# FLORID

# How to Write F-Logic Programs in Florid

## A Tutorial for the Database Language F-Logic

Wolfgang May Pedro José Marrón
{may|pjmarron}@informatik.uni-freiburg.de

Institut für Informatik, Universität Freiburg
Germany

December 1999

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

# Contents

# 1 Introduction

F-Logic [KLW95] is a deductive, object oriented database language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modelling capabilities supported by the object oriented data model.

In version 1.0 (1996) FLORID implemented the basic features of F-Logic. The theoretical foundations of F-Logic have been described in the F-Logic report [KLW95] and [FLU94]. The present tutorial describes how to *apply* F-Logic in the FLORID system.

Therefore, this tutorial explains the various features of F-Logic by example and shows how to use them for typical database problems. Section 2 gives a first taste of how F-Logic programs look like. The same simple model world taken from the Old Testament also serves as a background database throughout the tutorial. The following Sections 3 to 8 focus on data modeling and present the language concepts of F-Logic relevant for FLORID.

In order to write programs in FLORID, the reader also has to be aware of how FLORID implements some more difficult points. Questions regarding safety of rules, treatment of negation, non-monotonic inheritance and type checking are covered in Sections 10 and 12. In Section 14, a number of examples illustrate special features of FLORID and offer more material to start with.

As a major new feature, FLORID 2.0 (1998) provides access to Web documents. Its usage is described in Section 13. FLORID 2.1 and 2.2 (1999) contain several internal modifications that improve on speed and usability.

We assume that the reader of this tutorial is familiar with the basic concepts of deductive databases, e.g., Datalog [AHV95, CGT90, Ull89], and the principles of object oriented database systems [ABD+89]. We refer the reader to the user manual [MM99] for a description of FLORID system commands.
Please send enquiries, comments, suggestions, and bug reports to

`florid@informatik.uni.freiburg.de`

# 2 A First Example

Before explaining the syntax and semantics in detail, we give a first impression of F-Logic. The following F-Logic program models biblical persons and their relationships:

```
% facts
abraham:man.
sarah:woman.
isaac:man[father->abraham; mother->sarah].
ishmael:man[father->abraham; mother->hagar:woman].
jacob:man[father->isaac; mother->rebekah:woman].
esau:man[father->isaac; mother->rebekah].

/* rules
   consisting of a rule head and a rule body */
X[son->>{Y}] :- Y:man[father->X].
X[son->>{Y}] :- Y:man[mother->X].
X[daughter->>{Y}] :- Y:woman[father->X].
X[daughter->>{Y}] :- Y:woman[mother->X].

// query
?- sys.eval.  // This is a system command.
?- X:woman[son->>{Y[father->abraham]}].
```

(2.1)

The first part of this example consists of a set of facts to indicate that some people belong to the classes man and woman, respectively, and give information about the father and mother relationships among them. According to the object-oriented paradigm, relationships between objects are modeled by method applications, e.g., applying the method father on the object isaac yields the result object abraham. All these facts may be considered as the extensional database of the F-Logic program. Hence, they form the framework of an object base which is completed by some closure properties. For more details about an object base see Section 8.

The rules in the second part of Example (2.1) derive new information from the given object base. Evaluating these rules in a bottom-up way, new relationships between the objects, denoted by the methods son and daughter, are added to the object base as intentional information. Note that the methods son and daughter are *multi-valued*, which is indicated by the double-headed arrow "->>" and the curly braces enclosing the result, whereas the methods father and mother are *functional*, indicated by the simple arrow "->". As known from other logical database languages, object names always begin with a lowercase letter, e.g., sarah, mother, man[1] and variable names in general begin with an uppercase letter, e.g., X.

The third part of Example (2.1) contains a query to the object base. The expression "?- sys.eval." is a system command to start the evaluation of the program (see also the user manual [MM99]). The query shows the ability of F-Logic to nest method applications. It asks about all women and their sons, whose father is Abraham. The same question could be written as a conjunction of simple subgoals:

```
?- X:woman, X[son->>{Y}], Y[father->abraham].
```

Also, the above program shows the different comment formats available in F-Logic: As in C and C++, text between "/*" and "*/" is ignored, even if it is ranging over several lines. The symbols "//" (C++-Style) or "%" (Prolog and LaTeX style) mark the rest of a line as a comment.

[1] Methods and classes also are objects, see Sections 3.1.1 and 3.1.2.

# 3 Objects and their Properties

As we have already seen in Example (2.1) *objects* are the basic constructs of F-Logic. Objects model real world entities and are internally represented by *object identifiers* which are independent of their properties. According to the principles of object oriented systems these object identifiers are invisible to the user. To access an object directly the user has to know its *object name*. Every object name refers to exactly one object. However, an object may be denoted by more than one object name as we will see in Section 6.1.

Following the object oriented paradigm, objects may be organized in *classes*. Furthermore, *methods* represent relationships between objects. Such information about objects is expressed by *F-atoms*.

## 3.1 Object Names and Variable Names

Object names and variable names are also called *id-terms* and are the basic syntactical elements of F-Logic. To distinguish object names from variable names, the former always begin with a lowercase letter, whereas the latter always begin with an uppercase letter or an underscore (cf. Prolog). After the first letter, object names and variable names may both contain uppercase letters, lowercase letters, numerals or the underscore symbol "_" . Examples for object names are abraham, man, daughter, for variable names are X, Method, _42. There are two special types of object names that carry additional information: integers and strings. Every positive or negative integer may be used as an object name, e.g., +3, 3, -3, and also every string enclosed by quotation marks "".

Complex id-terms may be created by *function symbols* where other id-terms may be used as arguments, e.g., couple(abraham, sarah), f(X). An id-term that contains no variable is called a *ground id-term*.

### 3.1.1 Methods

In F-Logic, the application of a method to an object is expressed by *data-F-atoms* which consist of a host object, a method and a result object, denoted by id-terms. Any object may appear in any location: host object, result position, or method position. Thus, in our Example 3.2 the method names father and son are object names just like isaac and abraham. Variables may also be used be used at all positions of a data-F-atom, which allows queries about method names like

```
?- isaac[X->Y].
```

As every person has at most one father, the method father is defined as a *functional* (or *scalar*) method, represented by the single headed arrow "->". In contrast, the method son may result in more than one object; this is indicated by the double headed arrow "->>". Such methods are called *multi-valued*. Note that the *set* consisting of the result objects is **not** an object, thus we preserve first-order semantics.

