How to Write F-logic Programs in FLORID

A Tutorial for the Database Language F-logic

Version 2.0

Jürgen Frohn   Rainer Himmeröder
Paul-Th. Kandzia   Christian Schlepphorst

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

November 1997

Contents

List of Figures

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.

The theoretical foundations of F-logic have been described in the F-logic report [KLW95] and [FLU94], but we think that these sources are not the best choice to learn how to *apply* F-logic from a more practical point of view.

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 impression 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 the sections 10 and 11. In section 12 a number of examples illustrate special features of FLORID and offer more material to start with.

As a major new feature, FLORID 2.0 provides access to Web documents. Its usage is described in section 13.

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 in the case of questions about FLORID system commands.

If you have any further questions or problems with FLORID do not hesitate to contact us under the following e-mail address:

florid@informatik.uni-freiburg.de

Acknowledgement: First of all, we want to thank Georg Lausen, the head of our group, who made the FLORID project possible. Furthermore, our thanks go to Heinz Uphoff, Bertram Ludäscher and Wolfgang May for many ideas and discussions that helped developing FLORID, and to the students who were involved in the implementation.

2 A First Example

Before explaining the syntax and semantics in detail we want to 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 */                    (2.1)
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]}].
```

The first part of this example consists of a set of facts which express that some persons belong to the class `man` resp. `woman` and give information about the father and mother relationships between some of these persons. 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 the bottom-up style, new relationships between the objects, denoted by the methods `son` and `daughter`, are added to the object base as intensional 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 begin with an uppercase letter, e.g. `X`, or underscore.

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). 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].                      (2.2)
```

[1]Methods and classes also are objects, see section 3.2.1 and 3.2.2.

All three parts of the F-logic program in example (2.1) begin with a comment. There are three 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 style) mark the rest of a line as a comment.

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.

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 first always begin with a lowercase letter, whereas the latter always begin with an uppercase letter or underscore. The first letter in an object name or variable name may be followed by uppercase letters, lowercase letters, numerals or the underscore symbol "_" . Examples for object names are `abraham, man, daughter`, for variable names are `X, Method, _11`. 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* with a given arity, 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*.

3.2 Expressing Information about an Object

Following the object oriented paradigm, *methods* represent relationships between objects. Furthermore, objects may be organized in *classes*. Such information about objects is expressed by *F-molecules* (see section 4). 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}].          (3.1)
```

This F-molecule may be split into several *F-atoms* where each F-atom expresses exactly one property of an object, e.g. a class membership or a method specification:

```
isaac:man.
isaac[father->abraham].                                   (3.2)
isaac[son->>{jacob}].
isaac[son->>{esau}].
```

After this short introduction to F-molecules and F-atoms, the different kinds of F-atoms are explained in the following subsections.

3.2.1 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. There is no distinguished position in a F-atom, that is, all objects may occur in all places: host object, result position or method position. Thus in our example (3.1) the method names `father` and `son` are object names just like `isaac` and `abraham`. Furthermore, variables may be used at the method position 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.

To split F-molecules containing a multi-valued method, we divide the set of result objects into singleton sets. For singleton sets it is allowed to omit the curly braces enclosing the result set, so that the three expressions given in (3.3), (3.4) and (3.5) are equivalent, which means that they yield the same object base:

```
isaac[son->>{jacob,esau}].                                    (3.3)
```

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

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

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};                            (3.6)
      son@(bilhah)->>{dan, naphtali}].
```

It is possible to use more than one parameter for a method. 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].
```
(3.7)

In examples (3.6) and (3.7) 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 (3.1), questioning the sons of Isaac yields all his known sons:

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

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

X/jacob
X/esau
```
(3.8)

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.

Thus, such queries do not require a set comparison, but a set *inclusion*. 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->>{}].
```
(3.9)

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 12.6.

Also note that variables in a query may only be bound to objects, never to sets of objects. The following query about Isaac's sons binds the variable `X` to the individual objects, not to the set as a whole (there is no object representing this set):

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

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

X/jacob
X/esau
```
(3.10)

3.2.2 Class Membership and Subclass Relationship

Isa-F-atoms state that an object belongs to a class and *subclass-F-atoms* express the subclass relationship between two classes. 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 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 arbitrary 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.

The class membership (resp. the subclass relation) is denoted by a single colon (resp. a double colon). 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.                                        (3.11)
woman::person.
man::person.
```

In isa-F-atoms (resp. subclass-F-atoms), the object and the class (resp. the subclass and the superclass) are denoted by id-terms because **classes are objects** as well as methods. 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 the class position in an isa- or subclass-F-atom.

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

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].                          (3.12)
```

Given this inheritable data-F-atom together with the subclass relationships and class memberships from example (3.11) we can derive the following information:

```
abraham[believes_in->god].
isaac[believes_in->god].
sarah[believes_in->god].                            (3.13)
woman[believes_in*->god].
man[believes_in*->god].
```

Example (3.13) 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 as the method `believes_in` is explicitly defined for the object `ahab`.

```
ahab:person[believes_in->baal].
```
(3.14)

In F-logic even multiple inheritance is supported. For a more detailed discussion about inheritance see section 10.3.

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. This restrictions may be viewed as typing constraints.

Thus, all signature-F-atoms together with the class hierarchy form the schema of an F-logic database.

Here are some examples for signature-F-atoms:

```
person[father=>man].
person[daughter=>>woman].
man[son@(woman)=>>man].
```
(3.15)

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`.

As in data-F-atoms, the arrow head indicates if 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.

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 defined to be in *all* of those classes[2]:

```
person[father=>(man, person)].
```
(3.16)

```
person[father=>(man)].
person[father=>(person)].
```
(3.17)

Both expressions in the examples (3.16) and (3.17)[3] 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

[2]Note, that we do not require the class hierarchy to be a lattice, so that the intersection of the result classes may not exist in the hierarchy.

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

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=>()].                                          (3.18)
```

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 11.

Overloading F-logic supports overloading of methods. This means that methods denoted by the same object name may be applied to instances of different classes if corresponding signatures exist for those 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 (3.6) and as a functional method with two parameters in example (3.7). The corresponding signature-F-atoms look like this:

```
man[son@(woman)=>>(man)].
man[son@(woman,integer)=>(man)].                                  (3.19)
```

4 Nesting Object Properties

As already shown in example (3.1), 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).
jacob[(father:method)->isaac].
```
(4.3)

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.
```
(4.4)

To avoid ambiguities, the nested properties have to be included in parentheses if those properties belong to a method object[4], 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]).
```
(4.5)

4.1 F-molecules without any properties

If we want to represent an object without giving any properties, we have to attach an empty spcification list to the object name, e.g. "`abraham[].`". If we use an expression that consists solely of an object name as a molecule, e.g. "`abraham.`", this expression 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)

[4]Due to the introduction of path expressions in section 7, parentheses for F-molecules at the method position are necessary to avoid possible ambiguities.

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

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).      (5.3)
```

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

```
married(isaac,rebekah).
isaac[father->abraham].                      (5.4)
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, 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. In such a case the object names are said to be **equal**, indicated by the equality predicate "`=`", e.g. `abram = abraham`.[5] 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,

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

making static safety checking impossible[6]. Instead there is a built-in *predicate* `integer` in FLORID.

Within a query or a rule body, relations between integer numbers may be tested with the *comparison predicates*[7] "`<`","`>`", "`<=`" 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 not optional, `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`".

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.                                         (6.2)
```

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 F-logic program, an error is reported and the evaluation stops.

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:

strlen(arg_1, arg_2)
This predicate holds if arg_2 represents the length of the string arg_1. For practical reasons, variables at position of arg_1 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(arg_1, arg_2, arg_3)
strcat succeeds if arg_3 is the concatenation of arg_1 and arg_2. E.g. "`?- strcat("a","b",X).`"

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

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

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(arg_1, arg_2)
`substr` holds if arg_1 is a substring of arg_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($arg_1, arg_2, arg_3, arg_4$)
`match` finds all strings contained in arg_1 which match the pattern given by a regular expression in arg_2; arg_3 is a format string describing how the matched strings should be returned in arg_4. This feature is useful when using groups (expressions enclosed in `\(...\)`) in the regular expression of arg_2.

For example "`?- match("linux98","\([0-9]\)\([0-9]\)","\2swap\1",X).`" returns `X/"8swap9"` (the first and second match are swapped). In the format string arg_3, groups are referred to by their number: `\n`, where `n` ranges from 1 to 9. If arg_3 is the empty string (`""`) all matches are returned without formatting. All arguments of `match` are strings, optional for arg_1 which may also be a Web document. With the exception of arg_4 all arguments must be bound. A full explanation of the syntax and use of regular expressions is given in appendix B.

string(arg)
A built-in *class* string would contain an infinite number of instances. Since related to safety checking the same problems occur as in case of intergers, a built-in *predicate* `string` is used in FLORID.

6.4 Aggregation

Since version 2.0, aggregation has been implemented in FLORID. An aggregation term has the form

$$agg\{X[G_1, \ldots, G_n]; b\}$$

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). The list of grouping variables $[G_1, \ldots, G_n]$ is optional and may be omitted.

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. Some examples queries:

```
?- Z = max{X; john[salary@(Year)->X]}.
?- Z = max{X; john[salary@(Year)->X], Year < 1990}.                    (6.3)
?- 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 has a second 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.

