# LoPiX

Logic Programming in XML

# (Preliminary) User Manual

Wolfgang May

may@informatik.uni-freiburg.de

Institut für Informatik, Universität Freiburg
Germany

January 2001

# Contents

## Preface

LoPiX is an implementation of the XPathLog language; a logic-programming language based on XPath. The LoPiX system employs the evaluation component of Florid, an implementation of F-Logic whose data model and language syntax has many similarities with XML and XPath, respectively.

We assume that the reader of this tutorial is familiar with the basic concepts of deductive databases, e.g., Datalog [AHV95, CGT90, Ull89], or F-Logic [KLW95].

XPath [XPa99] is the standard querying language for XML data. Most XML querying an data transformation languages are based on XPath; e.g., XQL [Rob99], XSL(T) [XSL99], XML-QL [DFF+99], Quilt [CRF00], and the XLink/XPointer proposals. XPathLog extends XPath with logic-programming style variable and variable-binding concepts, and a semantics for XPath expressions in rule heads for updating an XML database.

The evaluation of programs in LoPiX is based on a set-oriented bottom-up computation, as an extension of the algorithm well known from Datalog [AHV95, CGT90, Ull89]. Here, the evaluation component of the Florid system is employed.

We assume that the reader is familiar with the basic notions of XPath and Logic Programming.

The structure of this manual is as follows:

The first part describes the installation and the environment of LoPiX: First, we give describe the installation of LoPiX for working from the unix shell or from within emacs. Section 1 describes how to run LoPiX in the unix shell or working with emacs. Section 2 gives an overview of the structure of LoPiX programs: Section 3 describes how XML documents in the Web can be accessed and presents the XPathLog query language. The syntax and semantics of XPathLog rules is described in Section 4. Section 5 describes additional built-in features, mainly concerning datatypes. Section 6 gives an overview of the theory of program semantivcs and evaluation.

# 1 Installation

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

```
gunzip lopix.tar.gz
```

The resulting file lopix.tar has to be unpacked by entering

```
tar -xvf lopix.tar
```

Now the directory lopix is created. It has the following subdirectories:

```
lopix/bin/
lopix/environment/
lopix/sgml/
lopix/doc/
```

The directory lopix/bin contains the binary, lopix/environment contains several files defining the LoPiX environment:

- lopix.cfg.in: the source of the configuration file
- lopix.his: history lines to preload
- lpx.el: the emacs lpx-mode definition file

configure LoPiX

First, the LoPiX configuration has to be adapted to the local system by changing to the directory lopix/environment/ and calling ./configure – this generates lopix.cfg from lopix.cfg.in.

The directory lopix/sgml contains several definitions needed when using SGML documents. The directory lopix/doc contains postscript files of this manual (lopix.ps) LoPiX are available from http://www.informatik.uni-freiburg.de/~dbis/Publications/.

## 1.1 Environment Variables

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

- LOPIXCFG
- LOPIXHIS

LOPIXCFG tells how to find the configuration file and LOPIXHIS 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. If one of these variables is not set and the respective command line option is missing, the system will print a warning.

UNIX Environment Variables

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

- LOPIXCFG : /home/db/lopix/environment/lopix.cfg
- LOPIXHIS : /home/db/lopix/environment/lopix.his

Additionally, the environment variables

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

have to be set for the *sp* SGML parser [Cla] which is employed in the current version for parsing XML files into LoPiX.

If you want to check if your settings are working, proceed with Section 1.3 now for a first example.

## 1.2 Settings for LoPiX in Emacs

Additionally to the shell, emacs provides a very user-friendly interface to LoPiX via the emacs flp-mode. The emacs lpx-mode is defined in the file `lpx.el` which is located in `lopix/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 lpx mode if a file with the suffix ".lpx" is loaded
(setq auto-mode-alist (cons '("\.lpx$" . lpx-mode) auto-mode-alist))

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

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

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

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

In `lpx.el` it has to be specified where to find the `lopix` executable. If the `lopix/bin/` directory is in the binary search path, .emacs: lpx mode

```
(defvar lpx-program-name "lopix"
      "*Program name for invoking an inferior LoPiX with `run-lpx'.")
```

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

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

## 1.3 LoPiX in the UNIX Shell

LoPiX is started from the shell by simply typing `lopix` at the unix prompt. To give an impression of the LoPiX system and its usage we present a short example session in the unix shell:

**Example 1 (Example Session)**
*The examples directory contains files smallmondial.{xml,dtd,lpx}; a small XML file, a suitable DTD, and a* LoPiX*program. If you call*

```
malta:~/db/lopix/examples> lopix smallmondial.flp
```

*the program is executed. It should tell you that it parses smallmondial.xml (1783 bytes), and then print some country names. You can leave* LoPiX *with*

```
?- sys.end.
```

*If the program does not work, check the four environment variables given in Section 1.1.*

