LDScript: a Linked Data Script Language

Olivier Corby, Catherine Faron Zucker, Fabien Gandon

LDScript: a Linked Data Script Language

Olivier Corby, Catherine Faron-Zucker, Fabien Gandon

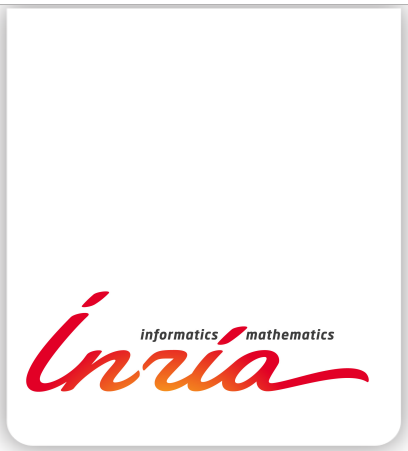
RESEARCH REPORT

N° 8982

November 2016

Project-Team Wimmics

ISSN 0249-6399 ISRN INRIA/RR--8982--FR+ENG



LDScript: a Linked Data Script Language

Olivier Corby*, Catherine Faron-Zucker†, Fabien Gandon‡

Project-Team Wimmics

Research Report n° 8982 — November 2016 — 18 pages

Abstract: In addition to the existing standards, Web of Data programmers would take advantage of a dedicated programming language enabling them to define functions on RDF terms, triples and graphs as well as SPARQL query results. In particular, this is the case when defining SPARQL extension functions, and the ability to capitalize complex SPARQL filter expressions into extension functions or to define and reuse dedicated aggregates would support modularity and maintenance of the code. Another use case is the definition of *functional* properties associated to RDF resources and the definition of procedural attachments as functions assigned to RDFS or OWL classes with the selection of the function to be applied to a resource depending on the type of the resource. To address these needs we define a *Linked Data Script language* on top of the SPARQL filter expression language. We provide the syntax and the semantics of the LDScript language.

Key-words: Script Language, Semantic Web, Web of Data, SPARQL

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

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

LDScript: a Linked Data Script Language

Résumé : Un langage complémentaire des standards existant, dont les entités de base seraient des termes, des triplets et des graphes RDF ainsi que les résultats de requêtes SPARQL serait utile pour la programmation du Web de données. Cela faciliterait en particulier la définition de fonction d'extension pour SPARQL ainsi que la définition d'opérateurs d'aggrégation supplémentaires et la capitalisation de filtres de requête complexes sous forme de fonctions. Un autre use case concerne la définition de propriétés fonctionnelles associées à des ressources RDF ainsi que les attachements procéduraux associés aux classes RDFS et OWL avec la sélection de la fonction appropriée en fonction du type de ressource. Pour résoudre tous ces problèmes nous définissons LDScript, un language de script pour le Web de données liées au dessus du langage de filtre de SPARQL. Nous donnons sa syntaxe et sa sémantique.

Mots-clés : Language de script , Web sémantique, Web de données, SPARQL

1 Introduction

RDF is the standard model recommended by the W3C to represent and interchange data on the Semantic Web. It is associated with RDF Schema and/or OWL for ontology-based data modelling and SPARQL for data and ontology querying. The development of the Web of data opens up a wide range of use cases where, in addition to the existing standards, the Semantic Web programmer would take advantage of a dedicated programming language enabling her to define functions on RDF terms or RDF graphs. This is the case, for instance, when defining SPARQL extension functions implemented for a special purpose and domain or application dependent. This would also be needed to capitalize a complex SPARQL filter expression or the definition of special purpose extension aggregates to be reused across queries or sub-queries. Another kind of use cases is the definition of *functional* properties associated to RDF resources, the results of which are computed on demand. For instance, the value of the surface property of a rectangular object could be computed as the product of the values of its width and length properties; the age of a person could also be computed from her date of birth. These use cases can also be extended to the definition of procedural attachments as functions assigned to RDFS or OWL classes. The selection of the function to be applied to a resource can then be made dependent on the types (classes) of the resource.

