FunSPARQL: Extension of SPARQL with a Functional Language

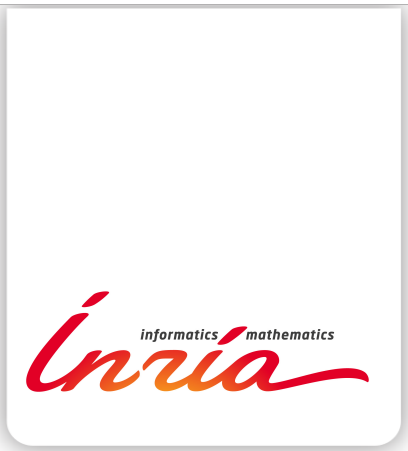
Olivier Corby, Catherine Faron-Zucker

FunSPARQL: Extension of SPARQL with a Functional Language

Olivier Corby, Catherine Faron-Zucker

RESEARCH REPORT
N° 8814
December 2015
Project-Team Wimmics

ISSN 0249-6399 ISRN INRIA/RR--8814--FR+ENG



FunSPARQL: Extension of SPARQL with a Functional Language

Olivier Corby*, Catherine Faron-Zucker†

Project-Team Wimmics

Research Report n° 8814 — December 2015 — 29 pages

Abstract: The SPARQL query language for RDF is the standard recommended by the W3C to query the Semantic Web. It enables to query and search RDF data by using query graph patterns and filters restricting the solutions produced by graph pattern matching according to the constraints they express. Constraints are either expressions defined according to the expression language specified in the W3C recommendation or function calls to SPARQL builtin functions or to extension functions. The use of extension functions in a SPARQL query, although compliant to the recommendation, limits the interoperability of the query. In this report we propose a lightweight extension of SPARQL to enable the *definition* of extension functions in the SPARQL filter language. We call it FunSPARQL, standing for Functional SPARQL, as it consists in providing SPARQL with a functional language for expressing constraints.

Key-words: Semantic Web, SPARQL, Functional Language

* Inria, I3S
† Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

FunSPARQL: Extension of SPARQL with a Functional Language

Résumé : SPARQL est le language de requête pour RDF recommandé par le W3C pour interroger le Web sémantique. Il permet d'interroger des graphes RDF avec des patterns de graphe et des filtres qui restreignent les solutions possibles à celles vérifiant des contraintes.

Les contraintes sont des expressions définies selon le language de filtre de SPARQL ainsi que des appels à des fonctions d'extension externes. Les fonctions d'extension, bien que conformes à la recommandation SPARQL, limitent l'interopérabilité car il n'y a pas de mécanisme d'extension standardisé.

Dans ce rapport nous proposons une légère extension qui permet de définir des fonctions dans le langage de filtre de SPARQL. Nous appelons cette extension FunSPARQL, pour SPARQL Fonctionnel, car elle propose un langage fonctionnel pour écrire des contraintes.

Mots-clés : Web sémantique, SPARQL, Langage Fonctionnel

1 Introduction

The SPARQL query language for RDF is the standard recommended by the W3C to query the Semantic Web [6]. It enables to query and search RDF data by using query graph patterns and filters restricting the solutions produced by graph pattern matching according to the constraints they express. Constraints are either (conjunctions or disjunctions of) relational expressions, or function calls to SPARQL builtin functions available for use or to extension functions defined by users.

As the Web of data grows, Semantic Web applications are facing the key problem of the quality and heterogeneity of data. As a first step of the management of RDF data crawled on the web of data or extracted from the deep web, this data should be cleaned up, data values with heterogeneous units and formats should be transformed according to XSD datatypes. The usual way to perform these transformations is to write SPARQL queries or SPARQL updates using special purpose extension functions that clean up the data. The development of the Web of data opens up a wide range of use cases which can be answered with user-defined extension functions. Among many others, extensions functions could be defined to translate decimal numbers into numbers in natural language and conversely numbers in natural language into decimal numbers; to convert binary numbers into decimal numbers and conversely decimal numbers into binary numbers; to perform approximate string comparison; etc.

In SPARQL, an extension function is named by a IRI, within a user defined extension namespaces; it takes RDF terms as arguments and returns an RDF term; it can be implemented in any programming language chosen by the developer, e.g. Java. Each SPARQL implementation has its own protocol to realize the binding between extension namespaces and function definitions; it is not standardized. As a result, the use of extension functions in a SPARQL query, although compliant to the recommendation, limits the interoperability of the queries written. Therefore, we adress the research question of *How can we enable the definition of SPARQL extension functions without limiting the interoperability of the SPARQL queries using them?*

To answer this question, we propose a lightweight extension of SPARQL filter expression language to enable the *definition* of extension functions, taking advantage of the fact that the SPARQL filter expression language potentially enables to express function definitions. As a result, we define a *functional programming language* on top of SPARQL filter expression language. Syntactically, it consists in additional statements: a `function` statement enabling the SPARQL developer to define functions and a `let` statement enabling him to introduce local variables in the filter language. We call our extension FunSPARQL, standing for Functional SPARQL, as it consists in providing SPARQL with a functional language for expressing constraints. We present the syntax and semantics of FunSPARQL as well as an implementation and we illustrate the simplicity as well as the expressive power of this extension with some examples of FunSPARQL extension functions.

This paper is organized as follows: In section 2 we present state-of-the-art approaches to define extension functions. In section 3 we present FunSPARQL. In section 4 we define the syntax and semantics of FunSPARQL. In section 5 we present several use cases of extension functions and we show how they can easily be addressed by using FunSPARQL. In section 6 we present our implementation of FunSPARQL within the Corese Semantic Web Factory. In section 7 we conclude and draw some perspectives of our work.