**Command Line Options.** When LoPiX is started in the unix shell, command line options can be given. A list of possible options is printed with `lopix -h`. If `-h` or an undefined option is given, LoPiX terminates after printing an appropriate message.

```
malta:~/lopix/bin> lopix -h
This is LoPiX

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`. Similarly, the option `-his` loads the given file into the readline-history instead of the default in `DEFAULTHIS`. For operating LoPiX with non-standard configuration files regularly, it is more convenient to change the environment variables (cf. Section 1.1). The option `-q` causes LoPiX to quit after execution of the command line instead of entering the interactive mode. This is helpful if the user wants to call LoPiX in batch mode, e.g., from a shell script: `lopix -q test.lpx`.

In addition to the options, a list of filenames of F-Logic-programs can be given. LoPiX consults them in a left-to-right order.

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.

## 1.4 Running LoPiX in Emacs

Running the system from the emacs not only offers high-level editing facilities but also integrates LoPiX into an environment where all kinds of tools (mailreader, newsreader, several compilers, TeX etc.) are used in a uniform way. In order to use LoPiX from emacs, the configuration steps in Section 1.2 must have been executed. When a file ending with `.lpx` is loaded into emacs, emacs automatically enters the XPathLog mode which provides syntax highlighting facility to make XPathLog programs.

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

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 LoPiX is started,

| | |
|---|---|
| `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 database and program) |
| `C-c C-b` | Consult whole buffer as region (without reset) |
| `C-c C-l` | Display buffer *lpx* and jump there |
| `C-` | Break evaluation or output |
| `C-c C-i` | Quit LoPiX process |

Table 1: Special keycodes in the lpx-mode

running the program given in the program buffer. Load examples/smallmondial.lpx into the emacs and type `Control-c Control-c`.

Then, the user can interact in the same way with LoPiX 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 LoPiX.

# 2 Programming with LoPiX

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 LoPiX.

After entering starting LoPiX with a program (or typing `C-c C-c` in an emacs program buffer holding the program), the following happens: LoPiX reads `foo.lpx` rulewise using the XPathLog parser.

- Facts und rules are added to the current program held by the the interpreter. Note that no evaluation takes place until the system query "`?- sys.eval.`" or "`?- sys.strat.doIt.`" is encountered.
- Queries are answered immediately:
  - System queries are executed by sending messages to interface objects. Syntactically, system commands are 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 database.
  - Other queries are passed to the evaluation component which answers them according to the current state of the database.
- In case of a syntax error, the process is terminated and the error is reported.

When the end of the program is reached, control is given back to the calling level. The database then contains a model of the program evaluated so far.

## 2.1 System Commands

The following built-in system commands are provided:

| | |
|---|---|
| `?- sys.consult@("foo.lpx").` | read the program file *foo.lpx* |
| `?- sys.load@("foo.lpx").` | load a program file containing only facts directly into the databases (e.g., for loading predicates from ASCII representations) |
| `?- sys.eval.` | evaluate the current program |
| `?- sys.strat.doIt.` | evaluate a stratum (see Sec. 2.1.1) |
| `?- 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 LoPiX |

A number of frequently used system commands are loaded to the readline history.

#### 2.1.1 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 XPathLog 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 database 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 a database. Then, *forgetProgram* clears the current program, the contents of the database remains unchanged and the second set of rules is evaluated wrt. this database.

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