**Empty Sets as Result of Multi-valued Methods.** The result position of a multi-valued data-F-atom may also consist of the empty set, e.g., "isaac[son->>{}].". This does not necessarily mean that isaac has indeed zero sons. Since the object base may contain other information about Isaac having sons, e.g., "isaac[son->>{jacob}].", the former data-F-atom

expresses that Isaac has at least zero sons. This can be viewed as a kind of declaration of the method son for the object isaac.

**Methods with Parameters.** Sometimes the result of the invocation of a method on a host object depends on other objects, too. For example, Jacob's sons are born by different women. To express this, the method son is extended by a *parameter* denoting the corresponding mother of each of Jacob's sons. Like methods, parameters are objects as well, denoted by id-terms. Syntactically a parameter list is always included in parentheses and separated by "@" from the method object.

```
jacob[son@(leah)->>{reuben, simeon, levi, judah, issachar, zebulun};
      son@(rachel)->>{joseph, benjamin};
      son@(zilpah)->>{gad, asher};
      son@(bilhah)->>{dan, naphtali}].
```
(3.1)

The syntax extends straightforwardly to methods with more than one parameter. If we additionally want to specify the order in which the sons of Jacob were born, we need two parameters which are separated by commas:

```
jacob[son@(leah,1)->reuben; son@(leah,2)->simeon;
      son@(leah,3)->levi; son@(leah,4)->judah;
      son@(bilhah,5)->dan; son@(bilhah,6)->naphtali;
      son@(zilpah,7)->gad; son@(zilpah,8)->asher;
      son@(leah,9)->issachar; son@(leah,10)->zebulun;
      son@(rachel,11)->joseph; son@(rachel,12)->benjamin].
```
(3.2)

In Examples (3.1) and (3.2) the method son is used with a different number of parameters. This so-called *overloading* (see also Section 3.4) is supported by F-Logic.

**Queries with Multi-Valued Methods.** If a query contains a multi-valued method with a variable at the result position, each result object of this method application in the object base is a possible binding for this variable. Given the object base described in Example 3.2, questioning the sons of Isaac yields all his known sons:

```
?- isaac[son->>X].

Answer to query : ?- isaac[son->>X].

X/jacob
X/esau
```

Note that variables in a query may only be bound to individual objects, never to sets of objects, i.e., the above query does *not* return X/{jacob,esau}.

In case of a query with a set of *ground* id-terms at the result position, however, it is only checked whether all these results are true in the corresponding object base; there may be additional result objects in the database. With the object base above, all the following queries yield the answer true.

```
?- isaac[son->>{jacob, esau}].
?- isaac[son->>{jacob}].
?- isaac[son->>{esau}].
?- isaac[son->>{}].
```

If we want to know if a set of objects is the *exact* result of a multi-valued method applied to a certain object, we have to use negation, see Example 14.6.

### 3.1.2 Class Membership and Subclass Relationship

*Isa-F-atoms* state that an object belongs to a class, *subclass-F-atoms* express the subclass relationship between two classes. Class membership and the subclass relation are denoted by a single colon and a double colon, respectively. In the following example the first three isa-F-atoms express that Abraham and Isaac are members of the class man, whereas Sarah is a member of the class woman. Furthermore, two subclass-F-atoms state that both classes man and woman are subclasses of the class person:

```
abraham:man.
isaac:man.
sarah:woman.                                    (3.3)
woman::person.
man::person.
```

In isa-F-atoms and subclass-F-atoms, the objects and the classes are also denoted by id-terms because **classes are objects** – as well as methods are objects. Hence, classes may have methods defined on them and may be instances of other classes which serve as a kind of metaclasses. Furthermore, variables are permitted at all positions in an isa- or subclass-F-atom.

In contrast to other object oriented languages where every object is instance of exactly one most specific class (e.g., ROL [Liu96]), F-Logic permits that an object is an instance of several classes that are incomparable by the subclass relationship. Analogously, a class may have several incomparable direct superclasses.

Thus, the subclass relationship specifies a partial order on the set of classes, so that the *class hierarchy* may be considered as a directed acyclic, but reflexive graph with the classes as its nodes.

Note that in analogy to HiLog [CKW93] a class name does **not** denote the set of objects that are instances of that class.

## 3.2 Expressing Information about an Object: F-Molecules

Instead of giving several individual atoms, information about an object can be collected in *F-molecules*. For example, the following F-molecule denotes that Isaac is a man whose father is Abraham and whose sons are Jacob and Esau.

```
isaac:man[father->abraham; son->>{jacob,esau}].
```

This F-molecule may be split into several F-atoms:

```
isaac:man.
isaac[father->abraham].
isaac[son->>{jacob}].
isaac[son->>{esau}].
```

For F-molecules containing a multi-valued method, the set of result objects can be divided into singleton sets (recall that our semantics is *multivalued, not set-valued*). For singleton sets, it is allowed to omit the curly braces enclosing the result set, so that the three expressions

given in (3.4), (3.5) and (3.6) are equivalent, which means that they yield the same object base:

```
isaac[son->>{jacob,esau}].                                    (3.4)

isaac[son->>{jacob}].
isaac[son->>{esau}].                                          (3.5)

isaac[son->>jacob].
isaac[son->>esau].                                            (3.6)
```

## 3.3 Behavioral Inheritance

In object oriented systems, an instance (resp. subclass) may inherit properties of its class (resp. superclass). We distinguish *structural inheritance*, i.e., propagation of a type restriction for a method from a superclass to its subclasses (see Section 3.4 and 8.3), and *behavioral inheritance*, i.e., propagation of results of a method application from a class to its instances and subclasses.

To express behavioral inheritance in F-Logic we use *inheritable methods*, indicated by a special arrow type: "*->" for inheritable functional methods and "*->>" for inheritable multi-valued methods. If an inheritable method is defined for a class, this method application and the corresponding result is propagated to every instance and subclass of that class unless it is overridden.

The following inheritable data-F-atom denotes that in our object base every person, i.e., every object that is an instance of the class person, believes in god.

```
person[believes_in*->god].                                    (3.7)
```

Given this inheritable data-F-atom together with the subclass relationships and class memberships from Example (3.3) we can derive the following information:

```
abraham[believes_in->god].
isaac[believes_in->god].
sarah[believes_in->god].                                      (3.8)
woman[believes_in*->god].
man[believes_in*->god].
```

Example (3.8) shows that inheritable methods remain inheritable when they are passed to subclasses, but they become non-inheritable when passed to instances.

The following example demonstrates overriding of a method application (non-monotonic inheritance). Even though Ahab is a person, he does not believe in god but in Baal. Thus, the method believes_in is explicitly defined for the object ahab:

```
ahab:person[believes_in->baal].
```

This explicit information overrides inheritance from the class person.
In F-Logic, even multiple inheritance is supported. For a more detailed discussion about inheritance see Section 11.