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 $X, G_1, \ldots, G_n$ 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.

Semantics The result of an aggregation

$$Z = agg\{X[G_1, \ldots, G_n]; body\}$$

can be viewed as a relation $r(Z, G_1, \ldots, G_n)$ where Z is the result of the aggregation with grouping variables fixed. Thus r fulfills the functional dependency $G_1, \ldots, G_n \to Z$.

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).

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 $V < Z, Z = agg\{X[G_1, \ldots, G_n]; body\}$.

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]}.                    (6.4)
?- 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 also).

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 12.3 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.                                   (6.5)

?- sys.strat.doIt[].  %% 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 side 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 (3.7):

```
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 at 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.
```

If a path expression occurs 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]].                           (7.3)
```

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 someone's grandfather. 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).                                         (7.4)
```

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.                                  (7.5)
(jacob.father).twice:person.
```

The path expression in the first F-molecule denotes `abraham` as in (7.4). As path expressions are evaluated left to right, the second and third F-molecule are equivalent. In our context[8], however, they are not meaningful because `jacob.father` is a person and not a method, so that `isaac` cannot be applied twice.

```
jacob:(god.people).
jacob:god.people.                                           (7.6)
(jacob:god).people.
```

In example (7.6) 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). The syntax for the appropriate F-molecules would be:

```
jacob:god.people[].
(jacob:god).people[].                                       (7.7)
```

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                             (7.8)
```

[8]Assume the object base defined by example (2.1) is given.

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.

7.2 Object Creation with Path Expressions

Path expressions in a rule head or a fact may 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. 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 appearing in the following facts do not create objects because the methods are already defined for the corresponding host objects:

```
isaac.father:person.
isaac[married_to->>jacob.mother].
```
(7.9)

However, when applying the method `father` to the object `abraham` as in example (7.10), we create a new object representing Abraham's father. Then the method `father` is defined for the object `abraham` and yields the new object `abraham.father`. The second fact in the example 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 its path expressions.

```
abraham.father:man.
jacob:(god.people).
```
(7.10)

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[9]. 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)
abraham.father
```
(7.11)

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. They can spare variables for intermediate results. If for example the grandfather of Isaac is asked, this query can be written as:

```
?- isaac.father[father->X].
```
(7.12)

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

Without path expressions we have to bind the variable `Y` explicitly to the object denoting Isaac's father[10]:

```
?- isaac[father->Y], Y[father->X].                                    (7.13)
```

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 example (7.12) the path expression `isaac.father` may be replaced by a new variable `Y` holding the result of the corresponding method application, i.e. `isaac[father->Y]`. Example (7.13) shows the resulting set of F-atoms.

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 mult-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 objects.

Every path expression is either functional or multi-valued. This so-called *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 (7.14) 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
                                                                      (7.14)
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 (7.14) 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 influence 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

[10]It is also possible to use don't-care variables for intermediate results that shall not appear in the answer set, see section 9.

F-logic. Instead, we consider each object in the set as one possible result of the method application. Let us explain this by an example. The query in (7.15) 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[11] and therefore yields the answer true.

```
?- sarah[son->>abraham..son].                                  (7.15)
```

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 12.6).

Since multi-valued path expressions do not denote sets, they may be used at any syntactical position in an F-molecule[12]. The following queries, for example, ask for those persons whose father is one of Abraham's sons, resp. 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].                            (7.16)
```

No Object Creation with Multi-valued Path Expressions Multi-valued path expressions may only be used 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 facts or 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 (7.17) denotes the object `god` if we consider the object base defined in example (3.12).

```
person!believes_in                                             (7.17)
```

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 every ground path expression has an *object value* according to a given object base. Based upon the definition of the object value for unnested path expressions (i.e. without any specifications) and simple F-molecules (i.e. without any path expressions), the object value is recursively defined for arbitrary complex expressions.

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

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

The object value of a ground unnested functional path expression is the result object of the corresponding method application to the host object. For a ground unnested multi-valued path expression the object value consists of each of the objects in the corresponding set of result objects. If the result of the method application is the empty set, the object value of the appropriate multi-valued path expression is `empty`. The object value of a ground unnested path expression, which may be functional or multi-valued, is empty, too, if the method application is not defined for the given host object in the underlying object base.

For a ground F-molecule with an id-term or a functional path expression as its host object the object value is defined as the object value of this host object if this satisfies all the specifications given in the F-molecule. If a multi-valued path expression appears at the position of the host object, the object value comprises only those objects that fulfill the given specifications. If none of the objects satisfies these specifications, the object value is empty.

Although we have defined object values 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.

Besides the object value F-molecules also have a *truth value*. Note that this truth value is not defined for path expressions. An F-molecule 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. Since a path expression has no truth value it may not be used as a subgoal in a rule. This will cause an error. However, a path expression can be extended to an F-molecule by adding empty pair of brackets. For example the path expression `jacob.father.mother` may be extended to the F-molecule `jacob.father.mother[]` to use it as a subgoal of a rule.

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). Path expressions have a dash in the truth value column. An empty object value is written as $\emptyset$. For the exact description of the syntax see appendix A.

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

Figure 1: Object values and truth values

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 the following subparagraphs.

In our notation, 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 I, we write $I \models t$.

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 (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 $o \in active(I)$ if an object name o is active in 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 I an arbitrary object base.

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

8.2 Closure Properties of Subclass Relationships

As already mentioned in section 3.2.2, 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 I an arbitrary object base.

- If $o \in active(I)$, then $I \models$ `o::o` (subclass reflexivity).
- If $I \models$ `o::p` and $I \models$ `p::q`, then $I \models$ `o::q` (subclass transitivity).
- If $I \models$ `o::p` and $I \models$ `p::o`, then $I \models$ `o=p` (subclass acyclicity).
- If $I \models$ `o:p` and $I \models$ `p::q`, then $I \models$ `o: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 each 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,` $\mathtt{a}_1,\ldots,\mathtt{a}_n$`, r, x` be object names (n $\geq$ 0) and I an arbitrary object base.

- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>r]` and $I \models$ `x::o`, then $I \models$ `x[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>r]` (type inheritance).
- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_i,\ldots,\mathtt{a}_n$`)=>r]` and $I \models$ `x::`$\mathtt{a}_i$, then $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{x},\ldots,\mathtt{a}_n$`)=>r]` (input-type restriction).
- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>r]` and $I \models$ `r::x`, then $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>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,` $\mathtt{a}_1,\ldots,\mathtt{a}_n$`, r, s` be object names (n $\geq$ 0) and I an arbitrary object base.

- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->r]` and $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->s]`, then $I \models$ `r=s` (analogously for inheritable functional methods).
- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->r]`, then $I \models$ `o.m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=r` (analogously for inheritable functional methods).
- If $o \in \mathit{active}(I)$, then $I \models$ `o[]`.
- If `o.m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)`$\in \mathit{active}(I)$, then $I \models$`o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->o.m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)]` (analogously for other types of path expressions).
- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->>{r}]`, then $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)->>{}]` (analogously for inheritable multi-valued methods).
- If $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>(r)]`, then $I \models$ `o[m@(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`)=>()]` (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 fulfilled, 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).

To express generic knowledge, it is very important to have *variables* that can pass 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 biblical 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
```

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. The following example shows the problem:

```
X[grandma->sarah] :- X:man.father[mother->sarah].
```
(9.3)

The rule body in (9.3) corresponds to a conjunction of F-atoms. Replacing the path expression by a new variable `Y`, the rule could also be written as:

```
X[grandma->sarah] :- X:man, X[father->Y], Y[mother->sarah].
```
(9.4)

If the dot and `father` in (9.3) are separated by a whitespace, this would be equal to the following rule and an additional fact.

```
X[grandma->sarah] :- X:man.
father[mother->sarah].
```
(9.5)

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 grandchilds of `sarah` could be queried ad hoc by transforming (9.4) into

```
?- X:man[father->_Y], _Y[mother->sarah].
```
(9.6)

avoiding path expressions.

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.

Negated Subgoals and Safety Similar to Datalog, a rule has to fulfill 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[13], 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`.

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

Additionally, a don't-care variable in a negated subgoal is considered as 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)
```
(9.7)

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 (9.8) and (9.9) 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.
```
(9.8)

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. 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.

```
X:adult :- X[age->Y], Y = Z, Z > 18.
X[older->>Y] :- X.age > Y.age.
X:person :- X.father[].
```
(9.9)

These rules are safe. 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.
```
(9.10)

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

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[14]. This evaluation of rules is continued as long as new information is obtained. As in the case of Datalog, the evaluation of an negation-free F-logic program reaches a fixpoint which concurs with the unique minimal object base of that program[15]. 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

[14]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.

[15]Note that this fixpoint is not necessarily finite, cf. example (9.10).

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 anymore but information inserted in the object base is never removed from there anymore. 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 12.10). 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.doIt[].`" which divides a program into two or more 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[16].

