FLORID

User Manual
Version 2.0

Jürgen Frohn Rainer Himmeröder
Paul-Th. Kandzia Christian Schlepphorst

Institut für Informatik, Universität Freiburg
Am Flughafen 17, 79110 Freiburg, Germany

November 1997

Contents

List of Tables

List of Figures

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 [7], 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. [8], GULOG [4] and ROL [9], nearly all of the distinctive F-logic features are realized in FLORID. In particular we emphasize that the prototype supports data driven schema definition, multiple, non–monotonic inheritance and furthermore path expressions [6] 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 [1, 3, 14].

When designing FLORID the objective was to provide a prototype environment where the practical programming aspects of such a powerful language can be tested. Therefore, we have not set great store by performance, nor does FLORID support secondary data storage. Nevertheless, both issues are topics of actual research.

The present prototype 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. To learn something about the language and data modeling with F-logic the reader is referred to the FLORID programming tutorial [5] and the F-logic report [7].

The structure of this manual is as follows: First, we give a short example of how to run FLORID and investigate an F-logic program. In section 2 the structure of the system and its operation from a text shell is explained. There are two 'smarter' interfaces which build on the FLORID-shell: a graphic interface based on Tcl/Tk [10, 15], and an integration of FLORID into the emacs/XEmacs editor. The installation and operation of these two interfaces are described in the sections 3 and 4, resp. Of course, running the graphic interface requires that Tcl/Tk is installed (cf. appendix C). Even if the user wants to work with the higher-level interfaces only, he should read section 2 to understand the basic philosophy of FLORID, as the interfaces build upon the text shell.

For installation hints see appendix A. Any comments or questions please send to
florid@informatik.uni-freiburg.de

Acknowledgement: 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 Heinz Uphoff, Bertram Ludäscher and Wolfgang May for many ideas and discussions that helped developing FLORID, and to the students who were involved in the implementation.

Changes in Version 2.0

FLORID 2.0 provides a semi-naive evaluation component. To facilitate writing and testing of F-logic programs, a number of helpful debug features have been added. System output has been reorganized to enable redirecting certain sorts of output. It is now bundled into six output channels, depending on its type and origin. Furthermore, FLORID has been extended to fetch and process Web documents declaratively (see tutorial [5] for details).

1 A First Example

To give an impression of the FLORID system and its usage we present a short example session. A simple F-logic program is loaded and evaluated. Throughout this manual, all FLORID input and output, as well as F-logic programs, filenames, class names etc. are shown in `typewriter face`. More example sessions are included in appendix D.

Let the file `first_example.flp` contain the following program:

```
chorydoras :: fish[water *-> sweet;
                perfectPH *-> 8;
                hardness *-> medium;
                brood *-> eggs;
                size *-> 7;
                diet *-> carnivore].

bob : chorydoras.
```

We want to evaluate the program `first_example.flp` and query the defined database.

```
naxos:~/florid/bin> florid                    invoke FLORID from shell
This is FLORID, Version 2.0 <September 1997>
Type 'sys.help[].' for further information.
?- sys.consult@("first_example.flp")[].       system command to load file
?- sys.eval[].                                start evaluation of current program
?- X[water->sweet].                           logical query to the model computed
Answer to query : ?- X[water -> sweet].       answer set is printed
X/bob
1 output(s) printed
?- sys.end[].                                 quit FLORID
Bye
naxos:~/flogik/bin>
```

As seen in this example session, FLORID is started from shell by simply typing `florid`[1] at the unix prompt. For possible options please see section 2.1.

The queries starting with "`?- sys.`" are system commands, they are described in section 2.5. Other queries refer to the evaluated model. In this case the answer set is printed.

2 Using FLORID from the Unix Shell

Starting with a short overview over the structure and the components of the system, in this section we describe how to run FLORID from a text shell. Even if the user wants to access FLORID via a graphic front-end only (see sections 3 and 4), he should also glance through this section to be acquainted with the basics of this operation mode as the advanced interfaces build upon the text shell.

[1]the name of the executable (of course the user is free to choose another name for the executable on operating system level).

2.1 Invoking FLORID

Provided that the system is installed (see Appendix A), FLORID is invoked from the (Unix) shell by entering `florid`. 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, Version 2.0 <September 1997>
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)
 -i                 : start with graphic interface
 -q                 : quit after executing command line
                      (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. sect. 2.13). 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 A.1).

In addition to the options a list of filenames of F-logic-programs can be given. FLORID will consult them in a left to right order (see section 2.9 for details about consulting). In the example from section 1 one could reduce typing by invoking `florid first_example.flp`. Then the consult command is automatically executed.

Option `-i` calls the Tcl/Tk GUI immediately after starting FLORID. This is more convenient than invoking the GUI via system command (cf. section 3).

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 1. The option `-q` causes FLORID to quit 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`.

If option -q is not given, FLORID enters the interactive mode after having processed the command line. Now the prompt "`?-`" is displayed, indicating that queries can be entered. See section 2.3 for a detailed description of the interactive mode.

To leave the prototype, enter "`?- sys.end[].`".

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 1: Parameters for Option -e

2.2 The Structure of the F-logic System

Figure 1 shows the major parts of an F-logic system. 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 2.4) which the user can access by sending messages to them. All modules shown in fig. 1 are interface objects, their names are given in parentheses. The system is generated in the configuration file (see section 2.13).

2.3 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 F-logic 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.

2.4 Interface Objects

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 components of the F-logic system seen in figure 1, as well as some other objects (e.g. integers, strings, or blocks, cf. section 2.10), are such objects available at the user interface. These 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*`].`

[2]In conventional systems, one would use variables for that purpose.

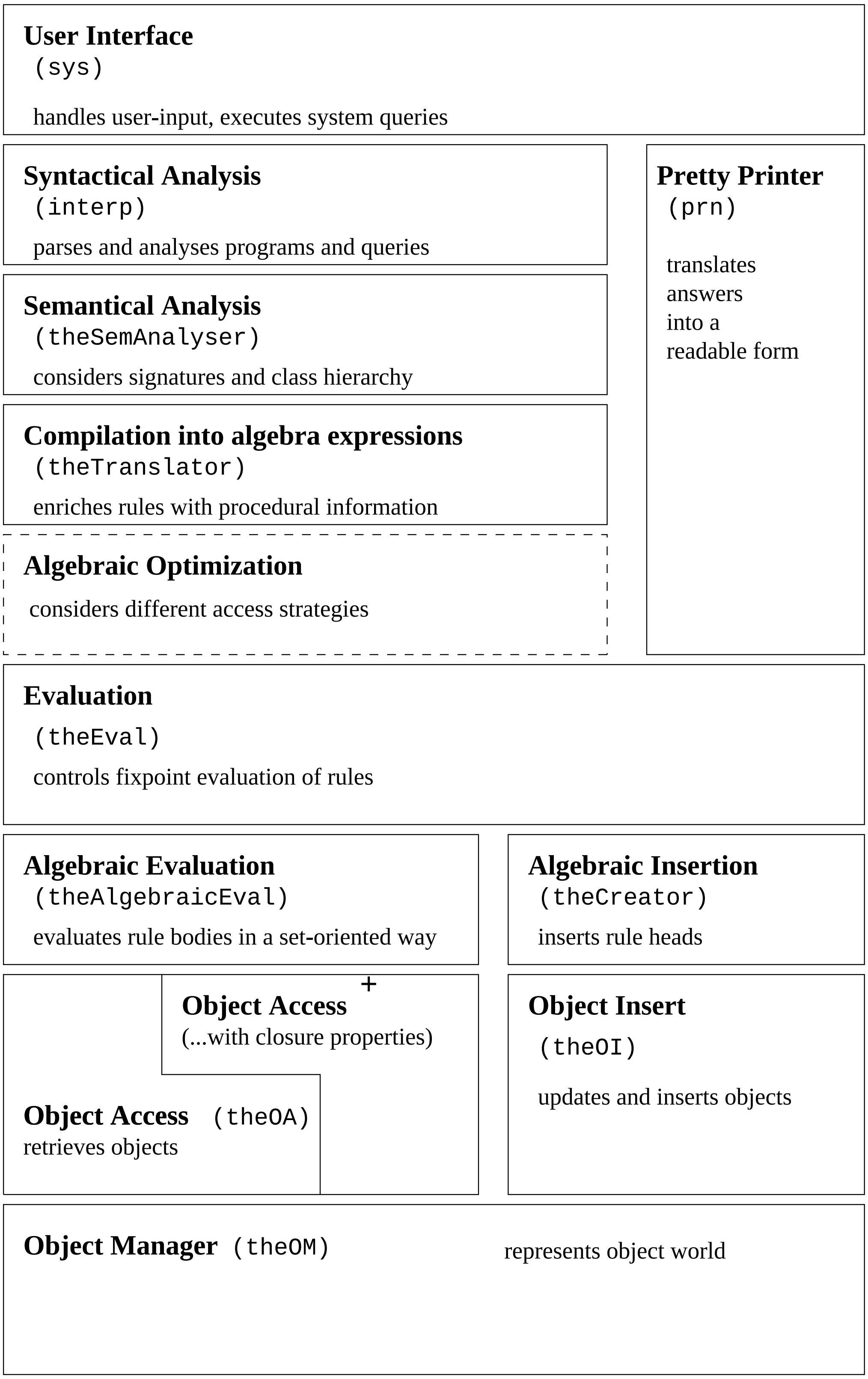

Figure 1: Structure of the F-logic system

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")].
```

There are also a number of *class* objects available at the user interface. They provide a construction kit to build an F-logic system according to individual needs. Class objects have 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 (Version 2.0 now provides a semi-naive evaluation module). FLORID can even hold several separate F-logic 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 2.13 for details of the system building process. Normally, the class objects are only used in configuration files.

2.5 System Commands

Besides assigning interface objects, as described above, system commands can send messages to interface objects. Syntactically, such commands are scalar path expressions of the form:

`?- sys.`*expression*`[].`

The object returned by *expression* is ignored. Therefore only expressions with side effects are of interest. For example the command