## 3.4 Signatures

*Signature-F-atoms* define which methods are applicable for instances of certain classes. In particular, a signature-F-atom declares a method on a class and gives type restrictions for

parameters and results. These restrictions may be viewed as typing constraints. Signature-F-atoms together with the class hierarchy form the *schema* of an F-Logic database. As in data-F-atoms, the arrow head indicates whether a signature F-atom describes a functional method "=>" or a multi-valued method "=>>". To distinguish signature-F-atoms from data-F-atoms, the arrow body consists of a double line instead of a single line. Here are some examples for signature-F-atoms:

```
person[father=>man].
person[daughter=>woman].
man[son@(woman)=>man].
```

The first one states that the functional method father is defined for members of the class person and the corresponding result object has to belong to the class man. The second one defines the multi-valued method daughter for members of the class person restricting the result objects to the class woman. Finally, the third signature-F-atom allows the application of the multi-valued method son to objects belonging to the class man with parameter objects that are members of the class woman. The result objects of such method applications have to be instances of the class man.

By using a list of result classes enclosed by parentheses, several signature-F-atoms may be combined in an F-molecule. This is equivalent to the *conjunction* of the atoms: the result of the method is required to be in *all* of those classes:

```
person[father=>(man, person)].                    (3.9)

person[father=>(man)].
person[father=>(person)].                         (3.10)
```

Both expressions in the Examples (3.9) and (3.10)[2] are equivalent and express that the result objects of the method father if applied to an instance of the class person have to belong to both classes man and person.

As a special case, empty parentheses are allowed at the result position of a signature-F-atom. This means that the result objects are not restricted to certain classes. The following atom defines believes_in as a functional method which may be applied to instances of the class person without any requirements for the class membership of the result objects.

```
person[believes_in=>()].
```

More information about type checking—i.e., how to ensure that every method application in the object base is covered by a corresponding signature—may be found in Section 12.

**Overloading** F-Logic supports method overloading. This means that methods denoted by the same object name may be applied to instances of different classes. Methods may even be overloaded according to their scalarity (functional or multi-valued), their arity, i.e., number of parameters, or their inheritability. For example, the method son applicable to instances of the class man is used as a multi-valued method with one parameter in Example (3.1) and as a functional method with two parameters in Example (3.2). The corresponding signature-F-atoms look like this:

```
man[son@(woman)=>>(man)].
man[son@(woman,integer)=>(man)].
```

[2]Of course, the result of a signature may be enclosed in parentheses as well, if it consists of just one object.

# 4 Nesting Object Properties

As already shown in Example 3.2, properties of an object may be expressed in a single, complex F-molecule instead of several F-atoms. For that purpose, a class membership or subclass relationship may follow after the host object. Then, a *specification list*, a list of method applications (with or without parameters) separated by semicolons, may be given. If a multi-valued method yields more than one result, those can be collected in curly braces, separated by commas; if a signature contains more than one class, those can be collected in parentheses, also separated by commas:

```
isaac[father->abraham; mother->sarah].
jacob:man[father->isaac; son@(rachel)->>{joseph, benjamin}].        (4.1)
man::person[son@(woman)=>>(man, person)].
```

The following set of F-atoms is equivalent to the F-molecules in (4.1):

```
isaac[father->abraham]
isaac[mother->sarah].

jacob:man.
jacob[father->isaac].
jacob[son@(rachel)->>{joseph}].                                     (4.2)
jacob[son@(rachel)->>{benjamin}].

man::person.
man[son@(woman)=>>(man)].
man[son@(woman)=>>(person)].
```

Besides collecting the properties of the host object, the properties of other objects appearing in an F-molecule, e.g., method objects or result objects, may be inserted, too. Thus, a molecule may not only represent the properties of one single object but can also include nested information about different objects, even recursively:

```
isaac[father->abraham:man[son@(hagar:woman)->>ishmael];
      mother->sarah:woman].
jacob:(man::person).                                                (4.3)
jacob[(father:method)->isaac].
```

The equivalent set of F-atoms is:

```
isaac[father->abraham].
abraham:man.
abraham[son@(hagar)->>ishmael].
hagar:woman.
isaac[mother->sarah].
sarah:woman.

man::person.
jacob:man.

jacob[father->isaac].
father:method.
```

F-Logic molecules are evaluated from left to right. Thus, nested properties have to be included in parentheses if those properties belong to a method object (cf. Section 7), class object or

superclass object. Note the difference between the following two F-molecules. The first one states that Isaac is a man and Isaac believes in god, whereas the second one says that Isaac is a man and that the *object* man believes in god (which is probably not the intended meaning).

```
isaac:man[believes_in->god].
isaac:(man[believes_in->god]).
```

Moreover, omitting parentheses at method or result position can lead to syntactically incorrect molecules, e.g.,

```
isaac[(father::ancestor)->abraham]    is correct, whereas
isaac[father::ancestor->abraham]      results in a parsing error, and

isaac[father->(abraham:man)]          is correct, whereas
isaac[father->abraham:man]            results in a parsing error.
```

### 4.1 F-molecules without any properties

If we want to represent an object without giving any properties, we have to attach an empty specification list to the object name, e.g., "abraham.". If we use an expression like this that consists solely of an object name as a molecule, it is treated as a 0-ary predicate symbol (see next section).

## 5 Predicate Symbols

In F-Logic, predicate symbols are used in the same way as in predicate logic, e.g., in Datalog, thus preserving upward-compatibility from Datalog to F-Logic. A predicate symbol followed by one or more id-terms separated by commas and included in parentheses is called a *P-atom* to distinguish it from F-atoms. Example (5.1) shows some P-atoms. The last P-atom consists solely of a 0-ary predicate symbol. Those are always used without parentheses.

```
married(isaac,rebekah).
male(jacob).
son_of(isaac,rebekah,jacob).
true.
```
(5.1)

Information expressed by P-atoms can usually also be represented by F-atoms, thus obtaining a more natural style of modelling. For example, the information given in the first three P-atoms in (5.1) can also be expressed as follows:

```
isaac[married_to->>rebekah].
jacob:man.
isaac[son@(rebekah)->>jacob].
```
(5.2)

Similar to F-molecules, P-molecules may be built by nesting F-atoms or F-molecules into P-atoms. The P-molecule

```
married(isaac[father->abraham], rebekah:woman).
```

is equivalent to the following set of P-atoms and F-atoms:

```
married(isaac,rebekah).
isaac[father->abraham].
rebekah:woman.
```

Note, that only F-atoms and F-molecules may be nested into P-atoms, but not vice versa.