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 and facts of the lower stratum are not considered anymore during the further evaluation. To illustrate this we extend example (10.1) by the stratification command:

```
isaac[father->abraham].
?- sys.strat.doIt[].
isaac:orphan :- not isaac[father->abraham].        (10.2)
?- sys.eval[].
?- isaac:orphan.
```

The first stratum of the program consists only of the fact that Abraham is Isaac's father. Hence, this is the only information inserted into the object base during the evaluation of the first stratum. Next, the rule belonging to the second stratum is evaluated. Since it is already derived that Abraham is the father of Isaac the rule does not fire anymore and Isaac is not said to be an orphan.

Negation of Complex Molecules Negated subgoals may consist not only of atoms but also of molecules as the following example shows. This rule states that two persons are not related to each other if they have no common ancestor[17]:

```
X[notrelated->>Y] :- X:person, Y:person,
                     not X[ancestor->>Y..ancestor].   (10.3)
```

Although FLORID can handle negated molecules, their decomposition is not as easy as it is for positive molecules. It is not possible to transform this rule into a single one consisting of F-atoms only without changing the semantics. On the one hand, negated conjunctions become disjunctions of negated atoms what cannot be expressed in F-logic. Path expressions on the other hand have to be decomposed by introducing existentionally quantified variables what is not possible in F-logic, either[18].

[16]There are other possibilities to ensure that the negated subgoal becomes active only after the information to be negated has been inferred, cf. example 12.10.

[17]To achieve the desired result the ancestor relation has to be reflexive, i.e. `X[ancestor->>X] :- X:person.`

[18]In version 2.0, don't-care variables have been introduced that can be viewed as existentionally quantified if

Instead, decomposing negated molecules requires an additional rule with a new predicate symbol in its head. The arguments of this predicate are all the variables appearing in the negated molecule and it is defined by a new rule the body of which consists of the decomposition of the molecule (see sections 4 and 7.3). Then, the negated molecule in the original rule is replaced by the negation of this new P-atom. The result of decomposing the negated molecule in example (10.3) is:

```
common_anc(X,Y)        :-  X[ancestor->>Z], Y[ancestor->>Z].
?- sys.strat.doIt[].                                            (10.4)
X[notrelated->>Y]      :-  X:person, Y:person, not common_anc(X,Y).
```

Note that `common_anc` has to be a predicate occurring nowhere else in the program containing the original rule.

Stratification to Enhance Efficiency The system command "`?- sys.strat.doIt[].`" may also be used in negation-free F-logic programs to speed up the evaluation. Consider the facts denoting the father and mother relationships in example (2.1) and assume that we want to define the relationship `ancestor` and its inverse `descendant`:

```
% facts defining the father and mother relationships
...
X[ancestor->>Y] :- X[father->Y].
X[ancestor->>Y] :- X[mother->Y].
% all direct ancestors are computed so these two rules
% do not have to be considered anymore
?- sys.strat.doIt[].                                            (10.5)
X[ancestor->>Y] :- X.father[ancestor->>Y].
X[ancestor->>Y] :- X.mother[ancestor->>Y].
% all ancestors are computed and the derivation of the descendant
% relation is deferred until now
?- sys.strat.doIt[].
X[descendant->>Y] :- Y[ancestor->>X].
```

The stratification command may enhance efficiency by dividing a negation-free program into several strata whenever every subgoal in a rule only depends on rules belonging to the same or lower strata. This way we can avoid to evaluate non-recursive rules again and again always deriving the same information.

Another strategy to avoid redundant derivation of information is semi-naive evaluation [Ban86]. An adoption of this strategy is included in FLORID version 2.0. However, it turned out in practice that reasonably subdivide the programs into strata is still advantageous and will keep the overhead of semi-naive evaluation small.

10.3 Inheritance

To improve efficiency new information inherited from a class to an instance or a subclass is only derived when this information is needed to evaluate a rule body or to answer a query.

they occur in negated subgoals. However, for safety reasons, negated don't-care variables cannot be propagated to another subgoal, so that this is no solution for flattening the rule (10.3). See also section 9.

For example, evaluating the facts in example (10.6) without any other rule or query does not infer that Abraham believes in god.

```
person[believes_in*->god].
abraham:person.
```
(10.6)

To implement this concept we define *inheritance triggers* which describe a method application to a certain object. Furthermore, information is only inherited if an inheritance trigger is *activated* and *fires*. Given an inheritable method, two conditions have to be fulfilled to activate an inheritance trigger:

1. There has to be an appropriate subgoal in a rule body or query which matches the inheritable method. With regard to example (10.6) the following query meets this condition. The corresponding inheritance trigger consists of the method `believes_in` applied to the object `abraham`.

   ```
   ?- X[believes_in->Y].
   ```
   (10.7)

2. The inheritable method is not yet defined for the object which would inherit the method.

However, *firing* an inheritance trigger is deferred until the evaluation of the program has reached a fixpoint. The reason is that by evaluating the rules of a program new information could be derived that overrides the inherited information and deactivates the inheritance trigger. Thus, logic rules are given priority over inheritance.

Example (10.8) illustrates how inheritance works in the case of overriding. This program states that all persons believe in Baal except for Abraham and his descendants, who believe in god.

```
abraham:person[believes_in->god; descendant->>{isaac:person}].
ahab:person.
person[believes_in*->baal].
X[believes_in->god] :- abraham[descendant->>X:person].
god[loves->>X] :- X[believes_in->god].
?- sys.eval[].
?- god[loves->>X].
```
(10.8)

Evaluating each rule and fact of the program once derives that Abraham, Isaac and Ahab are persons, Isaac is a descendant of Abraham, Abraham believes in god and persons in general believe in Baal. Note that the method `believes_in` cannot be inherited to Abraham because it is already defined. However, this method could be inherited to Isaac and since there is a rule with an appropriate subgoal, namely `X[believes_in->god]`, the inheritance trigger for applying the method `believe_in` to the object `isaac` is active. But this inheritance would lead to a wrong answer. Instead, the firing of this inheritance trigger is deferred until a fixpoint of the program is reached. Computing this fixpoint we derive that Isaac believes in god because he is a descendant of Abraham. Since the method `believes_in` is now defined for Isaac, the corresponding inheritance trigger is not active anymore and cannot fire. Only the inheritance trigger for `believes_in` applied to Ahab is still active and, since the evaluation has reached a fixpoint, this trigger fires.

However, there is another problem with respect to inheritance. If all active inheritance triggers fired at the same time it could be possible that contradictory information is derived.

Example (10.9) denotes that there are two kings, John and Peter, but only one of the them can live in the palace, the other one has to live abroad:

```
john:king.
peter:king.
king[lives*->palace].
peter[lives->abroad] :- john[lives->palace].          (10.9)
john[lives->abroad] :- peter[lives->palace].
?- sys.eval[].
?- X[lives->Y].
```

Since both kings, John and Peter, may inherit the method `lives` from their class `king` there are two active inheritance triggers when we consider the rules of the program. Firing both triggers would yield that both kings live in the palace, contradicting the rules (to ensure consistency, we would be forced to infer `palace = abroad`!). The solution is to fire only one inheritance trigger at a time. After having fired one of the two active inheritance triggers in our example we first compute a new fixpoint before firing the other inheritance trigger. However, the evaluation of the fixpoint deactivates the other inheritance trigger, so it does not fire anymore and the evaluation is finished.

The question is which of the two inheritance triggers fires and which one is deactivated afterwards. This problem is similar to dealing with a disjunction in a rule head of a logic program. Hence, it is *non-deterministic* which of the two instances inherits the method from the class (see also [Kan97]). So, there are two minimal models satisfying example (10.9), one of them stating that Peter lives in the palace and John lives abroad, the other one stating the opposite.

Let us summarize the concept of inheritance in FLORID. Having reached a fixpoint we determine which inheritance triggers are active and fire exactly one of them. Next, we check if the inherited information may be used to derive other information by evaluating all rules again until a fixpoint is reached. So, fixpoint computation and firing a single inheritance trigger at a time alternate until a fixpoint is reached where no inheritance triggers are active anymore.

This strategy avoids the inheritance of lots of information which is not needed for the evaluation of a given F-logic program. Nevertheless, the strategy is correct since every inherited method appearing in a rule body or query activates a trigger and other inherited methods not occurring in any rule body or query have no influence on the evaluation of the program.

On the other hand, the performance of evaluation may become very low if a lot of inheritance triggers are active because they are fired successively, only one at a time. Example 12.9 illustrates how evaluation may be improved in this case.

Multiple Inheritance Dealing with multiple inheritance is a common problem in object oriented systems. Multiple inheritance means that an inheritable method is defined for two different classes which have a common instance. We have to distinguish two cases:

If there is a subclass relationship between these two classes, the method result of the superclass is overridden by the one of the subclass. This means that the method defined by the subclass is inherited to the object. In example (10.10) the object `potiphar` inherits the method `believes_in` from the class `egyptian`, hence, Potiphar believes in the Pharaoh.

```
person[believes_in*->god].
egyptian::person[believes_in*->pharaoh].
potiphar:egyptian.                                        (10.10)
?- sys.eval[].
?- potiphar[believes_in->X].
```

In [MSL97] it is shown how interitance can also be used to model dynamic processes.

