

ESSLLI 2012

Ontology-based Interpretation of Natural Language

August 17,
2012

Application: Ontology-based question answering

Philipp Cimiano · Christina Unger

Semantic Computing Group

CITEC, Bielefeld University

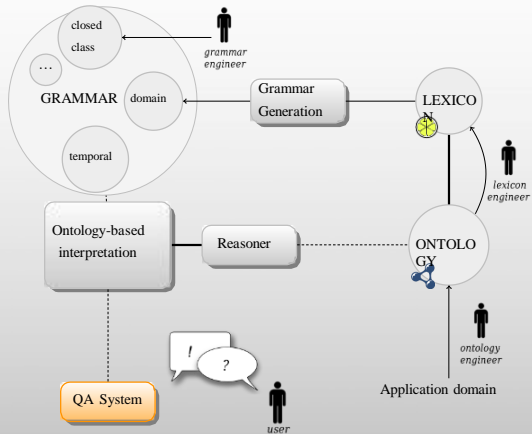
Edited by Manuel Fiorelli (fiorelli@info.uniroma2.it)

CITEC

**These slides are not the
original ones produced by
prof. Philipp Cimiano.**

**They have been edited by
Manuel Fiorelli
(fiorelli@info.uniroma2.it).**

Today



Question answering

Question answering is the task of automatically retrieving an answer to a natural language question.

- I Who was the first team to win the world championship?
- I Give me all Polish players from the 90s.
- I How many hexagons are on a soccer ball?

Sources:

- I unstructured data (newspaper articles, websites, etc.)
- I structured data (databases, Linked Data)

Question answering over RDF data

For a given natural language question, construct a SPARQL query that retrieves the answers from an RDF repository.

QA over RDF is relevant, because it provides an *intuitive interface* to data:

- Users express their information need in their own terms;
- Users do not have to know underlying schema, vocabulary or query language

Example

Which referees **arbitrated** the game Sweden against France?



```
1 SELECT ?ref WHERE {  
2   ?m rdf:type soccer:Match .  
3   ?m soccer:team soccer:Sweden .  
4   ?m soccer:team soccer:France .  
5   ?ref rdf:type Person .  
6   ?ref soccer:role ?r .  
7   ?r rdf:type soccer:RefereeRole .  
8   ?r soccer:match ?m .  
9 }
```



<http://sc.cit-ec.uni-bielefeld.de/essli2012/soccer#PedroProenca>

<http://sc.cit-ec.uni-bielefeld.de/essli2012/soccer#BertinoMiranda>

<http://sc.cit-ec.uni-bielefeld.de/essli2012/soccer#RicardoSantos>

Outline

Querying RDF data with
SPARQL

Ontology-based question answering

Other approaches

Outline

Querying RDF data with
SPARQL

Ontology-based question answering

Other approaches

SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is

- I a query language for RDF as well as a protocol
- I the current W3C recommendation
 - I <http://www.w3.org/TR/sparql11-query/>
 - I <https://www.w3.org/TR/sparql11-protocol/>

SPARQL 1.1 Update is an update language for RDF graphs. Its syntax was derived from the SPARQL RDF Query Language

- I the current W3C recommendation
 - I <https://www.w3.org/TR/sparql11-update/>

SPARQL

SPARQL is based on matching graph patterns against RDF graphs.

Example:

- 1 ?x rdf:type soccer:Goal .
- 2 ?x soccer:inGame soccer:Game7324 .
- 3 ?x soccer:team soccer:Austria .

Variables can be written as `?x` or `$x`, and can occur in subject, predicate and object position.

Example

Data:

- 1 soccer:Goal1107 rdf:type soccer:Goal .
- 2 soccer:Goal1107 soccer:match soccer:Game7324 .
- 3 soccer:Goal1107 soccer:team soccer:Austria .

