# Corese User Manual - 17/04/2007

Virginie BOTTOLLIER

Version: 1.0.2
Date: 17/04/2007
Objet: Corese User Manual



*Click here to know what is new in this version*

# Plan

## Introduction



## Using Corese



## Developing with Corese

# Corese Presentation

Corese stands for COnceptual REsource Search Engine. It is an RDF engine based on Conceptual Graphs (CG). It enables the processing of RDF Schema and RDF statements within the CG formalism (see the *note from T. Berners-Lee* on the subject).
Corese is written in Java; it provides an API for developers to add semantic to their applications. The main functionality of Corese is dedicated to retrieve web resources annotated in RDFS, by using a query language based on *SPARQL* and an inference rule engine.

On top of Corese, a *JSP tag library (Semantic Tags)* has been developed. And a set of web services are under development. These two libraries allow developers to use Corese in their web applications by wrapping Corese access (like Corese administration tasks, query submission and so on) into standard web tools (JSP Tags and Web services).

This manual presents how to use Corese in a standalone mode. After a short introduction about what Corese is and what it is used for, the main part describes how to use Corese (how to get the standalone version, how to load ontologies, how to write queries, which functionalities are added, how to write a new function, how to write rules), while the final part is more technical, discussing about how to develop with Corese (it describes the API, how to handle the results, how to manage the property files).
It is written for persons with a measurable knowledge in semantic web.

This manual describes the *CORESE_2007_04_17_v2_3_0.jar* version of Corese.

# Examples of Corese Applications

The standalone version of Corese can be used to:

- Test an ontology: look at all concepts and properties present in an ontology and find relations between them
- Test the rules language, see the effect of rules on an ontology
- Test the Corese search engine (with direct access to the engine)
- ..

The first version of the Corese prototype was created in 1999. Corese is now used, and has been used, as semantic search engine in more than 20 applications; here are some examples:

- *Semantic Tags*

Semantic Tag is a JSP tag library which allows the use of semantic web notions in a web application with the CORESE engine. This library, both easy to use and fast to take in hand, has been created for developers that have to write such applications. It is built upon the semantic web search engine Corese for the semantic side and upon the standard library (JSTL) for the JSP side.

- *Sweet Wiki*: Semantic Web Enabled Technologies for Wikis.

Wikis are social web sites enabling a large number of participants to modify any page or create a new page using their web browser. As they grow, wikis suffer from a number of problems (anarchical structure, large amount of pages, aging navigation paths, etc.). In SweetWiki we investigate the design of a wiki built around a semantic web server i.e. the use of semantic web technologies to support and ease the life cycle of the wikis.
Sweet Wiki uses Sewese and thus Corese.

- *Ewok-Hub*: Environment Web Ontology Knowledge Hub. (2006-2008)

The Ewok-Hub project aims at using semantic web to develop systems that allow (on Internet) the cooperation of different organizations implied in an engineer workflow. The future application will manage several project memories on the capture and the storage of CO2, while using technological watch's results.
In this project, Corese is used in the web service approach to make services composition and to suggest to end-users existing services (or composition of services) that can be used in a given context.

- *Palette*: European project on Pedagogically sustained Adaptative LEarning Through the exploitation of Tacit and Explicit knowledge. (2006-2008)

The Palette project aims at facilitating and augmenting individual and organisational learning in Communities of Practice (CoPs). Towards this aim, an interoperable and extensible set of innovative services as well as a set of specific scenarios of use will be designed, implemented and thoroughly validated in CoPs of diverse contexts.
Corese will be used to search information: documents, services, communities of practice..

- *SevenPro*: European project on Semantic Virtual Engineering Environment for Product Design. (2006-2008)

The SevenPro project will develop technologies and tools supporting deep mining of product engineering knowledge from multimedia repositories and enabling semantically enhanced 3D interaction with product knowledge in integrated engineering environments.
It aims at helping an engineer to design new objects by providing a virtual reality viewing of the object designed, informations on each part of the object (suggestions of other objects with same

properties can then be done) and information about repetitive design processes.
Corese can be used to retrieve information about each object; the approximated search functionality will be very useful to suggest objects with close properties of a given object.

- *Sealife*: A Semantic Grid Browser for the Life Sciences Applied to the Study of Infectious Diseases. (2006-2008)

The Sealife project aims at creating a semantic web browser for the medical environment which allows:
(i) highlighting domain concepts appearing in web pages,
(ii) links to external knowledge sources (FAQs, news, etc.),
(iii) recognition of the user by its navigation (and then a choice of the ontology to use),
(iv) push of knowledge by extending users queries using ontologies, etc.
In addition to the classical Information Retrieval scenario, Corese can be used for its rules, which could define (in combination with the navigation of the user) who the user is (a biologist, a chemist, a doctor..); it could also be used to realize technological watch (with queries -to the ontology chosen- on parent of a given concept).

- *KMP*: Knowledge Management Platform. (2003-2006)

KMP is a Semantic Web Service for the Cartography of Competences at Telecom Valley in Sophia Antipolis. The aim of the KMP project is to increase the portfolio of competences at Telecom Valley in Sophia Antipolis by helping actors in expressing their interests and needs in a common space. The solution relies on the specification, design, building and evaluation of an online customizable semantic web application.
There are two versions of KMP: a territorial one and another developed for Philips.
Corese is used for its semantic web processing capabilities.

Other applications using Corese can be found here: *http://www-sop.inria.fr/acacia/corese /applications.html*.

# Dependencies

Corese is developed with *Java 1.5*.

Annotations are RDF files, ontologies are RDFS or OWL (Lite or DL) files, and rules are .rul files.

We use the following APIs:

- *log4j-1.2.12.jar*: *Log4j* is a Java API used to insert log statements in the code, from *Apache*.
- *notio-1.0.a.jar*: *Notio* is a Java API designed to provide a library for manipulating Conceptual Graphs, from F.Southey.
- *arp-2.0.jar*: *Arp2* is "Another RDF Parser", from *HP*
- *icu4j-2.6.1.jar*: *Icu4j* is a Java API for the normalization, from IBM.
- *xercesImpl-2.8.0.jar*: *xercesImpl* is a Java XML Parser, from *Apache*.
- *xml-apis.jar*: *xml-apis* is used with xercesImpl-2.8.0, from *Apache*.

With the following licenses:

- *INRIA License* (Corese)
- *Apache License* (log4j-1.2.12.jar, xercesImpl-2.8.0.jar and xml-apis.jar)
- *Notio License* (notio-1.0.a.jar)
- *ARP License* (arp-2.0.jar)
- *IBM License* (icu4j-2.6.1.jar)

# Links

In Corese, we use RDFS (or OWL) for the ontology, RDF for annotations and SPARQL as a query language, you can find information about these four standards (SPARQL will soon become a standard, RDF, RDFS and OWL are already ones) on the following W3C documents:

- ***RDF*:** The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web.
- ***RDFS*:** RDF-Schema is a vocabulary description language used to describe RDF.
- ***OWL*:** The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans.
  We implement concepts defined by OWL-Lite and some of OWL-DL (see the *OWL-Corese documentation* for details)
- ***SPARQL*:** SPARQL Query Language is a standardized query language for RDF data.

We also use ***RDF Rules***; they will be explained later in this document.

# How to get corese-standalone.jar

To get the standalone version of Corese, go the *Corese download web page*, agree to the "Free of charge software agreement" by clicking on the "I AGREE" button, after having given your name, e-mail and organization.

On the next page, download "CORESE 2.3.0". Here can also be found licenses under which the software is provided, a set of examples of Corese, the Corese Javadoc, the libraries and the sources of Notio (the Conceptual Graph platform used to design Corese).

# How to load ontologies, annotations, and rules

To start the standalone version of Corese, double click on the JAR (for Windows OS) or use the command "*java -jar CORESE_2007_04_17_v2_3_0.jar*" (for any OS).

Once the application is launched, in the "Logging and messages" part, load an ontology by clicking on the "Load" button and by selecting the ontology (.rdfs file or .owl file) to load. Repeat this for all the ontologies you want to load. Note: it is also possible to load a whole directory in one time.
In the same way it is possible to load annotations (.rdf files) and rules (.rul files).