In the other case there is no subclass relationship, i.e. there are two classes, incomparable in the partial order induced by the class hierarchy, which may inherit information to the same object. . Inheriting a functional method from both classes would cause the result objects to be equated. Again, this problem is solved by firing only one inheritance trigger at a time. However, the choice which one will fire is non-deterministic. In example (10.11) Paul either serves Jesus or he serves Caesar but not both of them because inheriting the method `serves` once deactivates the trigger to inherit it from the other class, too.

```
paul:roman.
paul:christian.
roman[serves*->caesar].
christian[serves*->jesus].                                (10.11)
?- sys.eval[].
?- paul[serves->X].
```

11 Type checking

Although signature-F-atoms are supported, automatic type checking is not implemented in FLORID because static type checking is not possible for general F-logic programs. This is due to the possibility to generate new subclass relationships or equalities dynamically by rules. However, given an object base (e.g. the fixpoint of an F-logic program), type checking may be realized by some additional rules.

To verify whether every method in an object base is typed correctly we have to examine *type safety* and *type correctness*. Type safety means that there is no method application without a corresponding signature in the object base. Type correctness implies that every signature covering a given method application is fulfilled by the latter, i.e. the result object of a method application has to be an instance of *all* result classes of the appropriate signatures.

The following rules check if every functional method without parameters is type safe and type correct. Other kind of methods may be handled analogously. To defer the evaluation of the type checking rules until the fixpoint of the original program is computed we use the stratification command "`?- sys.strat.doIt[].`" (see section 10.2).

```
?- sys.strat.doIt[].
type_safe(O,M,R,funct) :- O[M->R], O:C, C[M=>()].
?- sys.strat.doIt[].
type_unsafe(O,M,R,funct) :- O[M->R], not type_safe(O,M,R,funct).
type_incorrect(O,M,R,funct) :- O[M->R], O:C, C[M=>D], not R:D.      (11.1)

?- sys.eval[].
?- type_unsafe(Object,Method,Result,KindOfMethod).
?- type_incorrect(Object,Method,Result,KindOfMethod).
```

12 Some Example Programs

The programs presented in this section demonstrate data modeling in F-logic and serve to explain some typical problems that may arise. All these programs are found in the directory `florid/examples/tutorial`. Just consult and query them. For more information about the system see the user manual.

12.1 Rules and Path Expressions

```
%% family.flp
%% Shows the use of scalar and multivalued methods and path expressions

/*
                  henry
                    |
           ------------------
           |                |
          tim              theo
          |                |
     ----------          ----
     |        |             |
   paul     peter          mike
     |        |            |
   ----     ---            ---
   |  |     |                |
john jack cliff             abe

*/

henry : person.
tim:person[father->henry].
theo:person[father->henry].
paul:person[father->tim].
peter:person[father->tim].
mike:person[father->theo].
john:person[father->paul].
jack:person[father->paul].
cliff:person[father->peter].
abe:person[father->mike].

/* RULES */
X[son->>Y]      :- Y[father->X].
X[brother->>Y]  :- X[father->Z], Y[father->Z], not X=Y.
% equivalent rule with path expressions:
% X[brother->>Y]  :- X[father->Y.father], not X=Y.

X[uncle->>Y]    :- X[father->Z] , Z[brother->>Y].
```

```
% equivalent rule with path expressions:
% X[uncle->>Y]    :- X.father[brother->>Y].

Y[nephew->>X]   :- X[uncle->>Y].
X[cousin->>Z]   :- X[uncle->>Y] , Z[father->Y].
X[ancestor->>Y] :- X[father->Y].
X[ancestor->>Y] :- X[ancestor->>Z[father->Y]].
% equivalent rule with path expressions:
% X[ancestor->>Y] :- X..ancestor[father->Y].

?- sys.eval[].

/* QUERY */
?- X[cousin->>Y].
?- X[ancestor->>Y].
```

This example models a (somewhat patriarchalic) family tree. The rules define the methods `son`, `brother`, `uncle`, `nephew`, `cousin` and `ancestor` in terms of the given method `father`. The negation in the `brother` rule is not critical because no user defined method or predicate, but only a built-in predicate is involved.

Variables only occurring in the rule body may often be avoided by using path expressions, as the alternative rules for `brother` and `uncle` show[19]. Replacing a variable in the rule head by a path expression, however, may alter the semantics. For example, if we change the `son` rule into

```
Y.father[son->>Y] :- Y:person.
```

a new object `henry.father` would be created.

12.2 Generic Methods

```
%% generic_methods.flp
%% Demonstrates a more complex use of path expressions

%% calculate transitive closure of a method M

X[(M.tc) ->> Y] :- X[(M:basicmethod)->Y].
X[(M.tc) ->> Y] :- X..(M.tc)[(M:basicmethod)->Y].

%% define the inverse of a method M

X[(M.inv) ->>Y] :- Y[(M:basicmethod)->X].

%% same generation problem for a method M

X[(M.sg) ->>X] :- X:object, M:basicmethod.
```

[19]In most cases, keeping the number of subgoals small will lead to a faster evaluation because the additional information from the complex molecules can be used in the matching algorithm.

```
X[(M.sg) ->>Y] :- X.(M:basicmethod)..(M.sg)[(M.inv)->>Y].

%% apply this to family.flp:

father:basicmethod.
person::object.

?- sys.consult@("family.flp")[].

?- X[(father.tc) ->> Y].
?- X[(father.inv) ->> Y].
?- X[(father.sg) ->> Y].
```

Here a similiar functionality as in `family.flp` is implemented in a more involved but general way. The variable `M` is bound to every instance of the class `basicmethod` and its transitive closure, inverse and same generation relation is calculated. Here, it is applied to the method `father` of the preceding example[20]. The new method `(father.tc)` is just the same as `anc`, and `(father.inv)` corresponds to `son`. However, the relation expressed by `(father.sg)` denotes a proper superset of the relation defined by `cousin` since it also contains the brother relation and the identity relation.

Note: The way F-logic handles variables ranging over methods is similar to HILOG [CKW93]. Note also that path expressions in rule heads may cause the creation of new objects (e.g. `father.tc`). As mentioned above, the names of such objects are path expressions.

12.3 Using Equality to Ensure Finiteness

```
%% cycles.flp
%% Eliminate cycles from pathes by equating

/* Schema */

edge[start=>node; end=>node].

path[start=>node; end=>node;
     add@(edge)=>path;
     add@(path)=>path].

edge::path.
cycle::path.

/* Facts */

e7:edge[start->n1; end->n2]. %% This is a simple 3-cycle
e8:edge[start->n2; end->n3].
e9:edge[start->n3; end->n1].
```

[20]The consult command assumes that FLORID is started from a directory containing `family.flp`

```
/* RULES */

%% add edges to paths
P.E:path[start->X; end->Z] :-
          P:path[start->X; end->Y],
          E:edge[start->Y; end->Z].

%% concatenate paths
P1[(P2.E)->P3]:-
       P1.P2[E->P3], E:edge.

%% detect cycles
P:cycle :- P:path[start->P.end].

%% eliminate cycles by equating
P.C = P :- P.(C:cycle)[].

?- sys.eval[].

/* Query */
?- P:path.
```

When scanning a graph that possibly has cycles, it is often necessary to avoid situations like in example (9.10) to keep the model finite. In such cases, the use of equating can be helpful. In the example above, the three edges `e7,e8,e9` form a cycle. The first two rules define the possible paths in this graph as usual. Due to object creation, this rules alone would not yield a finite model because cycling generates arbitrary long paths. The last two rules detect cycles in the EDB graph and equate paths that differ only in the number of cycles. This means that the path `e7.e8.e9.e7` is only a different (and longer) name for the object denoted by `e7` and thus the fixpoint remains finite.

12.4 Unintended Equality

```
%% blasphemy.flp
%% Example for side effects of equality in
%% combination with scalar methods

Z[arrow@(X)->Y]:- Z[X->Y].   %% This seems to be independent of the
Z[arrow@(X)->Y]:- Z[X*->Y].  %% rest of the program

creation[createdBy*->god]. %% God created everything
man::creation.
male::man.
female::man.
adam:male.
eve:female.
```

```
%% But the devil sent a snake...

snake::creation[createdBy->devil].

%% It turned man into a creation of the devil:

?- sys.eval[].

?- adam[createdBy->devil].
```

This example shows that an implicite object equating may have unexpected side effects to the rest of the program. The crucial point is that for the class `snake` the method `createdBy` is overloaded with respect to the inheritability. This means that the invocation of the method `createdBy` to the object `snake` yields two different results: Since `snake` is a subclass of `creation` it inherits the method `createdBy` which *remains inheritable* and results in `god`. On the other hand the method `createdBy` is directly defined as a *non-inheritable* method yielding `devil`.

The `arrow` rules are apparently independent of the rest of the program[21] but in fact they derive the equality `god = devil`. The first rule defining the method `arrow` derives the fact "`snake[arrow@(createdBy)->devil].`", the fact "`snake[arrow@(createdBy)->god].`" is derived by the second rule. Since the method `arrow` is functional and it is applied to the same object with the same parameter the result objects `god` and `devil` are equated. We can eliminate this effect by using a multi-valued method `arrow` instead.

Note: See also the section about debug modi in the user manual.