Query:

```
1 SELECT ?x WHERE {  
2   ?x rdf:type soccer:Goal .  
3   ?x soccer:match soccer:Game7324 .  
4   ?x soccer:team soccer:Austria .  
5 }
```

Result:

<http://sc.cit-ec.uni-bielefeld.de/ontologies/soccer#Goal1107>

Query types

- I **SELECT queries** return variable bindings (mappings from the set of variables to the set of RDF terms, in SPARQL Query Results XML Format).
- I **ASK queries** return a Boolean value. They test whether or not a graph pattern has an instantiation.
- I **CONSTRUCT queries** return RDF graphs specified by a graph template.

Simple ASK query

Data:

- 1 soccer:Goal1107 rdf:type soccer:Goal .
- 2 soccer:Goal1107 soccer:inGame soccer:Game7324 .
- 3 soccer:Goal1107 soccer:forTeam soccer:Austria .

Query:

```
1 ASK WHERE {  
2   ?x rdf:type soccer:Goal .  
3   ?x soccer:match soccer:Game7324 .  
4   ?x soccer:team soccer:Austria .  
5 }
```

Result: true

Filter conditions

Filter conditions restrict variable bindings to those for which the FILTER expression evaluates to true.

Example:

```
1 SELECT ?x WHERE {  
2   ?x rdf:type  soccer:Goal .  
3   ?x soccer:match  soccer:Game7324 .  
4   ?x soccer:atMinute  ?m .  
5   FILTER (?m < 10)  
6 }
```

Filter conditions

Filter conditions can be composed by the Boolean operators `&&` and `||`.

Example:

```
1 SELECT ?x WHERE {  
2   ?x rdf:type  soccer:Goal .  
3   ?x soccer:match  soccer:Game7324 .  
4   ?x soccer:byPlayer  ?p .  
5   ?p soccer:name  ?name .  
6   FILTER (regex(?name,'Janko','i')  
7           && !sameTerm(?p,soccer:MarcJanko)  
8 }
```

Notes on sameTerm (1/2)

xsd:boolean sameTerm (RDF term term1, RDF term term2)

Returns TRUE if term1 and term2 are the same RDF term as defined in Resource Description Framework (RDF): Concepts and Abstract Syntax; returns FALSE otherwise.

xsd:boolean RDF term term1 = RDF term term2

Returns TRUE if term1 and term2 are the same RDF term as defined in Resource Description Framework (RDF): Concepts and Abstract Syntax [CONCEPTS]; produces a type error if the arguments are both literal but are not the same RDF term *; returns FALSE otherwise.

Notes on sameTerm (2/2)

The behaviour of the operator = allows supplying extensions that perform the comparison based on the value represented by the arguments (based on their datatype).

In RDF 1.0, they are literals without datatype and optionally with a language tag

By default, SPARQL interprets plain literals, string, numeric, boolean and datetime.

The downside is that

`«XI»^^my:romanNumeral = «IX»^^my:romanNumeral`

`«XI»^^my:romanNumeral != «IX»^^my:romanNumeral`

Both produces an error; unless it has been loaded an extension for my:romanNumeral

Alternatively, use sameTerm and !sameTerm

Optional graph patterns

- I **Optional graph patterns** serve to add information to the answer where it is available.
- I If such a pattern does not match, no variable binding is created, instead of rejecting the whole instantiation.

Example

Query:

```

1 SELECT ?x ?y WHERE {
2   ?x rdf:type  soccer:Goal .
3   ?x soccer:match  soccer:Game7324 .
4   OPTIONAL{ ?y rdf:type  soccer:PenaltyKick .
5              ?y soccer:leadsTo  ?x .}
6 }

```

Result:

x	y
soccer:Goal1042	
soccer:Goal1043	soccer:PenaltyKick807
soccer:Goal1044	

Alternative graph patterns

The construct **UNION** can be used to build disjunction of graph patterns so that one of several alternative graph patterns may match. In this case, all possible instantiations are returned as answer.