# 6 Built-in Features

The FLORID implementation of F-Logic provides some built-in features, namely the *equality predicate*, the built-in class *integer*, several *comparison predicates*, the basic *arithmetic operators*, predicates for *string handling*, and *aggregate functions*.

## 6.1 Equality

Objects in F-Logic are uniquely determined by their *object identifiers* which are only used internally and are invisible to the user. Objects must be accessed by their *object names*. Every object name references exactly one object. However, there may be several object names denoting the same object. In such a case the object names are said to be **equal**, indicated by the equality predicate "=", e.g., abram = abraham.[3] The equality predicate induces a congruence relation on the set of object names and may be used in facts or rule heads to equate two object names, as well as in queries or rule bodies to ask about the equivalence of object names. Objects may also be equated *implicitly* by redefining a scalar method (since the functionality constraint forces both result objects to be equal) or by defining a circular subclass relationship. The equality predicate may also be used in connection with variables. If an equality predicate in a rule body has variables on both sides, safety of the rule is a critical issue (see Section 9).

## 6.2 Integers, Comparisons and Arithmetics

Objects denoting *integer* numbers are different from other objects because the usual comparison operators are defined for them, as well as several arithmetic functions. We have not implemented a built-in *class* integer because it would contain an infinite number of instances, making static safety checking impossible[4]. Instead there is a built-in *predicate* integer(<argument>) in FLORID.

Within a query or a rule body, relations between integer numbers may be tested with the *comparison predicates*[5] "<",">", "<=" or ">=". For example, the following query asks for the first three sons of Jacob:

```
?- jacob[son@(X,Y)->Z], Y <= 3.                    (6.1)
```

To guarantee safety (see Section 9), 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 FLORID. 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. By default, multiplication and division are prior to addition and subtraction. As usual, the evaluation order may be changed by using parentheses, e.g., "(3 + 2) * 3".

[3] Note that the equality predicate is used in infix notation in contrast to other predicate symbols.

[4] A subgoal like X:Y with variable Y bound yields an infinite answer set, if Y is bound to the object name integer.

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

Arithmetic expressions are only allowed in (in-)equality-P-atoms in a rule body, e.g., "X = Y + 2", "3 * X = Y + 4". Furthermore, every variable in an arithmetic expression has to be bound by another subgoal in the rule body (see Section 9). The following example contains the query whether Jacob has three sons born consecutively by the same woman.

```
?- jacob[son@(X,A)->Z1; son@(X,B)->Z2; son@(X,C)->Z3],
    B = A + 1, C = A + 2.
```

Objects denoting integers do not have to 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 F-Logic program, an error is reported.

## 6.3 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 large 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. Besides strings, Web documents (see Section 13.1) may also appear as second argument. Thus, substr can also be used to check if a certain string occurs in a Web document.

**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. A full explanation of the syntax and use of regular expressions is given in Appendix B.

For example,

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

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

**Example 1 (Wrapping by regular expressions)** *Assume that the author list of a paper is given on a Web page (see also Section 13.1) as*

*auth*$_1$, *auth*$_2$, ... , and *auth*$_n$: *title*. number $n$ in

Volume $v$ of *series*, pages $p_1 - p_2$, *year*.

*Then, the following predicate assigns the relevant substrings to the corresponding variables:*

```
pmatch(STRING,
       "/\A ([:]*): (.*)\.\s
         Number ([0-9]*) in Volume ([0-9]*) of ([a-Z]*),
         pages ([0-9-]*),([0-9]*)/",
       "[$1,$2,"$4($3)", $5, $6, $7]",
       "[AuthList, Title, Num, Series, Pages, Year]")
```

*Then, the string bound to the variable Num is of the form "volume(number)".*

## 6.4 Aggregation

Since version 2.0, aggregation has been implemented in FLORID. 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.,

```
?- count{C; C:person}.
```

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:

```
?- Z = max{X; john[salary@(Year)->X]}.
?- Z = max{X; john[salary@(Year)->X], Year < 1990}.
?- Z = count{Year; john.salary@(Year) <
                   max{X; john[salary@(Y2)->X], Y2 < Year}}.
```

The first query asks for the highest salary John ever got. The second query contains an additional aggregation goal restricting the aggregation to the years before 1990. The last query yields the number of years in which John earned less than in the best year before.

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[items->>{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 set of objects to aggregate has to be completely established before the actual aggregation. In doubt, the user has to ensure this by explicitely specifying a strata separation (see Section 10.2).

In the following simple example we want to compute the shortest path in a given graph. Of course, cyclic graphs may lead to nontermination problems. See Example 14.3 for a possible solution.

```
%% EDB graph
edge[from=>node; to=>node; dist=>int].
edge::path.
shpath::path. %% shortest path

%% Rule to calculate transitive closure
p(E,P):path[from->X; to->Z; dist->D] :-
    E:edge[from->X; to->Y], P:path[form->Y; to->Z],
    D = E.dist + P.dist.

?- sys.strat.dolt. %% separate the program strata
P:shpath :- P:path[from->X; to->Z; dist->M,
    M = min{D[X,Z]; P:path[from->X; to->Z; dist->D]}.

?- sys.eval.
?- X:path.
?- X:shpath[from->M; to->N; dist->D].
```

# 7 Path Expressions

Objects may be accessed directly by their object names. On the other hand it is also possible to navigate to them by applying a method to another object using *path expressions*. For example, the object described by the object name abraham may also be accessed by calling the method father on the object isaac. The corresponding path expression is "isaac.father".

Example (7.1) shows that path expressions may also contain methods with parameters and that it is possible to chain up path expressions by successively applying methods to the result object of the preceding method call. At the end of each line you find the object name of the result object that is denoted by the path expression. The underlying object base is taken from the Examples (2.1) and (3.2):

```
jacob.son@(rachel,11)            joseph
benjamin.father.father.mother    rebekah                    (7.1)
god.people                       ?
```

Some path expressions may even denote objects in the object world which have no id-term as object name. The last path expression in Example (7.1) defines a new object by applying the method people to the object god which is intended to be a class collecting the people of Israel. However, there is no direct object name denoting this object. The concept of object creation by path expressions is described in detail in Section 7.2.