```
?- sys.centralStatistic.show[].
```

will be 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 statistic handler to print the runtime statistics table.
In addition to the interface objects, *sys* also provides some built-in methods:

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

`?- sys.display[].`	show all existing interface objects
`?- sys.end[].`	exit FLORID
`?- sys.eval[].`	evaluate the current program, i.e. calculate a model (sec. 2.8)
`?- sys.tp[].`	apply T_P operator of current program once to the (partial) model (sec. 2.8)
`?- sys.showProgram[].`	show current program (facts and rules)
`?- sys.forgetProgram[].`	clear the current program
`?- sys.forgetIDB[].`	clear the Object Manager
`?- sys.consult@("foo.flp")[].`	read the program file *foo.flp* (sec. 2.9)
`?- sys.debugOn[].`	enter debug mode for consult
`?- sys.debugOff[].`	(see section 2.11)
`?- sys.switch@(interpreter,symTab)[].`	attaches an FLogicInterpreter to the system (used in *config.flp*, see sec. 2.13)
`?- sys.echo@("...")[].`	print argument string
`?- sys.open@("filename")[].`	opens a FLSysOStream to write to a file (sec. 2.7)
`?- sys.remove@("thisobject")[].`	deletes an interface object.

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). See section 2.7 for details about output channels.

Appendix D contains a number of example sessions that demonstrate the use of system commands more explicitely. A number of frequently used system commands are loaded to the readline history (see app. B). They can be accessed with the up and down arrow keys.

2.6 Syntactical Restrictions for System Commands

For system commands, only a subset of the full F-logic syntax [5] 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. Sending the `help` method to any interface object lists the methods that are defined for this object.
   ```
   ?- sys.theEval.foo[].
   Unknown method : foo
   ```

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. System queries may contain only one specification with at most one element.

```
?- sys.isOk[].display[].

System query with molecule within a path.
Properties are allowed only at the end of path reference.
```

2.7 Output Handling

All output that FLORID generates runs through so-called *output channels*. Output channels wrap standard C++ output streams and enable to centrally redirect them. 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.

Let's send all debug output to the file *logfile* in the current directory:

```
?- sys[mystream->sys.open@("logfile")].
?- sys.debugChannel.setStream@(mystream)[].
 ...
?- sys.debugChannel.resetStream[].
?- sys.remove@("mystream")[].
```

By removing the interface object `mystream`, 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)[].`".

2.8 F-logic Programs and Evaluation

Programs are collections of facts and rules. In FLORID, queries are not part of a program. The system holds a *current program* handled by the FLogicInterpreter `interp`. This interface object serves as an encapsulation for all logical operations dealing with programs and queries. Typing "`?- sys.showProgram[].`" displays the program and "`?- sys.forgetProgram[].`" discards it. To add new facts and rules to the current program, you have to consult a program file (see section 2.9). 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. As mentioned above, FLORID uses a bottom-up evaluation strategy. This algorithm iteratively deduces new facts from already established facts using a forward chaining technique [3]. A program P gives rise to an operator T_P on partial models (for F-logic defined in [7]). 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[4] 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.

[4] 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.

Thus, the model of P is not unique. For a formal treatment of the semantics of F-logic and implemented algorithm see [7]. Evaluation of the current program is started with the system command "`?- sys.eval[].`".

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. To start from scratch, call "`?- sys.forgetIDB[].`" before evaluating. A single application of T_P can be achieved with "`?- sys.tp[].`".

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 prototype 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: The configuration file of FLORID 2.0 now includes a new 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 [12] for details about semi-naive evaluation for F-logic.

To see the rewritten program, set the debug mode `"program"` (see section 2.11.1).

2.9 Consulting Program Files

By convention, the names of files containing F-logic programs end with the extension `.flp`. However, this is only relevant if FLORID is used in combination with the emacs editor, because emacs enters a special F-logic mode when loading a file with that extension. See section 4 for emacs operation.

After entering "`?- sys.consult@("foo.flp")[].`" the following happens: The user interface opens the file `foo.flp` and reads it rulewise using an F-logic parser. If a rule is a query, it will be answered immediately. In case of a system query, it will be executed (by sending messages to interface objects). Otherwise, the query is passed to the object `interp` which serves as an encapsulation for the model represented by the object manager. The answer reflects the current state of the OM. Facts und rules are added to the current program held by `interp`. No evaluation takes place until the system query "`?- sys.eval[].`" is encountered. When end of file is reached, control is given back to the calling level (interactive mode or another consult). A syntax error will terminate the consult process and is reported at the text shell.

This proceeding allows a user-stratification of programs by inserting the system command "`?- sys.strat.doIt[].`" between two strata. This means executing a *block* (cf. section 2.10) consisting of the two system queries

```
?- sys.eval[].
?- sys.forgetProgram[].
```

(which could be used directely, too.)

Note that the method *forgetProgram* clears only the current program (which can be displayed by "`?- sys.showProgram[].`"), the contents of the object manager remain unchanged. The command "`?- sys.forgetIDB[].`" clears the OM (and leaves the program unchanged).

F-logic 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) is found, the rest of the line is considered as a comment. These formats may be mixed deliberately.

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")[].
```

2.10 Blocks

Sequences of 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 extern files.

Syntactically, a block declaration is a set-valued method definition with host object *sys*. Between the curly braces stands a sequence of goals, i.e. queries without "`?-`". In contrast to the F-logic semantics of multi-valued molecules, the goals form a list here, not a set, that is, 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.

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. sect. 2.13.)

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

To separate the strata of a program, simply put the command

```
?- sys.strat.doIt[].
```

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[].
")
```

2.11 Debug Modes

Several components of the system provide a debug mode which can independently turned on or off. All debug output will be sent to the debug channel so it can be easily redirected (see sec. 2.7).

2.11.1 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, TP applications and reached fixpoints are reported. Every time a rule is matched, a dot is printed. So you can watch the progress of complex programs or programs dealing with large models.

It the outcome of your program is not what you expected, you can add parameters to the *debugOn* command, telling the evaluation component what to monitor. Possible parameters are:

- "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 TP round, the contents of the object manager are printed. This is sensible only in the case of very small models. Otherwise you have to specify frames to watch, 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 TP 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 will be printed after each TP 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 will be printed after each TP 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 TP 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 partial model that is already established.

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`.

If the object manager provides Web access (see tutorial [5]), any Web action can be monitored with

```
?- sys.theOM.debugOn[].
```

Standard object managers ignore this command.

2.11.2 Tracing the System Input

If you get parser errors or have problems with consulting files and system commands, 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.

2.11.3 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 brackets. 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)
```

2.12 Display Modes of the Pretty Printer

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:

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

"instance"	Display answers as fully instantiated molecules.
"bound"	Display answers in prolog-like style with variables and objects bound to them.
"html"	Same as "bound", but write answers to an html file and send this file to a netscape browser.

The html style is in particularly useful with the Web version of FLORID. This version has been extended to process WWW queries (see also [5]). 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[].`".