Example:

```
1 SELECT ?x WHERE {  
2   ?x rdf:type  soccer:Goal .  
3   ?x soccer:match  soccer:Game7324 .  
4   ?y soccer:leadsTo ?x .  
5   { ?y rdf:type  soccer:FreeKick .}  
6   UNION  
7   { ?y rdf:type  soccer:CornerKick .}  
8 }
```

Nesting constructs

Example:

```
1 SELECT ?x WHERE {  
2   ?x soccer:match soccer:Game7324 .  
3   ?x rdf:type soccer:Goal .  
4   { ?x soccer:atMinute ?m .  
5     FILTER (?m >= 80) }  
6   UNION  
7   { ?y rdf:type soccer:PenaltyKick .  
8     ?y soccer:leadsTo ?x .  
9   }  
10 }
```

Solution modifiers

I ORDER BY (ASC/DESC)

I PROJECTION

I DISTINCT

I REDUCED

I OFFSET

I LIMIT

Example:

Players that scored a goal (in any match)

```
1 SELECT ?p WHERE {  
2   ?a rdf:type soccer:SoccerAction .  
3   ?a soccer:byPlayer ?p .  
4   ?a soccer:leadsTo ?g .  
5   ?g rdf:type soccer:Goal .  
6 }
```

A player who scored more than one goal is returned multiple times

Solution modifiers

I ORDER BY (ASC/DESC)

I PROJECTION

I DISTINCT

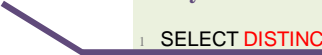
I REDUCED

I OFFSET

I LIMIT

Example:

Players that scored a goal (in any match)



```
1 SELECT DISTINCT ?p WHERE {  
2   ?a rdf:type soccer:SoccerAction .  
3   ?a soccer:byPlayer ?p .  
4   ?a soccer:leadsTo ?g .  
5   ?g rdf:type soccer:Goal .  
6 }
```

DISTINCT removes duplicate solutions

REDUCED permits that duplicate solutions are eliminated

Solution modifiers

- I ORDER BY (ASC/DESC)
- I PROJECTION
- I DISTINCT
- I REDUCED
- I OFFSET
- I LIMIT

Example:

Players that scored a goal (in any match) ordered by last and first name. Discard the first ten players and return at most five of the remaining.

```
1 SELECT DISTINCT ?p WHERE {  
2   ?a rdf:type soccer:SoccerAction .  
3   ?a soccer:byPlayer ?p .  
3   ?p soccer:lastName ?ln .  
4   ?p soccer:firstName ?fn .  
5   ?a soccer:leadsTo ?g .  
6   ?g rdf:type soccer:Goal .  
7 }  
8 ORDER BY ASC(?ln) ?fn ?p  
9 OFFSET 10  
10 LIMIT 5
```

Necessary for a predictable order in case of homonymous players

ASC is the default

The combined use of OFFSET and LIMIT allows to select a “slice” of the solution sequence. ORDER BY is necessary to predictably order the solutions.

Aggregates

- I Aggregates apply expressions over groups of solutions
- I By default, all solutions belong to a single group
- I GROUP BY introduces one or more expressions to group the solutions

Example:

Players and the number of goals they scored

```
1 SELECT ?p (COUNT(?x) as ?c) WHERE {  
2   ?x rdf:type  soccer:Goal .  
3   ?x soccer:byPlayer  ?p .  
4 }  
5 GROUP BY ?p
```

A select expression must be a variable in the GROUP BY or an aggregate

Aggregates

- I Aggregates apply expressions over groups of solutions
- I By default, all solutions belong to a single group
- I GROUP BY introduces one or more expressions to group the solutions

Example:

Players that scored at least five goals

```
1 SELECT ?p WHERE {  
2   ?x rdf:type  soccer:Goal .  
3   ?x soccer:byPlayer ?p .  
4 }  
5 GROUP BY ?p  
6 HAVING COUNT(?x) > 5
```

HAVING introduces one or more constraints on solution groups. In this context, expressions are subject to the same constraints as in the SELECT

Aggregates

- I Aggregates apply expressions over groups of solutions
- I By default, all solutions belong to a single group
- I GROUP BY introduces one or more expressions to group the solutions

Example:

The top 10 players and the number of goals they scored

```
1 SELECT ?p (COUNT(?x) as ?c) WHERE {  
2   ?x rdf:type soccer:Goal .  
3   ?x soccer:byPlayer ?p .  
4 }  
5 GROUP BY ?p  
6 ORDER BY DESC(?c)  
7 LIMIT 10
```

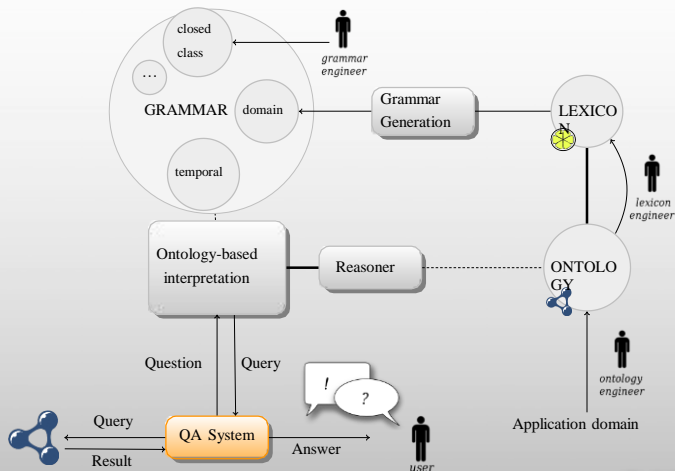
Outline

Querying RDF data with
SPARQL

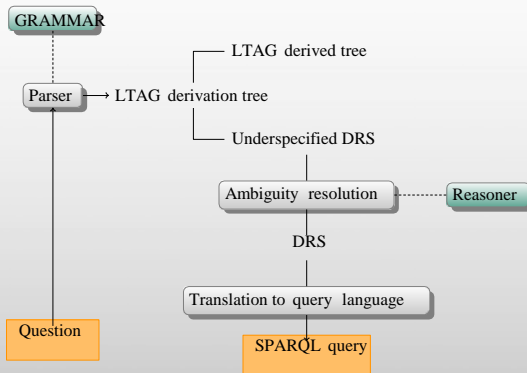
Ontology-based question answering

Other approaches

QA system

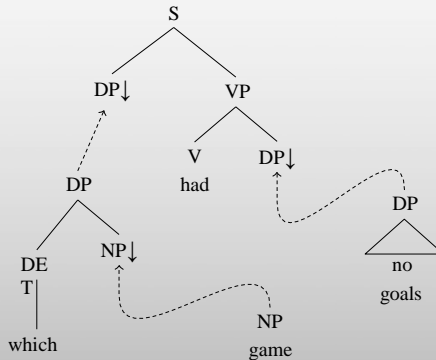


In more detail



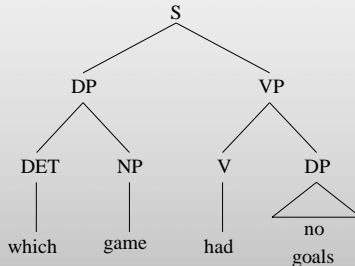
Example

- I Parsing along the lines of the Earley-type parser devised by Schabes & Joshi (1988) results in an LTAG **derivation tree**.



Example

- I Parsing along the lines of the Earley-type parser devised by Schabes & Joshi (1988) results in an LTAG **derivation tree**.
- I Next, syntactic and semantic composition rules apply in tandem in order to construct an LTAG **derived tree**...

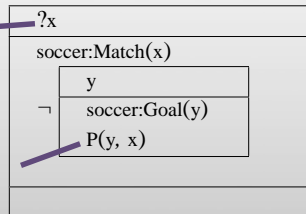


Example

- I Parsing along the lines of the Earley-type parser devised by Schabes & Joshi (1988) results in an LTAG **derivation tree**.
- I Next, syntactic and semantic composition rules apply in tandem in order to construct an LTAG **derived tree** and a **DUDE**.