# How to write queries in SPARQL

SPARQL is an RDF query language designed by the *W3C Data Access Working Group* to easily access to RDF stores.
According to *the SPARQL Working Draft*:
*"The SPARQL query language consists of the syntax and semantics for asking and answering queries against RDF graphs. SPARQL contains capabilities for querying by triple patterns, conjunctions, disjunctions, and optional patterns. It also supports constraining queries by source RDF graph and extensible value testing. Results of SPARQL queries can be ordered, limited and offset in number, and presented in several different forms."*

The *Corese tutorial* explains briefly what RDF, RDFS, and SPARQL are; before giving a tutorial on how to query Corese and how to use rules.

```
PREFIX tutor: <http://inria.fr/2006/tutorial#>
SELECT ?student ?name ?firstname
WHERE {
        ?student tutor:isStudentOf ?univ .
        {
          { ?univ tutor:siteweb "http//www.mit.edu" . }
          UNION
          { ?univ tutor:siteweb "http//www.stanford.edu" . }
        }
        ?student tutor:name ?name .
        OPTIONAL { ?student tutor:firstname ?firstname . }
        ?student tutor:age ?age .
        FILTER ( ?age > 21 )
}
ORDER BY ?name
LIMIT 20
```

*Example of a SPARQL query*

SPARQL is a future semantic web recommendation; that is why we have decided to use it in Corese to query annotations. Here are the main functionalities of SPARQL, used in Corese:

- *Basic query functionalities*
  - *PREFIX, BASE*
  - *FILTER*
  - *OPTIONAL*
  - *UNION*
  - *Blank Nodes*
- *Solution Sequences*
  - *ORDER BY*
  - *Projection*
  - *DISTINCT*
  - *OFFSET*
  - *LIMIT*
- *Result Forms*
  - *SELECT*
  - *CONSTRUCT*

- *DESCRIBE*
- *ASK*

- *RDF Dataset*
  - *FROM, FROM NAMED*
  - *GRAPH*

- *Operators*
  - *SPARQL Operators*
  - *User defined functions*

- *Syntax for Triple Patterns*
  - *Same subject and/or same property*
  - *Collections*

With Corese, we offer several other functionalities that are not in SPARQL; they are described in the next part, *"Additive functionalities"*.

# Additive functionalities

Corese query language is based on *SPARQL*, but it offers in addition some original statements among which approximated search that find best matches according to the RDF Schema, aggregation, path patterns...

## RDFS Entailment

*SPARQL specification* covers *simple Entailment*, but with Corese we cover RDFS-Entailment.

- when one ask for *(?x rdf:type e:Person)*, one will also find someone who is of type *e:Engineer* if it is a subclass of *e:Person*.
- when one ask for *(<uri-1> <P> <uri-2>)*, one will also find properties that use subproperties of *<P>*.
- we also exploit the domain and range of properties.

In Corese, edges that are inferred by a rule have a default source graph: `http://www.inria.fr/acacia/corese#engine`
To get inferred edges, it is then possible to ask:

```
SELECT * WHERE {
        GRAPH <http://www.inria.fr/acacia/corese#engine> { ?x ?p ?y }
}
```

## Approximated search

### Simple approximated search

With Corese, it is possible to ask for an approximated answer.
For example, an approximated search for "a teacher who has written a book" can retrieve "a researcher who has written an article".
Syntactically, the keyword *MORE* in the select clause of a Corese query asks for approximated answers. In this case, Corese basically approximates every concept types of the query.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:personA | e:Name | 'John' |
| e:personA | rdf:type | e:Teacher |
| e:personA | e:hasWritten | e:bookA |
| e:bookA | rdf:type | e:Book |
| | | |
| e:personB | e:Name | 'Lara' |
| e:personB | rdf:type | e:Researcher |
| e:personB | e:hasWritten | e:artB |
| e:artB | rdf:type | e:Article |
| | | |
| e:personC | e:Name | 'Lucio' |
| e:personC | rdf:type | e:MathTeacher |
| e:personC | e:hasWritten | e:artC |
| e:artC | rdf:type | e:Article |
| | | |
| e:MathTeacher | rdfs:subClassOf | e:Teacher |
| e:Teacher | rdfs:subClassOf | e:Person |
| e:Researcher | rdfs:subClassOf | e:Person |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT MORE ?name ?doc WHERE {
```

```
        ?person e:Name ?name .
        ?person rdf:type e:Teacher .
        ?person e:hasWritten ?doc .
        ?doc rdf:type e:Book
}
```

*Query: Approximated search*

| name | doc |
|---|---|
| John | e:bookA |
| Lara | e:artB |
| Lucio | e:artC |

*Query Result*

## Approximated search with some concepts that are specialized

It is possible to require the specialization of some concepts while approximating others by using *type operators*. For instance, by using the <=: operator, Corese is able to retrieve persons who are teachers (or a subclass of teacher) and who have written a book (or something close) by processing the following query.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:personA | e:Name | 'John' |
| e:personA | rdf:type | e:Teacher |
| e:personA | e:hasWritten | e:bookA |
| e:bookA | rdf:type | e:Book |
| | | |
| e:personB | e:Name | 'Lara' |
| e:personB | rdf:type | e:Researcher |
| e:personB | e:hasWritten | e:artB |
| e:artB | rdf:type | e:Article |
| | | |
| e:personC | e:Name | 'Lucio' |
| e:personC | rdf:type | e:MathTeacher |
| e:personC | e:hasWritten | e:artC |
| e:artC | rdf:type | e:Article |
| | | |
| e:MathTeacher | rdfs:subClassOf | e:Teacher |
| e:Teacher | rdfs:subClassOf | e:Person |
| e:Researcher | rdfs:subClassOf | e:Person |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT MORE ?name ?doc WHERE {
        ?person e:Name ?name .
        FILTER ( ?person <=: e:Teacher )
        ?person e:hasWritten ?doc .
        ?doc rdf:type e:Book
}
```

*Query: Approximated search (with a part of the query where the concept is specialized)*

| name | doc |
|---|---|
| John | e:bookA |
| Lucio | e:artC |

*Query Result*

## Score

The keyword *SCORE* allows us to control the level of approximation we want, for every part of the query.
The value 1 corresponds to perfect matches while 0 would correspond to the highest approximation.
We can use *SCORE* with 2 syntaxes:

- *SCORE ?s1 { triple pattern }*: ?s1 represents the score of the triple pattern; it can be used in filters, sorting or everywhere in the query.
- *score()*: The function score() corresponds to the score of the whole query; it can also be used in filters, sorting or everywhere in the query.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:personA | e:Name | 'John' |
| e:personA | rdf:type | e:Teacher |
| e:personA | e:hasWritten | e:bookA |
| e:bookA | rdf:type | e:Book |
| | | |
| e:personB | e:Name | 'Lara' |
| e:personB | rdf:type | e:Researcher |
| e:personB | e:hasWritten | e:artB |
| e:artB | rdf:type | e:Article |
| | | |
| e:personC | e:Name | 'Lucio' |
| e:personC | rdf:type | e:MathTeacher |
| e:personC | e:hasWritten | e:artC |
| e:artC | rdf:type | e:Article |
| | | |
| e:MathTeacher | rdfs:subClassOf | e:Teacher |
| e:Teacher | rdfs:subClassOf | e:Person |
| e:Researcher | rdfs:subClassOf | e:Person |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT MORE ?x ?doc WHERE {
        SCORE ?s1 { ?x rdf:type e:Teacher } .
        SCORE ?s2 { ?doc rdf:type e:Book } .
        ?x e:hasCreated ?doc .
        FILTER ( score() > 0.75 )
        FILTER ( ?s2 > 0.5 )
}
```

*Query: with SCORE*

| name | doc |
|---|---|
| John | e:bookA |
| Lucio | e:artC |

*Query Result*

In the example, we are looking for a teacher who has created a book (or something near); we are more flexible with the fact that the document is a book (?s2 > 0.5), than with the whole result of the query (score() > 0.75).

### Similarity and classSimilarity

The function *similarity* computes the similarity between two concepts; the result is a number between 0 (the highest approximation) and 1 (concept types are equal). If one concept subsumes the other, we still compute the similarity between the two concepts.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:p1 | rdf:type | e:Person |
| e:p2 | rdf:type | e:Engineer |
| e:p3 | rdf:type | e:Researcher |
| e:t1 | rdf:type | e:Truck |
| e:t2 | rdf:type | e:Car |
| | | |
| e:Engineer | rdfs:subClassOf | e:Person |
| e:Researcher | rdfs:subClassOf | e:Person |
| e:Truck | rdfs:subClassOf | e:Vehicle |

| | | |
|---|---|---|
| e:Car | rdfs:subClassOf | e:Vehicle |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT MORE * WHERE {
        ?x rdf:type ?t1 .
        ?y rdf:type ?t2
        FILTER ( ?x != ?y && similarity(?x, ?y) > 0.9 )
}
```

*We are looking for 2 different instances, that have similarity of 0.9.*

| x | t1 | y | t2 |
|---|---|---|---|
| e:p1 | e:Person | e:p2 | e:Engineer |
| e:p2 | e:Engineer | e:p1 | e:Person |
| e:p1 | e:Person | e:p3 | e:Researcher |
| e:p3 | e:Researcher | e:p1 | e:Person |
| e:p2 | e:Engineer | e:p3 | e:Researcher |
| e:p3 | e:Researcher | e:p2 | e:Engineer |
| e:t1 | e:Truck | e:t2 | e:Car |
| e:t2 | e:Car | e:t1 | e:Truck |

*Query Result*

The function *classSimilarity* do the same thing with two classes (and not two instances).

