Notes: INF3580

Veronika Heimsbakk
veronahe@ifi.uio.no

June 3, 2015

Contents

Introduction

This is my notes for repetition in the course INF3580–Semantic technologies in front of the exam. This document is based on lectures, own notes and the course books.

1 Resource Description Framework

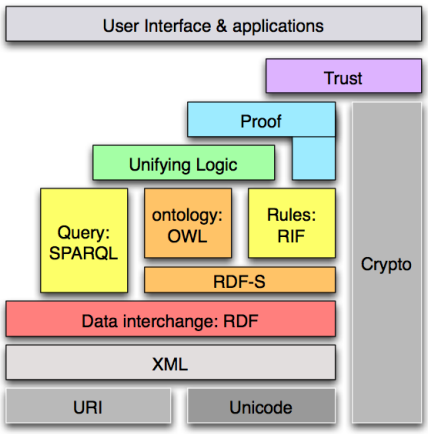


Figure 1: The semantic web stack

An Resource Description Framework (RDF) document describes a *directed graph*. Both nodes and edges are labeled with identifiers to distinguish them. RDF was intended to serve as a description language for data on the WWW, and being a graph structure—it's easy to combine RDF data from multiple sources.

All information in RDF is expressed using a *triple* pattern. A triple consist of a *subject*, a *predicate* and an *object*.

subject	predicate	object
Norway	has capital	Oslo
Oslo	has mayor	Fabian Stang
Fabian Stang	born year	1955

A another word for an RDF triple is a *statement* or *fact*.

1.1 URIs

RDF uses *Uniform Resource Identifiers* (URIs) as **names** to clearly distinguish resources from each other. URIs that are not URLs are sometimes called *Uniform Resource Name* (URNs). Details of protocols (as `http`) are not relevant when using URI only as a name, as in RDF. RDF allow encoding of data which are not URIs, and it features **blank nodes** which do not carry any name. Data values in RDF are represented by **literals**.

Almost everything is a *resource*. Resources are identified by URIs. For example `dbpedia.org`:

- Norway: `http://dbpedia.org/resource/Norway`

- has capital: `http://dbpedia.org/ontology/capital`

In RDF-syntax (with prefixes) the triple "Norway has capital Oslo" will be.

```
@prefix dbp:  <http://dbpedia.org/resource/>
@prefix dbp-ont:  <http://dbpedia.org/ontology/>

dbp:Norway dbp-ont:capital dbp:Oslo .
```

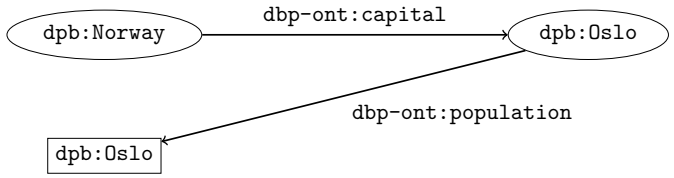
1.2 Literals

Data values are specified by a literal string in quotation marks, followed by `^^` and the URI of the datatype. Language information can be provided as semantically untyped literals. Supplied by means of the symbol `@`.

1.3 RDF graphs

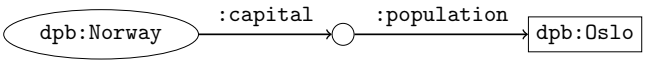
An RDF graph is a graph containing of at least two triples.

```
dbp:Norway dbp-ont:capital    dbp:Oslo .
dbp:Oslo   dbp-ont:population "629313"^^xsd:integer .
```



1.4 Blank nodes

RDF allow us to introduce nodes without any URI, called *blank nodes*. This feature is only available for subject and object of RDF triples. Blank nodes cannot be addressed globally by means of URIs, and they do not carry additional information within RDF graphs.



1.5 Vocabularies

Families of related notions are grouped into *vocabularies*. Usually the same namespace or prefix is shared. A well-known namespace (and prefix) is for example `rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`

A description is usually published at the namespace base URI. The prefix is not standardized, so you may give the prefix `rdf:<http://xmlns.com/foaf/0.1/>`. However, this would be highly irregular.