The requirements we propose for a programming language enabling such definitions of functions are:

- The objects of the language are RDF terms (URI, blank node and literal), RDF triples, RDF graphs, SPARQL solution mappings and lists of such objects.
- The statements of the language include SPARQL filter expressions and SPARQL queries (the SELECT and CONSTRUCT query forms and the SERVICE clause).
- The language provides function definition and function call on lists.

In this paper, we address the research question: *Can we define a standard-based programming language that meets the above described requirements?*

To answer this question, we define a *Linked Data Script language* on top of the SPARQL filter expression language, taking advantage of the fact that the SPARQL filter expression language potentially enables users to express function definitions. We then proceed to define a *programming language* on top of SPARQL filter expression language.

Syntactically, it consists in additional statements: a `function` statement enabling users to define functions and a `let` statement enabling him to introduce local variables in the filter language.

We call our extension LDScript, standing for Linked Data Script, as it consists in providing the Linked Data with a script language. We present the syntax and semantics of LDScript as well as an implementation and we illustrate the

simplicity as well as the expressive power of this extension with some examples of LDScript 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 an overview of LDScript. In Section 4 we define the syntax and semantics of LDScript. In Section 5 we present several use cases and we show how they can easily be addressed by using LDScript. In Section 6 we present our implementation of LDScript 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 one to represent both SPARQL queries (SPIN templates) and SPARQL extension functions (SPIN functions) [7]. SPIN is represented in RDF. In SPIN, a SPARQL extension function is identified by a resource of type `sp:Function` (with `sp` the prefix denoting the SPIN namespace) which is linked by property `sp:body` to its definition as a SPARQL query of the form SELECT or ASK.

Jena provides a `java` URI scheme for naming and accessing SPARQL extension functions implemented in Java. This enables one 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 example of a SPARQL query using the `f:myTest` extension function implemented in Java and enabling to filter the RDF triples matching the triple pattern to those for which a call to function `f:myTest` with the values bound to their subject and object as parameters 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 [8]. Functions are identified by URLs and their source code may be retrieved at run time by dereferencing their URL. It relies on an RDF schema enabling one 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`

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

(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 occurrence 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's IRI is computed from the extension function's IRI, based on a Function-to-Endpoint IRI pattern.

When compared to these four state-of-the-art proposals, the key idea of our proposal described in the following is to extend the SPARQL language, and more precisely its filter language, in order to enable the definition of extension functions in the SPARQL language *itself* and using its native syntax.

3 Overview of LDScript

Our goal is to define functions the objects of which are Linked Data entities (e.g. URI, RDF Literal, RDF triple, etc.) with the main objective of defining SPARQL extension functions and possibly extend SPARQL itself.

One possibility is to rely on an existing programming language, e.g. Java, and use the API of the SPARQL implementation of the RDF entities. However, this approach has several weaknesses. First, it is not interoperable because other SPARQL implementations of RDF entities do not use the same API. Hence, the functions cannot be reused by other SPARQL implementations. Second, one does not benefit of SPARQL native function library dedicated to RDF terms: isURI, isBlank, isLiteral, datatype, strdt, strlang, langMatch, uri, bnode, etc. Third, one must switch back and forth from SPARQL to Java environments with their compiler, project management environment, etc. and link the compiled functions to the SPARQL interpreter.

Another possibility would be to design a specific programming language the object of which would be RDF entities with a library implementing SPARQL functions. The weakness of this approach would be that users would have to learn yet another programming language.

We propose LDScript, a third way that synthetize these two approaches: a programming language the objects of which are RDF entities, compatible with SPARQL and embedding the complete SPARQL function library.