```
?- 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.
")
```

Stratification is an important matter when dealing with negation, when facts have to be computed completely before their negation is used. The system command "?- sys.strat.doIt." may also be used in negation-free XPathLog programs to speed up the evaluation.

**Interrupting an Evaluation.** A running evaluation can be canceled without terminating the whole system by pressing `Control-\`. Then, a `QUIT` signal is sent to the system, causing a break flag to be set. In order to return a somewhat consistent object world, the current $T_P$-round has to be finished, such 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.

# 3 Document Access and Queries in LoPiX Programs

The internal data model of LoPiX is an (in the current version, restricted) XML data model enhanced by some features known from the relational and the object-oriented data models: predicates, metadata (signatures and data for controlling document access), and a class hierarchy.

## 3.1 The XPathLog Language

XPathLog is based on XPath. Constants are interpreted as names and elements of the universe. For convention, constants start with lowercase letters whereas variables start with uppercase ones.

- an XPathLog *reference expression* is an XPath expression which satiesfies the following conditions:
  - every element name or attribute name in a navigation step may be replaced by $\mathit{variable}_1$, *name*->$\mathit{variable}_2$, or even $\mathit{variable}_1$->$\mathit{variable}_2$. Here, $\mathit{variable}_1$ is bound to the name, extending the "*" navigation wildcard, and $\mathit{variable}_2$ is bound to the node(s).
  - for reference attributes @*name*, additional navigation steps may follow (implicit dereferencing).

  - the id function is not used (since XPathLog supports implicit dereferencing).
  - filters do not contain logical or (to be replaced by two or more rules)
- arithmetic expressions, built-in predicates, and aggregations are described below.
- In the sequel, features which do not belong to the basic XML data model are described. Let $A$, $C$, $D$, $M$, and $X_i$ stand for XPathLog reference expressions.
- An *is-a assertion* is an expression of the form $O$ isa $C$ (object $O$ is a member of class $C$), or $C$ subcl $D$ (class $C$ is a subclass of class $D$).
  When parsing an XML document, for every element $e$ of type $t$, e isat holds.
- A *predicate* is an expression of the form $p(X_1, \ldots, X_n)$. The special equality predicate is written as $X_1 = X_2$.
- A *signature atom* is an expression of the form $C[M{=>}D]$ or $C[@A{=>}D]$; denoting that elements of type/class $C$ have subchildren tagged with $M$ of class $D$ (if the class relationships induced by XML documents is not changed, $M = D$), or that elements of type/class $C$ have an attribute named $A$ which results in a instance of class $D$ (if the class relationships induced by XML documents is not changed, $D \in \{$literal, object$\}$ without further distinction).
  *Signature atoms* describe which properties apply to instances of certain classes. Signature atoms together with the class hierarchy form the *schema* of an XPathLog database.
- A *method application* is an expression of the form $X.m@(X_1, \ldots, X_n)$. Method applications are also references whose value is defined via equality. A special kind of method applications are *active methods* (e.g., for accessing an url or for exporting a document) which are described below.
- in the rule head, further restrictions apply to yield a well-defined semantics, they are also described below.

### 3.2 Accessing Documents

Every resource available in the Web has a unique address, called *Uniform Resource Locator* (URL), which is used to initiate access to the document. There is a predefined class url containing Web address strings and a predefined built-in active method parse that, when applied to a member of url, accesses the corresponding Web document and integrates it into the database. The schema for Web access is:

  (url subcl string)[parse => xmldoc].

Whenever for an instance u of class url the method u.parse(...) is called by a rule head, the document at the address u is automatically loaded and analyzed according to the arguments:

- $u$.parse@("xml") parses the contents of $u$ as an xml document and assigns its root to the *reference* $u$.parse@("xml"). $u$.parse@("xml") has a child node which represents the outermost document node. In course, this node has children and attributes and so on, similar to the DOM representation (see below). This structure is then queried by XPathLog reference expressions.
- $u$.parse@("dtd") parses $u$ as a DTD and generates a suitable signature which applies to every document which uses this DTD. The signature is queried by signature expressions, or it can be dumped with the system command

  ?- sys.theOMAccess.export@("sig").

  The example `parse-dtd.lpx` illustrates DTD parsing and its result.

By evaluating rules of the form

$u$ isa url, $u$.get :- <body> ,

the internal database is extended by new XML documents. Thus, loading Web documents is completely data-driven. New documents are fetched depending on information and links (i.e., URLs or XLinks) found in already known documents.

As long as only a single document is considered, it is recommended to assign its (local) root to the global constant root which is used for evaluating expressions of the form //...; as it is done in smallmondial.lpx:

```
mondial[@xml->"file:smallmondial.xml" isa url].
U.parse@(xml) = root :- mondial[@xml->U].
```

If several documents have to be considered, it is useful to assign constants to their roots, e.g.

```
germany[@xml->"file:germany.xml" isa url].
france[@xml->"file:france.xml" isa url].
U.parse@(xml) = X :- X[@xml->U].
```

Then, the constants can be used as starting points for XPathLog reference expressions, e.g.

```
?- germany//city/name->C.
```

### 3.3 Querying the Database

In this section, we use the MONDIAL database [Mon] as an example (XML-files: `mondial-2.0.xml` and `mondial-europe-2.0.xml`; use `lopix/examples/mondial-example.flx`). The file is accessed by

```
mondial[@xml->"file:mondial-europe-2.0.xml" isa url].
U.parse@(xml) = root :- mondial[@xml->U].
```

and can then be queried by XPathLog reference expressions.

**Example 2 (XPathLog queries)**

**Pure XPath expressions:** *pure XPath expressions (i.e., without variables) are interpreted as* existential queries *which return* true *if the result set is non-empty:*

```
?- //country[name!text() = "Germany"]//city/name!text().
true
```

*since the country element which has a name element with the text contents "Germany" contains at least one city descendant with a name element with non-empty text contents.*

**Output Result Set:** *The query "?- xpath->N" for any xpath binds N to all nodes belonging to the result set of xpath:*

```
?- //country[name!text() = "Germany"]//city/name!text()->N.
N/"Stuttgart"
N/"Mannheim"
⋮
```

*If the result set does not contain literals, but elements, their internal ids are returned:*

```
?- //country[name!text() = "Germany"]//city/name!text()->N.
C/f0_1880
C/f0_1888
C/f0_1895
⋮
```

**Additional Variables:** *XPathLog allows to bind all nodes which are considered by an expression (both by the access path to the result set, and in the filters):*
*The following expression returns all tuples $(N1, C, N2)$ such that the city with name $N2$ belongs to the country with name $N1$ and car code $C$:*

```
?- //country[name!text()->N1 and @car_code->C]//city/name!text()->N2.
N2/"Stuttgart"  C/"D"  N1/"Germany"
N2/"Mannheim"  C/"D"  N1/"Germany"
N2/"Karlsruhe"  C/"D"  N1/"Germany"
⋮
```

**Local Variables:** *Queries can also contain local variables which are used for joins or conditions; their bindings do not occur in the output; local variables are of the form _X:*
*The following XPath expression returns all names of cities s.t. the city belongs to a country whose name is known and its population is higher than 100000, and its latitude is not known:*

```
?- //country[name!text()->N1]//city[@population->_P and not latitude]/name!text()->N2,
     _P > 100000.
```

*The semantics of this query is a set of variable bindings for $N1$ and $N2$.*
*An equivalent expression without local variable is*