## 7.1 Nesting of Path Expressions and F-Molecules

As mentioned before, every (ground) path expression corresponds to an object. This object is called the *object value* of a path expression. Thus, it is possible to nest path expressions in F-molecules as well as in P-molecules in any position where id-terms are allowed:

```
jacob.son@(rachel,11)[mother->rachel; father->jacob].
abraham[son->>{jacob.father}].
jacob[son@(joseph.mother)->>{benjamin}].                    (7.2)
male(jacob.father).
jacob.father.father = abraham.
```

F-Logic expressions are evaluated from left to right. Thus, if a path expression should occur at the method position or at the class position, resp. superclass position, it has to be enclosed by parentheses. To illustrate this we define a new "meta"-method twice by the following rule:

```
X[(M.twice)->Z] :- X[M->Y[M->Z]].
```

This new method may be invoked on other method names meaning that the original method is applied twice, e.g., the method denoted by the path expression father.twice would specify the "grandfather on the father's side" method. Hence, applying father.twice to Jacob yields Abraham as result. The other F-molecule denotes that Jacob belongs to god's people.

```
jacob[(father.twice)->abraham].
jacob:(god.people).                                         (7.3)
```

How parentheses affect the meaning of path expressions will become clear looking at the next two examples:

```
jacob.(father.twice):person.
jacob.father.twice:person.                                  (7.4)
(jacob.father).twice:person.
```

The path expression in the first F-molecule denotes abraham as in (7.3). As path expressions are evaluated left to right, the second and third F-molecule are equivalent. In our context[6], however, they are not meaningful (evaluating to false) because jacob.father is a person (isaac) and not a method, so that twice cannot be applied to this object.

[6] Assume the object base defined by Example (2.1) is given.

```
jacob:(god.people).
jacob:god.people.                                    (7.5)
(jacob:god).people.
```

In Example (7.5) the first F-molecule states that applying the method people to the object called god yields a class jacob belongs to. The second expression, which is equivalent to the third one, states that the object jacob is a member of the class god and denotes the application of the method people to the object jacob. However, the last two expressions are path expressions—not F-molecules—as they do not end with a specification list or an isa/subclass relationship (see Section 4.1).
Besides using path expressions instead of simple id-terms in F-molecules, it is also possible to nest path expressions and F-molecules the other way round: Intermediate objects in a path expression may have specification lists, turning them into F-molecules. As an example the path expression jacob.mother may be extended by specifying some properties for Jacob:

```
jacob:man[father->isaac].mother
```

In a rule body, this feature is useful to restrict the set of objects matching a path expression by selecting those with a certain property. For a formal analysis of such terms, see the *reference semantics* and *object semantics* of F-Logic expressions, e.g., in [LHL+98].

## 7.2 Object Creation with Scalar Path Expressions

Scalar path expressions are allowed in rule heads to induce the creation of new objects. This effect occurs whenever a path expression consists of a host object with a method application that has not been defined otherwise. When applying the method father to the object abraham, we create a new object representing Abraham's father:

```
abraham.father:man.
```

Here, the method father is defined for the object abraham and yields the new object abraham.father. Analogously,

```
jacob:(god.people).
```

creates a new object god.people by defining the method people for the object god. Both objects, abraham.father and god.people, have no ground id-term as object name; the user has to access these objects with their path expressions.

On the other hand, if the method application is defined somewhere else for the host object, the path expression evaluates to the corresponding result object. Assume the object base described by Example (2.1). Then, the path expressions in

```
isaac.father:person.
isaac[married_to->>jacob.mother].
```

do not create objects because the methods are already defined for the corresponding host objects. Instead, the facts abraham:person and isaac[married_to->>rebekah] are added.
Object creation with path expressions reveals its full power in connection with variables. See Section 9 for the use of variables in rules.

**Object Names and Path Expressions.** Because of the possibility to create new objects using path expressions we have to redefine the set of object names: now there are objects which are not associated to a ground id-term. For that purpose, the set of object names is defined by the union of the set of all ground id-terms and the set of all ground, unnested,

functional path expressions[7]. These are path expressions containing neither variables nor any F-molecules, therefore consisting just of a host object followed by one or more functional method applications (possibly with parameters). Examples for such path expressions are:

```
benjamin.father.father.mother
jacob.son@(rachel,1)                                    (7.6)
abraham.father
```

## 7.3 Path Expressions in Queries

Path expressions in a rule body or query help the user to describe the information in question more concisely, avoiding auxiliary variables for intermediate results. If for example the grandfather of Isaac is requested, this query can be written as

```
?- isaac.father[father->X].   instead of
?- isaac[father->Y], Y[father->X].
```

Path expressions may be eliminated from F-molecules in rule bodies or queries by decomposing the molecules into a set of F-atoms using new variables for the result values (in the above example, also a *don't-care* variable could be used which does not occur in the result set, see Section 9).

## 7.4 Multi-valued Path Expressions

Up to now only functional methods in path expressions have been considered. It is also possible to build path expressions with multi-valued methods, resulting in *multi-valued path expressions*. The use of a multi-valued method in a path expression is indicated by a double dot, e.g., "isaac..son". Such a path expression matches not just a single object (like a functional path expression does) but each object from a set of answers.

Every path expression is either functional or multi-valued. The *scalarity* of a path expression can be determined syntactically by considering the corresponding *unnested path expression* which is built by removing all isa and method specifications. Example (7.7) shows some path expressions and their corresponding unnested path expressions:

```
jacob[son->>{joseph}].(father.double):person
jacob.(father.double)

jacob[father->abraham..son].mother
jacob.mother                                            (7.7)

isaac:man..son[mother->rebekah]..son
isaac..son..son

jacob..son@(laban..daughter:woman)
jacob..son@(laban..daughter)
```

A path expression is multi-valued if its corresponding unnested path expression contains at least one multi-valued method application. Hence, in Example (7.7) the first two path expressions are functional and the last two are multi-valued. The second path expression is functional, although it contains a multi-valued path expression abraham..son. However, this multi-valued path expression appears at the result position of a specification and thus has no influ-

[7] Id-terms may be considered as a trivial case of path expressions (without any method application)

ence on the unnested path expression. The last path expression jacob..son@(laban..daughter) is multi-valued not only because of the host object but also because of the parameter that is denoted by a multi-valued path expression. Even if the method son were used as a functional method in this example, the path expression would be multi-valued due to the multi-valued parameter.

**Semantics of Multi-valued Path Expressions** As already mentioned, a multi-valued path expression describes a set of objects. Note that the set itself cannot be referenced in F-Logic. Instead, we consider each object in the set as one possible result of the method application. The query in (7.8) does not ask whether *all* of Abraham sons are sons of Sarah, too, but whether *one* of Abraham's sons is a son of Sarah[8] and therefore yields the answer true.

```
?- sarah[son->>abraham..son].                    (7.8)
```

Considering sets as a whole as in languages like LDL [NT89] requires stratification. The comparison of sets is not a built-in feature in F-Logic but can easily be done by an F-Logic program (see Section 14.6).