LDScript primarily relies on the SPARQL filter expression language. 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 argument if the first argument results in an effective value of true, or else the value of the third argument. A SPARQL filter restricts the solutions of a graph pattern matching to those satisfying the constraint it expresses: the filtered solutions result in the boolean value *true* when substituted into the filter expression.

We propose to define functions by taking advantage of the fact that the SPARQL filter language enables users to define expressions. Handbooks of programming languages and programming history explain that typical imperative languages include as commands: variables declaration, assignment, call, return, sequential blocks, iterative commands and if statements. For this reason, in this section we will introduce the corresponding statements in LDScript. Here are the namespaces and prefixes used in the definitions:

```
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 LDScript Function Definition

In LDScript, a *function definition* starts with the FUNCTION keyword. The first argument of the declaration is the name (a 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 LDScript 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 a another example of function definition. A call to the `us:status` function returns the status of the resource given as parameter. Its definition uses the SPARQL built-in IF function and EXISTS operator.

```
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 function form with a given value for variable `?n`.

The language for defining the body of a function is LDScript, i.e. the SPARQL filter expression language extended with statements presented in this document. Hence, to define extension functions, LDScript programmers can make use of the expressivity of the whole SPARQL filter expression language. In particular, this includes built-in SPARQL functions, among which the IF function form enabling to consider alternatives, and the EXISTS statement to test graph patterns. This also includes extension functions defined in LDScripts or externally defined in another language (as SPARQL allows it).

At compile time, 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 LDScript interpreter to retrieve the function definition at runtime for a function call. For example, the SPARQL query below is followed by the definition of function `us:fac` and the `WHERE` clause embeds a call to this function to search the 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 Local Variable Declaration

LDScript function definitions can embed *local variable declarations*. These are expressed in a LET statement. Its first argument declares a local variable and its value. Its second argument is an expression which is evaluated with the transient binding of the local variable declared. After completion of the expression evaluation, the binding vanishes. The result of a LET statement is the result of its second argument. For instance, the example below shows a `let` statement that returns the pretty-printing of a date, e.g. `"29/01/2017"`.

```
LET (?n = now()) {
  CONCAT(day(?n), "/", month(?n), "/", year(?n))
}
```

3.3 Loop Statements

In order to iterate a statement on the elements of a list of values, LDScript is provided with the FOR loop statement. For instance the following function iteratively calls the `xt:display` function on the prime numbers among a given list of natural numbers.

```
FOR (?n IN xt:list(1, 2, 3, 4, 5)) {
  IF (us:prime(?n)) { xt:display(?n) }
}
```

The FOR statement can iterate on the results of a SPARQL SELECT or CONSTRUCT query. In the case of a CONSTRUCT query, it iterates on the triples of the graph. For instance, the following function iteratively calls the `xt:display` function on RDF triples of the form <URI> `a foaf:Person`.

```
FOR (?t IN CONSTRUCT WHERE { ?x a foaf:Person }) {
  xt:display(?t)
}
```

The FOR statement can also bind a list of variables to a list of lists of values. In the case of the result of a CONSTRUCT query, the FOR statement can bind the subject, property and object of each triple to a list of three variables, like in the following example statement.

```
FOR ((?s, ?p, ?o) in
  CONSTRUCT WHERE { ?x a foaf:Person }) {
  xt:display(?s, ?o)
}
```

3.4 Function Evaluation

LDScript is provided with the FUNCALL function to call a function whose name is dynamically evaluated within a LET statement. For instance, the following LET statement enables to apply a function to a variable `?x`, whose body is randomly chosen.

```
LET (?fun = IF (rand() > 0.5, us:foo, us:bar)) {
  funcall(?fun, ?x)
}
```

LDScript is provided with the APPLY function to iteratively call a binary function on a list of arguments. For example, the following function call enables to compute the sum of the elements of a list of numbers with the binary `rq:plus` function.