12.5 Negation and Stratification

```
%% bachelor.flp
%% User stratification is needed to achieve the
%% intended negation semantics

/*
    john --- mary
          |
      ---------
      |       |
    tina    peter --- sally
                    |
              -----------
              |         |
            tim       jack
*/

john : person[spouse->mary].
```

[21] Initially the aim of these rules was to collect all methods defined for an object, whether inheritable or non-inheritable, so that they could be queried in a uniform way. The case that *both* are defined was not considered.

```
peter : person[father->john; mother->mary; spouse->sally:person].
mary : person.
tina : person[father->john; mother->mary].
tim : person[father->peter; mother->sally].
jack : person[father->peter; mother->sally].

X[spouse->Y] :- Y[spouse->X].
X : married :- X[spouse->Y].

X[par->>Y] :- X[father->Y].
X[par->>Y] :- X[mother->Y].

X[anc->>Y] :- X[par->>Y].
X[anc->>Y] :- X..anc[par->>Y].

married :: person.
bachelor :: person.

?- sys.strat.doIt[].

X : bachelor :- X : person, not X : married.

?- sys.eval[].

?- X : bachelor.
```

To use negation on user-defined information (i.e. information that is not part of the built-in concepts) you have to specify a stratification of the program. This is done by inserting the system command "`?- sys.strat.doIt[].`" between the strata. If all facts that may match the negated subgoal are derived in previous strata, not in the same stratum or higher strata, the program will yield the intended answer. In the example above, the class `married` is established in the first stratum and used negatively in the second one.

Note: Due to the expressive power of F-logic, no sufficiently large class of stratified programs exists which is statically decidable. So additional information has to be supplied by the user. It is topic of current research to provide this in a declarative way instead of the nondeclarative one described above.

12.6 Subset relationship

```
%% set_comparison.flp
%% Example for subset relationship of multi-valued methods
abraham[son->>{ishmael, isaac}].
sarah[son->>{isaac}].

% Is the set of Abraham's sons a subset of Sarah's sons?
?- sys.strat.doIt[].
nosubset(abraham,son,sarah,son) :- abraham[son->>Z], not sarah[son->>Z].
```

```
?- sys.strat.doIt[].
?- not nosubset(abraham,son,sarah,son).
```

Although variables in F-logic can only range over objects, not over sets, set operations may be expressed by rules. This example asks for the subset relationship of two sets given as the result of a multi-valued method application. The query whether the set of Abraham's sons is a subset of Sarah's sons (the answer is *false*) is evaluated in two steps. First, it has to be inferred whether there are any sons of Abraham who are not sons of Sarah, then the negation of this answer is taken as result.

Note the difference to the query

```
?- sarah[son->>abraham..son].
```

from example (7.15)

```
% Is the set of X's sons a subset of Y's friends?
?- sys.strat.doIt[].
nosubset(X,son,Y,friend) :- X[son->>Z], Y[friend->>{}], not Y[friend->>Z].
?- sys.strat.doIt[].
?- X[son->>{}], Y[friend->>{}], not nosubset(X,son,Y,friend).
```

All F-atoms containing the empty set as result are only needed for safety reasons to limit the variables occurring in the negated subgoal (see section 9).

12.7 Inheritance and Negation

```
%% jonathan.flp
%% negation together with inheritance triggering causes problems

bird::animal.
bird[can*->fly].
jonathan:bird.

?- sys.strat.doIt[].

X[can->swim] :- X:animal, not X[can->fly].

?- sys.eval[].

?- jonathan[can->fly].
?- jonathan[can->swim].
```

In this example the negation does not work as intended. The outcome of the program is that jonathan can swim but cannot fly. What happens is that in the first stratum the (active) trigger that makes jonathan inherit the method `can` from the class `bird` is not fired because there is no subgoal in this stratum activating this trigger. So in the second stratum `jonathan[can->swim]` is inferred and when asking the query the inheritance trigger is not active anymore. To avoid this we have to enforce inheritance triggering with a dummy rule in the first stratum:

```
bird::animal.
bird[can*->fly].
jonathan:bird.

dummy:- X[can->Y].    % dummy rule to enforce triggering

?- sys.strat.doIt[].

X[can->swim] :- X:animal, not X[can->fly].
```

This version will yield the correct result.

Note: Negation combined with inheritance triggering is a major problem. See also appendix B in [KLW95] for a theoretical approach.

12.8 Negation with Inflationary Semantics

```
%% adulterer.flp
%% Inflationary semantics of negation yields unsupported model
%% Here, the problem is caused by crosswise negation and equating

john[spouse->jane].
john[kissed->peter.sister].
john:adulterer:-john[kissed->Woman], not john[spouse->Woman].
john[spouse->Woman]:-john[kissed->Woman], not john:adulterer.

?- sys.eval[].
?- john:adulterer.
?- john[kissed->X].
?- peter[sister->Y].
```

The first evaluation round derives the facts that John is married to Jane and kissed Peter's sister. In the next round it is deduced that John is an adulterer because Jane is not known to be Peter's sister. At the same time it follows from the second rule that John is married to Peter's sister because John is not known to be an adulterer. The scalarity of the method `spouse` forces the equating of the two objects `jane` and "`peter.sister`".

In a third round none of the rules fires and the inferred facts are not removed from the object base. So John is an adulterer because he kissed his wife.

Note: This behavior is due to the pure inflationary treatment of negation and only avoidable by stratification. However, this program is not stratified because the corresponding dependency graph is cyclic [Ull89]. Another solution of this problem is be the application of the well-founded semantics [VGRS91]. See example 12.10.

12.9 Speed up Inheritance

```
%% tones.flp
%% Calculates the intervals between tones within one octave
%% Heavy use of inheritance triggering
```

```
%% base tones

a:tone[number->-1;chro->-2].
b:tone[number->0;chro->0].
c:tone[number->1;chro->1].
d:tone[number->2;chro->3].
e:tone[number->3;chro->5].
f:tone[number->4;chro->6].
g:tone[number->5;chro->8].

tone::note.

%% notes are derived from base tones by sharpening or flattening

sharp(X):note[number->X.number;chro->Y]:- Y=X:tone.chro + 1.
flat(X):note[number->X.number;chro->Y]:- Y=X:tone.chro - 1.

%% base intervals are defined by the distance between base tones

prime::interval[diff*->0].
second::interval[diff*->1].
third::interval[diff*->2].
fourth::interval[diff*->3].
fifth::interval[diff*->4].
sixth::interval[diff*->5].
seventh::interval[diff*->6].

%% there are derivates of base intervals with different real distance

plainPrime:prime[chrodiff->0].
augPrime:prime[chrodiff->1].

dimSecond:second[chrodiff->0].
smallSecond:second[chrodiff->1].
bigSecond:second[chrodiff->2].
augSecond:second[chrodiff->3].

dimThird:third[chrodiff->2].
smallThird:third[chrodiff->3].
bigThird:third[chrodiff->4].
augThird:third[chrodiff->5].

dimFourth:fourth[chrodiff->4].
plainFourth:fourth[chrodiff->5].
augFourth:fourth[chrodiff->6].

plainFifth:fifth[chrodiff->7].
```

```
dimFifth:fifth[chrodiff->6].
augFifth:fifth[chrodiff->8].

dimSixth:sixth[chrodiff->7].
smallSixth:sixth[chrodiff->8].
bigSixth:sixth[chrodiff->9].
augSixth:sixth[chrodiff->10].

dimSeventh:seventh[chrodiff->9].
smallSeventh:seventh[chrodiff->10].
bigSeventh:seventh[chrodiff->11].

%% (insert optimization here)

%% Two notes give an interval
X[to@(Y)->Z] :- X:note,Y:note,
                V=Y.chro-X.chro, U=Y.number - X.number,
                Z:interval[diff->U;chrodiff->V].

?- sys.eval[].

?- X[to@(Y)->Z].
```

There is no semantic problem here. The program works fine but is quite slow. The reason is that all intervals inherit the attribute `diff` from their respective base intervals. The trigger concept of F-logic requires that these triggers fire one at a time with computing a complete fixpoint afterwards. So 24 fixpoints are needed for this simple program and the time consuming last rule is evaluated over and over.

The first approach to an optimization is to separate the interval definitions from the last rule by stratification (inserting "`?- sys.strat.doIt[].`" before the last rule). Unfortunately this does not change anything because FLORID only fires a trigger when it is enabled by a subgoal[22]. However, as there is no appropriate subgoal no inheritance trigger fires in the first stratum. In the second stratum firing one trigger and computing a new fixpoint alternates, just as it was the case without stratification.

The solution is that all interesting triggers are enforced to fire already in the first stratum which can be achieved by a dummy rule:

```
dummy:-X:interval[diff->Z]. %% dummy rule forcing inheritance triggering
?- sys.strat.doIt[].        %% user stratification
```

The evaluation is sped up with this optimization approximately by factor 20.

Note: Version 2.0 of FLORID now contains an adoption of the semi-naive evaluation strategy. This also speeds up inheritance triggering, but it turned out that the effect of the dummy rule is by far greater. However, combining the dummy rule with semi-naive evaluation is even better and yields the best results.

[22] This is sort of a top-down element in the evaluation algorithm used.