```
?- //country[name!text()->N1]
              //city[@population > 100000 and not latitude]/name!text()->N2.
```

**Dereferencing:** *Reference attributes can be resolved in path expressions for navigating through the XML database:*
*For every organization, give the name of the seat city and all names and types of members:*

```
?- //organization[name!text()->N and @abbrev->A and @seat/name!text()->SN]
                  /members[@type->MT]/@country/name!text()->MN.
```

*One element of the result set is e.g.,*

```
N/"..."    A/"UN"    SN/"New York"    MT/"observer"    MN/"Switzerland"
```

*The following query is equivalent, using a join of literals connected by the local variable _Org instead of navigation:*

```
?- //organization->_Org[name!text()->N and @abbrev->A and @seat/name!text()->SN],
     _Org/members[@type->MT]/@country/name!text()->MN.
```

*The following query yields the abbreviations of all organizations which are seated in the capital of one of their members:*

```
?- //organization[@abbrev->A and @seat = members/@country/@capital].
```

*We can additionally add variables to bind the names of the organization, of the country, and of the seat city:*

```
?- //organization[name!text()->N and
                  @seat = members/@country[name!text()->CN]/@capital]
                                              /@seat/name!text()->CN.
```

**Negation in Filters:** *The following query yields all organizations which are not seated in a city in some member country*

```
?- //organization[name!text()->N and not @seat = members/@country//city].
```

*The answer includes those where no seat is known, since the filter predicate says "not those where (i) the seat attribute is defined, and (ii) references a city in a country which is a member".*

**Navigation Variables:** *Variables can also range over element or attribute names:*
*Are there elements which have a name subelement with the pcdata contents "Monaco", and of which type are they?*

?- //Type->X[name!text()="Monaco"]

| | |
|---|---|
| Type/country | X/*country-monaco* |
| Type/city | X/*city-monaco* |

*Note that we use the "=" predicate for comparison with a constant in the filter.*

**Axes:** LoPiX *support the axes* *self, child, parent, descendant, ancestor, attribute, and sibling. Since the current version does not actively support the ordering of children (although the evaluation strategy preserves that ordering in most cases) we do not distinguish preceding-sibling and following-sibling.*
*The file mondial-example.lpx contains many more instructive queries.*

# 4 XPathLog Rules

XPathLog rules are logical rules of the form

*head* :- *body*.

over XPathLog expressions (where we additionally allow conjunctions in the rule head). To cut long theory short, evaluation in LoPiX means bottom-up evaluation with user-defined stratification; i.e., an intuitive, easy-to-grasp rule based semantics. Given a program, i.e., a set of rules, the rule bodies (right-hand side) are evaluated like queries, yielding a *result set*. The variables in the rule heads are instantiated according to the result sets, specifying the facts to be added to the database. In a step, all rules in a program are evaluated simultaneously. As long as they yield new facts, the process is iterated ($T_P$ operator of Logic Programming).

Programs can be cut into subprograms by *strata* (as introduced in Section 2.1.1). Then, the strata are evaluated sequentially, each of tem starting with the database which results from the evaluation of the previous one (see program `reachable.flx` for an example). Section 6 gives a more detailed overview of the semantics.

**Right Hand Side.** The body of an XPathLog rule is a conjunction of XPathLog expressions. The evaluation of the body wrt. a given database yields variable bindings which are propagated to the rule head where facts are added to the database.

## 4.1 Left Hand Side

Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or Quilt where the structure to be *generated* is always specified by XML patterns (this implies that these languages do not allow for updating existing nodes – e.g., adding children or attributes –, but only for generating complete nodes). In contrast, in XPathLog, existing nodes are communicated via variables to the head, where they are *modified* when appearing at host position of atoms.

The head of an XPathLog rule is also a conjunction of XPathLog expressions using only the child and sibling axes, the // operator is allowed only in leftmost position, without negation, disjunction, indexing, and built-in functions (because all these things would cause ambiguities what update is intended). When used in the head, the / and / operators and the

[...] construct specify which properties should be added or updated (thus, [...] does not act as a filter, but as a *constructor*).

**Modification of Elements.** When using the child or attribute axis for updates, the host of the expression gives the element to be updated or extended; when a sibling axis is used, effectively the parent of the host is extended with a new subelement.

**Generation or Extension of Attributes.** An expression of the form $X$[@$a$->$V$] specifies that the attribute @a of the node $X$ should be set or extended with $V$. If $V$ is not a literal value but a node, a reference to $V$ is stored, and in case of generating output, @a is represented as an `IDREF` attribute.

**Example 3 (Attributes)** *We add the data code to Switzerland, and make it a member of the European Union:*

```
C[@datacode->"ch"], C[@memberships->O] :-
    //country->C[car_code="CH"], //organization->O[abbrev!text()="EU"].
```

*results in*

```
<country datacode="ch" car_code="CH"
        memberships="org-efta org-un org-eu ..." >    ...
</country>
```

We discuss insertion of new subelements below, after showing how to create elements.

**Creation of Elements.** Elements can either be created as *free* elements by atoms of the form //*name*[...] (meaning "some element of type *name*" – in the rule head, this is interpreted to create an element which is not a subelement of any other element) or as subelements.

**Example 4 (Creation of a free element)** *We create a new (free) country element (remember that in Section 3.2, we equated* root *to the root node of the parsed Web page):*