Since multi-valued path expressions do not denote sets, they may be used at any syntactical position in an F-molecule[9]. The following queries, for example, ask for those persons whose father is one of Abraham's sons, and for the sons of Jacob born by a daughter of Laban (Laban is the father of Leah and Rachel).

```
?- X:person[father->abraham..son].
?- jacob[son@(laban..daughter)->X].              (7.9)
```

In contrast to scalar path expressions, multi-valued path expressions *are only allowed in queries or rule bodies.* If we allowed multi-valued path expressions in facts or rule heads the following problem would occur: Since the multi-valued path expression abraham..son denotes the two objects isaac and ishmael the fact "abraham..son[lives->israel]." would be satisfied if Isaac, Ishmael or both lived in Israel. The last alternative would violate the minimality of the object base (see Section 10.1). If only one of Abraham's sons lives in Israel, it is not clear which one. This situation is similar to disjunctive expressions in a rule head, hence, we would have to handle a non-deterministic program. To avoid such problems, multi-valued path expressions are disallowed in rule heads.

## 7.5 Path Expressions with Inheritable Methods

Path expressions may contain inheritable methods, too. Such methods are indicated by one or two exclamation marks instead of dots. The path expression in example (7.10) denotes the object god if we consider the object base defined in example (3.7).

```
person!believes_in                               (7.10)
```

## 7.6 F-Molecules vs. Path Expressions

We have seen that F-molecules and path expressions may be combined and nested in several ways to obtain complex expressions. This is possible because every ground F-molecule and

[8] The semantics of this example differs from the semantics introduced in [FLU94].

[9] In this point the semantics of FLORID does not agree with [FLU94].

every ground path expression has an *object value* and a *truth value* according to a given object base.

An object base corresponds to an F-structure as introduced in [KLW95]. If an atom $t$ is true in a given object base or F-structure $\mathcal{I}$, we write $\mathcal{I} \models t$.

Although we define object values and truth value for both of them, the set of F-molecules and the set of path expressions are strictly disjunct: An F-molecule always ends with a specification, i.e., an isa specification denoting a class membership or subclass relationship, or a list of method specifications expressing the results of method applications to an object. A path expression always ends with a dot followed by a method application.

The object value and truth value is recursively defined for arbitrary complex expressions in an intuitive way corresponding to the evaluation of F-molecules:

**Definition 1 (Semantics [LHL+98])** *For an F-Logic database $\mathcal{I}$, the object values of ground expressions are given by the following mapping obj from ground expressions to sets of ground references:*

$$
\begin{array}{lcl}
obj(t) & := & t \text{ for a ground id-term } t \\
obj(r[spec]) & := & \{o \in obj(r) \mid \mathcal{I} \models o[spec]\} \\
obj(r:c) & := & \{o \in obj(r) \mid \mathcal{I} \models o:c\} \\
obj(c :: d) & := & \{c' \in obj(c) \mid \mathcal{I} \models c' :: d\} \\
obj(r.m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m\text{->}v]\} \\
obj(r..m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m\text{->>}\{v\}]\}
\end{array}
$$

*analogously for $r!m$ and $r!!m$.*

*The $\models$ relation extends from atoms to pure references and nested expressions as follows:*

$$
\begin{array}{lcl}
\mathcal{I} \models r & \Leftrightarrow & obj(r) \neq \emptyset \qquad \text{for a ground pure reference } r, \\
\mathcal{I} \models r_1[r_2@(p_1,...,p_n)\text{->}r_3] & \Leftrightarrow & \mathcal{I} \models o_1[o_2@(q_1,...,q_n)\text{->}o_3] \text{ for any } o_i \in obj(r_i),\ q_j \in obj(p_j) \\
 & & \text{analogously for ->>, =>>, and =>.} \\
\mathcal{I} \models r_1 : r_2 & \Leftrightarrow & \mathcal{I} \models o_2 : o_2 \text{ for any } o_1 \in obj(r_1),\ o_2 \in obj(r_2) \\
 & & \text{analogously for } ::.
\end{array}
$$

An F-Logic expression (i.e., an F-molecule or a path expression) is true if the corresponding object value contains at least one object, i.e., if one object has the appropriate properties. It is false if the object value is empty. Thus, also a path expression may be used as subgoals in a rule – evaluated to true if their object value is nontrivial.

Some examples showing F-molecules and path expressions as well as their object and truth values are presented in Figure 1. The underlying object world is the one defined in Example (2.1). An empty object value is written as $\emptyset$.

# 8 The Object Base

An object base is specified by a set of facts which is completed by additional information maintained by the system. These so-called closure properties express some basic features of the object-oriented paradigm and are discussed in detail in this section.

In contrast to [KLW95], in FLORID some closure properties are not defined for all possible object names but only for the set of *active object names*. This ensures that (if we disregard integers) the object model remains finite, enabling easier static safety checking. Remember that the set of object names includes the set of ground unnested functional path expressions

| expression | object value | truth value |
|---|---|---|
| isaac | isaac | true |
| isaac | isaac | true |
| isaac:man | isaac | true |
| isaac:woman | ∅ | false |
| isaac[son->>{jacob}] | isaac | true |
| isaac[son->>{abraham}] | ∅ | false |
| isaac.father | abraham | true |
| isaac.father | abraham | true |
| isaac..son | jacob, esau | true |
| abraham..son | isaac, ishmael | true |
| abraham..son[mother->sarah] | isaac | true |
| abraham..son:woman | ∅ | false |

Figure 1: Object values and truth values

(see Section 7.2). A ground id-term is an active object name if it appears at any syntactical position of one of the facts specifying the object base. A ground unnested functional path expression is an active object name if the appropriate method is defined in the object base. We write $o \in \mathit{active}(\mathcal{I})$ if an object name $o$ is active in $\mathcal{I}$.

## 8.1 Closure Properties of the Equality Predicate

The equality predicate determines a congruence relation over the set of object names which leads to the following implications: Let o, p, q be object names, $t, t'$ atoms and $\mathcal{I}$ an arbitrary object base.

- If $\mathsf{o} \in \mathit{active}(\mathcal{I})$, then $\mathcal{I} \models \mathsf{o}{=}\mathsf{o}$ (reflexivity).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$, then $\mathcal{I} \models \mathsf{p}{=}\mathsf{o}$ (symmetry).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$ and $\mathcal{I} \models \mathsf{p}{=}\mathsf{q}$, then $\mathcal{I} \models \mathsf{o}{=}\mathsf{q}$ (transitivity).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$ and $\mathcal{I} \models t$, then $\mathcal{I} \models t'$ where $t'$ is obtained from $t$ by replacing o with p (substitution).