2 Related work

SPIN is a W3C member submission which proposes a SPARQL-based rule and constraint language and, additionally, enables to represent both SPARQL queries (SPIN templates) and SPARQL extension functions (SPIN functions) [8]. It uses RDF as a syntax. In SPIN, a SPARQL extension function is represented by a resource of type `sp:Function` (with `sp` the prefix denoting the SPIN namespace); it is related with property `sp:body` to its definition which consists in a SPARQL query of the form SELECT or ASK. For instance, here is the definition in SPIN of the `ex:cardinality` extension function (with `ex` the prefix of a user defined namespace). It returns the number of values of a given property (`sp:_arg1`) for the current subject (`sp:_this`).

```
ex:cardinality
  a sp:Function ;
  rdfs:label "cardinality"^^xsd:string ;
  sp:constraint [
    a spl:Argument ;
    spl:predicate sp:_arg1 ] ;
  sp:body [
    a sp:Select ;
      sp:resultVariables (
        [ a sp:Count ;
          sp:expression sp:_object ] ) ;
      sp:where (
      [ sp:subject   sp:_this  ;
        sp:predicate sp:_arg1 ;
        sp:object    sp:_object ] ) ] .
```

For instance, here is a SPARQL query using the extension function `ex:cardinality` to filter the solutions of the triple pattern match to those for which the value bound to the subject of the triple pattern is the subject of at least one RDF triple with `ex:child` as property:

```
SELECT ?p
WHERE {
  ?p ex:child ?c
  FILTER (ex:cardinality(ex:child) > 0)
```

```
}
```

Jena provides the `java` URI scheme for naming and accessing SPARQL extension functions implemented in Java. This enables to dynamically load the bytecode implementing the function. By convention, the location of the Java class must be found in the Java classpath and the local name of the function must be the name of the Java class implementing it[1]. Here is an exemple of a SPARQL query using the `f:myTest` extension function implemented in Java and enabling to filter the solutions of the triple pattern match to those for which a call to function `f:myTest` with the values bound to the subject and object of the triple pattern returns true:

```
PREFIX f: <java:app.myFunctions.>
SELECT ?x
WHERE {
  ?x ?p ?y
  FILTER f:myTest(?x, ?y)
}
```

A proposal to implement SPARQL extension functions in JavaScript as an agreed-upon programming language and to share implementations among query engines by using an embedded JavaScript interpreter is described in [9]. Functions are identified by URLs and the source code may be retrieved at run time by dereferencing the URL. It relies on an RDF schema enabling to describe a SPARQL extension function and retrieve its source code at run time by dereferencing its URL. Here is an example of RDF statements describing a SPARQL extension function to compute a geographical distance in kilometers. The location of its JavaScript source code is the value of property `ex:source` (with `ex` the namespace prefix of the extension function schema) and the function name in the source code that should be called to execute the extension function is the value of property `ex:function`.

```
<http://example.com/functions/distance>
  a ex:Function;
  dc:description "Geographic distance in km";
  ex:source <http://example.com/distance.js>;
  ex:function "gdistance" .
```

A proposal to implement SPARQL extension functions based on both a generic extension function `wfn:call` and the SPARQL SERVICE clause is described in [1]. Function `wfn:call` is similar to the Lisp *funcall* function and takes the extension function to be evaluated as its first argument. Any occurence of the `wfn:call` function is replaced by a SERVICE call to delegate the evaluation of the extension function to the SPARQL endpoint implementing the function. The SPARQL endpoint IRI is computed from the extension function IRI based on a Function-to-Endpoint IRI pattern. For instance, to execute the following query:

[1] `https://jena.apache.org/documentation/query/writing_functions.html`

```
SELECT ?x
WHERE {
  ?x ?p ?y
  FILTER wfn:call(ex:myTest, ?x, ?y)
}
```

the compiler will change the FILTER line into the following SERVICE clause:

```
SERVICE ex:sparql {
  BIND (ex:myTest(?x, ?y) as ?res)
}
FILTER (?res)
```

When compared to these four state-of-the-art proposals to answer the question of the interoperability of SPARQL extension functions, the key idea of our proposal described in the following, is to extend the SPARQL language, and more precisely its constraint language, in order to enable the definition of extension function in the SPARQL language *itself*.

3 Definition of FunSPARQL

A SPARQL filter restricts the solutions of a graph pattern match to those satisfying the constraint it expresses: the filtered solutions result in an effective boolean value of true when substituted into the filter expression. A SPARQL filter is either (a disjunction or conjunction of) a relational expression or a call to a built-in or externally defined boolean function. Among the built-in SPARQL functions stands the IF ternary function which evaluates the first argument and returns the value of the second arguments if the first argument results in an effective value of true, or else the value of the third argument. Starting from the SPARQL filter expression language, we propose to extend it to define a *functional programming language* on top of it, taking advantage of the fact that the SPARQL filter language enables to define expressions.

We use the prefix and namespaces below.

```
prefix xt: <http://ns.inria.fr/sparql-extension/>
prefix us: <http://ns.inria.fr/sparql-extension/user/>
prefix rq: <http://ns.inria.fr/sparql-function/>
prefix dt: <http://ns.inria.fr/sparql-datatype/>
prefix ex: <htp://example.org/>
```

3.1 Function Definition

In FunSPARQL, we introduce *function definition* as an additional statement of the language. The first argument of the declaration is the name (the URI) of the function being defined followed by its argument list. The variables in the argument list play the usual role of function arguments. The second argument

is the body of the function being defined. It is a FunSPARQL expression or a sequence of expressions. For example, the *factorial* function `us:fac` is defined as follows, by using the SPARQL IF built-in function and embedding a recursive call:

```
function us:fac(?n) {
  IF (?n = 0, 1, ?n * us:fac(?n - 1))
}
```

Here is another example of function definition. The `us:compare` binary function compares two values given as parameters. Its definition uses a call to the IF SPARQL statement .

```
function us:compare(?x, ?y) {
   IF (?x < ?y, -1,
     IF (?x = ?y, 0, 1))
}
```

Here is a third example of function definition. A call to the `us:status` function return the status of the resource given as parameter. Its definition uses the SPARQL built-in statements IF and EXISTS.

```
function us:status(?x) {
  IF (EXISTS { ?x a foaf:Person },
     ex:Human, ex:Thing)
}
```

A call to a defined function returns the result of the evaluation of its body, with its arguments bound by the function call. In the body, the arguments are *local* variables in the sense that the variable bindings are local to the body of the function and exist only during the execution of the function. For instance, according to its above definition, a call to function `us:fac` will return the value returned by a call to the IF SPARQL built-in statement with a given value for variable `?n`.

The language for defining the body of a function is FunSPARQL, i.e. the SPARQL filter expression language extended with statements presented in this document. Hence, to define extension functions, FunSPARQL programmers can make use of the expressivity of the whole SPARQL filter expression language, including built-in SPARQL functions, among which IF statement enabling to consider alternatives, and the usefull EXISTS statement; they can call extension functions in a function definition, and define extension functions.

The `function` statement triggers the storage of the declared function in a table together with the number of its arguments. The same name can be used to declare different functions with different numbers of arguments. Later on, this table enables the FunSPARQL interpreter to retrieve the function definition at the time of a function call.