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

3.5 List Datatype

LDScript is provided with a `dt:list` datatype to manage lists of values. A `dt:list` datatype value is a list whose elements are RDF terms: URIs, literals, blank nodes or sublists of type `dt:list`. The elements of a list need not be of the same kind, neither of the same datatype. The `dt:list` datatype comes with a set of predefined functions among which `xt:size` returns the size of the list, `xt:get` returns the n^{th} element, `xt:sort` 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 one 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))
```

There are several variants of the MAPLIST function: MAP applies a function and returns true, MAPSELECT returns the list of elements such that the boolean function returns true.

4 LDScript Language

The previous section gave an overview of LDScript. In this section we formally define the syntax and semantics of this language.

4.1 LDScript Syntax

LDScript grammar is based on SPARQL[2]. The definition of `BuiltInCall` is extended with `let`, `for`, `map`, `funcall` and `apply` statements.

```
Function ::= 'function' iri ('()' | VarList) Body
Body ::= '{' '}' | '{' Expression (';' Expression)* '}'
VarList ::=  '(' Var (',' Var)* ')'
BuiltInCall ::= SPARQL_BuiltInCall
  | 'let' '(' Decl (',' Decl) * ')'        Body
  | 'for' '(' VarOrList 'in' ExpQuery ')'  Body
  | Map   '(' iri ',' Expression ')'
  | 'funcall' '(' Expression (',' Expression)* ')'
  | 'apply' '(' iri ',' Expression ')'
Decl      ::= VarOrList '=' ExpQuery
VarOrList ::= Var | VarList
ExpQuery  ::= Expression |
  SelectQuery | ConstructQuery | ServiceGraphPattern
Map ::= 'map' | 'maplist' | 'mapselect'
```

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

4.2 LDScript Semantics

As usually done for programming languages, we formally defined the semantics of the core of LDScript by a set of Natural Semantics inference rules [6]. 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 $\vdash$ symbol states that the expression on the right side is evaluated in the environment given 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. In addition, the environment contains a reference to the SPARQL dataset.

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

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

Rule 12 specifies the evaluation of an LDScript 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.

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

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

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

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

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

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

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

$$\frac{\mu, \rho \vdash e \rightarrow f \wedge \mu, \rho \vdash f(e_1, .. e_n) \rightarrow v}{\mu, \rho \vdash funcall(e, e_1, .. e_n) \rightarrow v} \quad (8)$$

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

$$\frac{\mu, \rho \vdash f() \rightarrow v}{\mu, \rho \vdash apply(f, ()) \rightarrow v} \quad (10)$$

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

$$\frac{sparql(\mu, \rho \vdash exp \rightarrow v)}{\mu, \rho \vdash exp \rightarrow v} \quad (12)$$

4.3 Summary

LDScript is a programming language where values are RDF terms, hence its use to define extension functions avoids to cast datatype values from RDF to the target language (e.g. Java) and back. It enables to associate function definitions directly to a SPARQL query, with no need to compile nor link code.

All standard SPARQL functions, among which `IF` and `EXISTS`, are natively available in LDScript and can be used directly in a LDScript function definition. The SELECT and CONSTRUCT SPARQL query forms, the SPARQL SERVICE clause are statements of the LDScript and can be used as well in the definition of functions. The result of these statements can also be manipulated directly in the language without any cast or additional operations. In addition, the language provides recursion, hence enabling recursive SPARQL queries: a function can execute a SPARQL query that can call the function.

5 Use Cases

In this section, we present the definition of several LDScript extension functions showing the expressive power and usability of the language. Some additional examples can be found at: `http://ns.inria.fr/sparql-extension` such as calendar functions that enable one to compute the week day of a date and a converter from decimal to Arabic numbers and reverse.

5.1 Extended Aggregates

LDScript enables us to simply define extended aggregates with a simple extension of the SPARQL interpreter. We introduce the `aggregate` function which is an additional generic aggregate operator. This function takes as arguments the expression to be aggregated (e.g. `?v`). The `aggregate` function aggregates the results of the expression into a `dt:list`. Then we can call a custom aggregation function with this list as argument. The example below defines `sort_concat`, a variant of the `group_concat` aggregate which sorts the elements before concatenation occurs. The `rq` prefix and namespace are used to assign a URI to each SPARQL standard function, hence `rq:concat` function is SPARQL `concat` function. Hence LDScript enables users to define inline custom aggregates for SPARQL.