12.10 Well-founded Semantics

```
%% wellfounded.flp
%% uses F-Logic triggers to emulate well-founded semantics

%---------  CONTROL RULES FOR WELL-FOUNDED DATALOG -------------

% inherit 'fixpoint'  after fixpoint is reached:
state[fixpoint *-> t].

% initial state:
0:state.

% after fixpoint of an active state [S] is reached, create [S+1]:
s(S):state :- S:active.fixpoint[].

% create all even states (while [S] is active)
0:even.
s(s(S)):even :- (S:state):even.

% [0] is active.
% [S+1] is active if [S] is an active even state.
% [S+2] is active if [S] is even and new tuples have been computed.
0:active.
s(S):active :- (S:active):even.
s(s(S)):active :- S:even, win(s(s(S)),X), not win(S,X).

% the first inactive even state is the final state
S:final :- (S:even).fixpoint[], not S:active.

% compute the final outcome:
%       everything in the (even) final state is true
%       everything in the previous but not in the final state is undefined
win_true(X) :- S:final, win(S,X).
win_undef(X) :- s(S):final, win(S,X), not win(s(S),X).


%------------ USER-DEFINED WELL-FOUNDED RULES ------------------------------

% EDB
move(a,b).
move(b,a).
move(b,c).
move(c,d).
move(d,e).
move(e,f).
```

```
% IDB:
% for negated subgoals use previous state!

win(s(S),X) :- s(S):state, move(X,Y), not win(S,Y).

//?- sys.theEval.debugOn@(cout)[].
?- sys.eval[].
?- win_true(X).
```

In this example, the procedural part of the inheritance definition via triggers is used to emulate the well-founded semantics for datalog. This serves as a wrap for any datalog program, here the prototypic example for the well-founded semantics, the win-move program is used. The method carries over to all F-logic programs that do not use inheritance. Programs with inheritance can be handled, too, if the control rules generating the next state are the last ones in the program. FLORID always fires the first active trigger found in the program. Thus, the next state is only generated when there is no more active trigger in the target program for the actual state.

13 Querying the World Wide Web with F-logic

With the growing importance of the World Wide Web, the problem of integrating database and Web technology has gained a lot of interest. It seems quite natural to have an abstract view of the Web as a labeled directed graph where the nodes represent documents and the labeled edges are provided by the hyperlinks. In contrast to SQL, languages for deductive databases are able to examine paths of arbitrary length in a graph (see also examples in section 12 for graph traversal). As the Web structure is unknown beforehand and can be quite involved, this is an important feature for Web access.

In this section, an integration of Web access into F-logic is presented which allows to explore, query, and restructure Web information, combining Web data and the local database in a uniform way. A full description of this approach, especially concerning the necessary extension of the semantics of F-logic, is found in [HLLS97].

13.1 Modeling the Web in F-logic

Every resource available in the Web has a unique address, called *Uniform Resource Locator* (URL). URLs can be used to initiate Web access. When a Web document is fetched, FLORID associates it with an object name derived from its URL by exploiting the object creation feature of scalar path expressions (see section 7). There is a predefined class url containing Web address strings and a predefined method get that, when applied to a member of url, yields the corresponding Web document. The schema for Web access is:

```
url::string[get ⇒webdoc].
```

Whenever for an instance u of class url the object u.get is created by a rule head, the document at the address u is automatically loaded and analyzed. Thus, loading Web documents is completely data-driven. New documents are fetched depending on information and links (i.e. URLs) found in already known documents. The requested documents can be specified by rules.

As a first example assume that `"http://www.informatik.uni-freiburg.de/~dbis/"` denotes an instance of class url. The object representing the corresponding document is created by the fact `"http://www.informatik.uni-freiburg.de/~dbis/".get[].`

Whenever the method get is defined for an instance u of class url, u.get denotes the corresponding Web document. Every fetched document has a set of predefined properties. The basic structure is shown by the signature of the class webdoc:

```
webdoc[self ⇒url; author ⇒string; modif ⇒string;
        type ⇒string; hrefs@(string) ⇒⇒url; error ⇒⇒string].
```

In the schema above, self denotes the URL, author the owner, modif the time of last modification, and type the type of the document. The method hrefs, parametrized with the link labels, returns the links in the document. The result type of hrefs is the class url.

If the Web access fails, error returns an explanation (e.g. *server does not exist*, *page not found*, *connection timed out*). In this case the other methods may be undefined. Error messages are also generated if the document can't be processed, e.g. in case of postscript files. In addition to these methods, the user can define other methods by rules.

13.2 Querying and Restructuring the Web

We present some examples to illustrate the practical usage of these concepts.[23] The first example deals with graph traversal of a limited area of the Web, starting at the homepage of our research group.

```
"http://www.informatik.uni-freiburg.de/~dbis/" = ourUrl.
ourUrl:url.
ourServer = "www.informatik.uni-freiburg.de/".                          (13.1)
X.get:ourPage :- X:ourUrl.
Y.get:ourPage :- X:ourPage[hrefs@(_) ->> Y], substr(ourServer,Y).
```

The facts in the first two lines declare the URL of our homepage to be a member of the class `url` and its subclass `ourUrl`. The third fact defines a shorthand for the hostname of our server, which is later used to restrict the graph traversal. Then the definition of the method `get` in the fourth fact implicitly triggers loading the document under `ourUrl` and puts it into the classe `ourPage`. The rule in line six resembles the well-known transitive closure example and fetches all pages from our server which are reachable by following links from our homepage. Rule one starts the search by requesting our homepage. The second rule finds the referenced URLs (in `Y`), and constrains them to those which reside on our server (`substr(...)`). As our server is not left, the search is limited and will terminate. Note that it would be a bad idea to omit the subclass `ourUrl` in the first fact and first rule, and to use the class `url` directly in the first rule:

```
X.get:ourPage :- X:url.                                                 (13.2)
```

As all URLs occurring in fetched pages are automatically members of class `url`, such a rule easily leads to non-termination.

After evaluating the program above, the properties of the loaded pages can be examined. In order to detect missing links or documents that are not processed, type

```
?- X:ourPage[error ->> Y].                                              (13.3)
```

If it is not known what properties are defined, you may use variables at method position, e.g:

```
?- X:ourPage[Y -> Z].
?- X:ourPage[Y ->> Z].                                                  (13.4)
```

In the context of example (13.1) it might be interesting to get all mail addresses found on the visited pages:

```
?- X:ourPage[hrefs@(_) ->> Y], substr("mailto:",Y).                     (13.5)
```

By the system command "`?- sys.prn.style@("html")[].`", a special output style is set that writes all query answers to an html file and calls a Netscape browser to display it. This has the advantage that URLs in the output are automatically tagged and highlighted, so they can be immediately examined with Netscape.

Another way to control graph traversals is to limit the search depth. Using the facts of example (13.1) again, all pages are visited that can be reached from the start in less than four steps:

[23]Before trying these examples, please check whether a Web version of FLORID runs on your system. For installation hints and special output or debug modes related to Web queries we refer to the manual.

```
X.get:ourPage[depth ->> 0] :- X:ourUrl.
Y.get:ourPage[depth ->> N] :- X:ourPage[depth ->> M;hrefs@(_)->>Y],   (13.6)
        N = M + 1, N < 4.
```

Note that the method `depth` is multi-valued because there are possibly different paths leading to the same document from the start. If these differ in length, `depth` may have more than one result value. In case of a scalar method different integer values would be equated, which leads to system error messages.

Sometimes it is useful to locate all Web pages which contain certain keywords. Then the search is normally started using search engines like Alta Vista or HotBot. Instead of browsing all hits by hand, we can use these search engines as a kind of index and let FLORID process the output.

```
"http://www.altavista.digital.com/cgi-bin/query?pg=q&text=yes
&what=web&kl=XX&q=logic&act=search":avurl.
avurl::url.
X.get:startpage :- X:avurl.
Y.get:keywordpage :- X:startpage[hrefs@(_) ->> Y],
        substr("www.altavista.digital.com",Y),
        substr("q=logic&stq=",Y).                                  (13.7)
Y.get:logicpage :- X:startpage[hrefs@(_) ->> Y],
        not substr("doubleclick",Y),
        not substr("www.altavista.digital.com",Y).
Y.get:logicpage :- X:keywordpage[hrefs@(_) ->> Y],
        not substr("doubleclick",Y),
        not substr("www.altavista.digital.com",Y).
```

The query to Alta Vista is given as instance of class `avurl`. The keyword "*logic*" appears as part `q=logic` of the URL. "`X.get:startpage`" represents the page returned by Alta Vista. Besides the URLs of the hits, the answer page of Alta Vista also contains other links that are not of interest here (e.g. advertisements). To avoid that these links are followed, the constraints "`not substr(...)`" are added. However, there are links to pages with more hits. These links have a special pattern "`q=logic&stq=`" in their URL. All pages containing the keyword "*logic*" are grouped into the class `logicpage`. In a further step they may be processed in greater detail.

In example (13.8), the pages of the DBPL (Databases & Logic Programming) server are used to get information about people who have published papers on database conferences. Search starts with the page that gives an overview of all listed database conference series. The initial rule has the same structure as in previous examples. As in example (13.7), the start page contains links that are not of interest. To avoid processing such links, the rule body contains three constraints. The first one ensures that the path to the conference pages is part of the links, whereas the second demands that the label string is part of the link's URL (which is true for links to database conferences on this server). The last one excludes the link to the directory of computer science journals.