The scope of a function definition is the SPARQL query where it is declared. In a SPARQL query, a function definition is stated after the end of the query. For

instance the following SPARQL query embeds the definition of function `us:fac` and a call to this function. It searches resources whose income is greater or equal to 10! = 3,628,800.

```
SELECT ?x ?i
WHERE {
  ?x ex:income ?i
  FILTER (?i >= us:fac(10))
}

function us:fac(?n) {
   if (?n = 0, 1, ?n * us:fac(?n - 1))
}
```

3.2 Function Export

To enable the reuse of extension functions in SPARQL queries, we introduce *function export* as an additional FunSPARQL statement. It takes as argument one or several function definitions, each one defining an extension function. For instance, the following query embeds the export of two function definitions: *factorial* and *fibonacci*.

```
SELECT *
WHERE {
}
export {
  function us:fac(?n) {
    IF (?n = 0, 1, ?n * us:fac(?n - 1))
  }

  function us:fib(?n) {
    IF (?n <= 2, 1, us:fib(?n - 2) + us:fib(?n - 1))
  }
}
```

When defined within an export, extension functions are exported in the runtime environment of the SPARQL interpreter and can be reused in other SPARQL queries during the current session. For instance, after the above query is executed, the query below can reuse the defined function `us:fac`.

```
SELECT *
WHERE {
  ?x ex:income ?i
  FILTER (?i >= us:fac(10))
}
```

When the session resumes, function definitions vanish from the runtime environment.

3.3 Local Variable Declaration

We introduce *local variable declarations* as an additional type of FunSPARQL expressions: a local variable declaration is a call to the `let` statement. It takes as first argument an equality term which first argument is the name of the local variable to be declared and which second argument is an expression which value is bound to the variable. The second argument of a `let` statement is an expression which is evaluated with the transient binding of the local variable. The local variable binding only exists during the evaluation of the second argument of the *let* statement and vanishes after it completes. A *let* statements evaluates to the value of its second argument.

In addition to function definition, the `let` statement can also be used within any SPARQL expression (SELECT, FILTER, BIND, HAVING, etc. clauses). The SPARQL query below embeds a call to `let` statement in its SELECT clause. It returns a binding of variable `?date` with a string pretty-printing the date of the day, e.g. "14/10/2015".

```
SELECT
(let (?n = now()){
   CONCAT(day(?n), "/", month(?n), "/", year(?n))
 }
AS ?date)
WHERE { }
```

Several bindings with `let` statement can be nested, enabling sequential evaluation with variable bindings, in functional programming style. Here is an example of such nested calls within the SELECT clause of a SPARQL query returning the same result as the preceding query.

```
SELECT (
  let (?n = now(),     ?d = day(?n),
       ?m = month(?n), ?y = year(?n)){
    CONCAT(?d, "/", ?m, "/", ?y)
  }
AS ?date)
WHERE { }
```

The `let` statement can also evaluate a SPARQL query as expression and bind a list of variables with the (first) result of the query. The binding of the variables of the `let` with the variables of the `select` is done by name.

```
function us:type(?s){
  let ((?t) = select * where { ?s a ?t }){
    ?t
  }
}
```

3.4 Loop

In order to iterate a statement on the elements of a list of values, we introduce the `for` loop statement.

```
for (?n in xt:list(1, 2, 3, 4, 5)){
  if (us:prime(?n)){
    xt:display(?n)
  }
}
```

The `for` statement can iterate on the result of a SPARQL `select` or `construct` query. In case of `construct`, it iterates on the triples of the graph.

```
for (?t in construct where { ?x a foaf:Person }){
  xt:display(?t)
}
```

The `for` statement can bind a list of variables according to the results of the expression. If the result elements are triples, the `for` statement can bind the subject, property and object of each triple as a list of variables.

```
for ((?s, ?p, ?o) in construct where { ?x a foaf:Person }){
  xt:display(?s, ?o)
}
```

3.5 Eval

The `eval` function enables users to call a function whose name is the result of the evaluation of an expression.

```
let (?fun = if (rand() > 0.5, us:foo, us:bar)){
  eval(?fun, ?x)
}
```

3.6 Apply

The `apply` function enables users to apply a binary function to a list of arguments. For example, it enables to compute the sum of the elements of a list of numbers given the binary `rq:plus` function.

```
apply(rq:plus, xt:list(1, 2, 3, 4, 5))
```

3.7 The List Datatype

In order to leverage the functional language, we introduce a `dt:list` datatype to manage lists of values and the `maplist` function to apply a function to the elements of a list. A `dt:list` datatype value is a list whose elements are RDF terms: URI, Literal or Blank Node. The elements of a list need not to be of the same kind, neither of the same datatype.

The `dt:list` datatype is provided with a set of predefined functions among which `xt:size` returns the size of the list, `xt:get` returns the nth element, `xt:get` sorts the list according to the `order by` rules of SPARQL, `xt:iota` returns the list of n first integers, `xt:cons` adds an element to the head of the list, etc.

The `maplist` function enables to apply a function to the elements of a list and return the list of the results. For instance, the call to function `maplist` shown below returns the list of the results of the calls to function `us:fac` on the first ten integers.

```
maplist(us:fac, xt:iota(10))
```

We define an `unnest` statement to enumerate the elements of a list in a BIND clause of a SPARQL query. For instance, the SPARQL query below returns the values of the first four integers.

```
SELECT ?f
WHERE {
 BIND (unnest(xt:iota(4)) AS ?f)
}
```

More precisely, a call to the `unnest` function in a BIND clause generates a solution sequence S with one solution for each value of the list and performs a join of S with the result of the evaluation of the rest of the BGP. For instance, the above query returns the same solution than the following SPARQL query with a VALUES clause. In other terms, `unnest` emulates a VALUES clause with evaluable expressions.

```
SELECT ?f
WHERE {
 VALUES ?f {1 2 3 4}
}
```

3.8 Annotated and Linked Function

On the Web the core naming mechanism is the URIs. Applied to functions, URI do not only mean they can be universally identified it also means that they become subject to dereferenciation and annotation. Dereferenciation means one can go to that URI and discover the definition of a function thus enabling a distributed architecture for exchanging functions. Annotation means we can

attach metadata to the function (documentation, technical characteristics, inputs, outputs, certification, etc.) and we can link them to other resources (e.g. Datasets), execution and provenance traces (e.g. PROV-O annotations) and even other functions (e.g. specialization, versioning, composition compatibility, alternatives, etc.) This enables a Web of linked functions as shown below.

```
us:newTest a xt:Method, prov:SoftwareAgent;
  xt:name us:fun ;
  xt:input (foaf:Person) ;
  xt:output xsd:string ;
  rdfs:label "just an example for procedural attachement" ;
  us:extends us:test;
  eg:certifiedBy <http://www.inria.fr> ;
  dc:creator <http://ns.inria.fr/fabien.gandon#me> ;

:text890
   a prov:Entity;
   prov:wasGeneratedBy :computation998.

:computation998
   a prov:Activity;
   prov:startedAtTime    "2015-08-15T01:02:01Z"^^xsd:dateTime;
   prov:endedAtTime      "2015-08-15T01:02:02Z"^^xsd:dateTime;
   prov:wasAssociatedWith :us:newTest;
   prov:used              :Person828.
```

4 FunSPARQL Language

In this section we present the syntax and semantics of the FunSPARQL language.