Figure 2: Answer display in html style

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

Consulting the program yields the following:

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

X/o5   Y/o9

1 output(s) printed

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

X/peter   Y/o9

1 output(s) printed

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

X/peter   Y/peter.father

1 output(s) printed

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

X/peter   Y/mary.husband

1 output(s) printed

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

peter[father -> mary.husband]

1 output(s) printed
```

2.13 System Configuration in `config.flp`

The FLORID prototype was implemented in C++ [13]. Due to the object oriented programming style supported by C++ it seemed reasonable to transfer this modeling style to the user interface, too ([11]). This resulted in a flexibly configurable system. Important internal classes and objects of the prototype are directly accessible from the user interface and are combined from there. Thus the structure of the F-logic system reflects at the interface (Fig. 1)[5]. 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` first. Normally this is the file `config.flp`. It generates the components of a standard F-logic prototype and links them to build the system. 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 customisations to the config file instead of typing them again and again at the FLORID shell.

config.flp:

```
// Configuration file (FLORID 2.0)
// C. Schlepphorst 11.12.95, 30.04.96, 29.07.97

?- sys.centralErrorHandler.setOutputFilter@(2)[]. // short error messages
?- sys[theOM->classOM.new].
?- sys[theOA->classOA.new@(theOM)].
?- sys[theOAplusISA->classOAplusISA.new@(theOA)].
?- sys[theOAplusSig->classOAplusSig.new@(theOAplusISA)].
?- sys[theOAplusBI->classOAplusBuiltIn.new@(theOAplusSig)].
?- sys[theOI->classOI.new@(theOM,theOAplusISA)].
?- sys[theOAplusInherit->classOAplusInherit.new@(theOAplusBI,theOA,theOI)].
?- sys[theCreator->classCreator1.new@(theOI,theOAplusInherit)].
?- sys[theAlgebraicEval->classAlgebraicEval.new@(theOAplusInherit)].
?- sys[theTranslator->classTranslator.new].
?- sys[theMatcher->
       classMatcher.new@(theTranslator,theCreator,theAlgebraicEval)].
?- sys[theEval->classPrSemiNaive.new@(theMatcher,theOAplusInherit,theOM)].
?- sys[theSemAnalyser->classSemanticAnalyser.new].
?- sys[prn->classPrettyPrinter.new@(theOM)].
?- sys[interp->classFLogicInterpreter.new@(theEval,prn,theSemAnalyser)].
?- sys.switch@(interp,theOM.giveSymTab)[].
?- sys[theNormalizer->classNormalizer.new@(interp)].
?- sys[strat->>{sys.eval[],sys.forgetProgram[]}].
?- sys[interf->classGraphicInterface.new@(sys,theOAplusSig)].
```

[5]Algebraic optimization is optional and not implemented at the moment

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[theOA->classOA.new@(theOM)].
```

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

```
?- sys[theOAplusISA->classOAplusISA.new@(theOA)].
?- sys[theOAplusSig->classOAplusSig.new@(theOAplusISA)].
?- sys[theOAplusBI->classOAplusBuiltIn.new@(theOAplusSig)].
```

Three derivations of the OA provide F-logic's closure properties regarding class hierarchy and some built-in data types. Later a fourth derivation is generated, the OAplusInherit.

```
?- sys[theOI->classOI.new@(theOM,theOAplusISA)].
```

The object inserter encapsulates write access to the object manager.

```
?- sys[theOAplusInherit->classOAplusInherit.new@(theOAplusBI,theOA,theOI)].
```

This derivation of the OA adds the trigger concept (cf. [7]) for inheritance. For inserting the resulting new facts to the OM, the OI is passed as a parameter.

```
?- sys[theCreator->classCreator1.new@(theOI,theOAplusInherit)].
?- sys[theAlgebraicEval->classAlgebraicEval.new@(theOAplusInherit)].
?- 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,theOAplusInherit,theOM)].
```

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@(theOM)].
```

The pretty printer displays answer sets in readable form. To do this, it needs direct 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 F-logic 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 FLSysInterpreter.

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

With this command, a block for user stratification is defined.