```
//country[name->"Bavaria" and @car_code->"BAV" and
        @capital->X and city->X and city->Y] :-
    //city->X[name!text()->"Munich"],   //city->Y[name!text()->"Nurnberg"].
```

*Note that the two city elements are* linked *as subelements. This operation has no equivalent in the "classical" XML model: these elements are now children of* two *country elements.* Copying *elements is described below. Thus, changing the elements effects both trees. By* linking *elements, it is possible to introduce cycles in the document hierarchy.*

*For the subsequent examples, we associate aliases with the "Bavaria" and "Germany" elements:*

```
C = bavaria :- //country->C[name!text()->"Bavaria"].
C = germany :- //country->C[name!text()->"Germany"].
```

**Insertion of Subelements and Attributes** As shown above, elements are extended with subelements or attributes by using filter syntax in the rule head:

**Example 5 (Subelements)** *The following two rules are equivalent to the above ones:*

//country[name->“Bavaria”].
C[@car_code->“BAV” and @capital->X and city->X and city->Y] :-
//city->X[name!text()->“Munich”], //city->Y[name!text()->“Nurnberg”],
//country->C[name!text() = “Bavaria”].

*Here, the first rule creates a free element, whereas the second rule uses the variable binding to C for inserting subelements and attributes.*

**Generation of Elements by Path Expressions.** Additionally, subelements can be created by *path expressions* in the rule head which create nested elements which satisfy the given path expression.

**Example 6 (Generation of Subelements)** *The ethnicgroups subelements of Germany are* copied *as subelements of Bavaria:*

bavaria/ethnicgroups[!text()->E and @percentage->P] :-
germany/ethnicgroups[!text()->E and @percentage->P].

*Here, new elements are created which can be changed independently from the original ones.*

Path expressions can be arbitrarily nested:

**Example 7 (Generation of Nested Elements)**
*The path expression n/a/b/c[name->>m] generates the XML tree fragment*

<a> <b> <c>
<name *attributes of* $m$> *contents of* $m$ </name>
</c> </b> </a>

In another example, we show how data can be restructured:

**Example 8 (Creation of free elements)** *The use of variables allows to create elements whose name is given by an attribute of another element (casting strings silently into names, which is very useful for data restructuring and integration): From the elements*

<water type=“river” name=“Mississippi”> ... </water>
<water type=“sea” name=“North Sea”> ... </water> ,

*the rule* //T[name!text()->N] :- //water[@type->T and name!text()->N]
*creates* <river name=“Mississippi”/> and <sea name=“North Sea”/> .

*Attributes and contents are then transformed by separate rules which use* name!text() *for identification. Properties are copied by using variables at element name and attribute name position:*

X[@A->V] :- //water[@type->T and name!text()->N and @A->V], //T->X[name!text()->N].
X[S->V] :- //water[@type->T and name!text()->N and S->V], //T->X[name!text()->N].

# 5 Built-in Features

The LoPiX implementation of XPathLog provides some built-in features, namely a comfortable handling of `PCDATA` contents, the *equality predicate*, the built-in class *integer*, several *comparison predicates*, the basic *arithmetic operators*, predicates for *string handling*, and *aggregate functions*.

### 5.1 Handling of Literal Values

Literal values can either be represented as attributes or as PCDATA contents. Similar to XPath queries, XPathLog supports automatical casting from PCDATA-only elements into literals in queries and in comparison predicates. Casting can be turned on and off using the system command

```
?- sys.annotatedLiterals@("on").
?- sys.annotatedLiterals@("off").
```

When annotatedLiterals is on, variables which are bound to elements which PCDATA contents, are *output* as literal values (this does not effect the *variable binding* which is used when propagating values to the rule head – there, the variable is still bound to the node). Also, in arithmetics and comparisons, the literal value is used.

**Example 9 (Handling of Literals)** *Consider the following XML fragment (smallmondial.xml):*

```
<country car_code"D" >
    <population year="1997">83536115</population>
</country>
```

*Here, population is an* annotated literal, *i.e., it can act as a literal or as an element:*

```
?- //country[@car_code->C and population->P[year->Y] > 1000000].
    N/"CH"    Y/1997    P/83536115
```

*Note that the query*

```
?- 83536115[year->Y].
false
```

*does* not *yield any results.*
*In a rule, e.g.,*

```
//bigcountry[@car_code->C and population->P] :-
    //country[@car_code->C and population->P[year->Y] > 1000000].
```

*P is bound to the node, and the new free element*

```
<bigcountry car_code"D" >
    <population year="1997">83536115</population>
</bigcountry>
```

*is created (where the population subelement is not copied, but linked from the old one).*

**Example 10** *The equality predicate "=" does* not *support annotated literals; instead,* equiv *has to be used:*