```
SELECT (aggregate(?v) as ?list)
  (us:sort_concat(?list) as ?res)
WHERE {
  ?x rdf:value/rdf:rest*/rdf:first ?v
}
FUNCTION us:sort_concat(?list){
  apply(rq:concat, xt:sort(?list))
}
```

5.2 Application-specific processing

Another use case of LDScript extension functions is user-defined calculation such as metrics for 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 LDScript extension function is to define it once and reuse it across queries.

5.3 Procedural Attachment

LDScript 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 computing surfaces, `us:surfaceRectangle` 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 the `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 now 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 .
```

5.4 Calendar

We wrote LDScript functions to compute the day of the week given a date literal of type xsd:date[3] and functions to generate a calendar given a year[4]. We

[3] `http://ns.inria.fr/sparql-extension/calendar`
[4] `http://corese.inria.fr/srv/template?transform=st:calendar`

designed a dynamic Web page generated from DBpedia events were events of a given year are placed into the calendar[5]. The performance of LDScript is such that the Web page is computed and displayed in real time.

5.5 Data Shape

As part of a DataShape validator, we wrote an interpreter for W3C DataShape Property Path language[6]. The function below recursively rewrites a property path shape expression ?pp as a LDScript list.

```
function sh:path(?shape, ?pp){
  let ((?q, ?path) =
      select ?shape ?pp ?q ?path where {
        graph ?shape {
            # rdf:rest is for a sequence
            values ?q {
                sh:inversePath sh:alternativePath
                sh:zeroOrMorePath sh:oneOrMorePath
                sh:zeroOrOnePath rdf:rest }
            ?pp ?q ?path
        }
      } ) {

      if (! bound(?q)){
        if (isURI(?pp)){ ?pp }
        else { error() }
      }
      else if (?q = rdf:rest) {
        xt:list(sh:sequence, sh:list(?shape, ?pp)) }
      else { xt:list(?q, sh:path(?shape, ?path)) }
   }
}
```

The function below rewrites recursively an RDF list of path expressions ?pp (e.g. a sequence) as a LDScript list.

```
function sh:list(?shape, ?pp){
   let ( (?l) =
    select ?shape ?pp (aggregate(sh:path(?shape, ?e)) as ?l)
    where { graph ?shape { ?pp rdf:rest*/rdf:first ?e }}) {
    ?l
   }
}
```

[5] http://corese.inria.fr/srv/template?profile=st:calendar3
[6] http://ns.inria.fr/sparql-extension/datashape

These two exampes shows the natural integration of SPARQL queries into LDScript program. In particular LDScript variables are seamlessly shared with SPARQL variables.

6 Implementation and test

LDScript 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 LDScript compiler, function definitions are recorded, taking into account the fact that the same function name can be used with different numbers of arguments. The notion of local variable is defined.

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"*[7].

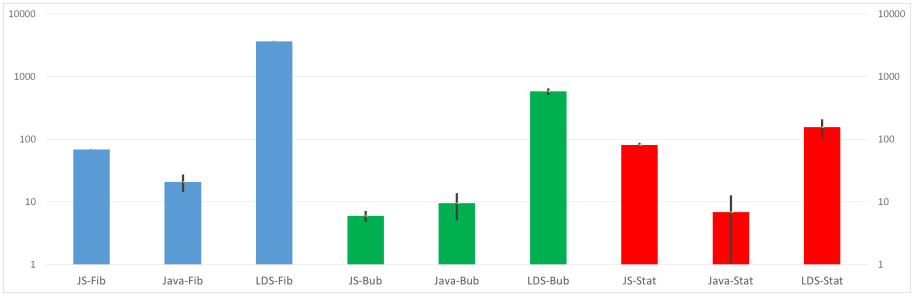


Figure 1: Comparison of mean times and their mean absolute differences for JavaScript (JS), Java and LDScript (LDS) computing recursive Fibonacci (-Fib in Blue), Bubble sort (-Bub in green) and statistics (-Stat in red) using a logarithmic scale.