4.1 FunSPARQL Syntax

FunSPARQL grammar is basically that of SPARQL[2]. The definition of `QueryUnit` is extended with `function` and `export` statements, `BuilInCall` is extended with `let`, `for`, `map`, `eval` and `apply` statements.

```
QueryUnit ::= Query FunDecl*

FunDecl ::=
  | Function
  | 'export' '{' Function + '}'

Function ::=
```

[2] http://www.w3.org/TR/sparql11-query/#grammar

```
  'function' iri ('()' | VarList) Body

Body ::= '{' '}' | '{' Expression (';' Expression)* '}'

VarList ::=  '(' Var (',' Var)* ')'

BuilInCall ::= SPARQL_BuiltInCall
  | 'let' '(' Decl (',' Decl) * ')'        Body
  | 'for' '(' VarOrList 'in' ExpQuery ')'  Body
  | Map   '(' iri ',' Expression ')'
  | 'eval' '(' Expression (',' Expression)* ')'
  | 'apply' '(' iri ',' Expression ')'

Decl      ::= VarOrList '=' ExpQuery
VarOrList ::= Var | VarList
ExpQuery  ::= Expression |
  SelectQuery | ConstructQuery | ServiceGraphPattern
Map ::= 'map' | 'maplist' | 'mapselect' | 'mapany' | 'mapevery'

Bind ::=  SPARQL_Bind
  | 'BIND' '(' 'unnest' '(' Expression ')' 'AS' Var ')'
```

4.2 FunSPARQL Semantics

We define the semantics of the core of FunSPARQL by a set of Natural Semantics inference rules [7]. These rules enable us to define the semantics of the evaluation of the expressions of the language in an environment with variable bindings. The bottom of the rule is the conclusion and the top is the condition. The $|-$ symbol states that the expression on the right side is evaluated in the environment on the left side. The $\rightarrow$ symbol represents the evaluation of the expression on the left side into the value on the right side. An environment is a couple (μ, ρ) where μ is the BGP solution mapping and ρ represents local variable bindings.

Rule 1 states that local variables are evaluated within ρ which is managed as a stack, latest variable binding first; rule 2 states that global variables are evaluated within μ which is a BGP solution.

Rules 3 & 4 specify the evaluation of function calls. The $\Rightarrow$ symbol represents a function definition lookup for the function name on the left side. The solution mapping environment is empty during function body evaluation: there are no global variables. Each function call creates a fresh environment with function parameters (if any) as local variables.

Rule 5 specifies the evaluation of the *let* clause which declares a local variable to be added to environment ρ. Hence, a declared local variable may hide a function parameter or a BGP variable. BGP variables are accessible in a *let* statement (e.g. in a filter), unless the *let* statement is inside a function, in which case the μ environment is empty.

Rule 6 specifies the evaluation of the *for* by evaluating the first expression

that returns a list of values and then binds the variable successively with each element of the list and evaluate the second expression with each local binding. The result of the *for* statement is always *true* by convention.

Rules 7, 8 and 9, 10, 11 specify *map*, *eval* and *apply* statements.

Rule 12 specifies the evaluation of a FunSPARQL expression. The semantics is that of standard SPARQL expression evaluation, except that the overall environment comprises an environment for local variables in addition to the standard environment for BGP variables.

$$\frac{}{\mu, \rho[x = v] \mid - \ x \ \rightarrow v} \tag{1}$$

$$\frac{x \notin \rho}{\mu[x = v], \rho \mid - \ x \ \rightarrow v} \tag{2}$$

$$\frac{f() \Rightarrow f() = body \ \wedge \ \phi, \phi \mid - \ body \ \rightarrow res}{\mu, \rho \mid - \ f() \ \rightarrow res} \tag{3}$$

$$\frac{\begin{array}{l} f(e_1, ... e_n) \Rightarrow f(x_1, ... x_n) = body \\ \mu, \rho \mid - \ e_1 \ \rightarrow v_1 \\ ... \\ \mu, \rho \mid - \ e_n \ \rightarrow v_n \\ \phi, [x_1 = v_1; ... x_n = v_n] \mid - \ body \ \rightarrow res \end{array}}{\mu, \rho \mid - \ f(e_1, ... e_n) \ \rightarrow res} \tag{4}$$

$$\frac{\mu, \rho \mid - \ e_1 \ \rightarrow v_1 \ \wedge \ \mu, \rho\,[x = v_1] \mid - \ e_2 \ \rightarrow res}{\mu, \rho \mid - \ let(x = e_1, \ e_2) \ \rightarrow res} \tag{5}$$

$$\frac{\begin{array}{l} \mu, \rho \mid - \ e \ \rightarrow (v_1, ... v_n) \\ \mu, \rho\,[x = v_1] \mid - \ b \ \rightarrow r_1 \\ ... \\ \mu, \rho\,[x = v_n] \mid - \ b \ \rightarrow r_n \end{array}}{\mu, \rho \mid - \ for(x = e, \ b) \ \rightarrow true} \tag{6}$$

$$\frac{\begin{array}{l} \mu, \rho \mid - \ e \ \rightarrow (v_1, .. v_n) \\ \mu, \rho \mid - \ f(v_1) \ \rightarrow r_1 \\ .. \\ \mu, \rho \mid - \ f(v_n) \ \rightarrow r_n \end{array}}{\mu, \rho \mid - \ map(f, e) \ \rightarrow true} \tag{7}$$

$$\frac{\mu, \rho \mid - \ e \ \rightarrow f \ \wedge \ \mu, \rho \mid - \ f(e_1, .. e_n) \ \rightarrow v}{\mu, \rho \mid - \ eval(e, e_1, .. e_n) \ \rightarrow v} \tag{8}$$