```
?- sys[interf->classGraphicInterface.new@(sys,theOAplusSig)].
```

The object implementing the GUI is defined. Note that its name has to be `interf` for the `-i` option to run correctly.

3 A Graphic Interface Based on Tcl/Tk

3.1 Basics

The programming language Tcl ("tickle") together with the graphic toolkit Tk [10, 15] has recently become widely used because it provides quick and effective designing facilities for graphic user interfaces with high functionality. In what follows, a Tcl/Tk-based FLORID interface (GUI) is presented which builds upon the text shell described above. Additionally, some Tcl scripts are needed. The environment variable SCRIPTPATH must contain the path to a directory containing these scripts (cf. appendix A). Further information about Tcl/Tk can be found in appendix C.

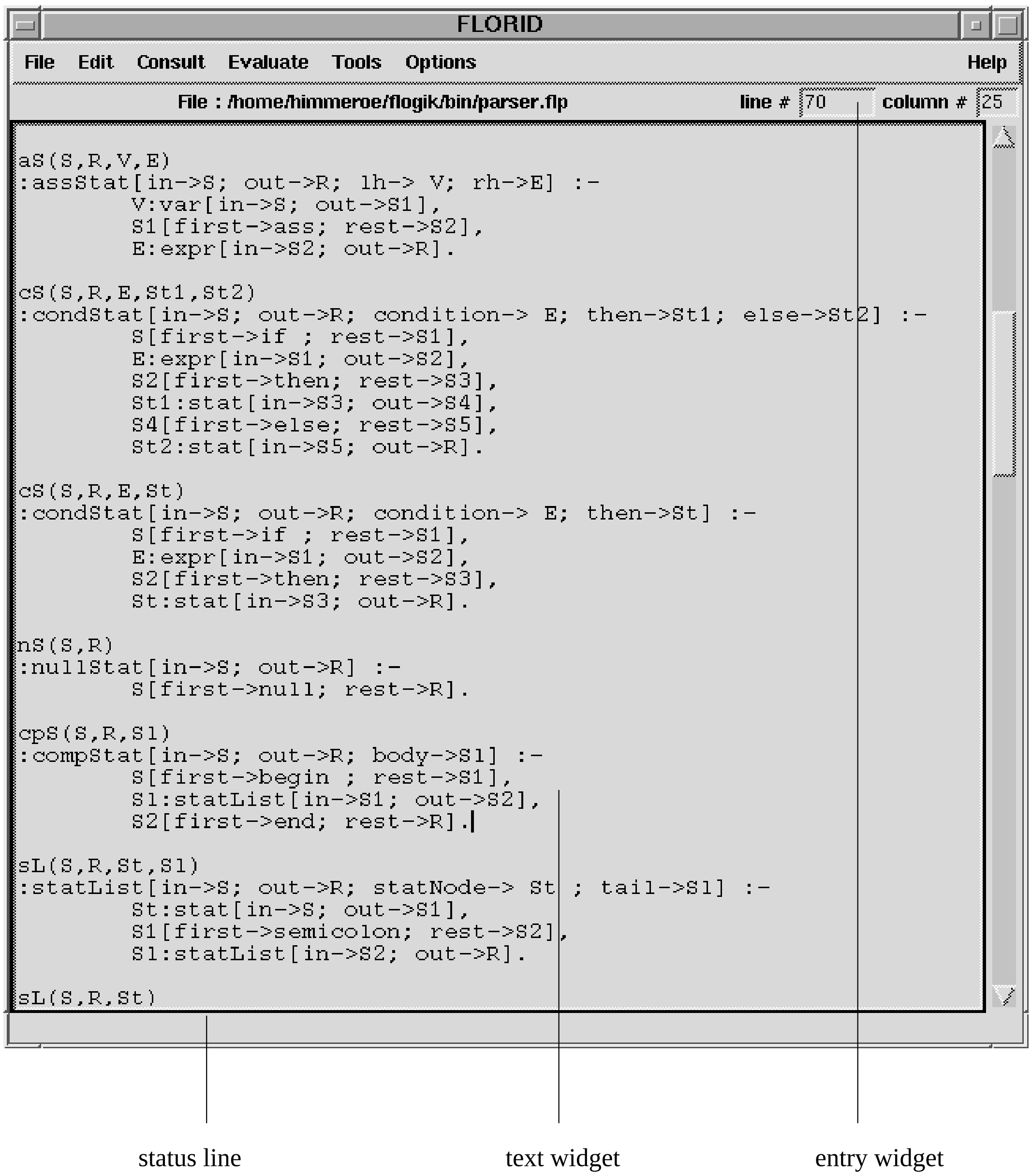

Figure 3: The main window of the Tcl GUI

By the system command " `?- sys.interf.run[].`" the graphic interface is launched. Note that the configuration file (see section 2.13) must contain a line creating `interf`. Then the main window appears as shown in figure 3.[6]

The interplay between the graphic interface and the underlying prototype is very simple. Certain actions of the user, for example consulting files, evaluating programs etc. are mapped onto system commands which are sent to the prototype. The other way round, messages of FLORID like answer sets, errors and statistics are displayed within Tcl-controlled windows.

Changes in the configuration of FLORID (cf. sect. 2.13) may require that the corresponding system commands that the Tcl scripts send to the prototype are customized too. Otherwise the mapping described above might fail.

Other user actions like file handling and editing take place on the GUI level without any information exchange with the prototype. In version higher 4.0, the Tk editor (text widget) provides a number of predefined key commands which are listed in appendix C.1.

Note that when using the GUI, it is possible to interrupt the evaluation process or cancelling FLORID by pressing `Control-\` or `Control-c` *at the shell FLORID was started from.*

3.2 The File Menu

After choosing the **File** item from the menu bar, the file menu opens (Fig. 4). It offers the usual editing facilities: The subitem **Load** (File/Load) loads a program file into the interface's buffer. To do that, a new file dialog window is generated presenting a list of the current files and directories (see sect. 3.2.1) from which you can choose a file. New files can be generated, too.
Subitem **Save** (File/Save) writes the buffer contents back to the file it was loaded from. If the cursor is located within the text widget, you may also use the key code `Control-s`.
To save the buffer under another file name, use **Save As** (File/Save-As). Then a file dialog pops up. The **Include** (File/Include) subitem inserts a file at the current cursor position. **Empty document** (File/Empty-document) clears the buffer if you want to start a new program from scratch. The last subitem **Exit** (File/Exit) leaves the GUI and returns to the FLORID text shell.

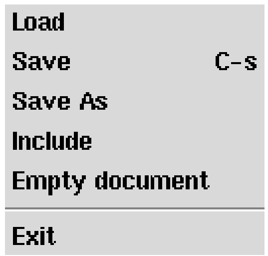

Figure 4: The file menu

3.2.1 The File Dialog

The file dialog shown in figure 5 opens if one of the subitems File/Load, File/Save-As or File/Include are chosen. (The button inscriptions differ of course in the Save case). There are predefined key kombinations for the entry widgets (see appendix C.1).

You can select files and directories by editing the contents of the entry widgets as well as by mouseclicking within the list boxes. Doubleclicking on a line in the **Directories** listbox changes the path in the entry widget appropriately. A single click on a filename in **Files** transfers the name to the entry widget whereas doubleclicking starts the action (load, include, save as) on this file and removes the file dialog. Pressing the action button in the lower left corner or hitting return in the filename entry widget has the same effect. If the user has no access permission for the chosen file and action, an error message is reported.

[6]Please see the installation hints in appendix A before starting the Tcl/Tk GUI the first time.

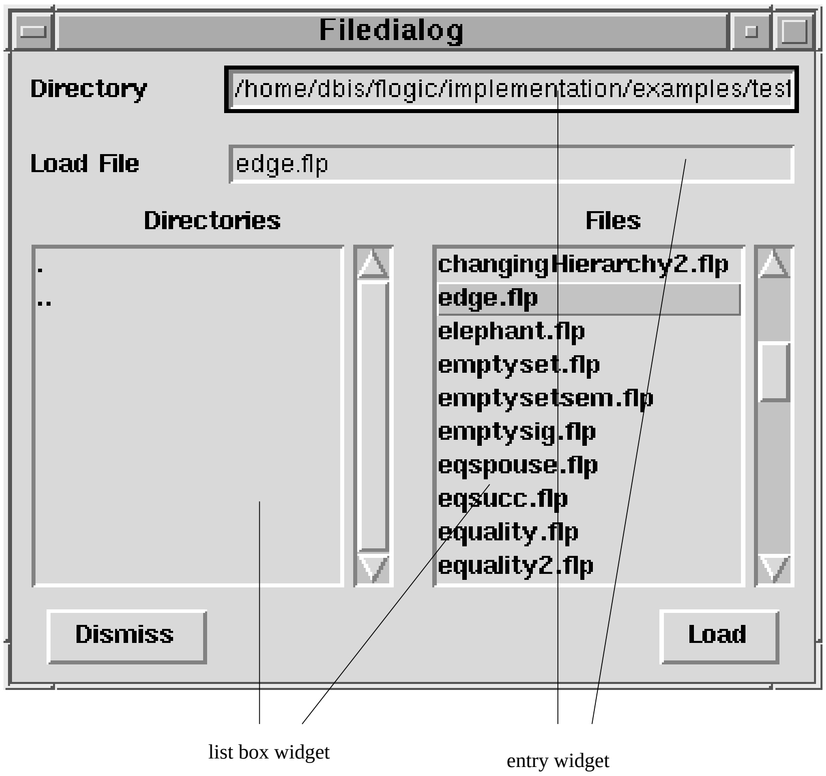

Figure 5: The file dialog for the File/Load subitem

In the load case, the file dialog offers the possibility to generate new files, too. Just enter the new filename into the **Load** entry widget and press return. If the file does not exist, a pop-up dialog will ask for confirmation. On confirmation, the file is created.

Pressing **Dismiss** removes the file dialog.

3.3 The Edit Menu

By choosing **Edit** from the menu bar, the menu seen in figure 6 opens. As long as no text portion is selected, the subitems **Cut** (Edit/Cut) and **Copy** (Edit/Copy) are deactivated. **Cut** kills the selected text portion and buffers it, whereas **Copy** only buffers the selection. If the copy/paste buffer is not empty, **Paste** (Edit/Paste) becomes activated. Selecting it inserts the text portion from the copy/paste buffer at the current cursor position.

The use of **Search&Replace** and **Search via scrollbar** is explained in the subsequent sections. Text portions found are highlighted (tagged). To remove those tags, choose **Delete tags** (Edit/Delete tags).

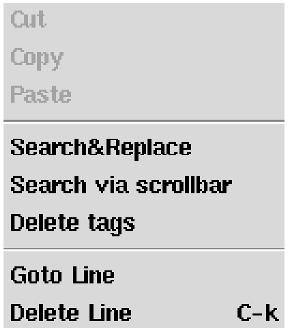

Figure 6: The edit menu