```
?- //country[@car_code->C and population = 83536115].
false
?- //country[@car_code->C and equiv(population,83536115)].
C/"D"
```

### 5.2 Equality

Objects in XPathLog are uniquely determined by their *object identifiers* which are only used internally and are invisible to the user, who has to access objects by their *object names.* Every object name references exactly one object. However, there may be several object names denoting the same object; often, such equalities are derived during program execution.

**Example 11** *Consider two data sources,* gs *and* terra*, which provide geographical information.* cia *uses english names, whereas and* terra *uses german names. Then, city objects can be equated,* fusing *the contents:*

X = Y :- gs//city->X[equiv(name,"Munich")], terra//city->Y[equiv(name,"Muenchen")].

*or*

X = munich :- gs//city->X[equiv(name,"Munich")].
X = muenchen :- terra//city->X[equiv(name,"Muenchen")].
muenchen = munich.

*Then, in the scond case, the same element is addressed by the expressions*

?- munich.
?- muenchen.
?- gs//city->X[equiv(name,"Munich")].
?- terra//city->X[equiv(name,"Muenchen")].

*and has all subelements which the individual elements had before, and also the attributes are the union of the original attributes:*

?-munich/name!text()->N.
N/"Muenchen"
N/"Munich"

## 5.3 Integers, Comparisons and Arithmetics

Objects denoting *integer* or *float* numbers, or *strings* are different from other objects because the usual comparison operators are defined for them, as well as several datatype-specific functions (arithmetics, string handling). There is no built-in *class* integer because it would contain an infinite number of instances. Instead there is a built-in *predicate* integer(<argument>) in LoPiX; analogously for float and string.

Within a query or a rule body, relations between integer numbers may be tested with the *comparison predicates*[1] "<",">", "<=" or ">=".

To guarantee safety variables that appear in a comparison P-atom have to be bound by another atom or molecule in the rule body. Comparison predicates are not allowed in rule heads.

The basic arithmetic operations addition "+", subtraction "-", multiplication "*" and integer division "/" are also implemented in LoPiX. Arithmetic expressions may be constructed in the usual way, even complex expressions, e.g., "3 + 5 + 2" or "3 + 2 * 3" are possible. Note that the blanks between an arithmetic operator and its operands are mandatory, 2+2 leads to a parser error message.

Arithmetic expressions are only allowed in (in-)equality-predicate atoms in a rule body or in filters, e.g., "X = Y + 2", "3 * X = Y + 4".

Objects denoting integers must not be equated to other integer or string objects because their object identity is not independent from their object name. Unfortunately, avoiding this by a static check is impossible since integer objects may be bound to variables in a rule head. If an equation of two different integer objects is derived during the evaluation of an XPathLog program, an error is reported.

[1]Note that comparison predicates are used in infix notation in contrast to other predicate symbols.

## 5.4 String handling

Analogously to integers, there are several predefined operations for strings. These are provided by the built-in predicates which all have a fixed arity. Using them with a wrong arity causes a parser error message. Furthermore these predicates can only be used in rule bodies:

**string(<arg>)** is true, if <arg> is a string.

**strlen(<string>, <value>)** holds if <value> (which can be given a constant or a variable) represents the length of <string>. For practical reasons, variables at position of <string> have to be bound by other molecules in the rule body (otherwise <strlen> fails), because a query like "*show all strings with a certain arity*" e.g., "?- strlen(X,40)." may lead to an answer set that is not infinite but too huge to be handled. Normally strlen is used in the following way to return the length of a given string:
"?- strlen("logic",X)."

**strcat(<string$_1$>, <string$_2$>, <string$_3$>)** succeeds if <string$_3$> is the concatenation of <string$_1$> and <string$_2$>. E.g., "?- strcat("a","b",X)." returns the binding X/"ab" whereas "?- strcat("a",Y,"ab")." leads to Y/"b". If the arguments contain more than one variable, e.g., "?- strcat("a",Y,X).", either Y or X have to be bound by other molecules in rule body, otherwise strcat fails. The reason for this limitation is the same as in the case of strlen.

**substr(<string$_1$>, <string$_2$>)** holds if <string$_1$> is a substring of <string$_2$>. Comparison between the two strings is case insensitive, for example "?- substr("DaTA","database")." returns true. If the arguments contain any variable, it has to be bound by other molecules in the rule body, otherwise substr fails. For "?- substr("logic",X)." the corresponding answer set would be infinite.

**match(<string>, <pattern>, <fmt-list>, <variable-list>),**

**pmatch(<string>, <pattern>, <fmt-list>, <variable-list>)** finds all strings contained in <string> (which may be a string or a Web document) which match the pattern given by a regular expression in <pattern>. <fmt-list> is a format string describing how the matched strings should be returned in <variable-list>. This feature is useful when using groups (expressions enclosed in `\(...\)`) in <pattern>. In the format string <fmt-list>, groups are referred to by their number: `$n`,where `n` ranges from 1 to 9. If <fmt-list> is the empty string ("") all groups are returned without formatting. With the exception of <variable-list>, all arguments must be bound. pmatch does the same, using Perl regular expressions; we recommend using pmatch.
For example,

```
?- match("linux98","\([0-9]\)\([0-9]\)","$2swap$1",X).
```

returns X/"8swap9" (the first and second match are swapped).

## 5.5 Data Conversion

LoPiX internally distinguishes objects (e.g., john from literals (e.g., 42, 3.14, or "John"). The literal types are further distinguished: strings are always given in quotes ("john"), integer objects are given as-is, and floats have to be distinguished by #, from literals (#3.14).

**Example 12** *Note that 3.14 and #3.14* actually *are different things: In the first example, #3.14 is interpreted as a float:*