## 8.2 Closure Properties of Subclass Relationships

As already mentioned in Section 3.1.2, the subclass relationship specifies a partial order on the set of object names. Besides that, every object belonging to a class is always an instance of every superclass of this class. These properties are stated by the following implications: Let o, p, q be object names and $\mathcal{I}$ an arbitrary object base.

- If $o \in \mathit{active}(\mathcal{I})$, then $\mathcal{I} \models \mathsf{o}{::}\mathsf{o}$ (subclass reflexivity).
- If $\mathcal{I} \models \mathsf{o}{::}\mathsf{p}$ and $\mathcal{I} \models \mathsf{p}{::}\mathsf{q}$, then $\mathcal{I} \models \mathsf{o}{::}\mathsf{q}$ (subclass transitivity).
- If $\mathcal{I} \models \mathsf{o}{::}\mathsf{p}$ and $\mathcal{I} \models \mathsf{p}{::}\mathsf{o}$, then $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$ (subclass acyclicity).
- If $\mathcal{I} \models \mathsf{o}{:}\mathsf{p}$ and $\mathcal{I} \models \mathsf{p}{::}\mathsf{q}$, then $\mathcal{I} \models \mathsf{o}{:}\mathsf{q}$ (subclass inclusion).

### 8.3 Closure Properties of Signatures

In combination with a given class hierarchy, a signature-F-atom implies other signature-F-atoms (e.g., the definition of a method for a class is valid for every subclass of that class, too). Other implications concern the replacement of a parameter class by a subclass or the result class by a superclass. This leads to the following implications (analogously for multi-valued signatures =>>): Let o, m, $a_1$,...,$a_n$, r, x be object names (n $\geq$ 0) and $\mathcal{I}$ an arbitrary object base.

- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)=>r] and $\mathcal{I} \models$ x::o, then $\mathcal{I} \models$ x[m@($a_1$,...,$a_n$)=>r] (type inheritance).
- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_i$,...,$a_n$)=>r] and $\mathcal{I} \models$ x::$a_i$, then $\mathcal{I} \models$ o[m@($a_1$,...,x,...,$a_n$)=>r] (input-type restriction).
- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)=>r] and $\mathcal{I} \models$ r::x, then $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)=>x] (output-type relaxation).

### 8.4 Miscellaneous Properties

Some more closure properties hold for every object base in F-Logic. The first one states that a functional method may not yield different result objects if it is applied to the same host object with the same parameter objects. Furthermore, every functional path expression is equated to the corresponding result object. The third statement ensures that active object names without any properties are true in every object base. The fourth one says that if a path expression is active, it is a result of the corresponding method application. Finally the empty set, resp. empty signature, is implied as a result by any data-F-atom with a multi-valued method, resp. any signature-F-atom.
Let o, m, $a_1$,...,$a_n$, r, s be object names (n $\geq$ 0) and $\mathcal{I}$ an arbitrary object base.

- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)->r] and $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)->s], then $\mathcal{I} \models$ r=s (analogously for inheritable functional methods).
- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)->r], then $\mathcal{I} \models$ o.m@($a_1$,...,$a_n$)=r (analogously for inheritable functional methods).
- If $o \in active(\mathcal{I})$, then $\mathcal{I} \models$ o.
- If o.m@($a_1$,...,$a_n$)$\in active(\mathcal{I})$, then $\mathcal{I} \models$o[m@($a_1$,...,$a_n$)->o.m@($a_1$,...,$a_n$)] (analogously for other types of path expressions).
- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)->>{r}], then $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)->>{}] (analogously for inheritable multi-valued methods).
- If $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)=>(r)], then $\mathcal{I} \models$ o[m@($a_1$,...,$a_n$)=>()] (analogously for multi-valued signatures).

## 9 Rules and Queries

**Rules.** Based upon a given object base (which can be considered as a set a facts), rules offer the possibility to derive new information, i.e., to extend the object base *intensionally*. Rules encode generic information of the form: "Whenever the precondition is satisfied, the conclusion also is". The precondition is called *rule body* and is formed by a conjunction of subgoals, i.e., possibly negated P- or F-molecules. The conclusion, the *rule head*, is a conjunction of P- and F-molecules. Syntactically the rule head is separated from the rule

body by the symbol ":-" and every rule ends with a dot followed by a whitespace (blank, tab or return).[10]

Non-ground rules use *variables* for passing information between subgoals and to the head. Variables can be considered as implicitly universally quantified and range over one rule at a time. Thus, using the same variable in different subgoals in fact defines a join condition.

Assume an object base defining the methods father and mother for some persons, e.g., the set of facts given in Example (2.1). The rules in (9.1) compute the transitive closure of these methods and define a new method ancestor:

```
X[ancestor->>Y]  :-  X[father->Y].
X[ancestor->>Y]  :-  X[mother->Y].
X[ancestor->>Y]  :-  X[father->Z], Z[ancestor->>Y].
X[ancestor->>Y]  :-  X[mother->Z], Z[ancestor->>Y].
```
(9.1)

**Queries.** A query can be considered as a special kind of rule with empty head. For syntactical distinction, a query begins with the query symbol "?-" followed by a rule body and ends with a dot followed by a whitespace. The following query asks about all female ancestors of Jacob:

```
?- jacob[ancestor->>Y:woman].
```
(9.2)

The answer to a query consists of all variable bindings such that the corresponding ground instance of the rule body is true in the object base. Considering the object base described by the facts of Example (2.1) and the rules in (9.1), the query (9.2) yields the following variable bindings:

```
Y/rebekah
Y/sarah
```

**Don't Care Variables.** When asking queries, it is often useful to have variables for intermediate results that do not show up in the answer set. This can be achieved with *don't care variables*, that is, variable names starting with the underscore symbol, e.g., _1,_Y,_little. In rules, such a variable is not propagated to the head (using don't care variables in rule heads will cause an error message). The grandchildren of sarah could be queried ad hoc by the equivalent queries

```
?- X:man.father[mother->sarah].        and
?- X:man[father->_Y], _Y[mother->sarah].
```
(9.3)

The variable "_" (a single underscore) plays a special role: every occurrence is considered as a *distinct* don't care variable (internally, they are transformed into distinct variable names like _#1,_#2).

Don't care variables can often be avoided by using path expressions, but not in every case. Don't care variables can even spare additional rules when occurring in negated subgoals.

