}].
X[anc ->> {Y}] :- X..anc[father ->> {Y}].

?- sys.forgetProgram.                     discard the current program
?- sys.showProgram.
/* This FProgram is empty */
?- sys.forgetIDB.                         clear OM (discard model)
?- sys.consult@("test.flp").              read test.flp
?- sys.eval.                              and evaluate it
?-
```

As it is seen from this session, "`?- sys.consult@("...").`" does not discard the current program: it adds new rules.

Scenario 4 : After evaluating `edge.flp` display the runtime statistics.

```
naxos:~/florid/bin> florid edge.flp
This is FLORID
Type 'sys.help.' for further information.
?- sys.eval.
?- p(e1,P):path.                            query the model

Answer to query : ?- p(e1,P):path.


P/e2
P/p(e2,e5)
P/p(e2,p(e5,e6))

3 output(s) printed
?- sys.centralStatistic.show.             show runtime statistics
```

```
Time spent in semantic analyses : 0.03sec real    0.02sec user
(Time spent in retrieving       : 0.16sec real    0.15sec user)
Number of TP iterations         : 5
Total time spent in evaluation  : 0.34sec real    0.24sec user
Time spent in computing answers : 0.01sec real    0.01sec user
Time spent in pretty-printing   : 0.22sec real    0.00sec user

?- p(e1,P):path.

Answer to query : ?- p(e1,P):path.

P/e2
P/p(e2,e5)
P/p(e2,p(e5,e6))

3 output(s) printed
?- sys.centralStatistic.show.          show statistics again

Time spent in semantic analyses : 0.03sec real    0.02sec user
(Time spent in retrieving       : 0.16sec real    0.15sec user)
Number of TP iterations         : 5
Total time spent in evaluation  : 0.34sec real    0.24sec user
Time spent in computing answers : 0.02sec real    0.01sec user
Time spent in pretty-printing   : 0.00sec real    0.00sec user

?- sys.centralStatistic.clear.         reset statistics
?- sys.centralStatistic.show.          and display again
?-                                     statistic is empty
```

This scenario shows that the runtime statistics entries accumulate until the user discards them explicitely. The first column (`real`) lists the absolute time difference between start and end, while the second column (`user`) yields the CPU time consumed by the task. The difference may vary due to other tasks running in parallel (e.g., kernel, window manager). Additional to the measured time the number of T_P iterations is displayed. If new facts were derived through inheritance, the number of these facts (fired inheritance trigger) is printed, too.

C.2 Help mechanism

Starting point of all system commands is the host object `sys`, the so called FLSysInterpreter. Therefore all system commands begin with "`?- sys`". Besides the built-in methods of the interpreter which can be listed by "`?- sys.help.`", its attributes (the interface objects) are handled as methods, too.

Sometimes it may not be clear for the user which interface object understands which methods. Therefore all interface objects representing instances of C++ classes provide the method `help`. Those objects representing the classes itself do not have this method[4]. The next two scenarios shall explain how to navigate through the user interface.

[4]Per convention, the names of these objects start with the prefix `class`.

Scenario 1 : Which attributes and methods has the interpreter after building the system in the standard configuration file?

```
This is FLORID
Type 'sys.help.' for further information.
?- sys.display.                               show attributes of interpreter


FLSysInterpreter{
  answerChannel : anOutputChannel
  centralErrorHandler : anErrorHandler
  centralStatistic : aStatisticHandler
  classBlock : Block
  classCreator1 : Creator1
  classFLogicInterpreter : FLogicInterpreter
  classGraphicInterface : GraphicInterface
  classMatcher : Matcher
  classPrSemiNaive : PrSemiNaive
  classPrettyPrinter : PrettyPrinter
  .
  .    // we don't list all interface objects here
  .
  classSemanticAnalyser : SemanticAnalyser
  classStatisticHandler : StatisticHandler
  cout : aFLSysOStream
  debugChannel : anOutputChannel
  devnul : aFLSysOStream
  errorChannel : anOutputChannel
  helpChannel : anOutputChannel
  interf : aGraphicInterface
  interp : aFLogicInterpreter
  prn : aPrettyPrinter
  statisticChannel : anOutputChannel
  strat : aBlock
  theAlgebraicEval : anAlgebraicEval
  theCreator : aCreator1
  theEval : aPrSemiNaive
  theMatcher : aMatcher
  theNormalizer : aFLSysObject
  theOI : anOI
  theOM : aFLSysObject
  theSemAnalyser : aSemanticAnalyser
  theTranslator : aTranslator
}
?- sys.help.                                 show built-in-methods of interpreter

                    Online help for class FLSysInterpreter
```

```
You can apply the following member functions to an instance of
class FLSysInterpreter via F-Logic shell :

    - help
    - isOk
        returns  0 (object is corrupted)  or  1
    - isA@(String)
        String : classname
        returns  1 (object belongs to class)  or  0
    - print
        prints type of instance
    - display
        displays components
    - end
        leave the flogic-system
    - eval
        evaluate the current program
    - tp
        apply TP-operator once
    - showProgram
        show the current program
    - forgetProgram
        clear the current program
    - forgetIDB
        clear the object manager
    - switch
        link a new FLogicInterpreter to sys
    - debugOn
        echo lines read while consulting
    - debugOff
    - interpreteLine@(String)
        String: command in flogic syntax
    - interpreteFile@(String)
        String: filename
    - open@(String)
        opens a FLSysOStream pointing to a file.
        String: filename
        Usage: ?- sys[mystream->sys.open@("logfile")].
    - echo@(String)
        String: displayed on output stream
    - remove@(String)
      String: name of interface object to remove.
```

In the output generated by "`?- sys.display.`" both the C++ classes and instances appear. The line `classOI : OI` means that `classOI` represents the C++ class `OI` at the user interface.

Furthermore, `theOI : anOI` describes that `theOI` is an instance of the class `OI`. To all interface object displayed on the left side of "`:`" (e.g., theOI, prn, theEval, etc.) you can apply `help` to see which methods are defined for them.

Scenario 2 : Which methods can be sent to the interface object `prn`?