In the second rule the conference series pages are processed. Given such a page, the the program page of all years should be visited. The corresponding links to these pages are labeled with "`contents`," or "`inhalt`" or "`program`". Additionally, the conference series name has to be part of the link. The `match` predicate is used to extract the year from the link. In the rule head, this information is added to the fetched document.

Finally, in the third rule, all documents representing a single database conference are processed. The labels of links referring to the author index are used as identifiers for authors in the rule head, except labels containing the string "author".

```
"http://www.informatik.uni-trier.de/~ley/db/conf/index.html":
dbconfurl.
"http://www.informatik.uni-trier.de/~ley/db/conf/" = dbconfpath.
"contents":usefullabel.
"inhalt":usefullabel.
"program":usefullabel.
dbconfurl::url.
X.get:dbconfpage :- X:dbconfurl.
Y.get:confseries[seriesname -> L] :-
        X:dbconfpage[hrefs@(L) ->> Y], substr(dbconfpath,Y),
        substr(L,Y), not substr("journals",L).
Y.get:conf(Y)[series@(YEAR)-> X] :-
        C:confseries[seriesname -> X;hrefs@(L) ->> Y],
        substr(USE:usefullabel,L), substr(X,Y),
        substr(dbconfpath,Y), match(Y,"[0-9][0-9]","",YEAR).
L:dbauthor[publishedAt@(YEAR) ->> X; authorpage ->> Y] :-
        ANY:conf(C)[series@(YEAR) -> X; hrefs@(L) ->> Y],
        substr("/indices/a-tree/",Y), not substr("author",L).
```
(13.8)

After evaluation of (13.8), the extracted information from different Web pages about conferences and authors is grouped into class `dbauthor`.

Since conference pages also contain links to editors of conference proceedings or panelists, class `dbauthor` may not only contain authors of papers. A distinction between authors and editors is not done here for simplicity.

Possible queries of interest are "*who has published on more than 3 different conferences in thes same year ?*", or "*who has published on VLDB and SIGMOD in the same year ?*".

```
?- 3 < countX[L,Y]; L:dbauthor[publishedAt@(Y) ->> X].
?- L:dbauthor[publishedAt@(X) ->> "VLDB","SIGMOD"].
```
(13.9)

Obviously navigation through certain parts of the Web as in example (13.8) requires more knowledge about the structure as the simple transitive closure example (13.1). Since most properties of Web documents are represented by strings, built-in predicates (see section 6.3) play an important role when dealing with Web queries.

The last example makes heavy use of Web access and built-in predicates in order to retrieve data from a Web server and to structure it. The swiss television (SF) maintains Web pages dedicated to weather information. There is information on temperature, wind, air pressure, and a general verbal description of the situation for several places in Switzerland and Europe. However, the data is distributed over several pages; e.g. to find air pressure and temperature for one place one has to search two pages. Within the pages data is organized as HTML-tables.

The following program retrieves the tables from the pages and recognizes the respective rows. To that aim regular expressions are used which specify identifying patterns. Next, from every row, object data is generated, again using regular expressions. Last, some queries show how the object base allows complex browsing not possible in the original pages.

```
% Web document handling
srcTempE =
  "http://www.sfdrs.ch/sendungen/meteo/karten/temperatur_europa_heute.html".
srcSituationE =
  "http://www.sfdrs.ch/sendungen/meteo/karten/wetter_europa_heute.html".
srcPressureE =
  "http://www.sfdrs.ch/sendungen/meteo/karten/luftdruck_europa_heute.html".
srcTempCh =
  "http://www.sfdrs.ch/sendungen/meteo/karten/temperatur_ch_heute.html".
srcSituationCh =
  "http://www.sfdrs.ch/sendungen/meteo/karten/wetter_ch_heute.html".
srcWindCh =
  "http://www.sfdrs.ch/sendungen/meteo/karten/wind_ch_heute.html".

srcTempE:source.
srcSituationE:source.
srcPressureE:source.
srcTempCh:source.
srcSituationCh:source.
srcWindCh:source.
source::url.

X.get[] :- X:source.

/* retrieve information ("wrapper") */

% get tables from the sources
T:tableTempE :-
  match(srcTempE.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).
T:tableSituationE :-
  match(srcSituationE.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).
T:tablePressureE :-
  match(srcPressureE.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).
T:tableTempCh :-
  match(srcTempCh.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).
T:tableSituationCh :-
  match(srcSituationCh.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).
T:tableWindCh :-
  match(srcWindCh.get,"</B><BR><BR></TD></TR>\(.\|\n\)*<TD COLSPAN","",T).

% get rows from the tables
R:rowTempE :- T:tableTempE,match(T,"<TR>\n.*\n.*\n</TR>","",R).
R:rowSituationE :- T:tableSituationE,match(T,"<TR>\n.*\n.*\n</TR>","",R).
R:rowPressureE :- T:tablePressureE,match(T,"<TR>\n.*\n.*\n</TR>","",R).
R:rowTempCh :- T:tableTempCh,match(T,"<TR>\n.*\n.*\n</TR>","",R).
R:rowSituationCh :- T:tableSituationCh,match(T,"<TR>\n.*\n.*\n</TR>","",R).
R:rowWindCh :- T:tableWindCh,match(T,"<TR>\n.*\n.*\n.*\n</TR>","",R).
```

```
/* building an object base from the rows ("mediator") */
europe::location.
ch::location.

% process rows, take care that patterns are unique within a row
C:europe[temp->T] :-
   R:rowTempE,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD ALIGN.*><.*>\(.*\)</FONT></TD>\n","\1",T).

C[pressure->P] :-
   R:rowPressureE,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD ALIGN.*><.*>\(.*\)</FONT></TD>\n","\1",P).

C[situation->S] :-
   R:rowSituationE,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD WIDTH.*><.*>\(.*\)</FONT></TD>\n","\1",S).

C:ch[temp->T] :-
   R:rowTempCh,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD ALIGN.*><.*>\(.*\)</FONT></TD>\n","\1",T).

C[situation->S] :-
   R:rowSituationCh,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD WIDTH.*><.*>\(.*\)</FONT></TD>\n","\1",S).

C:ch.wind[velocity->V;direction->D] :-
   R:rowWindCh,
   match(R,"<TD VALIGN.*><.*>\(.*\) </FONT></TD>\n","\1",C),
   match(R,"<TD ALIGN.*><.*>\(.*\)</FONT></TD>\n<TD","\1",D),
   match(R,"<TD ALIGN.*><.*>\(.*\)</FONT></TD>\n</TR>","\1",V).


?- sys.theOM.debugOn[].   // trace Web access
?- sys.eval[].

%?- sys.prn.style@("html")[].

?- C:location[temp->T;situation->S;pressure->P].
?- C:ch[temp->T].wind[velocity->V;direction->D].
?- L:ch.M[].
```

A Grammar of F-logic Syntax in Backus-Naur-Form

A.1 Lexical structure

Before defining how F-logic expressions are correctly built we describe the basic lexical structure of F-logic in figure 2:

RuleDelimiter = “.” Whitespace
Whitespace = *Blank*
| *Tabulator*
| *Return*
ISASymbol = “:” | “::”
ImplicationSymbol = “:-”
QuerySymbol = “?-”
MethodArrow1 = “->” | “*->”
MethodArrow2 = “->>” | “*->>”
MethodArrow3 = “=>” | “=>>”
Dot = “.” | “..” | “!” | “!!”
Predicate = *character string starting with a lower case letter*
BuiltInPredicate = “integer”
InfixBuiltInPred = “>” | “<” | “=” | “>=” | ” “<=”
BuiltInOperator = “+” | “-” | “*” | “/”
Functor = *character string starting with a lower case letter*
Variable = *character string starting with an upper case letter or “_”*
String = *character string enclosed in ” ”*
Integer = *integer*

Figure 2: Lexical structure

A.2 Grammar of F-logic syntax

Figure 3 shows the BNF grammar of the F-logic syntax. Expressions enclosed in curly braces are optional.

```
           Program  =  { ListOfRules }
       ListOfRules  =  Rule RuleDelimiter { ListOfRules }
              Rule  =  Head { ImplicationSymbol Body }
             Query  =  QuerySymbol Body RuleDelimiter
              Head  =  ListOfMolecules
   ListOfMolecules  =  Molecule { "," ListOfMolecules }
              Body  =  ListOfLiterals
    ListOfLiterals  =  Literal { "," ListOfLiterals }
           Literal  =  { "not" } Molecule
          Molecule  =  F-Molecule | P-Molecule

        P-Molecule  =  Predicate { "(" ListOfExpressions ")" }
                    |  BuiltInPredicate "(" ListOfExpressions ")"
                    |  ArithExpr InfixBuiltInPred ArithExpr
 ListOfExpressions  =  Expression { "," ListOfExpressions }
        Expression  |  PathExpression
                    |  F-Molecule
                    |  Aggregate
         ArithExpr  =  Expression
                    |  ArithExpr BuiltInOperator ArithExpr
                    |  "(" ArithExpr ")"
         Aggregate  =  ID-Term "{" Variable { "[" ListOfVars "]" }
                       ";" ListOfLiterals "}"
        F-Molecule  =  PathExpression Specification
    PathExpression  =  ID-Term
                    |  "(" Expression ")"
                    |  PathExpression Dot MethodApplication
                    |  F-Molecule Dot MethodApplication

     Specification  =  ISASpecification "[" { ListOfMethods } "]"
                    |  ISASpecification
                    |  "[" { ListOfMethods } "]"
  ISASpecification  =  ISASymbol ID-Term
                    |  ISASymbol "(" Expression ")"
 MethodApplication  =  ID-Term { "@(" ListOfExpressions ")" }
                    |  "(" Expression ")" { "@(" ListOfExpressions ")" }
     ListOfMethods  =  MethodApplication MethodResult { ";" ListOfMethods }
      MethodResult  =  MethodArrow1 Expression
                    |  MethodArrow2 Expression
                    |  MethodArrow2 "{" { ListOfExpressions } "}"
                    |  MethodArrow3 Expression
                    |  MethodArrow3 "(" { ListOfExpressions } ")"
           ID-Term  =  BasicID-Term
                    |  Functor "(" ListOfExpressions ")"
      BasicID-Term  =  Functor | Variable | String | Integer
        ListOfVars  =  Variable { "," ListOfVariables }
```