[10] The condition that every rule or query has to end with a dot followed by a whitespace is necessary to avoid ambiguities because the dot is also used in path expressions:

```
X[grandma->sarah] :- X:man.father[mother->sarah].
```

Note that if the dot and father in that rule would be separated by a whitespace (typing error!), this would be equal to the following rule and an additional fact.

```
X[grandma->sarah] :- X:man.
father[mother->sarah].
```

**Negated Subgoals and Safety.** Similar to Datalog, a rule has to satisfy some constraints to guarantee that the number of newly derived facts remains finite. A rule is called *safe* if all its variables are limited. A variable in a rule is *limited* if it is limited by at least one subgoal. A subgoal containing a variable X limits this variable if it is positive and none of the following cases applies:

- The subgoal is an equality P-molecule where the variable X or the molecule X is one of the arguments and the other argument is
  - either an arithmetic expression containing a variable that is not limited by another subgoal in the same rule, e.g., X = Y + 3
  - or it is just a variable, say Y, or a molecule of the form Y, where Y is not limited by another subgoal in the same rule, e.g., X = Y, X = Y, X = Y.
- The variable X or the molecule X appears as a an operand in an arithmetic expression[11], e.g., Z = X + 2, Z = 3 * X.
- The subgoal is a subclass F-molecule where both arguments are exactly a variable or a variable followed by empty brackets and none of them is limited by another subgoal in the same rule, e.g., X::Y, X::(Y).
- The subgoal is an F-molecule of the form X::Y.
- The subgoal is just the variable X followed by an empty specification, i.e., X.
- The subgoal consists of the built-in predicate integer with the variable X or the molecule X as its argument, i.e., integer(X), integer(X).
- The variable X or the molecule X is one of the arguments of a comparison P-molecule, e.g., X > 5, X > 5, X > Y.

Additionally, a don't-care variable in a negated subgoal is considered limited, if this is the only occurrence of this variable in the rule.

Note that the concept of limited variables is slightly different from Datalog since equality as a reflexive relation is defined only for active object names (see Section 8). Hence, in F-Logic the variables in the subgoal f(X) = Y are limited, while in Datalog, they are not. Moreover, a variable appearing in a path expression or an F-molecule is always limited, e.g., the variable X is limited by each of the following subgoals:

```
X[father->Y] = Z
X.father = Z
X.age > 5
integer(X.age)
```
(9.4)

However, the restriction to active object names does not solve all problems because the number of active object names is infinite due to the existence of the integer objects. For this reason a subgoal like X = Y does not limit the variables X or Y. But if just one of the two variables is limited by another subgoal, the other one is limited, too. This analogously holds for subgoals of the form X::Y.

The following Examples (9.5) and (9.6) illustrate the concept of limited variables and safe rules:

```
X[father->Y] :- X:person.
X:bachelor :- X:person, not X[spouse->Y].
X:object :- X.
isaac[age->X] :- rebekah[age->Y], Y = X - 10.
```
(9.5)

[11] Remember that arithmetic expressions are only allowed in equality P-molecules

None of these rules is safe. In the first rule, the variable Y only appears in the rule head and therefore it cannot be limited by a subgoal. The second rule contains a negated subgoal with the variable Y which is not limited by any other subgoal. Nevertheless, replacing $Y$ by a don't care variable makes it safe:

X:bachelor :- X:person, not X[spouse->_Y].

In the third rule, the variable X is not limited because it is used as a single variable with an empty specification and due to the integer objects the number of variable bindings for X is infinite. However, by replacing the subgoal by f(X) the rule would become safe because there is only a finite number of active object names matching the term f(X). Finally, the last rule contains an equality P-atom with an arithmetic expression as argument. Since the variable X is not limited by another subgoal the rule is not safe. Actually in the last case the problem is not the derivation of an infinite number of new facts. However, the evaluation strategy for arithmetic expressions requires that all their variables are bound. The rule can be made safe by rewriting the equation s.t. $Y$ (bound) occurs in the arithmetic part instead of $X$:

isaac[age->X] :- rebekah[age->Y], X = Y + 10.

These following rules are safe:

```
X:adult :- X[age->Y], Y = Z, Z > 18.
X[older->>Y] :- X.age > Y.age.                    (9.6)
X:person :- X.father.
```

In the first one, the variables X and Y are limited by the first subgoal. The equality P-atom limits the variable Z because Y is already limited. In the second rule the arguments of the comparison predicate are not single variables, so they are limited. This becomes even more obvious by rewriting the rule body as X[age->A], Y[age->B], A > B. In the third rule the variable X is limited because of the method father since in any object base this method is only defined for a finite number of objects.

**Infinite number of object names.** Another problem is the creation of an infinite number of new object names by the use of function symbols or path expressions. Consider the following rule:

X.father:person :- X:person.

If our object base contains at least one object belonging to the class person, an infinite number of new objects is created by evaluating this rule again and again. Every time another father object is created and the evaluation will never stop. As this kind of recursion may also occur indirectly involving several rules, it cannot be fixed by a static check.

# 10 Programs and Evaluation

An F-Logic 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**.

## 10.1 Fixpoint Semantics

The evaluation strategy for F-Logic programs (without inheritance) is basically the same as for Datalog programs. The bottom-up evaluation of an F-Logic 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[12]. 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 F-Logic program reaches a fixpoint which coincides with the unique minimal model of that program[13]. The minimal object base of an F-Logic program is defined as the smallest set of P- and F-atoms such that all closure properties and all facts and rules of the program are satisfied.

## 10.2 Negation and Stratification

Negation in FLORID 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.
Consider the following program:

```
isaac[father->abraham].
isaac:orphan :- not isaac[father->abraham].
?- sys.eval.
?- isaac:orphan.
```
(10.1)

You may be surprised to get the answer true. However, this is correct under the inflationary semantics because in the first iteration of the evaluation the object base still is empty, so the negated subgoal evaluates as true and the information that Isaac is an orphan is inserted into the object base as well as the fact that Abraham is Isaac's father. In the second iteration, the rule does not fire any more, but information inserted in the object base is never removed from there. Hence, the answer to the query is true.

As in the case of Example (10.1) 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 three-valued model is not supported by FLORID but can easily be simulated [MLL97] (cf. Example 14.10). A very common approach is to *stratify* logic programs and to compute the perfect model [ABW88, Prz88]. Unfortunately, due to the powerful syntax of F-Logic, the class of stratified programs is very small. Thus, automatic stratification cannot be done by FLORID for a reasonable large class of programs. To enable the programmer to specify an explicit stratification, FLORID provides a system command "?- sys.strat.dolt." 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[14].

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

[13] Note that this fixpoint is not necessarily finite, cf. Example 9.

[14] There are other possibilities to ensure that the negated subgoal becomes active only after the information to be negated has been inferred, cf. Example 14.10.