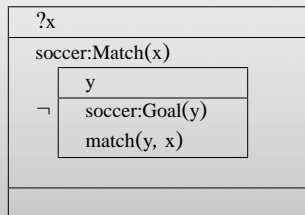
Marked discourse referent
introduced by the meaning
representation of Wh-forms

metavariable
originating from the fact that
“Had” denotes an underspecified
relation



Example

- I Parsing along the lines of the Earley-type parser devised by Schabes & Joshi (1988) results in an LTAG **derivation tree**.
- I Next, syntactic and semantic composition rules apply in tandem in order to construct an LTAG **derived tree** and a **DUDE**. Disambiguation then
- I yields a **Discourse Representation Structure**...



Example

- I Parsing along the lines of the Earley-type parser devised by Schabes & Joshi (1988) results in an LTAG **derivation tree**.
- I Next, syntactic and semantic composition rules apply in tandem in order to construct an LTAG **derived tree** and a **DUDE**.
- I Disambiguation then yields a **Discourse Representation Structure...**
- I ...which is finally translated into a **SPARQL query**.

```

1 SELECT ?x WHERE {
2   ?x a soccer:Match .
3   optional {
4     ?y a soccer:Goal .
5     ?y soccer:match ?x .
6   }
7   FILTER !BOUND(?y)
8 }
```

!BOUND(?y) is true iff ?y is not bound to an RDF term (i.e. the optional graph pattern did not match)



Implementation of the quantifier *no* in a negation-as-failure fashion



Negation-as-failure in SPARQL 1.1

SPARQL 1.0

```
1 SELECT ?x WHERE {  
2   ?x a soccer:Match .  
3   OPTIONAL {  
4     ?y a soccer:Goal .  
5     ?y soccer:match ?x .  
6   }  
7   FILTER !BOUND(?y)  
8 }
```

SPARQL 1.1

```
1 SELECT ?x WHERE {  
2   ?x a soccer:Match .  
3   FILTER NOT EXISTS {  
4     ?y a soccer:Goal .  
5     ?y soccer:match ?x .  
6   }  
7 }
```

FILTER EXISTS in SPARQL 1.1

SPARQL 1.0

Players that scored a goal (in any match)

```
1 SELECT DISTINCT ?p WHERE {  
2   ?a rdf:type soccer:SoccerAction .  
3   ?a soccer:byPlayer ?p .  
4   ?a soccer:leadsTo ?g .  
5   ?g rdf:type soccer:Goal .  
6 }
```

SPARQL 1.1

Players that scored a goal (in any match)

```
1 SELECT ?p WHERE {  
2   ?p rdf:type soccer:Person .  
3   FILTER EXISTS {  
4     ?a soccer:byPlayer ?p .  
5     ?a soccer:leadsTo ?g .  
6     ?g rdf:type soccer:Goal .  
7   }
```

FILTER EXISTS {...} tests whether the given pattern matches. It does not generate additional bindings.

Examples: more than

Which stadium has more than 30,000 seats?

```
SELECT ?x WHERE {  
  ?x a soccer:Stadium .  
  ?x soccer:capacity ?y .  
  FILTER (?y > 30000)  
}
```

soccer:capacity is a
datatype property

Which team won more than two games?

```
SELECT ?x WHERE {  
  ?x a soccer:Team .  
  ?y a soccer:Match .  
  ?y soccer:winner ?x .  
}  
GROUP BY ?x  
HAVING (COUNT(?y) > 2)
```

We count the subjects
of the property
soccer:winner

Examples: the most

Which stadium has the most seats?

```
SELECT ?x WHERE {  
    ?x a soccer:Stadium .  
    ?x soccer:capacity ?y .  
}  
ORDER BY DESC(?y)  
LIMIT 1
```

soccer:capacity is a
datatype property

Which team won the most games?

```
SELECT ?x WHERE {  
    ?x a soccer:Team .  
    ?y a soccer:Match .  
    ?y soccer:winner ?x .  
}  
GROUP BY ?x  
ORDER BY DESC(COUNT(?y))  
LIMIT 1
```

We count the
subjects of the
property
soccer:winner

Pythia

Demo:

- I <http://greententacle.techfak.uni-bielefeld.de/~cunger/pythia/>

Datasets:

- I Geobase
- I MusicBrainz
- I (DBpedia)

Unger, Christina, and Philipp Cimiano. *"Pythia: Compositional meaning construction for ontology-based question answering on the Semantic Web."* Natural Language Processing and Information Systems. Springer Berlin Heidelberg, 2011. 153-160.
<http://pub.uni-bielefeld.de/download/2278529/2674859>

Geobase

Geobase (Raymond Mooney et al.) comprises geographical information about the U.S. ...