After selecting **Goto Line** (Edit/Goto Line) the **line #** entry field becomes active (see fig. 3). Enter a line number and press return to position the cursor in this line of the text widget. Alternatively the **line #** field can be activated by clicking into it, just as the **column #** field.

3.3.1 The Subitem *Edit/Search&Replace*

Choosing the subitem **Search&Replace** (Edit/Search&Replace), you get an additional frame below the text widget (see fig. 7). The **Dismiss** button removes the frame again. Before

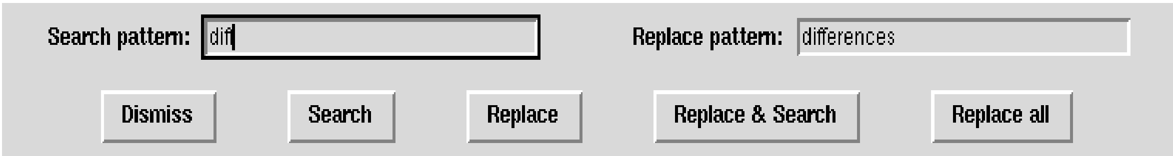

Figure 7: The Search&Replace frame

using the second button, you have to specify a search pattern in the **Search pattern** field. Otherwise you will get a reminder message (fig. 3). The search pattern may be a *regular expression* as well.

Examples:

Search pattern	Meaning
`showProgram`	Looks for the string `showProgram`
`di.*]\.$`	Looks for any string beginning with `di` and ending with `].` at a line end. Between `di` and `].` any text except newline is allowed

The search always starts at the current cursor position. You may use the **Replace** button only if there is a pattern specified in the **Replace pattern** entry field and a search succeeded before. If a match for the search pattern is found, it is highlighted (tagged). Pressing **Replace** then changes the highlighted part into the replace pattern. **Replace & Search** works just like **Replace** but starts the search again to get the next match. Pressing the fifth button (**Replace all**) replaces all matches of the search pattern in the whole text buffer at once.

If an error occurs during any action in the Replace & Search frame, it is displayed in the status line at the bottom of the window (see fig. 3).

3.3.2 The Subitem *Edit/Search via scrollbar*

After choosing the subitem **Search via scrollbar** (Edit/Search-via-scrollbar), the window shown in fig. 8 opens. Enter a search pattern in the entry field **Search pattern** and press

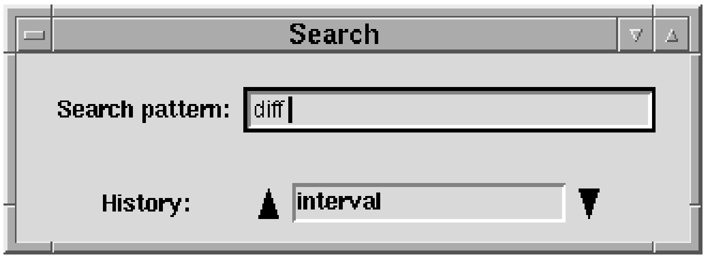

Figure 8: The Search via scrollbar frame

return. Then the search takes place in the whole text buffer and all matches will be highlighted. As in Edit/Search&Replace you may use regular expressions, too, as search patterns.

Additionally, Edit/Search-via-scrollbar has as history of all patterns searched in a session. By clicking onto the arrow symbols left and right of the history field you step through the list. A single mouseclick copies the pattern into the **Search Pattern** field and after pressing return, the search starts.

3.4 The Consult Menu

The subitems of this menu have a direct mapping to system commands of the underlying prototype. When an item is chosen, the GUI sends the corresponding command sequence to FLORID. For detailed information about the system commands see sect. 2.5. These subitems are mapped onto the following system commands:

Consult File	`?- sys.consult@("filename")[].`
Show Program	`?- sys.showProgram[].`
Forget Programm	`?- sys.forgetProgram[].`
Forget IDB	`?- sys.forgetIDB[].`

Figure 9: The consult menu

Consult File opens a file dialog (see sect. 3.2.1) to let you choose a filename. Then FLORID consults this file without changing the contents of the text buffer. By doing so it is possible to hold and edit some rules and queries in the text buffer and consult a big file containing facts without having to load it first. **Consult Buffer** on the other hand simply sends the buffer contents to FLORID without any file operation. In the example above, you could use **Consult Buffer** to add the rules in the buffer to the facts already consulted and to execute the queries. Note that consulting a file does not start evaluation unless the file contains the system command "`?- sys.eval[].`" or similar.
Reset System is the combination of the subitems **Forget IDB** and **Forget Program**.

3.5 The Evaluate Menu

Choosing **Evaluate Program** (Evaluate/Evaluate Program) sends the command "`?- sys.eval[].`" to the prototype and thus starts the evaluation of the program (note that this is not necessarily the same as the program displayed in the text widget, it is what you have *consulted* before. You may display it by choosing **Show Program** from the consult menu).
Likewise, **TP-Operator once** (Evaluate/TP-Operator once) sends "`?- sys.tp[].`" to FLORID. The subitem **Ask Ad Hoc Query** lets you examine the calculated model directly. To do this, a new frame opens below the text widget (fig. 11).

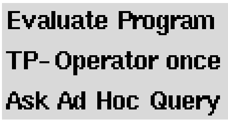

Figure 10: The evaluate menu

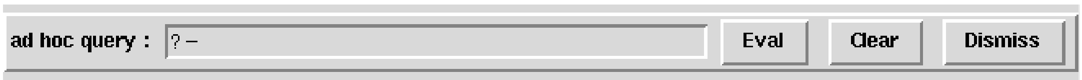

Figure 11: The Ad Hoc Query frame

By clicking on the **Eval** button you send the query to the prototype. Hitting return does the same. The **Clear** button empties the entry field and pressing **Dissmiss** removes the frame again.

Besides logical queries to examine the model, system queries can be sent, too. Since not all system commands (queries) are represented as Tcl/Tk menue items, in some cases (e.g setting debug modes) it is necessary to type in system commands directly. This should be used with care, however, not to interfere with the interplay between FLORID and the GUI. Of course, you can use any query inside the consulted files.

After a query has been sent, another frame opens below the query frame: the answer frame (fig. 12). By clicking on the arrows you can change the size proportion of answer frame and text frame. It is also possible to drag the separation bar with the mouse.

Pressing **Clear** empties the frame whereas **Dismiss** removes it.