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:p1 | rdf:type | e:Person |
| e:p2 | rdf:type | e:Person |
| e:p2 | rdf:type | e:Engineer |
| e:p2 | rdf:type | e:Actress |
| e:p3 | rdf:type | e:Person |
| e:p3 | rdf:type | e:Dancer |
| e:p3 | rdf:type | e:Researcher |
| | | |
| e:Engineer | rdfs:subClassOf | e:Scientific |
| e:Researcher | rdfs:subClassOf | e:Scientific |
| e:Actress | rdfs:subClassOf | e:Artist |
| e:Dancer | rdfs:subClassOf | e:Artist |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT MORE * WHERE {
        ?x rdf:type e:Person .
        ?x rdf:type ?t1 .
        ?x rdf:type ?t2 .
        FILTER ( classSimilarity(?t1, ?t2) < 0.8 )
}
```

*We are looking for 2 different types (with similarity less than 0.8) for a person*

| x | t1 | t2 |
|---|---|---|
| e:p2 | e:Actress | e:Engineer |
| e:p2 | e:Engineer | e:Actress |
| e:p3 | e:Dancer | e:Researcher |
| e:p3 | e:Researcher | e:Dancer |

*Query Result*

## Select expressions

It is possible to have functions or expressions in the select clause. The syntax is **function(?x) as ?fun**.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:p1 | e:age | '2'^^xsd:integer |

| | | |
|---|---|---|
| e:p1 | e:FamilyName | 'Smith' |
| e:p1 | e:birthdate | '1982-10-10'^^xsd:date |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?p ?y
        datatype(?y) as ?datatype
        (?age + 10) as ?inTenYears
WHERE {
        e:p1 ?p ?y
        e:p1 e:age ?age
}
```

*We want to know the datatype of values linked to e:p1 by a property*

| ?p | ?y | ?datatype | ?inTenYears |
|---|---|---|---|
| e:age | 2 | xsd:integer | 12 |
| e:FamilyName | 'Smith' | xsd:string | 12 |
| e:birthdate | 1982-10-10 | xsd:date | 12 |

*Query Result*

## Distinct variables

In *SPARQL*, the keyword *DISTINCT* is described as follows:
*"[Distinct] ensures that every combination of variable bindings (i.e. each solution) in the sequence is unique"*.
Example: if we have the results aab aab abb aba abb, we will only keep aab, abb, and aba.

With Corese, in addition to the SPARQL keyword *DISTINCT*, we have add the keyword *DISTINCT SORTED*, which keep only distinct values in variable binding sequence. In fact, we first sort the values in the results and then make the distinct.
Example: if we have the results aab aab abb aba abb, we will only keep aab and abb (and not aba, because it is not "distinct sorted" with aab).

| @prefix e: <http://example/of/ontology#> | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| e:p1 | e:FirstName | 'a' | | e:p2 | e:FirstName | 'a' |
| e:p3 | e:FirstName | 'b' | | e:p4 | e:FirstName | 'b' |
| | | | | | | |
| e:p1 | e:hasFriend | e:p2 | | e:p1 | e:hasFriend | e:p3 |
| e:p2 | e:hasFriend | e:p3 | | e:p2 | e:hasFriend | e:p4 |
| e:p3 | e:hasFriend | e:p2 | | e:p3 | e:hasFriend | e:p4 |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT DISTINCT SORTED ?n1 ?n2 ?n3 WHERE {
        ?x e:FirstName ?n1 .
        ?x e:hasFriend ?y .
        ?y e:FirstName ?n2 .
        ?y e:hasFriend ?z .
        ?z e:FirstName ?n3 .
        FILTER (?x != ?y && ?x != ?z && ?y != ?z)
}
```

*DISTINCT SORTED Example*

| n1 | n2 | n3 |
|---|---|---|
| a | a | b |
| a | b | b |

*Query Result: aab, aab, abb, abb, aba, bab => aab, abb*

## Results

## Display results

By default, results are presented as described by the W3C Working Group: *SPARQL Query Results XML Format*.

```
PREFIX e: <http://example/of/ontology#>
SELECT ?doc ?person WHERE {
        ?doc rdf:type e:Document .
        ?doc e:CreatedBy ?person
}
```

*Example of simple query*

```
<results>
        <result>
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-31.html</
                <binding name='person'><uri>http://www.inria.fr/rose.dieng</uri></bindi
        </result>
        <result>
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-31.html</
                <binding name='person'><uri>http://www.inria.fr/equipes/acacia.en.html</
        </result>
        <result>
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-26.html</
                <binding name='person'><uri>http://www.inria.fr/stéphane.lapalut</uri></
        </result>
        <result>
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-33.html</
                <binding name='person'><uri>http://www.inria.fr/equipes/orion.en.html</u
        </result>
</results>
```

*Standard presentation of results (XML presentation)*

The *DISPLAY* clause enables to tune the pretty-printer that generates RDF/XML markup.
It is possible to print in RDF format by using the *DISPLAY RDF* statement.

```
PREFIX e: <http://example/of/ontology#>
SELECT DISPLAY RDF ?doc ?person WHERE {
        ?doc rdf:type e:Document .
        ?doc e:CreatedBy ?person
}
```

*Query with DISPLAY RDF*

```
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:data='file:D:/corese/data/'
   xmlns:e='http://example/of/ontology#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'>
        <e:ResearchReport rdf:about='http://www.inria.fr/rapports/sophia/R-31.html'/>
        <e:Employee rdf:about='http://www.inria.fr/rose.dieng'>
                <rdf:type rdf:resource='http://example/of/ontology#Manager'/>
                <rdf:type rdf:resource='http://example/of/ontology#Researcher'/>
        </e:Employee>
</rdf:RDF>
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:data='file:D:/corese/data/'
   xmlns:e='http://example/of/ontology#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'>
        <e:ResearchReport rdf:about='http://www.inria.fr/rapports/sophia/R-31.html'/>
        <e:ProjectGroup rdf:about='http://www.inria.fr/equipes/acacia.en.html'/>
</rdf:RDF>
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:data='file:D:/corese/data/'
   xmlns:e='http://example/of/ontology#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'>
        <e:ResearchReport rdf:about='http://www.inria.fr/rapports/sophia/R-26.html'/>
```

```
        <e:Person rdf:about='http://www.inria.fr/stéphane.lapalut'/>
</rdf:RDF>
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:data='file:D:/corese/data/'
   xmlns:e='http://example/of/ontology#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'>
        <e:ResearchReport rdf:about='http://www.inria.fr/rapports/sophia/R-33.html'/>
        <e:ProjectGroup rdf:about='http://www.inria.fr/equipes/orion.en.html'/>
</rdf:RDF>
```

*Presentation of results in RDF/S*

Note: with the statements *CONSTRUCT* and *DESCRIBE*, the graphs constructed are presented in RDF format.

### Merge

The keyword *MERGE* merges all elementary solutions (projections) into one result.

```
PREFIX e: <http://example/of/ontology#>
SELECT MERGE ?doc ?person WHERE {
        ?doc rdf:type e:Document .
        ?doc e:CreatedBy ?person
}
```

*Query with MERGE*

```
<results>
        <result>
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-31.html</
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-26.html</
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-33.html</
                <binding name='doc'><uri>http://www.inria.fr/rapports/sophia/R-34.html</
                <binding name='person'><uri>http://www.inria.fr/rose.dieng</uri></bindi
                <binding name='person'><uri>http://www.inria.fr/equipes/acacia.en.html</
                <binding name='person'><uri>http://www.inria.fr/stéphane.lapalut</uri></
                <binding name='person'><uri>http://www.inria.fr/equipes/orion.en.html</u
                <binding name='person'><uri>http://www.inria.fr/régis.vincent</uri></bi
                <binding name='person'><uri>http://www.inria.fr/monique.thonnat</uri></b
                <binding name='person'><uri>http://www.inria.fr/myriam.ribière</uri></bi
        </result>
</results>
```

*Results are merged*

## Aggregation

### Group

In SPARQL, results are presented in a list, but with Corese, it is possible to group the results that have the same value for given variables with the keyword *GROUP*.
It is possible to group on several variables; in this case, we will group first according to the first group, then according to the second one, and so on.

```
@prefix e: <http://example/of/ontology#>
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| e:doc1 | rdf:type | e:Document | | e:doc1 | e:createdBy | e:p1 | | e:doc1 | e:date | '2006-10-10' |
| e:doc2 | rdf:type | e:Document | | e:doc2 | e:createdBy | e:p1 | | e:doc2 | e:date | '2006-10-10' |
| e:doc3 | rdf:type | e:Document | | e:doc3 | e:createdBy | e:p1 | | e:doc3 | e:date | '1999-12-08' |
| e:doc4 | rdf:type | e:Document | | e:doc4 | e:createdBy | e:p2 | | e:doc4 | e:date | '2006-10-10' |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?author ?date ?doc GROUP ?author GROUP ?date WHERE {
```

```
        ?doc rdf:type e:Document .
        ?doc e:createdBy ?author .
        ?doc e:date ?date
}
```

*Example: group documents by author and then by date*

| author | date | doc |
|---|---|---|
| e:p1 | 2006-10-10 | e:doc1, e:doc2 |
| e:p1 | 1999-12-08 | e:doc3 |
| e:p2 | 2006-10-10 | e:doc4 |

*Query Result*

## Count, Sum, Avg, CountItem

These four functions can be applied when results are grouped (if not, it does not make sense).
The function *count* is used to count the number of different values for a variable in a result.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://tania.dupont> | e:FirstName | 'Tania' |
| <http://tania.dupont> | e:hasFriend | <http://aurelie.morel> |
| <http://tania.dupont> | e:hasFriend | <http://susan.trott> |
| <http://tania.dupont> | e:hasFriend | <http://christina.meyer> |
| <http://aurelie.morel> | e:FirstName | 'Aurelie' |
| <http://aurelie.morel> | e:hasFriend | <http://tania.dupont> |
| <http://john.hartman> | e:FirstName | 'John' |
| <http://john.hartman> | e:hasFriend | <http://paul.dupont> |
| <http://john.hartman> | e:hasFriend | <http://aurelie.morel> |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?fname GROUP ?x WHERE {
        ?x e:FirstName ?fname .
        ?x e:hasFriend ?friend .
        FILTER ( count(?friend) > 2 )
}
```

*Select persons who have more than 2 friends*

| fname |
|---|
| Tania |

*Query Result*

Functions *sum* and *avg* can be applied only to numbers (if not, the returned value will be null). They are implemented as described in the document *"XQuery 1.0 and XPath 2.0 Functions and Operators"*.