- I **Classes:** state, city, road, mountain, lake, river
- I **Relations:** borders, inS tate, flows Through, population etc.

...together with 880 user questions annotated with queries.

- I How many people live in New Mexico?
- I How many states border Alaska?
- I What is the largest city in Texas?
- I How long is the longest river in California?
- I How many cities named Austin are there in the USA?

Geobase: original predicates in Prolog

state(name, abbreviation, capital, population, area,
state_number, city1, city2, city3, city4)

city(state, state_abbreviation, name, population)

river(name, length, [states through which it flows])

border(state, state_abbreviation, [states that border it])

highlow(state, state_abbreviation, highest_point,
highest_elevation, lowest_point, lowest_elevation)

mountain(state, state_abbreviation, name, height)

road(number, [states it passes through])

lake(name, area, [states it is in])

Results



Pythia has been evaluated on 865 of the 880 questions originally associated with Geobase.

The 15 discarded questions are out of the scope of the ontology: e.g. Which rivers do not run through USA? Remember: the dataset only deals with US geography

Pythia could process 624 of 865 questions.

I Recall: 67 %

I Precision: 82 %

Pythia-external failures

I Syntactically ill-formed questions

- I What is capital of Iowa?
- I What are the capital city in Texas?

I Semantically ill-formed questions

- I Which states **border** the Missouri river?

In the ontology, states
only border other states

I Data incompleteness

- I `highest_point` \equiv location with maximum height

The dataset explicitly uses `highest_point`.
Although it is extensionally equivalent to “location with maximum height”, the absence of data (e.g. the highest location is not described in the dataset) may cause erroneous results.

Pythia-internal failures

I Incomplete coverage (lexical and structural)

- I Of the states **washed by** the Mississippi river, which has the lowest point?
- I How many states have cities **or** towns named Springfield?

I Non-compositionality

- I How many cities are there **in the US**?
- I Which river flows through **the most states**?



DBpedia extracts structured information from Wikipedia and makes it available as RDF data.

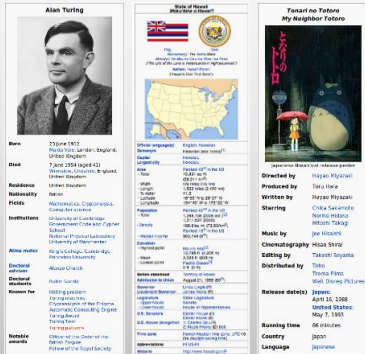
DBpedia ontology:

- I shallow, cross-domain ontology

I manually created based on the most commonly used infoboxes within Wikipedia

I 320+ classes, 1650 properties

<http://wiki.dbpedia.org/Ontology>



Outline

Querying RDF data with
SPARQL

Ontology-based question answering

Other approaches

Other approaches to question answering over Linked Data

The larger the domain, the less feasible is the manual creation of an ontology lexicon.

Task:

Map natural language expressions to ontology concepts without a domain-specific grammar.

Major challenges

- I **Lexical gap**: the gap between natural language expressions and ontology labels.
 - I game \rightarrow soccer:match
 - I victorious \rightarrow soccer:winner

Major challenges

I **Lexical gap**: the gap between natural language expressions and ontology labels.

I game \rightarrow soccer:match

I victorious \rightarrow soccer:winner

I **Structural gap**: the gap between the semantic structure of the natural language question and the structure of the data

I Who is the coach of Denmark?

```
1 SELECT ?p WHERE {  
2   ?p soccer:role ?r .  
3   ?r rdf:type soccer:CoachRole .  
4   ?r soccer:team soccer:Denmark .  
5 }
```