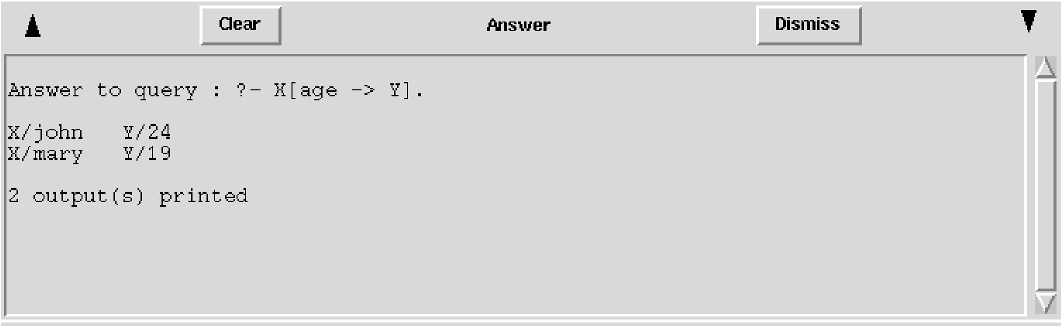

Figure 12: The Answer frame

3.6 The Tools Menu

By choosing the subitem **Statistics** (Tools/Statistic) you get a new frame (fig. 13). Here, the time consumption of the last evaluation calls are displayed. **Clear** resets the values stored and **Dismiss** removes the frame.

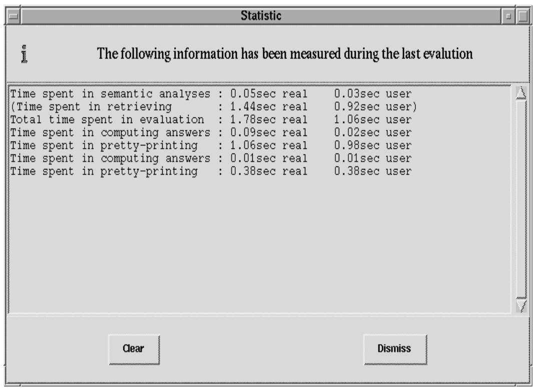

Figure 13: The Statistic frame

Clicking on the subitem **Browser** (Tools/Browser), a window for navigation through the object world opens (figure 14). In the Browser menu the items **Open object** and **Open predicate** create windows displaying the corresponding frames of objects and predicates. **Raise Object-Frame** moves an existing object-frame window to the front. If the state of the object manager has changed after you have displayed some frames, you can get the current state by **Refresh all**. So the Browser is a helpful means for monitoring the state of objects while using **TP-Operator once** (Evaluate/TP-Operator once) if you refresh the display after each TP-Iteration. **Close all** removes all frames in the browser, and **Quit** leaves it.

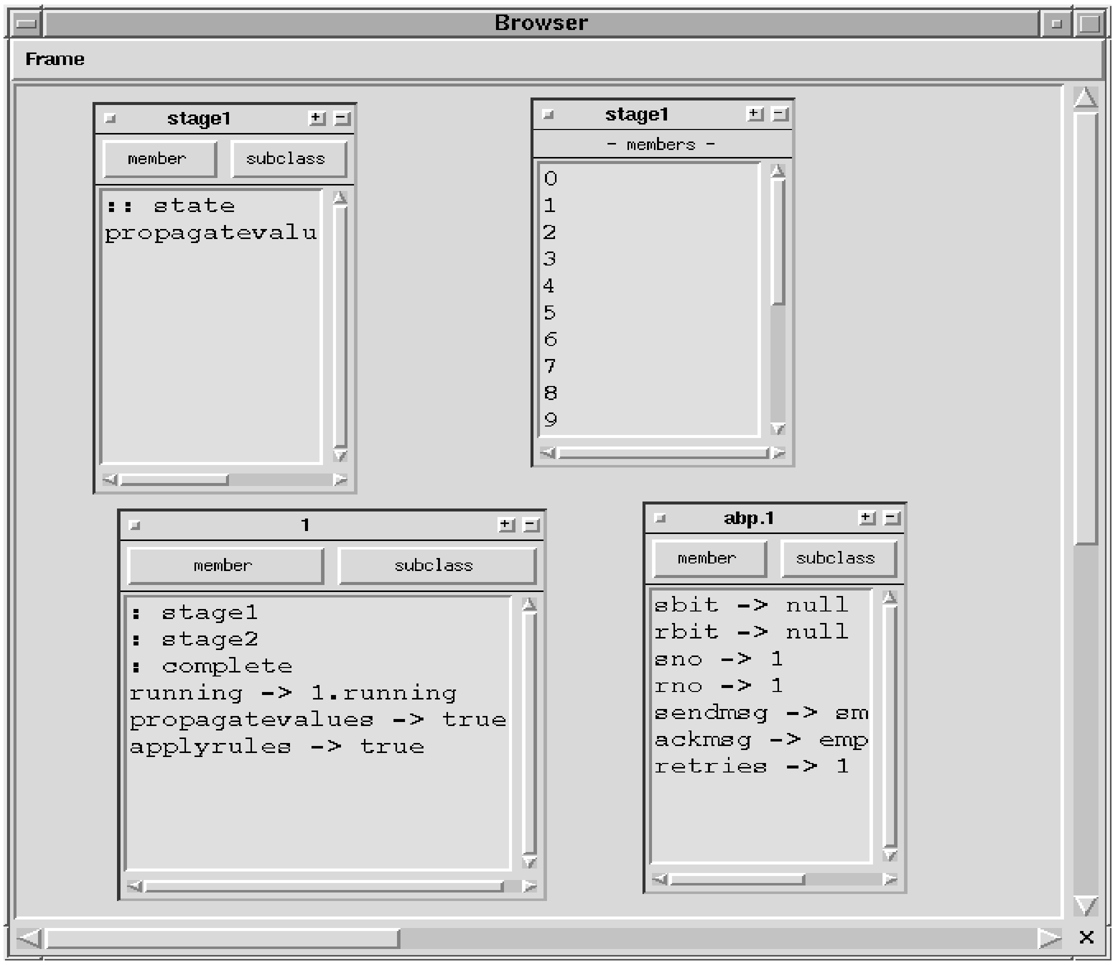

Figure 14: Browser window

Not all information belonging to an object is displayed in a single object frame. Information about members, subclasses, parametrized and set-valued methods is shown in so-called *subframes* (figure 15). By clicking on objects listed in a frame, their corresponding frame opens. When an object frame is closed, all depending subframes close, too.

There is no special debugging tool that covers all debug features of interface objects (see section 2.11). Instead, all debug messages are displayed in a special window.

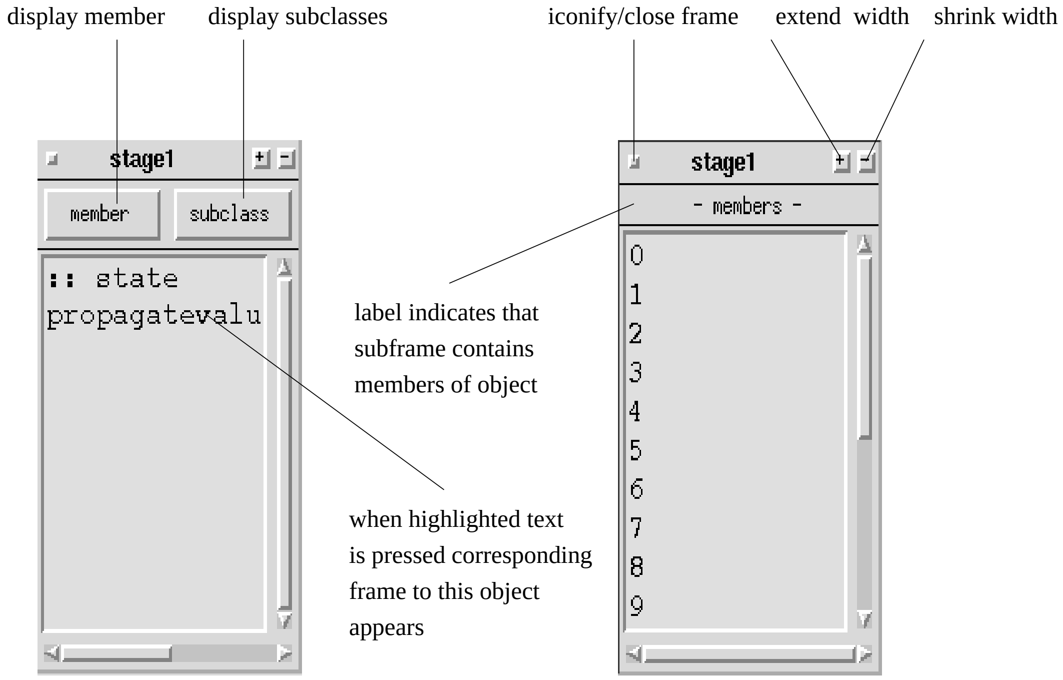

Figure 15: Frame and subframe of object

3.7 The Options Menu

In this menu you have the possibility to customize some parameters of the GUI. The subitem **Fonts** (Options/Fonts) lets you choose a font for the text widget. The window shown in fig. 17 opens and the desired font can be selected. By selecting **Highlighting** (Options/Highlighting) a submenu pops up that lets you turn the highlighting mode on or off. The option **Brackets** means that matching Brackets are signalled by flashing. The subitems **Answer Mode** (Options/Answer Mode) and **Answer Output** (Options/Answer Output) customize the pretty printer. See the description in section 2.12. If **Equality Trace** (Options/Equality Trace) is on, equating of objects is reported during evaluation. The corresponding messages appear on the error frame.

Figure 16: The Options menu

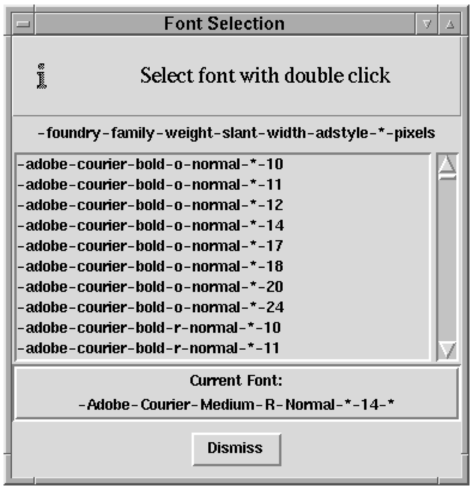

Figure 17: The Fonts window

3.8 System Messages of the Graphic Interface

As mentioned before, system messages are displayed in the status line of the main window (fig. 3). However, there are some errors and warnings which are displayed in top-level windows instead. If such a message appears (as shown in fig. 18), the GUI is blocked until the **OK** button is pressed.

Figure 18: Error window

In certain situations the GUI needs the user to make a decision. Then a top-level window pops up (fig. 19) presenting two buttons labelled **Yes** and **No**. Only after selecting one of them, the user can go on working with the GUI.

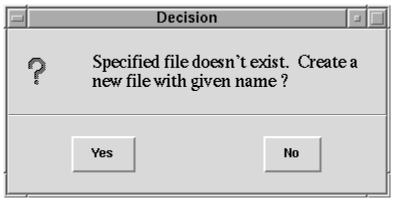

Figure 19: Decision window

4 Embedding FLORID into the Emacs Editor

Operating the prototype 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.[7]

In order to use FLORID from emacs, emacs must have access to the file `flp.el`. See appendix A for details.

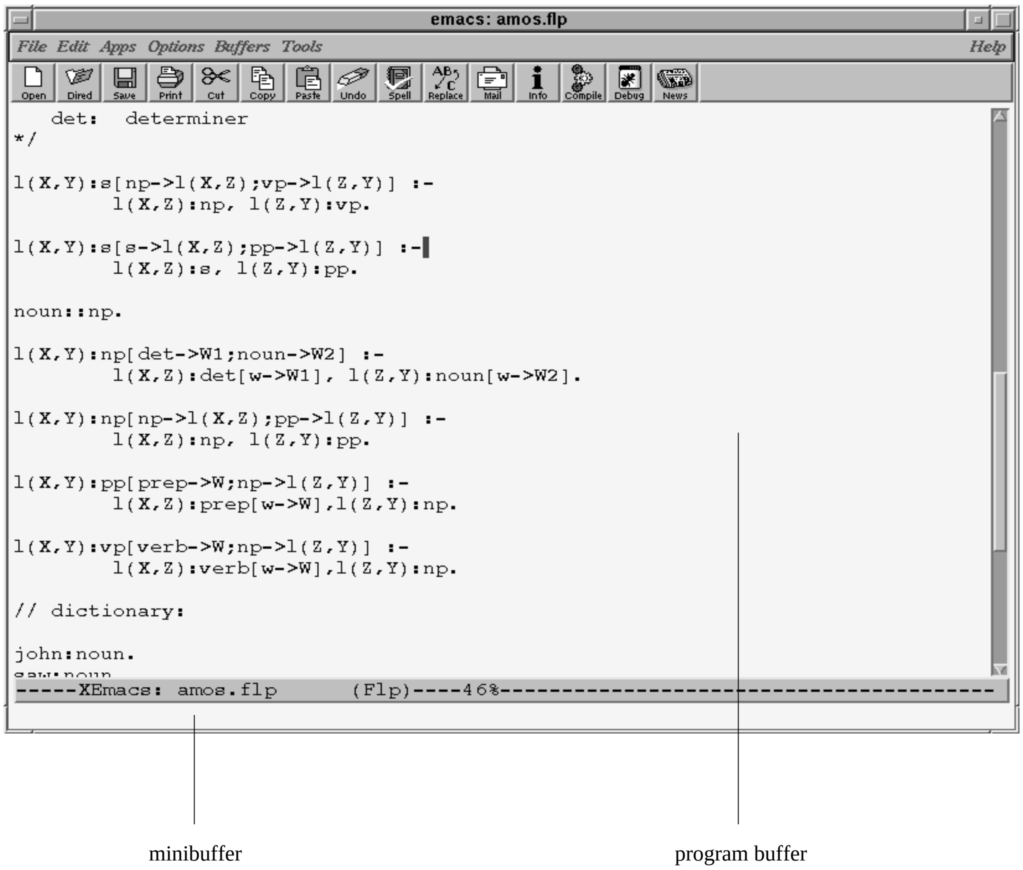

Figure 20: XEmacs 19.13

When the emacs edits a file ending with `.flp`, it automatically enters the F-logic mode (fig. 20). If the key combination `Control-c Control-c` is pressed in this buffer, a new buffer is created with a FLORID prototype attached to it. FLORID's output is printed here (fig.21). It gets the contents of the first buffer as a file to consult.

If the edited file does not end with `.flp`, the prototype can be started manually by the command `Meta-x run-flp`. On pressing `Meta-x` the emacs cursor jumps into the minibuffer where the command is entered.

Then again a new buffer is created as in fig. 21, from which you can use FLORID as described in section 2. Unlike in the shell, step through the history by Meta-P (Previous Item) and Meta-N (Next Item).

[7]The first adaption of the prototype to emacs was done by Heinz Uphoff by customizing the emacs-Prolog interface.

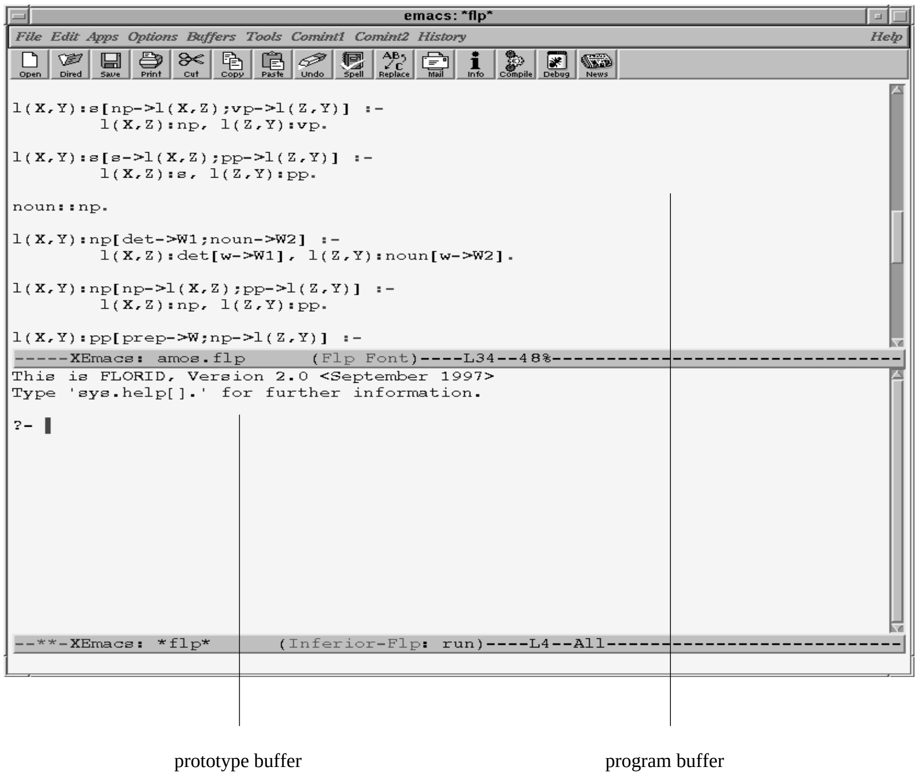

Figure 21: XEmacs 19.13 in flp mode

In the flp mode (i.e. in the program buffer, *not* in the prototype buffer) the following key codes are defined for interaction with FLORID:

`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 2: Special keycodes in the flp-mode

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.

The flp-mode provides a syntax highlighting facility to make F-logic programs more readable. It uses the font-lock package shipped with XEmacs 19.13 or higher. If font-lock is not available in your version of emacs, syntax highlighting does not work.

Please set Syntax Highlighting:Colors and Auto-Fontify in the XEmacs Options menu.

A Installation

By downloading FLORID from ftp you get a packed and compressed file. In the following we call this file `florid.tar.gz`[8]. First uncompress it 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/doc
florid/examples
florid/examples/tutorial
florid/lib/config
florid/lib/tcltk
```