The function *countItem* is used to count the number of values for a variable in a result. In opposition to the function *count*, *countItem* counts all values for a variable, and not only the different ones.
The function *sum* computes the sum of all values for a variable in a result.
The function *avg* computes the average of all values for a variable in a result.
*avg(?x) = sum(?x) / countItem(?x)*

| @prefix e: <http://example/of/ontology#> | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| e:teamA | rdf:type | e:Team | | e:teamB | rdf:type | e:Team |
| e:teamC | rdf:type | e:Team | | | | |
| | | | | | | |
| e:p1 | e:isMemberOf | e:teamA | | e:p1 | e:age | 30 |
| e:p1 | e:FirstName | 'James' | | | | |
| e:p2 | e:isMemberOf | e:teamA | | e:p2 | e:age | 30 |
| e:p2 | e:FirstName | 'John' | | | | |
| e:p3 | e:isMemberOf | e:teamA | | e:p3 | e:age | 28 |
| e:p3 | e:FirstName | 'Laura' | | | | |

| | | | | | |
|---|---|---|---|---|---|
| e:p4 | e:isMemberOf | e:teamB | e:p4 | e:age | 20 |
| e:p4 | e:FirstName | 'Jennifer' | | | |
| e:p5 | e:isMemberOf | e:teamB | e:p5 | e:age | 22 |
| e:p5 | e:FirstName | 'Alice' | | | |
| e:p6 | e:isMemberOf | e:teamB | e:p6 | e:age | 23 |
| e:p6 | e:FirstName | 'Bianca' | | | |
| e:p7 | e:isMemberOf | e:teamB | e:p7 | e:age | 28 |
| e:p7 | e:FirstName | 'Pierre' | | | |
| e:p8 | e:isMemberOf | e:teamC | e:p8 | e:age | 40 |
| e:p8 | e:FirstName | 'Steve' | | | |
| e:p9 | e:isMemberOf | e:teamC | e:p9 | e:age | 45 |
| e:p9 | e:FirstName | 'Johan' | | | |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?t ?x ?age GROUP ?t WHERE {
        ?t rdf:type e:Team .
        ?x e:isMemberOf ?t .
        ?x e:FirstName ?fname .
        ?x e:age ?age .
        FILTER ( avg(?age) < 30 && sum(?age) > 80 )
}
ORDER BY countItem(?age)
```

*Select teams (with their members and ages of their members) where the average age of the members is less than 30 years and the sum of the ages of the members is more than 80 and sort the results on the number of age per team.*

| t | x | age | countItem(?age) |
|---|---|---|---|
| e:teamA | e:p1, e:p2, e:p3 | 30, 30, 28 | 3 |
| e:teamB | e:p4, e:p5. e:p6, e:p7 | 20, 22, 23, 28 | 4 |

*Query Result*

Functions *sum*, *countItem* and *avg* use all values for a variable in a result.
If there is a duplication, the result is biased, but it is the responsability of the person who asks the query.

For example, if we add the following triple to data, results of the query will be biased, because a person (e:p4) will be counted twice (because he/she has two names).

| | | |
|---|---|---|
| e:p4 | e:FirstName | 'Jenny' |

*Triple added*

| t | x | age | countItem(?age) |
|---|---|---|---|
| e:teamA | e:p1, e:p2, e:p3 | 30, 30, 28 | 3 |
| e:teamB | e:p4, e:p4, e:p5. e:p6, e:p7 | 20, 20, 22, 23, 28 | 5 |

*Query Result*

## Operators in filters

### Type operators

Type comparators enable us to specify constraints on some types in a query: strict specialization (<:), specialization or same type (<=:), same type (=:), generalization or same type (>=:), strict generalization (>:).

For instance, by using the <: operator in the following example, we constrain the document to be a strict specialization of a thesis (e.g. a PhD thesis, a MSc thesis, etc.).

| @prefix e: <http://example/of/ontology#> | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| e:doc1 | rdf:type | e:Document | e:p1 | e:hasWritten | e:doc1 |
| e:doc2 | rdf:type | e:Thesis | e:p2 | e:hasWritten | e:doc2 |
| e:doc3 | rdf:type | e:PhDThesis | e:p2 | e:hasWritten | e:doc3 |

| | | | | | |
|---|---|---|---|---|---|
| e:doc4 | rdf:type | e:MasterThesis | e:p3 | e:hasWritten | e:doc4 |
| e:PhDThesis | rdfs:subClassOf | e:Thesis | | | |
| e:MasterThesis | rdfs:subClassOf | e:Thesis | | | |
| e:Thesis | rdfs:subClassOf | e:Document | | | |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        ?x e:hasWritten ?doc .
        ?doc rdf:type ?type .
        FILTER ( ?doc <: e:Thesis )
}
```

*Ask for a strict specialization of a thesis*

| x | doc | type |
|---|---|---|
| e:p2 | e:doc3 | e:PhDThesis |
| e:p3 | e:doc4 | e:MasterThesis |

*Query Result*

The following example excludes students and their sub-types.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:p1 | rdf:type | e:Student |
| e:p2 | rdf:type | e:MasterStudent |
| e:p3 | rdf:type | e:Engineer |
| e:Student | rdfs:subClassOf | e:Person |
| e:MasterStudent | rdfs:subClassOf | e:Student |
| e:Engineer | rdfs:subClassOf | e:Person |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        ?p rdf:type e:Person .
        ?p rdf:type ?type .
        FILTER (!( ?p <=: e:Student))
}
```

*Example of the negation of a type operator*

| p | type |
|---|---|
| e:p3 | e:Engineer |

*Query Result*

## Other operators

In addition to the *existing operators in SPARQL*, we have defined some other operators specifically for Corese, that applies for string, literals and URIs:

- contains: ~
- begins with: ^

In this query, we are looking for titles that either contains the word 'Potter' or begins with 'Witch'.

| @prefix e: <http://example/of/ontology#> | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| e:book1 | e:title | 'Witches and Wizards'@en | | e:book2 | e:title | 'Harry Potter 2' |
| e:book3 | e:title | 'The Witch Book' | | e:book4 | e:title | 'Harry Potter 1' |
| e:book5 | e:title | 'Everything about Harry Potter' | | e:book6 | e:title | 'Magic' |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?book ?title WHERE {
        ?book e:title ?title .
        FILTER ( (?title ^ 'Witch') || (?title ~ 'Potter') )
}
```

*Example of use of other operator*

| book | title |
|---|---|
| e:book1 | Witches and Wizards |
| e:book2 | Harry Potter 2 |
| e:book4 | Harry Potter 1 |
| e:book5 | Everything about Harry Potter |

*Query Result*

These operators can be useful to manipulate URIs, for example to find every properties that have for prefix "http://www.w3.org/1999/02/22-rdf-syntax-ns#" (which corresponds to the prefix "rdf:" in this manual).

```
SELECT ?p WHERE {
        ?p rdf:type rdfs:Property .
        FILTER ( ?p ^ 'http://www.w3.org/1999/02/22-rdf-syntax-ns#' )
}
```

*Find properties defined in the rdf syntax*

## Paths in graphs

### Oriented paths

A query with an oriented path of given maximum length n between two resources generates n queries (written here as UNION) and stops at the first query that succeeds.
The syntax for oriented path is to add *[* <INTEGER> *]* after the property.
Example: *e:hasFriend[n]* for a path of maximum length n.
We stop the query at the first result found.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://maria> | e:hasFriend | <http://tania> |
| <http://tania> | e:hasFriend | <http://leo> |
| <http://leo> | e:hasFriend | <http://sophie> |
| <http://john> | e:hasFriend | <http://nicolas> |
| <http://john> | e:hasFriend | <http://maria> |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?y WHERE {
        <http://maria> e:hasFriend[2] ?y . FILTER(?y ~ 'o')
}
```

*generates*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?y WHERE {
        { <http://maria> e:hasFriend ?y . FILTER(?y ~ 'o') }
        UNION
        { <http://maria> e:hasFriend ?z . ?z e:hasFriend ?y . FILTER(?y ~ 'o') }
}
```