Figure 3: Grammar of F-logic syntax

B Syntax of Regular Expressions

The following description of regular expression was taken from a xemacs[24] info file. Regular expressions have a syntax in which a few characters are special constructs and the rest are "ordinary". An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`+`', '`?`', '`[`', '`]`' and '`\`'; no new special characters will be defined. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does not match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, you can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. To do something nontrivial, you need to use one of the following special characters:

- '`. (Period)`'
  is a special character that matches any single character except a newline. Using concatenation, you can make regular expressions like '`a.b`', which matches any three-character string which begins with '`a`' and ends with '`b`'.

- '`*`'
  is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches one '`f`' followed by any number of '`o`'s. The case of zero '`o`'s is allowed: '`fo*`' does match '`f`'.

  '`*`' always applies to the smallest possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'.

  The matcher processes a '`*`' construct by immediately matching as many repetitions as it can find. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

- '`+`'
  is a suffix character similar to '`*`' except that it requires that the preceding expression be matched at least once. For example, '`ca+r`' will match the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' would match all three strings.

- '`?`'
  is a suffix character similar to '`*`' except that it can match the preceding expression either once or not at all. For example, '`ca?r`' will match '`car`' or '`cr`'; nothing else.

[24] xemacs is free software under the GNU General Public License.

- ‘[...]’
  ‘[’ begins a ”character set”, which is terminated by a ‘]’. In the simplest case, the characters between the two form the set. Thus, ‘[ad]’ matches either one ‘a’ or one ‘d’, and ‘[ad]*’ matches any string composed of just ‘a’s and ‘d’s (including the empty string), from which it follows that ‘c[ad]*r’ matches ‘cr’, ‘car’, ‘cdr’, ‘caddaar’, etc.

  You can include character ranges in a character set by writing two characters with a ‘-’ between them. Thus, ‘[a-z]’ matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in ‘[a-z$%.]’, which matches any lower-case letter or ‘$’, ‘%’, or period.

  Note that inside a character set the usual special characters are not special any more. A completely different set of special characters exists inside character sets: ‘]’, ‘-’, and ‘^’.

  To include a ‘]’ in a character set, you must make it the first character. For example, ‘[]a]’ matches ‘]’ or ‘a’. To include a ‘-’, write ‘---’, which is a range containing only ‘-’. To include ‘^’, make it other than the first character in the set.

- ‘[^ ...]’
  ‘[^’ begins a ”complement character set”, which matches any character except the ones specified. Thus, ‘[^a-z0-9A-Z]’ matches all characters except letters and digits.

  ‘^’ is not special in a character set unless it is the first character. The character following the ‘^’ is treated as if it were first (‘-’ and ‘]’ are not special there).

  Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.

- ‘^’
  is a special character that matches the empty string, but only if at the beginning of a line in the text being matched. Otherwise, it fails to match anything. Thus, ‘^foo’ matches a ‘foo’ that occurs at the beginning of a line.

- ‘$’
  is similar to ‘^’ but matches only at the end of a line. Thus, ‘xx*$’ matches a string of one ‘x’ or more at the end of a line.

- ‘\’
  does two things: it quotes the special characters (including ‘\’), and it introduces additional special constructs.

  Because ‘\’ quotes special characters, ‘\$’ is a regular expression that matches only ‘$’, and ‘\[’ is a regular expression that matches only ‘[’, and so on.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, ‘*foo’ treats ‘*’ as ordinary since there is no preceding expression on which the ‘*’ can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

Usually, ‘\’ followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by ‘\’, are special constructs. Such characters are always ordinary when encountered on their own. Here is a table of '\' constructs.

- '\|'
  specifies an alternative. Two regular expressions A and B with '\|' in between form an expression that matches anything A or B matches.

  Thus, 'foo\|bar' matches either 'foo' or 'bar' but no other string.

  '\|' applies to the largest possible surrounding expressions. Only a surrounding '\(... \)' grouping can limit the grouping power of '\|'.

  Full backtracking capability exists to handle multiple uses of '\|'.

- '\(... \)'
  is a grouping construct that serves three purposes:

  1. To enclose a set of '\|' alternatives for other operations. Thus, '\(foo\|bar\)x' matches either 'foox' or 'barx'.

  2. To enclose a complicated expression for the postfix '*' to operate on. Thus, 'ba\(na\)*' matches 'bananana', etc., with any (zero or more) number of 'na' strings.

  3. To mark a matched substring for future reference.

  This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same '\(... \)' construct because in practice there is no conflict between the two meanings. Here is an explanation:

- '\DIGIT'
  after the end of a '\(... \)' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '\' followed by DIGIT to mean "match the same text matched the DIGIT'th time by the '\(... \)' construct."

  The strings matching the first nine '\(... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. '\1' through '\9' may be used to refer to the text matched by the corresponding '\(... \)' construct.

  For example, '\(.*\)\1' matches any newline-free string that is composed of two identical halves. The '\(.*\)' matches the first half, which may be anything, but the '\1' that follows must match the same exact text.

- '\`'
  matches the empty string, provided it is at the beginning of the buffer.

- '\''
  matches the empty string, provided it is at the end of the buffer.

- '\b'
  matches the empty string, provided it is at the beginning or end of a word. Thus, '\bfoo\b' matches any occurrence of 'foo' as a separate word. '\bballs?\b' matches 'ball' or 'balls' as a separate word.

- ‘\B’
  matches the empty string, provided it is not at the beginning or end of a word.
- ‘\<’
  matches the empty string, provided it is at the beginning of a word.
- ‘\>’
  matches the empty string, provided it is at the end of a word.
- ‘\w’
  matches any word-constituent character. The editor syntax table determines which characters these are.
- ‘\W’
  matches any character that is not a word-constituent.
- ‘\sCODE’
  matches any character whose syntax is CODE. CODE is a character which represents a syntax code: thus, ‘w’ for word constituent, ‘-’ for whitespace, ‘(’ for open-parenthesis, etc.
- ‘\SCODE’
  matches any character whose syntax is not CODE.

Here is a complicated regexp used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to enable you to distinguish the spaces from the tab characters. In Lisp syntax, the string constant begins and ends with a double-quote. ‘\"’ stands for a double-quote as part of the regexp, ‘\\’ for a backslash as part of the regexp, ‘\t’ for a tab and‘\n’ for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This regexp contains four parts: a character set matching period, ‘?’ or ‘!’; a character set matching close-brackets, quotes or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab or two spaces; and a character set matching whitespace characters, repeated any number of times.

References

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 40 – 57. North-Holland/Elsevier Science Publishers, 1989.

[ABW88] Krzysztof R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89 – 148. Morgan Kaufmann, 1988.

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

[Ban86] F. Bancilhon. Naive evaluation of recursively defined relations. In L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 165 – 178. Springer, 1986.

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

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[DOO97] *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, LNCS, Montreux, 1997. Springer.

[FLU94] Jürgen Frohn, Georg Lausen, and Heinz Uphoff. Access to objects by path expressions and rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. Intl. Conference on Very Large Data Bases*, Santiago de Chile, 1994.

[HLLS97] Rainer Himmeröder, Georg Lausen, Bertram Ludäscher, and Christian Schlepphorst. On a declarative semantics for web queries. In *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)* [DOO97].

[Kan97] Paul-Th. Kandzia. Nonmonotonic Reasoning in Florid. In *Proc. Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, number 1265 in LNAI. Springer, 1997.

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

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM Symposium on Principles of Database Systems*, pages 231–239, 1988.

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

[MLL97] Wolfgang May, Bertram Ludäscher, and Georg Lausen. Well-founded semantics for deductive object-oriented database languages. In *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)* [DOO97].

[MSL97] Wolfgang May, Christian Schlepphorst, and Georg Lausen. Integrating dynamic aspects into deductive object-oriented databases. In Andreas Geppert and Mikael Berndtsson, editors, *Proc. of the 3nd Intl. Workshop on Rules in Database Systems (RIDS)*, LNCS, Skövde, Sweden, 1997.

[NT89] S. Naqvi and S. Tsur. *A logical Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 191 – 216. Morgan Kaufmann, 1988.

[Ull89] Jeffrey 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.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620 – 650, July 1991.