```
This is FLORID
Type 'sys.help.' for further information.
?- sys.prn.help.                               apply the method help to prn

                     Online help for class PrettyPrinter

You can apply the following member functions to an instance of
class PrettyPrinter via F-Logic shell :

    - help
    - isOk
          returns  0 (object is corrupted)  or  1
    - isA@(String)
          String : classname
          returns  1 (object belongs to class)  or  0
    - print@(OutputChannel)
          OutputChannel : e.g., answerChannel
    - display@(OutputChannel)
          OutputChannel : e.g., answerChannel
    - style@(String)
          String: describes output style, possible arguments are:
               1. "bound"   : answers are printed in Prolog style
               2. "instance": answers are printed in F-Logic syntax
               3. "html"    : answers (in Prolog style) are written to an
                              html file that is displayed by netscape
    - mode@(String)
          String: describes representation of objects,
                  possible arguments are:
                     1. "poor" : only OID's are printed
                     2. "const": only strings of constants are printed
                     3. "first": first string for object is printed
                     4. "best" : best string for object is printed
    - clear
          remove stored information and clear html file of html style


?- sys.prn.style@("instance").          set fact style for output
?- sys.prn.isOk.                        send method isOk to prn
The return value of isOK is not printed by the user interface, you have to
apply the method display@(cout) onto the return value.
This applies to all methods returning strings or integers.
?- sys.prn.isOk.display@(cout).         send isOk to prn
```

and print the return value on cout

C.3 System commands in files

Placing system commands into files is useful in the following situations:
- system configuration (Sec. 8.4),
- automation of processes to avoid unnecessary typing,
- generation of output for test purposes,
- user stratification by hand (Sec. 4.2).

Scenario 1 : Evaluate the file `edge.flp` and examine the model without having to type "`?- sys.eval.`" and the query it at the shell. Then the file `edge.flp` is as follows:

```
e1:edge[1->n1; 2->n2].
e2:edge[1->n2; 2->n4].
e3:edge[1->n1; 2->n3].
e4:edge[1->n3; 2->n4].
e5:edge[1->n4; 2->n5].
e6:edge[1->n5; 2->n6].

edge::path.

p(E,P):path[1->X; 2->Z] :- E:edge[1->X; 2->Y],P:path[1->Y; 2->Z].
?- sys.eval.
?- p(e1,P):path.
```

Every system command can occur in a program file as well. Most often, however, commands for controlling evaluation are used in program files.

By using the command "`?- sys.strat.doIt.`" the user can partition the program into strata by hand (see Sec. 4.2). This is helpful and sometimes unavoidable when negation is used. Other applications, e.g., for gaining efficiency, are possible, too.

Scenario 2 : Before using a negated goal in a rule, set a stratum. Document this by printing a message.

```
X[L->>Y] :- Y[(L:line)->>X].

X[reaches->>{X}] :- X:stop.

c(nil,X,Y,L)[from->X[reaches->>{Y}]; to->Y] :-
  X[(L:line)->>Y].

?- sys.strat.doIt.
?- sys.echo@("Stratum").

c(N,X,Z,L)[from->V[reaches->>{Z}]; to->Z] :-
  c(N,X,Y,L)[from->V], Y[(L:line)->>Z], not V=Z.
```

The interface object `strat` is a block (see Sec. 4.2). By calling its method `doIt`, the block is consulted. In the case of `strat` the two commands `sys.eval.` and `sys.forgetProgram.` are executed, thus stratifying the program.

References

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer, 1990.

[DT95] Gillian Dobbie and Rodney Topor. On the declarative and procedural semantics of deductive object-oriented systems. *Journal of Intelligent Information Systems*, 4(2):193–219, 1995.

[FHM$^+$00] J. Frohn, R. Himmeröder, W. May, P.-Th. Kandzia, and C. Schlepphorst. How to write F-Logic programs in FLORID, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/florid`.

[FLU94] Jürgen Frohn, Georg Lausen, and Heinz Uphoff. Access to objects by path expressions and rules. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 273–284, 1994.

[FR94] Brian Fox and Chet Ramey. GNU readline library, edition 2.0, for readline library version 2.0. Free Software Foundation, Cambridge, MA, 1994.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[Law93] Michael Lawley. A Prolog Interpreter for F-Logic. Technical report, Griffith University, Australia, 1993.

[Liu96] M. Liu. ROL: A typed deductive object base language. In *Intl. Conference on Database and Expert Systems Applications (DEXA)*, 1996.

[May00] W. May. Handling XML with FLORID, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/florid`.

[Pfe95] Thorsten Pferdekämper. Eine flexible Konfigurations- und Benutzerumgebung für F-Logik. Diplomarbeit Universität Mannheim, 1995.

[Sch97] Christian Schlepphorst. Semi-Naive Evaluation of F-Logic Programs. Technical Report 85, Universität Freiburg, 1997.

[Str92] Bjarne Stroustrup. *The C++ Programming Language.* Addison–Wesley, 2nd edition, 1992.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, 1989.