The directory `florid/bin` contains several executables. The reason is that executables with Tcl/Tk interface and/or Web access may not run on every system (e.g. if Tcl/Tk is not installed on your computer, or your computer is not connected to the Internet).

Executables with Tcl/Tk-GUI need a certain version of the X11 libraries (X11R6.1), causing a segmentation fault if not found. You should modify the file `config.flp` in the following way, depending on the executable you use:

- For a version with Web access replace `?- sys[theOM->classOM.new].` by
  `?- sys[theOM->classWebOM.new].`
- For a version without Tcl/Tk-GUI, remove the line
  `?- sys[interf->classGraphicInterface.new@(sys,theOAplusSig)].`

The directory `florid/doc` contains postscript files of the user manual (`manual.ps`) and the F-logic tutorial (`tutorial.ps`). Publications related to F-logic and FLORID are available from `http://www.informatik.uni-freiburg.de/~dbis/Publications/` .

In the directory `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`. `florid/lib/config` contains the following:

- `config.flp` the configuration file (section 2.13)
- `default.his` history lines to preload
- `flp.el` the emacs flp-mode definition file

The Tcl scripts needed to run the Tcl/Tk GUI are stored in the directory `florid/lib/tcltk`. The installation hints below refer to this directory structure. Any problems you have either in installing the system or in using it can be reported by e-mail to

`florid@informatik.uni.freiburg.de`

Comments, suggestions, and bug reports are welcome.

[8] If your archive name is `floridXXX.tar.gz`, please substitute `floridXXX` for `florid` in the following lines.

A.1 Environment Variables

Shell environment variables are used to set paths leading to FLORID's system files. These are needed to configure the prototype automatically. It is also possible to specify the pathes by command line options (see sect. 2.1). The following variables should be set before starting FLORID:

- SCRIPTPATH
- DEFAULTCFG
- DEFAULTHIS

`SCRIPTPATH` specifies the path to the Tcl/Tk scripts (see sect. 3), `DEFAULTCFG` says 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 2.13 for details.

If Tcl/Tk is not installed or you don't want to use the GUI, `SCRIPTPATH` is not needed. If one of the other variables is not set and the respective command line option is missing, the system will print a warning. In case of missing `DEFAULTCFG` the F-logic system has to be built "by hand".

```
naxos:~/florid/bin> florid
This is FLORID, Version 2.0 <September 1997>
Type 'sys.help[].' for further information.
Environment variable DEFAULTCFG doesn't contain path to configuration file.

Environment variable DEFAULTHIS doesn't contain path to history file.
```

In what follows we assume that the main directory `florid` is located at `/home/db/`. Then the environment variables are

- SCRIPTPATH : /home/db/florid/lib/tcltk/hdforms.tcl
- DEFAULTCFG : /home/db/florid/lib/config/config.flp
- DEFAULTHIS : /home/db/florid/lib/config/default.his

You can set the variables either directly from the shell or automatically in an operating system file. For the *bash* shell use the commands

```
export SCRIPTPATH="/home/db/florid/lib/tcltk/hdforms.tcl"
export DEFAULTCFG="/home/db/florid/lib/config/config.flp"
export DEFAULTHIS="/home/db/florid/lib/config/default.his"
```

A.2 Using the Tcl/Tk GUI

We assume that the user has already installed Tcl/Tk. By default, the file `hdforms.tcl` (`florid/lib/tcltk/hdforms.tcl`) is executed when the GUI starts up. The first line of this file calls the Tcl interpreter *wish*:

```
#!/usr/X11R6/bin/wish -f
```

Change the path /usr/X11R6/bin to the path leading to *wish* on your system. Otherwise the interpreter would fail. The scripts are based on Tcl 7.4 and Tk 4.0. Older version cannot be used and execution terminates if such are detected.

Further the Tcl/Tk environment variables Tcl_Library and Tk_Library have to be set to point to the Tcl/Tk libraries on your system. After downloading FLORID the Tcl interpreter used by FLORID contains paths to libraries of our Tcl/Tk installation. If you don't customize Tcl_Library and Tk_Library, starting the Tcl/Tk GUI fails.

If you change the Tcl scripts or add some new scripts, you have to delete the file florid/lib/tcltk/tclIndex before starting the Tcl/Tk GUI. Otherwise your changes will have no effect.

A.3 Using FLORID from Emacs

The emacs flp-mode is defined in the file flp.el (florid/lib/config/flp.el). To tell emacs about this mode, extend your local .emacs file by the following lines:

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

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

To be sure that emacs actually finds flp.el you should either add the lines

```
(setq load-path
        (cons "/home/db/florid/lib/config/flp.el"
              load-path))
```

or copy flp.el into a directory where emacs looks for files.

In flp.el you must specify where to find the florid executable on your system. Please change the first command to the actual path.