```
b[m->#3.14].
```

```
?- sys.strat.doIt.
?- b[m->C].
?- b[m->C], D = C + 1.
% Answer to query : ?- b[m -> C].
C/#3.14
% Answer to query : ?- b[m -> C], D = C + 1.
C/#3.14   D/#4.14
```

*In the second example, 3.14 is* not a float*:*

```
a[m->#3.14].
?- sys.strat.doIt.
?- a[m->C].
?- a[m->C], D = C + 1.
% Answer to query : ?- a[m -> C].
C/3.14
% Answer to query : ?- a[m -> C], D = C + 1.
false
```

*But, what happens there? Obviously, the program is syntactically correct. So, lets ask what the database looks like then:*

```
?- sys.theOM.dump.
3 [14 -> 3.14].
a [m -> 3.14].
```

*... which is correct: 3.14 is interpreted as the result (an anonymous object) of applying the method 14 to the object 3.*

Often, a conversion between datatypes is needed. In LoPiX, data conversion is provided by built-in predicates:

**string2integer(A,B)** is true, if B is the integer obtained when reading the string $A$ from left to right as far as possible. e.g., the following holds:
string2integer("42",42).
string2integer("3D",3).
string2integer("3.14",3).

**string2float(A,B)** is true, if B is the float obtained when reading the string $A$ from left to right as far as possible (a leading # is ignored, so both "normal" floats and the XPathLog representation of floats can be read). E.g., the following holds:
string2float("42",42).
string2float("3D",3).
string2float("3.14",#3.14).
string2float("#3.14",#3.14).
string2float("3.14D",#3.14).
Note that integers are seamlessly integrated with floats: LoPiX never prints 3 as #3, but always understands #3 as 3.

**string2object(A,B)** is true, if B is the object whose id is $A$ (converted to lowercase letters).

```
?- string2object("john",O).
% Answer to query : ?- string2object("john",O).
O/john

?- string2object("John",O).
% Answer to query : ?- string2object("John",O).
O/john

?- string2object(S,john).
% Answer to query : ?- string2object(S,john).
S/"john"
```

If the given string denotes an integer or a float, B is the corresponding integer or float object.

Note that the above predicates are "bidirectional", but at least one of the arguments has to be bound:

```
?- string2float("3.14",B).
% Answer to query : ?- string2float("3.14",B).
B/#3.14

?- string2float(X,#3.14).
% Answer to query : ?- string2float(X,#3.14).
X/"3.14"
```

## 5.6 Aggregation

An aggregation term has the form

agg{X[$G_1$,...,$G_n$]; body}

where *agg* is one of the usual aggregation operators min, max, count, and sum and $b$ is the aggregation body (that is, a conjunction of literals).

agg{X[$G_1$,...,$G_n$]; body}

returns one value for every vector of values for [$G_1$,...,$G_n$]: All variable bindings satisfying body are calculated (intentionally yielding bindings for X,$G_1$,...,$G_n$). Then, the X's are grouped by [$G_1$,...,$G_n$] and for every group, agg is calculated and returned.
The list of grouping variables [$G_1$,...,$G_n$] is optional and may be omitted, e.g.,

?- N = count{C; //country->C}.

If the aggregation body contains other variables than the grouping variables, these are local to the aggregation. The grouping variables may occur anywhere in the rule body (respectively the higher level aggregation body).

Like arithmetic expressions, aggregation terms may only occur in the built-in predicates "=", "<", ">", "<=", ">=". However, the aggregation body may contain built-in predicates with other aggregation terms. Thus, aggregation can be nested.

The semantics of an aggregation term in a comparison predicate, e.g., the inequality $V < agg\{X[G_1, \ldots, G_n]; body\}$ is defined as the conjunction $V < Z, Z = agg\{X[G_1, \ldots, G_n]; body\}$.

**Syntactic Restrictions** For obvious reasons the following syntactic restrictions apply:

- The aggregation variable and all grouping variables must occur in the aggregation body.
- The variables X,$G_1$,...,$G_n$ are pairwise distinct.
- The aggregation body has to obey the safety restrictions, i.e., no variable may represent an infinite answer set.

Violating these restriction will cause an error.

The operator count gives the total number of variable bindings to the aggregation variable. Note that, different from count, the operators min, max and sum ignore objects other than integer without producing any error message or warning.