$$\frac{\begin{array}{l} \mu, \rho \mid - \ e \ \rightarrow (v_1, .. v_n) \\ \mu, \rho \mid - \ apply(f, (v_1, .. v_n)) \ \rightarrow v \end{array}}{\mu, \rho \mid - \ apply(f, e) \ \rightarrow v} \tag{9}$$

$$\frac{\mu, \rho \mid - \ f() \ \rightarrow v}{\mu, \rho \mid - \ apply(f, ()) \ \rightarrow v} \tag{10}$$

$$\frac{\begin{array}{l}\mu, \rho \mid - \ apply(f, (v_2, ..v_n)) \ \rightarrow r \\ \mu, \rho \mid - \ f(v_1, r) \ \rightarrow v\end{array}}{\mu, \rho \mid - \ apply(f, (v_1, ..v_n)) \ \rightarrow v} \tag{11}$$

$$\frac{sparql(\mu, \rho \mid - \ exp \ \rightarrow v)}{\mu, \rho \mid - \ exp \ \rightarrow v} \tag{12}$$

5 Examples of FunSPARQL Functions

In this section, we present the definition of several FunSPARQL extension functions showing the expressive power and usability of the language.

5.1 Arabic-Roman Numerals Converter

Appendix 8.1 defines parsers from arabic to roman numbers and converse. The SPARQL query embeds the definition of two FunSPARQL extension functions: function `spqr:roman` which enables to convert Arabic numerals into Roman numerals and its inverse function `spqr:arabic` which enables to convert Roman numerals into Arabic numerals. For instance, when called with *1959* as parameter, the `spqr:roman` function returns *MCMLIX* and conversely function `spqr:arabic` returns *1959* when called with *MCMLIX*.

5.2 Calendar

Appendix 8.2 defines calendar functions. The SPARQL query embeds the definition of the FunSPARQL extension function `cal:day` which enables to compute the day of any calendar date (after year 400 and before year 9999). It is the implementation of Zeller's congruence[3]. For example, when applied to `"1930-01-29"^^xsd:date`, function `cal:day` returns `"Wednesday"`.

5.3 SPARQL Interpreter Tuning

Callback functions associated to events can be defined as FunSPARQL extension functions to tune the behaviour of a SPARQL interpreter. In the Corese Semantic Web factory, we define two callbacks : one for query triple evaluation (`xt:candidate`) and one for solutions (`xt:result`).

The example below shows a `xt:candidate` extension function that displays candidate triples if the property of the query triple is `rdf:type`. The SPARQL interpreter detects the callback definition and calls it for each candidate triple. The definition of `xt:candidate` is given below, where `?q` is the current query

[3] https://en.wikipedia.org/wiki/Zeller%27s_congruence

triple, `?t` is the current candidate triple, `?b` is a boolean the value of which is the success (or the failure) of the match between the query and the target triples, and `xt:property` is an extension function which returns the property of a triple.

```
function xt:candidate(?q, ?t, ?b) {
   if (xt:property(?q) = rdf:type,
       xt:display(?t), true)
}
```

Similarly, the example below shows a `xt:result` extension function callback, to be called by the SPARQL interpreter to trace each solution of a query. Here is its definition with `?r` a query result and `xt:display` an extension function defined to display a result.

```
function xt:result(?r) {
   xt:display(?r)
}
```

Another use case of FunSPARQL extension functions is user-defined approximate matching. We defined the `us:match` extension function which definition is given in the following SPARQL query where `?q` and `?t` are two RDFS classes. Given a triple pattern with property `rdf:type` and a type, i.e., a class (e.g. `ex:Researcher`), a type in an RDF triple matches the type in the triple pattern when it is a subtype of it (line (09) emulates class subsumption by using a path of properties `rdfs:subClassOf`), but also when it is a supertype of it (line 10), or when it shares a common supertype (line 11). For instance (line 05), `ex:Person` and `ex:Engineer` will match `ex:Researcher` if `ex:Researcher` is declared as a subtype of `ex:Person` in the ontology, and both `ex:Researcher` and `ex:Engineer` as subtypes of `ex:Scientist`.

```
(01) SELECT * WHERE {
(02)   ?x a ?tx .
(03)   ?x ex:author ?d .
(04)   ?d a ?td
(05)   FILTER us:match(ex:Researcher, ?tx)
(06)   FILTER us:match(ex:Report, ?td)
(07) }
(08) function us:match(?q, ?t) {
(09)   EXISTS { {?t rdfs:subClassOf* ?q } UNION
(10)   { ?q rdfs:subClassOf* ?t } UNION
(11)   { ?q rdfs:subClassOf/^rdfs:subClassOf ?t }})
(12) }
```

This kind of approximate matching could be coded in standard SPARQL, without extension function, but the interest to write a FunSPARQL extension function is to reuse it in the query and to export it to reuse it in other queries.

In addition, the above query illustrates another significant advantage of FunSPARQL: the language to define functions extends the SPARQL filter expression language and therefore embeds the EXISTS clause which has a very high expressive power. In particular, it enables to define recursive graph pattern matching that goes beyond the expressivity of SPARQL property path. A property path enables to search resources related by a path of triples where the sequence of the properties in the path follows a regular expression, e.g. `foaf:knows*`. We can generalize the notion of property path to a Basic Graph Pattern path (BGP path) where two sets of resources, e.g. `(?x, ?y)` and `(?z, ?t)`, are related by a BGP path. For instance, let us consider the following BGP:

```
?x ex:p ?z . ?x ex:q ?y .
?y ex:p ?t . ?z ex:q ?t
```

The following SPARQL query embeds a FunSPARQL extension function `us:bgp` which is recursive and implements the basic graph pattern path BGP*.

```
PREFIX ex: <http://example.org/>
SELECT ?x ?y ?z ?t
WHERE {
  function us:bgp(?x, ?y, ?z, ?t) {
    if (?x = ?z && ?y = ?t, true,
    EXISTS {
      ?x ex:p ?a . ?x ex:q ?y .
      ?y ex:p ?b . ?a ex:q ?b .
      [] ex:p ?z . [] ex:p ?t
      FILTER (us:bgp(?a, ?b, ?z, ?t))
    }
  )}

?x ex:p+ ?z . ?x ex:q ?y
?y ex:p+ ?t . ?z ex:q ?t
FILTER us:bgp(?x, ?y, ?z, ?t)
}
```