*Oriented path: we are looking for a friend of Maria or a friend of a friend of Maria*

| y |
|---|
| <http://leo> |

*Query Result*

## Non oriented paths

A query with a non oriented path of given maximum length n between two resources generates 2^(n+1)-2 queries (written here as UNION) and stops at the first query that succeeds.
*?x e:hasFriend{n} ?y* search the paths of maximum length n between ?x and ?y; both (?x e:hasFriend ?y) and (?y e:hasFriend ?x) are valid triples to construct the path.
Note: This statement should be used with care because of the large number of triples that it generates.
We stop the query at the first result found.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://maria> | e:hasFriend | <http://tania> |
| <http://tania> | e:hasFriend | <http://leo> |
| <http://leo> | e:hasFriend | <http://sophie> |
| <http://john> | e:hasFriend | <http://nicolas> |
| <http://john> | e:hasFriend | <http://maria> |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        <http://maria> e:hasFriend{2} ?y FILTER(?y ~ 'o')
}
```

*generates*

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        { <http://maria> e:hasFriend ?y . FILTER(?y ~ 'o') }
        UNION
        { ?y e:hasFriend <http://maria> . FILTER(?y ~ 'o') }
        UNION
        { <http://maria> e:hasFriend ?v . ?v e:hasFriend ?y . FILTER(?y ~ 'o') }
        UNION
        { <http://maria> e:hasFriend ?v . ?y e:hasFriend ?v . FILTER(?y ~ 'o') }
        UNION
        { ?v e:hasFriend <http://maria> . ?y e:hasFriend ?v . FILTER(?y ~ 'o') }
        UNION
        { ?v e:hasFriend <http://maria> . ?v e:hasFriend ?y . FILTER(?y ~ 'o') }
}
```

*Non oriented path*

| y |
|---|
| <http://john> |

*Query Result*

## All

When using paths, we stop by default at the first query that succeeds, but it is possible to test all paths with the property qualifier *all::*. This is possible both for oriented and not oriented paths.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://maria> | e:hasFriend | <http://tania> |
| <http://tania> | e:hasFriend | <http://leo> |
| <http://leo> | e:hasFriend | <http://sophie> |
| <http://john> | e:hasFriend | <http://nicolas> |
| <http://john> | e:hasFriend | <http://maria> |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        <http://maria> all::e:hasFriend{2} ?y FILTER(?y ~ 'o')
}
```

*Usage of the property qualifier all::*

| y |
|---|
| <http://john> |
| <http://leo> |
| <http://nicolas> |

*Query Result*

## Direct

The *direct::* property qualifier can be used either on rdfs:subClassOf or on rdf:type.
When it is used with rdfs:subClassOf, the aim is to limit the query to direct rdfs:subClassOf by ignoring occurrences produced by the transitive nature of rdfs:subClassOf.
It is also possible to use this property with rdf:type to ask the most precise type(s) of ?x.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| e:Thesis | rdfs:subClassOf | e:Document |
| e:MasterThesis | rdfs:subClassOf | e:Thesis |
| e:MasterThesis | rdfs:subClassOf | e:Document |
| e:Engineer | rdfs:subClassOf | e:Person |
| e:p1 | rdf:type | e:Person |
| e:p2 | rdf:type | e:Engineer |
| e:p2 | rdf:type | e:Person |

*Data*

```
SELECT * WHERE {
        { ?class direct::rdfs:subClassOf ?super }
        UNION
        { ?x direct::rdf:type ?class }
}
```

*Usage of the property qualifier direct::*

| x | class | super |
|---|---|---|
| | e:Thesis | e:Document |
| | e:MasterThesis | e:Thesis |
| | e:Engineer | e:Person |
| e:p1 | e:Person | |
| e:p2 | e:Engineer | |

*Query Result*

## Statements

### Describe

The *DESCRIBE* statement is not well defined in *SPARQL*:
*Current conventions for DESCRIBE return an RDF graph without any specified constraints. Future SPARQL specifications may further constrain the results of DESCRIBE, rendering some currently valid DESCRIBE responses invalid. As with any query, a service may refuse to serve a DESCRIBE query.*

With Corese, when we ask for describing the variable ?x (respectively the URI <uri>), we transform the query in a *SELECT MERGE ?x ?p ?v DISPLAY RDF* statement (respectively *SELECT MERGE <uri> ?p ?v DISPLAY RDF*) that contains *{ { ?x ?p ?v } UNION { ?v ?p ?x } }* (respectively *{ { <uri> ?p ?v } UNION { ?v ?p <uri> } }*).

```
        DESCRIBE <uri>
is transformed in
        SELECT MERGE <uri> ?p ?v DISPLAY RDF
        WHERE { { <uri> ?p ?v } UNION { ?v ?p <uri> } }

        DESCRIBE ?x WHERE { ?x rdf:type rdfs:Class }
```

```
is transformed in
        SELECT MERGE ?x ?p ?v DISPLAY RDF
        WHERE { ?x rdf:type rdfs:Class . { { ?x ?p ?v } UNION { ?v ?p ?x } } }
```

*Transformation of the DESCRIBE statement in Corese*

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://tania.dupont> | rdf:type | e:Person |
| <http://tania.dupont> | rdf:type | e:Engineer |
| <http://tania.dupont> | e:FamilyName | 'Dupont' |
| <http://tania.dupont> | e:FirstName | 'Tania' |
| <http://tania.dupont> | e:age | 25 |
| <http://aurelie.morel> | e:hasFriend | <http://tania.dupont> |
| <http://aurelie.morel> | e:age | 28 |

*Data*

```
    DESCRIBE <http://tania.dupont>
```

*Exemple of a describe query*

```
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema#'
   xmlns:e='http://example/of/ontology#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'>
<e:Person rdf:about='http://tania.dupont'>
   <rdf:type rdf:resource='http://example/of/ontology#Engineer'/>
   <e:FamilyName rdf:datatype='http://www.w3.org/2001/XMLSchema#string'>Dupont</e:Family
   <e:FirstName rdf:datatype='http://www.w3.org/2001/XMLSchema#string'>Tania</e:FirstNar
   <e:age rdf:datatype='http://www.w3.org/2001/XMLSchema#integer'>25</e:age>
</e:Person>
<e:Person rdf:about='http://aurelie.morel'>
   <e:hasFriend rdf:about='http://tania.dupont'/>
</e:Person>
</rdf:RDF>
```

*Query Result*

### Construct

The *CONSTRUCT* statement is defined in *SPARQL*
Here are some details about CONSTRUCT in Corese:
It is an error, when a variable is used in the CONSTRUCT while it is not defined in the WHERE clause.
It is not possible to use a property that is undefined in the ontology.

## Projection

It is possible to restrict the maximum number of positive elementary projections with the keyword *PROJECTION*.
A projection is an elementary result, before being (possibly) grouped.

| @prefix e: <http://example/of/ontology#> | | |
|---|---|---|
| | | |
| <http://tania.dupont> | rdf:type | e:Person |
| <http://tania.dupont> | rdf:type | e:Engineer |
| <http://susan.trott> | rdf:type | e:Gardener |
| <http://john.dotty> | rdf:type | e:Person |
| <http://john.dotty> | rdf:type | e:Researcher |
| <http://christina.meyer> | rdf:type | e:Engineer |

*Data*

```
SELECT ?x ?type PROJECTION 4 WHERE {
        ?x rdf:type ?type
}
```

*Search all the possible types, but stop running the query after 4 elementary projections*

| x | type |
|---|---|
| <http://tania.dupont> | e:Person |
| <http://tania.dupont> | e:Engineer |
| <http://susan.trott> | e:Gardener |
| <http://john.dotty> | e:Person |

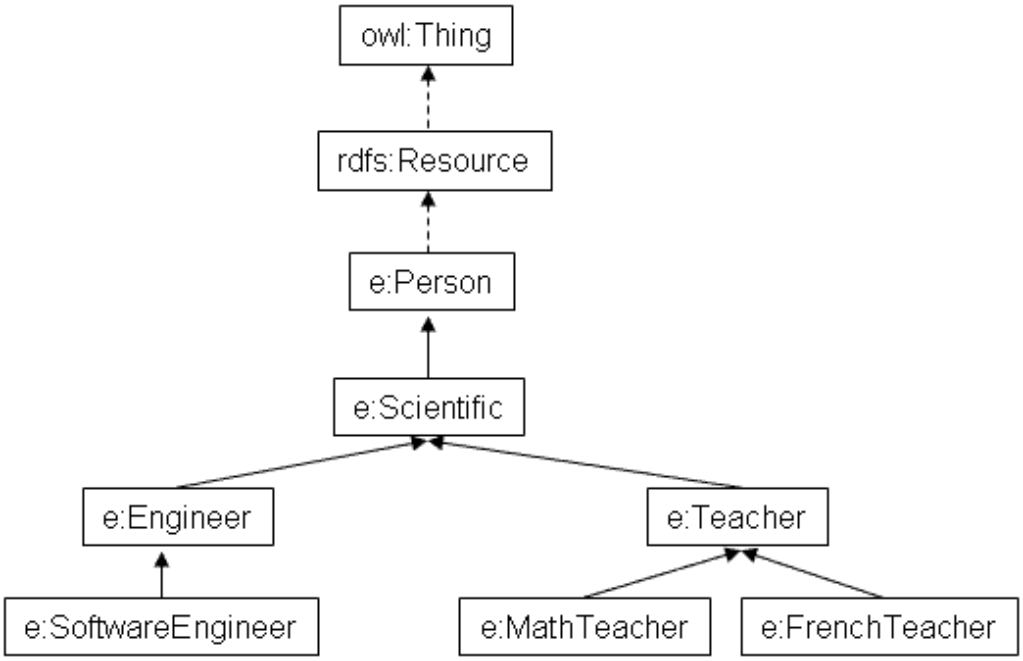
*Query Result*

## Corese functions

For the functions *score*, *similarity* and *classSimilarity*, see the *"Approximated search"* paragraph.
For the functions *count*, *sum*, *avg* and *countItem*, see the *"Aggregation"* paragraph.

### Depth

This function returns the depth of a class in the hierarchy.




*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?class WHERE {
        ?class rdfs:subClassOf e:Person .
        FILTER ( depth(?class) > 4 )
}
```