```
myset[item ->{10,40,apple,27,cheese}].
?- Z = count{X; myset[items->X]}.
?- Z = sum{X; myset[items->X]}.
```

will yield 5 and 77.

**Aggregates and Stratification** As in the case of negation, the facts to be used in the aggregation body has to be completely established before the actual aggregation.

### 5.7 Output Handling

For every node $N$, the active method

$N$.export@(*doctype*,"*filename*"). % (in the head of a rule, or as a fact)

generates XML output of the subtree rooted in $N$, consisting of all subelements and attributes which are specified by the currently stored signature, to the given *filename*. The DTD metadata (public/system, and the url) has to be set by methods, e.g.,

```
country isa doctype.
country.public = "mondial-europe-2.0.dtd".
?- sys.strat.doIt.
germany.export@(country,"exp1").
?- sys.strat.doIt.
```

If *filename* is the empty string, the output is done at standard output. Similarly,

?- sys.theOMAccess.export@("xml", "*filename*", "*doctype*", "*object*").

as a system command allows to export a given element[2]. The example `parse-dtd.lpx` illustrates the export functionality. The signature can either be given by parsing a DTD, or as facts in the program.

## 6 Evaluation Semantics

An XPathLog program is a collection of facts and rules in arbitrary order. Evaluating these facts and rules bottom-up, an object base is computed which may then be queried. Note, however, that **queries are not part of a program**. They are evaluated (respectively executed in case of system commands) when parsing the program (respectively thze stratum) – *before* evaluating the program/stratum.

[2]note that the system command version can be given interactively; whereas only the fact version accepts variables

**Program evaluation.** As already mentioned, LoPiX 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 (as defined for XPathLog in []). 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), $T_P$ is applied iteratively until a fixpoint $T_P^\infty(\emptyset)$ is reached.

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).

Note that the database 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 2.1.1).

**Semi-naive Evaluation:** LoPiX 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 (this may be changed in the configuration file `lopix.cfg`). 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 frequently change the class hierarchy or equate objects, because this makes dependancy analysis very hard. To see the rewritten program, set the debug mode `"program"`.

## 6.1 Fixpoint Semantics

The evaluation strategy for XPathLog programs (without inheritance) is basically the same as for Datalog programs. The bottom-up evaluation of an XPathLog program starts with a given object base. Initially, this is the empty object base. Facts are rules with an empty body, therefore always considered as true. The rules and facts of a program are evaluated iteratively in the usual way. If there are variable bindings such that the rule body is valid in the actual object base, these bindings are propagated into the rule head. New information corresponding to the ground instantiations of the rule head or deduced due to the closure properties is inserted into the object base[3]. This evaluation of rules is continued as long as new information is obtained. As in the case of Datalog, the evaluation of a negation-free XPathLog program reaches a fixpoint which coincides with the unique minimal model of that program[4]. The minimal object base of an XPathLog program is defined as the smallest set of atoms such that all closure properties and all facts and rules of the program are satisfied.

[3]To avoid redundancy, in LoPiX most of the information generated by the closure properties is not inserted into the object manager explicitely, but deduced when retrieving information.

[4]Note that this fixpoint is not necessarily finite.

### 6.2 Negation and Stratification

Negation in LoPiX is handled according to the *inflationary semantics* [KP88]. Remember that in a safe rule, every variable in a negated subgoal has to be limited by other subgoals. Thus, only ground instantiated negated subgoals have to be considered during the evaluation. Such an instantiation of a negated subgoal is evaluated as true if and only if the intermediate object base given in the moment of the evaluation of the rule does not contain the corresponding information. Inflationary semantics often yields unintended results. Hence, there are other concepts to handle negation in logic programs. One of the most general solutions is the three-valued *Well-Founded Semantics* [VGRS91]. A very common approach is to *stratify* logic programs and to compute the perfect model [ABW88, Prz88]. Unfortunately, due to the powerful syntax of XPathLog, the class of stratified programs is very small. Thus, automatic stratification cannot be done by LoPiX for a large class of programs. Instead, programs have to be stratified explicitly by the user with the system command "?- sys.strat.doIt." which divides a program into several strata. Information queried by a negated subgoal has always to be derived in lower strata than the stratum of the rule containing the negated subgoal.
The stratification command causes the evaluation of the rules in the higher stratum to be deferred until the fixpoint of the lower stratum is computed. Moreover, the rules of the lower stratum are not considered any more during the further evaluation.

## 7 Legacy: The Florid System

LoPiX is based on components of the Florid system [FHK+97], an implementation of F-Logic [KLW95]. When LoPiX is called with the `-florid` option, e.g.,

```
malta:~/db/florid/examples> lopix -florid test.flp
```

it acts as Florid.

**Acknowledgements.** First of all, we want to thank Georg Lausen, the head of our group. Furthermore, our thanks go to the former team members Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Bertram Ludäscher, Christian Schlepphorst, and Heinz Uphoff who developed Florid up to version 2.0 together with the students Thomas Beier, Thorsten Pferdekämper, Bernhard Seckinger, Markus Seilnacht, and Till Westmann from the universities at Mannheim and Freiburg.

## References

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, 1988.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

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

[Cla] J. Clark. SP: An SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language. `http://www.jclark.com/sp`.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB 2000*, pp. 53–62, 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference.* W3C, 1999. World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, `www.w3.org/TR/NOTE-xml-ql`.

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F–Logic. 1997.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[KP88] P. Kolaitis and C. Papadimitriou. Why not Negation by Fixpoint? pp. 231–239, 1988.

[Mon] The MONDIAL Database. `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann, 1988.

[Rob99] J. Robie. XQL (XML Query Language). `http://www.metalab.unc.edu/xql/xql-proposal.html`, 1999.

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

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620 – 650, July 1991.

[XPa99] XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[XSL99] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, 1999.