For instance, this query would match the following RDF graph:

```
x1 ex:p x2 . x2 ex:p x3 . x3 ex:p x4 .
y1 ex:p y2 . y2 ex:p y3 . y3 ex:p y4 .
x1 ex:q y1 . x2 ex:q y2 . x3 ex:q y3 . x4 ex:q y4 .
```

with, among the six results:

```
(?x = x1, ?y = y1, ?z = x4, ?t = y4)
```

5.4 Extended Aggregates

FunSPARQL enables to simply define extended aggregates with a simple extension of the SPARQL interpreter. We introduce the `aggregate` function which acts as an additional generic aggregate. This function takes as arguments the expression to be aggregated (e.g. `?v`) and the name of an aggregation function (e.g. `us:sort_concat`). The `aggregate` function aggregates the results of the expression into a list and calls the aggregation function with this list as argument. The example below shows a variant of the `group_concat` aggregate which sorts the elements before concatenation occurs. The `rq:` prefix and namespace are used to assign an URI to each SPARQL standard function, hence `rq:concat` function is SPARQL `concat` function.

```
select (aggregate(?v, us:sort_concat) as ?res)
where {
  ?x rdf:value/rdf:rest*/rdf:first ?v
}

function us:sort_concat(?list){
  apply(rq:concat, xt:sort(?list))
}
```

5.5 Procedural Attachment

FunSPARQL enables users to perform procedural attachment to RDF resources. The idea is to annotate the URI of a function to declare that it is a method associated to a class. In the example below, we annotate two functions, `us:surface Rectangle` and `us:surfaceCircle` and declare that they implement the method `us:surface` for `us:Rectangle` and `us:Circle` respectively.

```
us:surfaceRectangle a xt:Method ;
  xt:name us:surface ;
  xt:input (us:Rectangle) ;
  xt:output xsd:double .

us:surfaceCircle a xt:Method ;
  xt:name us:surface ;
  xt:input (us:Circle) ;
  xt:output xsd:double .
```

Then, we can call a method on a resource.

```
select * (eval(xt:method(us:surface, ?x), ?x) as ?m)
where {
  ?x a us:Figure
}
```

The `xt:method` function below retrieves function `?fun` implementing method `?m` by finding the type `?t` of the resource (04) and then finding a method attached to the type, or a superclass of the type (05). In the latter case, this implements method inheritance following `rdfs:subClassOf` relation.

```
(01) function xt:method(?m, ?x){
(02)   let ((?fun) =
(03)     select * where {
(04)     ?x rdf:type/rdfs:subClassOf* ?t .
(05)     ?fun a xt:Method ; xt:name ?m ; xt:input(?t)})
(06)   { ?fun }
(07) }
```

We define below the functions whose URI are annotated as methods.

```
function us:surfaceRectangle(?x){
  let ((?w, ?l) = select * where { ?x us:width ?w ; us:length ?l }){
    ?w * ?l
  }
}

function us:surfaceCircle(?x){
  let ((?r) = select * where { ?x us:radius ?r }){
     3.14159 * power(?r, 2)
  }
}
```

Below are some RDF descriptions of figures for which we can compute the surface using procedural attachment.

```
us:Circle    rdfs:subClassOf us:Figure
us:Rectangle rdfs:subClassOf us:Figure

us:cc a us:Circle ;
us:radius 1.5 .

us:rr a us:Rectangle ;
us:width 2 ;
us:length 3 .
```

5.6 Extended Datatypes

FunSPARQL enables to simply define extended datatypes with a simple extension of the SPARQL interpreter. When evaluating a term with extended datatypes, the interpreter searches an extension function the name of which is the name of the operator (e.g. `plus` for '+') in the namespace of the datatype: e.g. `http://ns.inria.fr/sparql-datatype/roman#plus`.

For instance, the following query embeds the definition of an extended datatype for roman numerals with the usual arithmetic operations.

```
PREFIX dt: <http://ns.inria.fr/sparql-datatype/>
PREFIX rm: <http://ns.inria.fr/sparql-datatype/roman#>
PREFIX spqr: <http://ns.inria.fr/sparql-extension/spqr/>
SELECT
  ("II"^^dt:roman * "X"^^dt:roman + "V"^^dt:roman  AS ?res)
WHERE {}

export {
function rm:equal(?x, ?y)        {rm:arabic(?x) =  rm:arabic(?y)}
function rm:diff(?x, ?y)         {rm:arabic(?x) != rm:arabic(?y)}
function rm:less(?x, ?y)         {rm:arabic(?x) <  rm:arabic(?y)}
function rm:lessEqual(?x, ?y)    {rm:arabic(?x) <= rm:arabic(?y)}
function rm:greater(?x, ?y)      {rm:arabic(?x) >  rm:arabic(?y)}
function rm:greaterEqual(?x, ?y) {rm:arabic(?x) >= rm:arabic(?y)}

function rm:plus(?x, ?y)  {rm:roman(rm:arabic(?x) + rm:arabic(?y))}
function rm:minus(?x, ?y) {rm:roman(rm:arabic(?x) - rm:arabic(?y))}
function rm:mult(?x, ?y)  {rm:roman(rm:arabic(?x) * rm:arabic(?y))}
function rm:divis(?x, ?y) {rm:roman(rm:arabic(?x) / rm:arabic(?y))}

function rm:roman(?x)  {strdt(spqr:roman(?x), dt:roman)}
function rm:arabic(?x) {spqr:arabic(?x)}
}
```

5.7 RDF Data Transformation

We used FunSPARQL extension functions in the STTL language [3, 4]. STTL is a transformation language for RDF based on SPARQL. It introduces a TEMPLATE query form which enables users to describe a text pattern that is instantiated using variable bindings of query solutions. For instance, the following STTL template generates the functional syntax of an OWL `SubClassOf` statement.

```
TEMPLATE {
  "SubClassOf(" ?x " " ?y ")"
}
WHERE {
  ?x rdfs:subClassOf ?y
}
```

Templates are compiled as standard SPARQL queries of the SELECT form by compiling the variables in the TEMPLATE clause into `st:process` function calls. For instance, the above template is compiled into the following SPARQL query:

```
SELECT (CONCAT ("SubClassOf(",
    st:process(?x), " ", st:process(?y), ")")
  AS ?out)
WHERE {
  ?x rdfs:subClassOf ?y
}
```

By default, the `st:process` extension function is defined to generate the Turtle format of RDF terms. This definition can be overloaded for specific transformation needs in a `st:profile` template. For instance, in the example below, the `st:process` function is defined to apply the transformation engine on blank nodes, using the `st:apply-templates` function, and to display literals and URIs in Turtle format using the `st:turtle` function.

```
function st:process(?x) {
  if (isBlank(?x), st:apply-templates(?x),
     st:turtle(?x))
}
```