*Example: Looking for a subclass of e:Person, which is at a depth greater than 4 in the ontology*

| class |
|---|
| e:SoftwareEngineer |
| e:MathTeacher |
| e:FrenchTeacher |

*Query Result*

The depth is relative to the root class.
In Corese, the root class is owl:Thing (depth 0); rdfs:Resource (depth 1) is a subclass of owl:Thing; concepts in the ontology can be at depth 2 or more.

### IsSource

This function is used to ask if a URI is a source (i.e a URI that represents a graph).

| | | | |
|---|---|---|---|
| *# Named graph: http://example/of/graphs/graphA* | | | |
| @prefix e: <http://example/of/ontology#> | | | |
| _:a | e:FirstName | 'Alice' | |
| _:a | e:FamilyName | 'Dupont' | |
| _:a | e:hasFriend | _:b | |
| _:b | e:FirstName | 'Bob' | |
| <http://example/of/graphs/graphA> rdf:type e:Graph | | | |
| | | | |
| *# Named graph: http://example/of/graphs/graphB* | | | |
| @prefix e: <http://example/of/ontology#> | | | |
| _:a | e:FirstName | 'Aline' | |
| _:a | e:hasFriend | _:b | |
| _:b | e:FirstName | 'Bianca' | |
| <http://example/of/graphs/graphB> rdf:type e:Graph | | | |

*Data*

```
SELECT ?x WHERE {
        ?x ?p ?y .
        FILTER ( isSource(?x) )
}
```

*This query looks for sources*

| x |
|---|
| <http://example/of/graphs/graphA> |
| <http://example/of/graphs/graphB> |

*Query Result*

## Cardinality and Occurrence

The function *cardinality* must be used on a property; if not, the query will fail.
It returns the number of occurrences of this property (and its subproperties) in the RDF base.

| | | |
|---|---|---|
| @prefix e: <http://example/of/ontology#> | | |
| | | |
| <http://tania.dupont> | rdf:type | e:Person |
| <http://tania.dupont> | rdf:type | e:Engineer |
| e:teamA | rdf:type | e:Team |
| <http://tania.dupont> | e:FirstName | 'Tania' |
| <http://aurelie.morel> | e:FirstName | 'Aurelie' |
| <http://tania.dupont> | e:FamilyName | 'Dupont' |
| <http://tania.dupont> | e:FamilyName | 'Morel' |
| <http://tania.dupont> | e:age | 25 |
| <http://tania.dupont> | e:hasBestFriend | <http://aurelie.morel> |
| <http://tania.dupont> | e:hasFriend | <http://susan.trott> |
| <http://tania.dupont> | e:hasFriend | <http://christina.meyer> |
| <http://tania.dupont> | e:hasFriend | <http://chris.bilon> |
| <http://aurelie.morel> | e:hasFriend | <http://tania.dupont> |
| <http://tania.dupont> | e:isMemberOf | e:teamA |
| e:hasBestFriend | rdfs:subPropertyOf | e:hasFriend |

*Data*

```
SELECT ?p WHERE {
        ?p rdf:type rdf:Property .
}
ORDER BY DESC ( cardinality(?p) )
```

*This query looks for the properties the most used in the base graph*

| p | cardinality(?p) |
|---|---|
| e:hasFriend | 5 |
| rdf:type | 3 |
| e:FirstName | 2 |
| e:FamilyName | 2 |
| e:age | 1 |
| e:isMemberOf | 1 |

| | |
|---|---|
| e:hasBestFriend | 1 |
| rdfs:subPropertyOf | 1 |

*Query Result*

The function *occurrence* do the same thing but only for the property in argument, NOT its subproperties. With the previous example, with the function *occurrence*, we would have found only 4 occurences of the property `e:hasFriend` instead of 5.

Note: The function *occurrence* also works with classes: when asking for occurrence(?c) with ?c of type rdfs:Class, we retrieve the exact number of instances of the class ?c, without taking its subClasses.

## Other functions

These functions are taken from "*XQuery 1.0 and XPath 2.0 Functions and Operators*".
Report to this document to have the full description of the functions.

### Date functions

- *year*: returns an integer that represents the year of a given date
example: year("1999-05-31") = 1999
- *month*: returns an integer that represents the month of a given date
example: month("1999-05-31") = 05
- *day*: returns an integer that represents the day of a given date
example: day("1999-05-31") = 31

Note: `func-year-from-dateTime`, `func-month-from-dateTime` and `func-day-from-dateTime` have been renamed (respectively) `year`, `month` and `day`.

### String functions

- *contains*: returns a boolean indicating if a value is contained in the other.
example: contains("tattoo", "t") = true
contains("tattoo", "ttt") = false
- *startswith*: returns a boolean to say if a value starts with the other.
example: startswith("edelweiss", "e") = true
startswith("edelweiss", "ew") = false
- *endswith*: returns a boolean to say if a value ends with the other.
example: endswith("edelweiss", "ss") = true
endswith("edelweiss", "e") = false
- *stringlength*: returns an integer that corresponds to the length (in characters) of the value
example: stringlength("edelweiss") = 9
- *substring* Note: The first character of a string is located at position 1, not position 0.
example: substring("motor car", 6) = " car"
substring("metadata", 4, 3) = "ada"
- *substringbefore*
example: substringbefore("tattoo", "attoo") = "t"
substringbefore("tattoo", "tatt") = ""
- *substringafter*
example: substringafter("tattoo", "attoo") = ""
substringafter("tattoo", "tatt") = "oo"

# User defined function in SPARQL queries

## Function Definition

Write a function (example: *isOdd()*) in a class (example: *myClass.java*), in a package (example: *example.of.path*).
In our example, the function's access path will be *example.of.path.myClass.isOdd()*.

Note: The function must have an *"IDatatype"* as a result and zero, one or several *"IDatatype"* as arguments; to make this possible, include CORESE_2007_04_17_v2_3_0.jar in the classpath of the project.
To return a IDatatype, compute the result and use the DatatypeFactory; three ways are possible:

- `return DatatypeFactory.newInstance(result);`
  if result is a String, a boolean, a double, a float, an int, a long or a Date
- `return DatatypeFactory.newInstance(result, "fr");`
  if result is a literal with a language (the second argument represents the language: "fr", "en",...)
- `return DatatypeFactory.newInstance(result, CoreseType.INTEGER);`
  where CoreseType is a java 1.5 enumerated type defined by

```
public static enum CoreseType {STRING, BOOLEAN, XMLLITERAL, DOUBLE, FLOAT,
DECIMAL, INTEGER, LONG, LITERAL, DATE, URI, BNODE};
```

See the *Corese Javadoc* for more documentation about IDatatype.

```
/**
* say if the number is odd or not
* @param dt
* @return true if the argument is odd, false if not
* @throws CoreseDatatypeException
*/
public IDatatype isOdd(IDatatype dt) throws CoreseDatatypeException {
        boolean result = (dt.getDoubleValue() % 2 == 1);
        return DatatypeFactory.newInstance(result);
}
```

*Example of function*

Note: Functions have to be public

After having created the function, compile it and make a jar file.
It is possible to use *ant*, or simply the java command:
*jar -cvf jar_name.jar classes/**
where *jar_name.jar* is the name of the jar and *classes/** indicates where the compiled class is.

## Function Execution

In order to recognize the new jar, add it to the java classpath; either add it to the classpath of the project in the IDE (if a new application based on Corese is being created), or add it to the classpath when launching the Corese jar (*java -classpath "/path/of/the/newFunctionProject.jar;." -jar CORESE_2007_04_17_v2_3_0.jar*)
Note: To execute this command, we have to be in the CORESE_2007_04_17_v2_3_0.jar directory; the dot in the classpath means that we use the jar placed in the current directory (that is to say CORESE_2007_04_17_v2_3_0.jar).