Instead of the original lines

```
;; This path has to be set at installation of the F-logic-System!!!
(defvar flp-program-name "/home/dbis/flogic/implementation/bin/florid"
  "*Program name for invoking an inferior Flp with `run-flp'.")
```

you would write (according to the assumptions above)

```
(defvar flp-program-name "/home/db/florid/bin/florid"
  "*Program name for invoking an inferior Flp with `run-flp'.")
```

After that, create the file /home/.florid-history as a copy of default.his. The existence of this file is essential for emacs to work properly. After these installation steps emacs has to be started again to let the changes take effect.

B Readline and History Commands

The following tables list the key commands available for the readline and the history packages, they were taken from [2]. 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 $\uparrow$ key.
`C-n`	Move 'down' through the history list. You can also use the $\downarrow$ 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 Tcl/Tk

Tcl/Tk is a programming system developed by John Ousterhout at the University of California, which has very useful graphic interface facilities. Tcl is the basic programming language, while Tk is a ToolKit of so-called widgets, graphic objects similar to those in other GUI toolkits such as Xlib, Xview and Motif. Unlike with many of the other toolkits, it is not necessary to use C or C++ in order to manipulate the widgets, and complex applications can be built very rapidly with some knowledge about the Tcl/Tk.

If you have not installed Tcl/Tk you can get it via ftp from

ftp.smli.com

The release is stored in /pub/tcl and the Draft of [15] is in /pub/tcl/welch.

You can get the release also via Sun's Tcl/Tk developement homepage

http://sunscript.sun.com/

The newsgroup related to Tcl/Tk is comp.lang.tcl.

C.1 Default Bindings

The following table lists the default bindings of text widgets for Tcl 7.4 and Tk 4.0.

<Any-Key>	Insert normal printing characters.
<Button-1>	Set the insert point, clear the selection, set focus.
<Control-Button-1>	Set the insert point without affecting the selection.
<B1-Motion>	Sweep out a selection from the insert point.
<Double-Button-1>	Select the word under the mouse.
<Triple-Button-1>	Select the line under the mouse.
<Shift-Button-1>	Adjust the end of selection closest to the mouse.
<Shift-B1-Motion>	Continue to adjust the selection.
<Button-2>	Paste the selection, or set scrolling anchor.
<B2-Motion>	Scroll the window.
<Key-Left>	Move the cursor left on character. Clear the selection.
<Control-b>	
<Shift-Left>	Move the cursor and extend the selection.
<Control-Left>	Move the cursor by words. Clear the selection.
<Control-Shift-Left>	Move the cursor by words. Extend the selection.
<Key-Right>	All Right bindings are analogous to Left bindings.
<Control-f>	
<Meta-b>	Same as <Control-Left> and <Control-Right> .
<Meta-f>	
<Key-Up>	Move the cursor up one line. Clear the selection.
<Control-p>	

`<Shift-Up>`	Move the cursor up one line. Extend the selection.
`<Control-Up>`	Move the cursor up by paragraphs, which are a group of lines separated by a blank line.
`<Contol-Shift-Up>`	Move the cursor up by paragraph. Extend selection.
`<Key-Down>`	All `Down` bindings are analogous to `Up` bindings.
`<Control-n>`	
`<Next>`	Move the cursor by a screenful. Clear the selection.
`<Prior>`	
`<Shift-Next>`	Move the cursor by a screenful. Extend the selection.
`<Shift-Prior>`	
`<Home>`	Move the cursor to line start.
`<Control-a>`	
`<Shift-Home>`	Move the cursor to line start. Extend the selection.
`<End>`	Move the cursor to line end. Clear the selection.
`<Control-e>`	
`<Shift-End>`	Move the cursor to line end. Extend the selection.
`<Control-Home>`	Move the cursor to the beginning of text. Clear selection.
`<Meta-less>`	
`<Control-End>`	Move the cursor to the end of text. Clear the selection.
`<Meta-greater>`	
`<Select>`	Set the selection anchor to the position of the cursor.
`<Control-space>`	
`<Shift-Select>`	Adjust the selection to the position of the cursor.
`<Control-Shift-space>`	
`<Control-slash>`	Select everything in the text widget.
`<Control-backslash>`	Clear the selection.
`<Delete>`	Delete the selection, if any. Otherwise delete the character to the right of the cursor.
`<Backspace>`	Delete the selection, if any. Otherwise delete the character to the left of the cursor.
`<Control-h>`	
`<Control-d>`	Delete character to the right of the cursor.
`<Meta-d>`	Delete word to the right of the cursor.
`<Control-k>`	Delete from cursor to the end of the line. If you are at the end of line, delete the newline character.
`<Control-o>`	Insert a newline but do not advance the cursor.
`<Control-w>`	Delete the word to the left of the cursor.
`<Control-x>`	Deletes the selection, if any.
`<Control-t>`	Transpose the characters on either side of the cursor.

Table 5: Default bindings for the text widget

The following table lists the default bindings of entry widgets for Tcl 7.4 and Tk 4.0.

`<Button-1>`	Set insert point in start a selection.
`<B1-Motion>`	Drag out a selection.
`<Double-Button-1>`	Select a word.
`<Triple-Button-1>`	Select all text in the entry.
`<Shift-B1-Motion>`	Adjust the ends of the selection.
`<Control-Button-1>`	Set insert point, leaving selection as is.
`<Button-2>`	Paste selection at the insert cursor.
`<B2-Motion>`	Scroll horizontally.
`<Left>`	Move insert cursor one character left. Start selection.
`<Control-b>`	
`<Shift-Left>`	Move cursor left and extend selection.
`<Control-Left>`	Move cursor left one word. Start selection.
`<Meta-b>`	Same as `<Control-Left>` .
`<Control-Shift-Left>`	Move cursor left one word and extend the selection.
`<Right>`	The bindings for Right correspond to the Left key.
`<Control-f>`	
`<Meta-f>`	Same as `<Control-Right>` , move right one word.
`<Home>`	Move cursor to beginning of entry.
`<Control-a>`	
`<Shift-Home>`	Move cursor to beginning and extend the selection.
`<End>`	Move cursor to end of entry.
`<Control-e>`	
`<Shift-End>`	Move cursor to end and extend the selection.
`<Select>`	Anchor the selection at the insert cursor.
`<Control-Space>`	
`<Shift-Select>`	Adjust the selection to the insert cursor.
`<Control-Shift-Space>`	
`<Control-slash>`	Selects all the text in the entry.
`<Control-backslash>`	Clears the selection in the entry.
`<Delete>`	Delete the selection or delete next character.
`<Backspace>`	Delete the selection or delete previous character.
`<Control-h>`	
`<Control-d>`	Delete next character.
`<Meta-d>`	Delete next word.
`<Control-k>`	Delete to the end of entry.
`<Control-w>`	Delete previous word.
`<Control-x>`	Delete the section, if it exists.
`<Control-t>`	Transpose characters.

Table 6: Default bindings for the entry widget

D 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*.

D.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, Version 2.0 <September 1997>
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:~/flogik/bin>
```

Scenario 2 : Evaluate `edge.flp`, discard the computed model and evaluate again by single step TP 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, Version 2.0 <September 1997>
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, Version 2.0 <September 1997>
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, Version 2.0 <September 1997>
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 taskes running in parallel (e.g. kernel, window manager). Additional to the measured time the number of TP iterations is displayed. If new facts were derived through inheritance, the number of these facts (fired inheritance trigger) is printed, too.

D.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[9]. The next two scenarios shall explain how to navigate through the user interface.

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

```
This is FLORID, Version 2.0 <September 1997>
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
}
```

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

```
?- 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, Version 2.0 <September 1997>
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*

D.3 System commands in files

Placing system commands into files is useful in the following situations:

- system configuration (sect. 2.13).
- automation of processes to avoid unnecessary typing
- generation of output for test purposes
- user stratification by hand

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, such commands are used in files which control evaluation.

By using the command "`?- sys.strat.doIt[].`" the user can partition the program into strata by hand. 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 sect. 2.10). 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

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[2] C. R. Brian Fox. GNU readline library, edition 2.0, for readline library version 2.0. Free Software Foundation, Cambridge, MA, 1994.

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

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

[5] J. Frohn, R. Himmeröder, P.-T. Kandzia, and C. Schlepphorst. How to write F-Logic programs in FLORID, Version 2.0, 1997. Available from ftp://ftp.informatik.uni-freiburg.de/pub/florid/tutorial.ps.gz.

[6] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. Intl. Conference on Very Large Data Bases*, Santiago de Chile, 1994.

[7] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

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

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

[10] J. Ousterhout. Tcl and the Tk toolkit. Addison-Wesley, Reading, Massachusetts, 1994.

[11] T. Pferdekaemper. Eine flexible Konfigurations- und Benutzerumgebung für F-Logik. Diplomarbeit Universität Mannheim, 1995.

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

[13] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 2nd edition, 1992.

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

[15] B. Welch. Practical programming in Tcl and Tk. Prentice Hall, 1995.