The calendar functions defined in section 5.2 enables to generate the calendar of a year with a single STTL template.

```
PREFIX cal: <http://ns.inria.fr/sparql-extension/calendar/>
TEMPLATE {
  "\n" cal:month(?m) "\n"
  "Mo Tu We Th Fr Sa Su \n"
  GROUP {
    IF (?n = 1, us:space(cal:num(?day) - 1), "")
    IF (?n < 10, " ", "") ?n " "
    IF (?day = "Sunday", "\n", "")
  ; separator = ""
  }
  ; separator = "\n"
}
WHERE {
  BIND (unnest(xt:iota(12)) AS ?m)
  BIND (unnest(xt:iota(cal:days(?y, ?m))) AS ?n)
  BIND (xsd:date(CONCAT(?y, "-", ?m, "-", ?n)) AS ?date)
  BIND (cal:day(?date) AS ?day)
}
GROUP BY ?m
ORDER BY ?m
VALUES ?y { 2000 }

function us:space(?n) {
  if (?n = 0, "",
    concat("   " , us:space(?n - 1)))
}
```

Figure 1 is the result of a similar STTL template that generates a HTML calendar.

Figure 1: Calendar

6 Implementation of FunSPARQL

FunSPARQL is implemented using the SPARQL interpreter of the Corese Semantic Web Factory [5, 2]. The *function*, *let* and other statements are implemented by the SPARQL parser, compiler and interpreter. For the FunSPARQL compiler, function definitions must be recorded, taking into account the fact that the same function name can be used with different numbers of arguments. The notion of local variable must be defined.

The evaluation of expressions takes two environments as arguments: a solution of a BGP for global variables and a stack of bindings for local variables. A local variable is such that its value does not come from a BGP solution but from parameter passing, by a function call, `let` or `for` statements. Function parameter passing is done using a stack of variable bindings that allows nested and recursive function calls.

When executing a function call, the FunSPARQL interpreter does:

1. fetch the function definition corresponding to the function name and the number of arguments,
2. bind in a stack the arguments of the definition with the value of the parameters of the function call,
3. execute the body of the function with the variable bindings,
4. pop argument bindings from the stack,
5. return the result of the execution of the body.

It must be noted that this extension may lead to safety problems since it enables code injection into a SPARQL interpreter. In particular, it may lead a SPARQL endpoint into an infinite loop if users define and execute non terminating functions. It may be wise to switch off this extension in server mode for user defined queries.

Should an error occur, function evaluation resumes in error mode, on the same model as SPARQL evaluation error. In a FILTER, the filter fails. In a SELECT or a BIND clause, "*If the evaluation of the expression produces an error, the variable remains unbound for that solution but the query evaluation continues*"[4].

FunSPARQL has been validated on the functions described in section 5 and extensively used in several STTL transformations on a server available online[5]. We measured the performance of our implementation of FunSPARQL on the execution of the extension function `fib` implementing the Fibonacci sequence:

$$fib(n) = if\ (n \leq 2,\ 1,\ fib(n-2) + fib(n-1)) \tag{13}$$

The computation of fib(30) = 832040 requires 1,664,079 function calls and takes 0.55 sec on a laptop, which represents 3 million function calls per second.

7 Conclusion and Future Work

We propose a lightweight extension to SPARQL — FunSPARQL — that enables users to define extension functions directly into SPARQL. The key point of our proposal is that a functional language can easily be integrated in SPARQL to define extension functions. FunSPARQL reuses the language of SPARQL filter expressions and augment it with `function`, `let`, `for` and other statements. We provide its syntax and its formal semantics. We have implemented FunSPARQL in the Corese Semantic Web Factory and we have developed a set of functions to validate our approach.

As future work, we plan to investigate Linked functions and go further in the definition of a functionnal programming language for SPARQL. We may

[4] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/#assignment
[5] `http://corese.inria.fr`

consider type checking function definition to ensure a certain level of safety. We may also consider compiling functions into target programming languages such as Java.

Acknowledgment

We thank Fabien Gandon for the ideas of function annotation and Linked function.

References

[1] Maurizio Atzori. Toward the web of functions: Interoperable higher-order functions in sparql. In *The Semantic Web – ISWC 2014*, volume 8797 of *Lecture Notes in Computer Science*, pages 406–421. Springer International Publishing, 2014.

[2] Olivier Corby and Catherine Faron-Zucker. The KGRAM Abstract Machine for Knowledge Graph Querying. In *IEEE/WIC/ACM International Conference on Web Intelligence*, Toronto, Canada, September 2010.

[3] Olivier Corby and Catherine Faron-Zucker. STTL: A SPARQL-based Transformation Language for RDF. In *Proc. 11th International Conference on Web Information Systems and Technologies, WEBIST 2015*, Lisbon, Portugal, May 2015.

[4] Olivier Corby, Catherine Faron-Zucker, and Fabien Gandon. A Generic RDF Transformation Software and its Application to an Online Translation Service for Common Languages of Linked Data. In *Proc. 14th International Semantic Web Conference, ISWC*, Bethlehem, Pennsylvania, USA, October 2015.

[5] Olivier Corby, Alban Gaignard, Catherine Faron-Zucker, and Johan Montagnat. KGRAM Versatile Data Graphs Querying and Inference Engine. In *Proc. IEEE/WIC/ACM International Conference on Web Intelligence*, Macau, China, December 2012.

[6] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Recommendation, W3C, 2013. http://www.w3.org/TR/sparql11-query/.

[7] G. Kahn. Natural Semantics. In *Proc. of 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[8] Holger Knublauch. SPIN - SPARQL Syntax. Member Submission, W3C, 2011. http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/.

[9] Gregory T. Williams. Extensible SPARQL Functions With Embedded Javascript. In *Scripting for the Semantic Web, ESWC Workshop*, Innsbruck, Austria, May 2007. http://ceur-ws.org/Vol-248/paper7.pdf.