To use the new function in a query, add a prefix, based on the following one (myurl).

```
function:// -> To inform Corese that the function is user defined
```

```
example.of.path.MyClass -> The path of the class where the function is defined

PREFIX humans: <http://www.inria.fr/2006/02/11/humans.rdfs#>
PREFIX myurl: <function://example.of.path.MyClass>
SELECT ?x ?age WHERE {
        ?x humans:age ?age
        filter(myurl:isOdd(?age))
}
```

*Example of a query with a user defined function call*

# RDF rules

The Corese rule language is based on the triple model of RDF and SPARQL. The syntax of a rule is the following.

```
<cos:rule>
        <cos:if>
                RDF Query
        </cos:if>
        <cos:then>
                RDF Pattern
        </cos:then>
</cos:rule>
```

where cos: is the predefined prefix for the Corese namespace (http://www.inria.fr/acacia/corese#) and where the triples correspond to RDF statements whose conjunction is translated into a conceptual graph.

In the *IF* part, it is possible to have prefix, defined like in SPARQL. The prefix is valid in the *THEN* part. It is however possible to define the previous prefix again, to redefine it (with a new URI), or to define a new prefix in the *THEN* part.

The *IF* part follows the SPARQL syntax of the *WHERE* clause, i.e. *GroupGraphPattern*. Hence *OPTIONAL, UNION* and *FILTER* can be used in the IF part.

The *THEN* part follows the SPARQL syntax of *BasicGraphPattern*, hence *OPTIONAL, UNION* and *FILTER* cannot be used in the *THEN* part.

```
<cos:rule>
        <cos:if>
                PREFIX s: &lt;http://example/of/ontology#&gt;
                { ?m rdf:type s:Person .
                ?m s:head ?t .
                ?t rdf:type s:Team .
                ?t s:hasMember ?p .
                ?p rdf:type s:Person }
        </cos:if>
        <cos:then>
                { ?m s:manage ?p }
        </cos:then>
</cos:rule>
```

*Rule example*

Here is the translation of the previous rule:
IF we have a person m (?m rdf:type s:Person) who is at the head (?m s:head ?t) of a team (?t rdf:type s:Team), with another person p (?p rdf:type s:Person) who is a member of this team (?t s:hasMember ?p),
THEN the person m manages the person p (?m s:manage ?p).

Rules are processed in forward chaining mode until saturation of the RDF base. The inference engine stops when nothing new is deduced. The deductions are stored within a default source which is: http://www.inria.fr/acacia/corese#engine. Hence the SPARQL graph statement enables to retrieve deductions of rules.

There are also Constraint Rules that enable to test whether constraints hold within the RDF graph. The condition describes a situation that should not happen, in which case the conclusion infer an error, by means of a predefined cos:Error system class. In this case, an error message is delivered, the resource (here ?m) is *not* typed to cos:Error, i.e. it is syntactic sugar.

```
<cos:rule>
        <cos:if>
                PREFIX s: &lt;http://example/of/ontology#&gt;
                { ?m rdf:type s:Adult .
```

```
                    ?m s:age ?age .
                   filter(?age < 18)}
        </cos:if>
        <cos:then>
                { ?m rdf:type cos:Error }
        </cos:then>
</cos:rule>
```

*Constraint Rule example*

Tip: As the document is an RDF/XML file, pay attention to write:

- < and > as < and > (in URIs and in filters)
- && (in filters) as &&

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE rdf:RDF [
<!ENTITY cos   "http://www.inria.fr/acacia/corese#">
<!ENTITY rdf   "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<!ENTITY s      "http://www.inria.fr/acacia/schema#">
]>

<rdf:RDF   xmlns:rdf="&rdf;"  xmlns:cos="&cos;"  xmlns:s='&s;' >

<cos:rule cos:name='r1'>
        <cos:if>
                PREFIX s: &lt;http://example/of/ontology#&gt;
                { ?m rdf:type s:Person .
                ?m s:head ?t .
                ?t rdf:type s:Team .
                ?t s:hasMember ?p .
                ?p rdf:type s:Person }
        </cos:if>
        <cos:then>
                { ?m s:manage ?p }
        </cos:then>
</cos:rule>

</rdf:RDF>
```

*Complete example*

# Differences with SPARQL

Corese is compliant with SPARQL, except in two cases:

- *OPTIONAL*: Corese executes firstly mandatory triples, and only after, optional part of the query, while in *SPARQL* it is precised that:
*"The OPTIONAL keyword is left-associative : pattern OPTIONAL { pattern } OPTIONAL { pattern } matches the same as { pattern OPTIONAL { pattern } } OPTIONAL { pattern }"*
OPTIONAL is, in Corese, a unary operator; optionals are evaluated later, at the end of the query.
Example: when we have *A . OPTIONAL { B } . C* we reorganize triples as *A . C . OPTIONAL { B }*

| @prefix e: <http://example/of/ontology#> | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| e:x1 | e:p | e:y1 | | e:a1 | e:q | e:y1 | e:x1 | e:q | e:a1 |
| e:x2 | e:p | e:y2 | | e:a2 | e:q | e:y2 | | | |
| e:x3 | e:p | e:y3 | | | | | e:x3 | e:q | e:a3 |

*Data*

```
PREFIX e: <http://example/of/ontology#>
SELECT ?x ?y ?z ?a WHERE {
        ?x e:p ?y .
        OPTIONAL { ?a e:q ?y } .
        ?x e:q ?a .
}
```

*Example of different results with OPTIONAL*

*Corese Result:*

| x | y | a |
|---|---|---|
| e:x1 | e:y1 | e:a1 |
| e:x3 | e:y3 | e:a3 |

*SPARQL Result:*

| x | y | a |
|---|---|---|
| e:x1 | e:y1 | e:a1 |

*Query Result*

- *Combination of OPTIONAL, UNION, and BOUND*: With the syntax *PAT option { PAT1 union PAT2 }*, if a filter *bound()* has for argument a variable from PAT1 and/or PAT2 and if this variable is bound by the option (and not bound before), this combination is not accepted by Corese, and hence, we do not evaluate these queries.
Here is an example:

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        ?x e:p ?y .
        OPTIONAL { { ?x e:q ?z } UNION { ?x e:r ?z } } .
        FILTER ( ! bound(?z) )
}
```

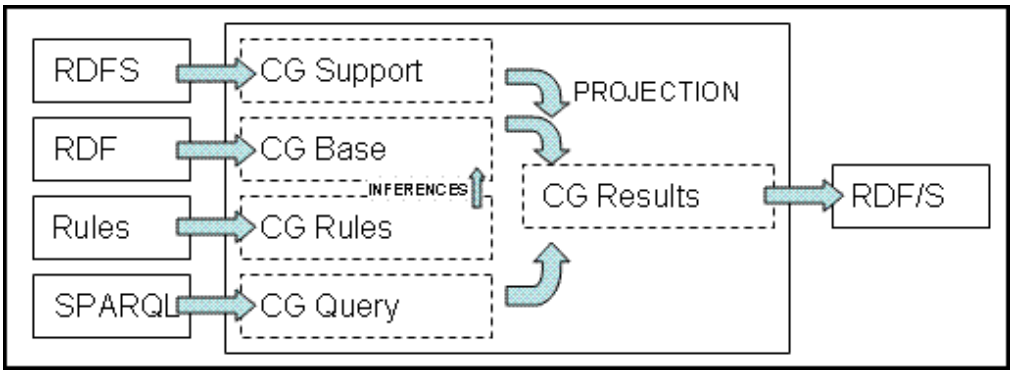
Note: it is possible to write the query in another way to get the results:

```
PREFIX e: <http://example/of/ontology#>
SELECT * WHERE {
        ?x e:p ?y .
        OPTIONAL { ?x e:q ?z1 }
        OPTIONAL { ?x e:r ?z2 }
        FILTER ( !bound(?z1) && !bound(?z2) )
}
```

# Global view of Corese

The processor of the query language is the conceptual graph projection. It takes advantage of RDF Schema by using subsumption relations (rdfs:subClassOf and rdfs:subPropertyOf) and processes datatype values (i.e. not the lexical form). It also exploits property signature (domain and range). The query language can query the RDF schema itself.

The Corese engine internally works on conceptual graphs. When matching a query with an annotation, according to their common ontology, both RDF graphs and their schemas are translated in the conceptual graph model. Through this translation, Corese takes advantage of the existing work of the knowledge representation community leading to reasoning capabilities of this language.




*Corese general principle*

As can be seen in the figure, Corese takes RDFS (ontologies), RDF (annotations), Rules, and SPARQL (queries) in entries. It constructs a conceptual graph with RDFS, RDF, and rules (using inferences). When a query (SPARQL) is submitted to Corese, we also transform it in a conceptual graph and then make projections on the first graph created (the one which represents ontology + annotations + rules). We finally find conceptual graph results, that we transform into RDF/S to pretty print the results.

# Corese API

## Create an instance of the engine

To create an instance of the engine, create an instance of the EngineFactory first.

```
EngineFactory ef = new EngineFactory();
```

It is possible, but not mandatory, to set some properties before creating the instance of IEngine.