LDScript has been validated on the functions described in Section 5 and extensively used in several STTL transformations on a server available online[8][3, 4]. We measured the performance of our implementation of LDScript on the execution of (1) a recursive Fibonacci function to test an exponential number of calls, (2) on the Bubble sort algorithms to evaluate loops and (3) on statistics functions for the calculation aspect. We compared with Java and Javascript implementations which, of course, benefit from many years of optimizations. Our goal here is just to show that a direct implementation on top of a SPARQL engine without dedicated optimizations is already usable. The results are shown using a logarithmic scale in figure 1. The extension function `fib` implements the

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

Fibonacci sequence. The computation of fib(35) = 9227465 on a HP EliteBook laptop takes 3650 ms in LDScript, 68.8 ms in JavaScript and 24.9 ms in Java.

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

Bubble sort on an array of 1000 items takes 580.1ms in LDScript, 6ms in JavaScript and 9.5ms in Java. Three statistics functions together (average, median and standard deviation) on an array containing 100000 integer values takes 157.1 ms in LDScript, 80.5 ms in JavaScript and 6.9 ms in Java.

7 Conclusion and Future Work

Dedicated programming language enabling Semantic Web programmers to define functions on RDF terms, RDF graphs or SPARQL results can improve modularity, reuse and maintenance of the code produced for Linked Data. This is the case when defining SPARQL extension functions, complex SPARQL filter expressions, *functional* properties associated to RDF resources and procedural attachments as functions assigned to classes. To address these needs we detailed in this article a lightweight extension of SPARQL filter expression language to enable the *definition* of extension functions and we defined a *Linked Data Script language* on top of the SPARQL filter expression language. Compared to state-of-the-art we directly extend the SPARQL language in order to enable the definition of extension functions in the SPARQL language *itself* and using its native syntax, building on a well-known and widely accepted component of the Web of data. The key point of our proposal is that a programming language can easily be integrated in SPARQL to define extension functions. LDScript reuses the language of SPARQL filter expressions and extends it with several classical programming statements, among which FUNCTION, LET, FOR, EVAL and APPLY. We first provided an overview of the LDScript language and then we detailed the formal definition of the grammar of its syntax and the Natural Semantics inference rules of its semantics. We also provide a full implementation in the Corese Semantic Web Factory and we have developed a set of functions to validate our approach, some example of which were presented in this article.

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

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.

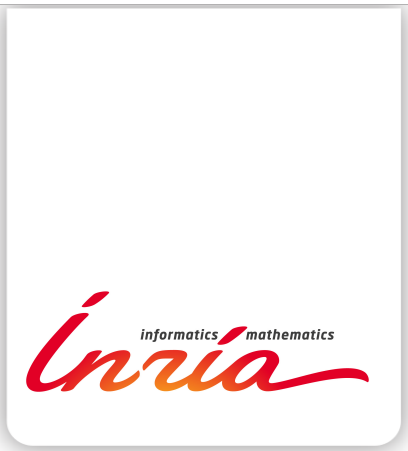
[4] Olivier Corby, Catherine Faron-Zucker, and Fabien Gandon. A Generic RDF Transformation Software and its Application to an Online Translation Service for Common Languages of Linked Data. In *Proc. 14th International Semantic Web Conference, ISWC*, Bethlehem, Pennsylvania, USA, October 2015.

[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] 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.

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

[8] 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.



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