8 Appendix

8.1 Romain & Arabic Numbers

```
prefix spqr: <http://ns.inria.fr/sparql-extension/spqr/>
select
  (1959 as ?n)
  (spqr:romain(?n) as ?r)
  (spqr:arabic(?r)  as ?d)
where {
}
export {
function spqr:romain(?n) { spqr:spqr(?n) }
function spqr:arabic(?r) { spqr:parse(?r) }

function spqr:div(?a, ?b) {
  xsd:integer(floor(?a / ?b)) }
function spqr:mod(?a, ?b) {
  xsd:integer(?a - (?b * spqr:div(?a, ?b))) }

function spqr:rep(?s, ?n) {
  if (?n = 0, "",
  if (?n = 1, ?s,
  concat(?s, spqr:rep(?s, ?n - 1))))
 }

function spqr:r1(?n) {
    spqr:num(?n, "I", "V", "X")
 }

function spqr:r10(?n) {
  spqr:num(?n, "X", "L", "C")
 }

function spqr:r100(?n) {
  spqr:num(?n, "C", "D", "M")
 }

function spqr:r1000(?n) {
  spqr:rep("M", ?n)
 }

function spqr:num(?n, ?u, ?f, ?t) {
  if (?n <= 3, spqr:rep(?u, ?n),
  if (?n = 4, concat(?u, ?f),
  if (?n < 9, concat(?f, spqr:rep(?u, ?n - 5)),
```

```
  if (?n = 9, concat(?u, ?t), ""))))
 }

function spqr:spqr(?n) {
  if (?n < 10, spqr:r1(?n),
  if (?n < 100,
    let (?c = spqr:div(?n, 10),
         ?r = spqr:mod(?n, 10)){
      concat(spqr:r10(?c), spqr:spqr(?r))
    },

  if (?n < 1000,
    let (?c = spqr:div(?n, 100),
         ?r = spqr:mod(?n, 100)){
      concat(spqr:r100(?c), spqr:spqr(?r))
    },

  if (?n < 10000,
    let (?c = spqr:div(?n, 1000),
         ?r = spqr:mod(?n, 1000)){
      concat(spqr:r1000(?c), spqr:spqr(?r))
    },
    ?n))))
}

# parse romain number

function spqr:parse(?s) {
if (strlen(?s) = 0, 0,
  let (?f = substr(?s, 1, 1)){
    if (?f = "I",
      spqr:step(?s, "I", "V", "X", 1, 5, 10),
    if (?f = "V",
      5 + spqr:parse(substr(?s, 2)),
    if (?f = "X",
      spqr:step(?s, "X", "L", "C", 10, 50, 100),
    if (?f = "L",
      50 + spqr:parse(substr(?s, 2)),
    if (?f = "C",
      spqr:step(?s, "C", "D", "M", 100, 500, 1000),
    if (?f = "D",
      500   + spqr:parse(substr(?s, 2)),
    if (?f = "M",
      1000  + spqr:parse(substr(?s, 2)),
    0)))))))
  }
```

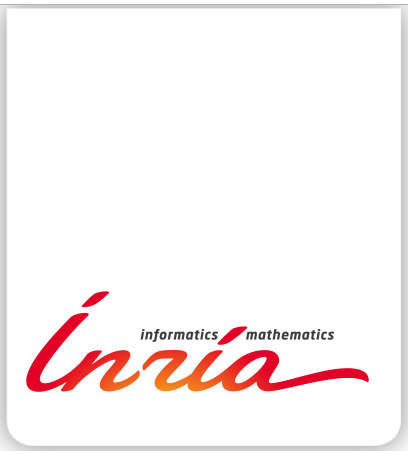
```
)
}

function spqr:step(?s, ?su, ?sc, ?sd, ?u, ?c, ?d) {
  if (strlen(?s) = 1, ?u,
    let (?r = substr(?s, 2, 1)){
      if (?r = ?sc,   ?c - ?u + spqr:parse(substr(?s, 3)),
        if (?r = ?sd, ?d - ?u + spqr:parse(substr(?s, 3)),
          ?u + spqr:parse(substr(?s, 2))))
    }
  )
}
}
```

8.2 Calendar

```
prefix cal: <http://ns.inria.fr/sparql-extension/calendar/>
select (cal:day(now()) as ?day)
where  { }
export {
function cal:day(?d)  { cal:en(cal:find(?d))}

function cal:jour(?d) { cal:fr(cal:find(?d)) }

function cal:div(?a, ?b) {
  xsd:integer(floor(?a / ?b))
}

function cal:mod(?a, ?b) {
  xsd:integer(?a - (?b * cal:div(?a, ?b)))
}

function cal:bisextile(?y) {
    ((cal:mod(?y, 4) = 0) &&
      ((cal:mod(?y, 100) != 0)
      || (cal:mod(?y, 400) = 0)))
}

function cal:ab(?y) { cal:div(?y, 100)}
function cal:cd(?y) { cal:mod(?y, 100)}
function cal:k(?y)  { cal:div(cal:cd(?y), 4)}
function cal:q(?y)  { cal:div(cal:ab(?y), 4)}

function cal:monthday(?m, ?y) {
  if (?m <= 2,
    if (cal:bisextile(?y),
```

```
      if (?m = 1, 3, 6),
      if (?m = 1, 4, 0)),
  if (?m in (3, 11), 0,
    if (?m in (6), 1,
      if (?m in (9, 12), 2,
        if (?m in (4, 7), 3,
          if (?m in (10), 4,
            if (?m in (5), 5, 6)))))))
}

function cal:get(?y, ?m, ?d) {
  let (?n = cal:k(?y) + cal:q(?y) + cal:cd(?y) +
        cal:monthday(?m, ?y) + ?d + 2
        + 5 * cal:ab(?y))
    {
      cal:mod(?n, 7)
    }
}

function cal:find(?d) {
  cal:get(year(?d), month(?d), day(?d))
}

function cal:en(?n) {
  if (?n = 0, "Sunday",
  if (?n = 1, "Monday",
  if (?n = 2, "Tuesday",
  if (?n = 3, "Wednesday",
  if (?n = 4, "Thursday",
  if (?n = 5, "Friday",
  if (?n = 6, "Saturday", "Unknown")))))))
}

function cal:num(?day) {
    if (?day in( "Lundi", "Monday"), 1,
    if (?day in( "Mardi", "Tuesday"), 2,
    if (?day in( "Mercredi", "Wednesday"), 3,
    if (?day in( "Jeudi", "Thursday"), 4,
    if (?day in( "Vendredi", "Friday"), 5,
    if (?day in( "Samedi", "Saturday"), 6,
    if (?day in( "Dimanche", "Sunday"), 7, 0)))))))
}

function cal:days(?y, ?m) {
let (?list = xt:list(31, 28, 31, 30, 31, 30, 31,
      31, 30, 31, 30, 31)){
```

```
  if (?m != 2, xt:get(?list, ?m - 1),
      if (cal:bisextile(?y), 29, 28))
}
}
}
```



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399