Properties that can be set in the EngineFactory are:

- **EngineFactory.PROPERTY_FILE:** This is the name of Corese configuration file; it should be located in the same place as the other data (rdf/s files, owl files, and rul files). More information about the *configuration file* are available further in this document
- **EngineFactory.DATAPATH:** This is also a String; it indicates the path of the directory where data are located. The path must be absolute.

```
Exemple of path with Windows:
        String path = "D:/toto/corese/data";

Exemple of path with Linux:
        String path = "/0/user/toto/corese/data";
```

- **EngineFactory.ENGINE_SCHEMA:** Ontologies to load (RDFS or OWL files or directories that contains rdfs or owl files)
- **EngineFactory.ENGINE_DATA:** Annotations to load (RDF files or directories that contains rdf files)
- **EngineFactory.ENGINE_RULE:** Rules to load
- **EngineFactory.ENGINE_RULE_RUN:** Boolean to say if rules will be run automatically
- **EngineFactory.ENGINE_LOG4J:** The path (absolute or relative) to indicate where to find the log4j configuration file; it can be a .properties or a .xml file

```
ef.setProperty(EngineFactory.PROPERTY_FILE, "corese.properties");
ef.setProperty(EngineFactory.DATAPATH, "D:/toto/corese/data");
ef.setProperty(EngineFactory.ENGINE_LOG4J, "prop/log4j.properties");
ef.setProperty(EngineFactory.ENGINE_SCHEMA,
        "data/humans.rdfs ontology/owlOntology.owl onto");
ef.setProperty(EngineFactory.ENGINE_DATA, "data/humans.rdf");
ef.setProperty(EngineFactory.ENGINE_RULE, "data/humans.rul");
ef.setProperty(EngineFactory.ENGINE_RULE_RUN, "true");
```

*Example of properties settings*

Note: if a property is defined both in the properties file and by the setProperty function, the value defined by the setProperty function will be considered, while the value defined by the properties file will be left aside.

Finally, create an IEngine by using the `newInstance()` method.

```
IEngine engine = ef.newInstance();
```

If you have not specified any files to load (with EngineFactory.ENGINE_SCHEMA, EngineFactory.ENGINE_DATA or EngineFactory.ENGINE_RULE), or if you want to load additional files, it is possible after the creation of the IEngine.
It is also possible to say explicitly when rules have to be executed.

```
try {
        // load only one file (the suffix has to be .rdfs .owl .rdf or .rul)
        engine.load("path/to/the/file/to/load.rdfs");
```

```
        // load a whole directory (only .rdfs .owl .rdf or .rul will be loaded)
        engine.loadDir("path/to/the/directory/to/load");
        // run rules
        engine.runRuleEngine();
} catch (EngineException e) {
        e.printStackTrace();
}
```

Note: paths can be relative or absolute

## Execute the query

To execute the query, use the method SPARQLQuery of the interface IEngine.

```
try {
        IResults res = engine.SPARQLQuery(queryString);
} catch (EngineException e) {
        e.printStackTrace();
}
```

`queryString` is a string which represents the query asked to Corese; `engine` is a IEngine object.
While executing the query, exceptions may occur. They all extend EngineException:

- QueryLexicalException: Lexical exceptions; it is used when errors are detected by the lexical analyser.
  Example: when using the character "\\" which is undefined in the SPARQL grammar
- QuerySyntaxException: This exception is thrown when there is an error in the syntax of the query
  Example: "SELECT * WHERE { ?x ?x rdf:type ?y }"
- QuerySemanticException: This exception is used for semantic errors: (one exception is raised for all the semantic errors that may occur)
  - Undefined prefix
  - Undefined property: in case of a triple of kind "A c:prop B", we check if the property "c:prop" exists
  - Undefined class: in case of a triple of kind "A rdf:type B", we check if B is a defined class
- QuerySolvingException: This exception is thrown during query processing
  Example: When using a function with a wrong number of argument or an undefined function

It is also possible to validate the query before executing it with the function SPARQLValidate:

```
try {
        boolean success = engine.SPARQLValidate(queryString);
} catch (EngineException e) {
        e.printStackTrace();
}
```

## Handling the results

Results can be printed. They will be displayed as asked by the *DISPLAY* statement (in XML SPARQL Result Format by default, except for CONSTRUCT and DESCRIBE where results are printed in RDF format).

```
IResults res = engine.SPARQLQuery(queryString);
System.out.println(res);
```

But it is also possible to access results by means of an API.
It is possible to get variables name and their corresponding value.

```
try {
    // get the results of the query
    IResults res = engine.SPARQLQuery(queryString);
    // get the list of all the selected variables
    String[] variables = res.getVariables();
    // go through all results
    for (Enumeration<IResult> en = res.getResults(); en.hasMoreElements();) {
        // get a result
        IResult r = en.nextElement();
        // go through this result
        for (String var : variables) {
            if (r.isBound(var)) {
                // get result values for each selected variable
                IResultValue[] values = r.getResultValues(var);
                for (int j = 0; j < values.length; j++)
                    System.out.println(var + " = " + values[j].getStringValue());
            } else {
                System.out.println(var + " = Not bound");
            }
        }
    }
} catch (EngineException e) {
    e.printStackTrace();
}
```

To know if there are results, it is possible to use the function *getSuccess()*

```
try {
    IResults res = engine.SPARQLQuery(queryString);
    if (res.getSuccess()) {
        //do something because there is one or more results!
    }
} catch (EngineException e) {
    e.printStackTrace();
}
```

# Properties files

## Corese configuration file: corese.properties

Corese can be configurated with the file corese.properties. Here are explained some of the main properties:

**CORESE PROPERTIES**

| **Input files** |
|---|
| **corese.schema**=ontology/humans.rdfs owlOntology.owl<br>Description: Ontologies to load<br>Values: rdfs files, owl files |
| **corese.data**=annot/data annot/data2 annot/testrdf.rdf<br>Description: Annotations to load<br>Values: rdf files, or directories that contain .rdf files |
| **corese.rule**=rule/testrule.rul<br>Description: Rules to load<br>Values: rul files |
| **corese.rule.run**=true<br>Description: Boolean to say if we run rules automatically (otherwise use the function runRuleEngine() described above)<br>Values: true or false; Default: true |
| **corese.gui.namespace**=a http://www.inria.fr/acacia# i http://www.inria.fr#<br>Description: Predefined prefix associated with a namespace<br>Values: (see above) |
| **Runtime parameters** |
| **corese.debug**=false<br>Description: If true, prints a maximum of information about the inferences it is making<br>Values: true or false; Default: false |
| **corese.log4j.conf**=data/log4j.properties<br>Description: The path (absolute or relative to the launching directory) to indicate where to find the log4j configuration file; it can be a .properties or a .xml file<br>Values: String |
| **corese.source.regex**=.*data2.f.* http://www.inria/acacia/corese/data/fff.rdf/<br>Description: Assign a source URI to a set of URL by means of a regular expression. The triple from the set of URL are assigned the given source URI.<br>Values: String |
| **Result parameters** |
| **corese.result.projection.max**=10000<br>Description: Maximum number of projections computed to answer a query<br>Values: Integer; Default: 10000 |
| **corese.result.max**=100<br>Description: Maximum number of result returned after possibly grouping projections<br>Values: Integer; Default: 100 |
| **corese.result.show.max**=1000<br>Description: Maximum number of statements printed by the RDF pretty printer<br>Values: Integer; Default: 1000 |
| **corese.result.join**=false<br>Description: When this property is set to true, Corese groups projections that share the same first concept into one result.<br>Values: true or false; Default: false |

Note: Corese can run without corese.properties; properties then take their default values.
A complete example of corese.properties can be found *here*.
Paths given (in the three first properties for example), are relative to the launching directory (i.e where Corese starts) and to the datapath.

## Logger configuration file: log4j.properties or log4j.xml

Corese uses the *log4j logging system*, from Apache.

- First, we look if there is a file named log4j.xml or log4j.properties at the root level; if there is one, we configure the logger with it
- Else, we look if the properties "corese.log4j.conf" is set (by the file corese.properties or by the function setProperty(EngineFactory.ENGINE_LOG4J, "...") of the EngineFactory interface), and we configure the logger with the file found there
- Finally, if no configuration file is found, we take the Corese default configuration (Level = INFO; Appender = ConsoleAppender)

When using the jar in an application, it is possible to get the logger and then to define an appender (where messages will be displayed), a layout (the format of displayed messages) or a level (between TRACE, DEBUG, INFO, WARN, ERROR and FATAL).

```
// get the default logger, from whom every other logger herits
Logger logger = Logger.getLogger("fr.inria.acacia.corese");

// remove appenders
logger.removeAllAppenders();

// define a new Appender (example: a WriterAppender)
WriterAppender wa = new WriterAppender(new PatternLayout("%m%n") , System.out);

// add this appender to the default logger
logger.addAppender(wa);
```

*Example of a redefinition of the root logger*

Log4j javadoc can be found here: *http://logging.apache.org/log4j/docs/api/index.html*.

# Corese Grammar

Corese grammar is inspired by *SPARQL Grammar*, but as mentioned in the part *additive functionalities*, we have added some other functionalities; moreover, to try to be compliant with the last query language of Corese, we are more flexible with some aspects (for example, it is not mandatory to have a dot between triples).