1.5.1 Examples with RDF and RDFS

```
rdf:Statement  rdfs:Class
rdf:subject    rdfs:subClassOf
               rdfs:subPropertyOf
rdf:predicate  rdfs:domain
rdf:object     rdfs:range
rdf:type       rdfs:label
```

Example of use

```
dbp:Oslo   rdf:type   dpt-ont:Place .
dbp:Norway rdfs:label "Norge"@no .
```

1.6 RDF Serializations

There are several serializations for the RDF data model.

RDF/XML this is the W3C standard.

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/">

  <foaf:Person rdf:about="http://example.org/ifi/veronahe">
  </foaf:Person>
</rdf:RDF>
```

Turtle is convenient, human readable and writable. This is used in the course.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix ifi: <http://example.org/ifi/>

ifi:veronahe rdf:type foaf:Person .
```

N-triples one triple per line. No abbreviations.

```
<http://example.org/ifi/veronahe> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
```

1.7 Turtle

1.7.1 URI references

URIs are surrounded by < and >.

- `<http://dbpedia.org/resource/Oslo>`

Statements are triples terminated by a period

```
<http://dbpedia.org/resource/Oslo>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://dbpedia.org/ontology/Place> .
```

Use `a` to abbreviate `rdf:type`

```
<http://dbpedia.org/resource/Oslo> a <http://dbpedia.org/ontology/Place> .
```

1.7.2 Namespaces

Namespace prefixes are declared with `@prefix`, may also declare default namespaces.

```
@prefix dbp: <http://dbpedia.org/resource/> .
@prefix : <http://dbpedia.org/ontology/> .

dbp:Oslo a :Place .
```

1.7.3 Literals

Literals are enclosed in double quotes `" ... "`. Possibly with type or language information. Numbers and booleans may be written without quotes.

```
dbp:Oslo   :officialName  "Oslo" .
dbp:Norway rdfs:label     "Norge"@no .
dbp:Oslo   :population    "629313"^^xsd:integer .
dbp:Oslo   :population    629313 .
dpb:Oslo   :isCapital     true .
```

1.7.4 Sharing elements

May list statements that share subject using `;`.

```
dbp:Oslo   :officialName  "Oslo" ;
           :population    629313 ;
           :leaderName    dbp:Fabian_Stang .
```

And we may list statements that share both subject and predicate using `,`.

```
dbp:Norway rdfs:label "Norway"@en   ,
                      "Norwegen"@de ,
                      "Norge"@no .
```

1.7.5 Blank nodes

Blank nodes are designated with underscore or `[ ... ]`.

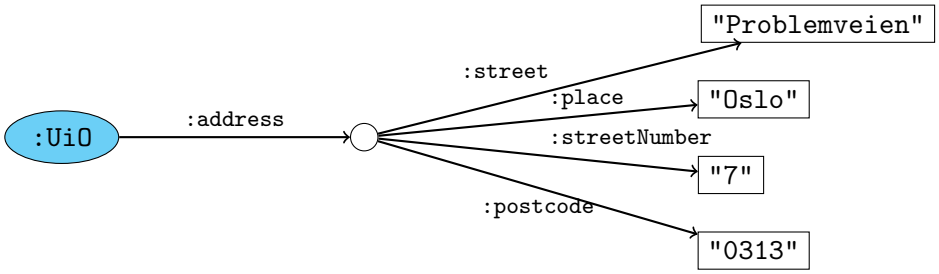
```
dbp:Norway  :capital    _:someplace .
_:someplace :population 629313 .
```

There is a place with official name Oslo

```
[] a             :Place ;
   :officialName "Oslo" .
```

UiO has address Problemveien 7, 0313 Oslo

```
:UiO :address [ :street       "Problemveien" ;
                :streetNumber "7" ;
                :place        "Oslo" ;
                :postcode     "0313" ] .
```



The blank node in this example has no name. In turtle you can name blank nodes with *blank node identifiers*, like `_:someplace`. This makes it easier to re-use blank nodes in several triples.

1.7.6 Other

- Use `#` to comment.
- Use \ to escape special characters.

  ```
  :someGuy foaf:name "James \"Mr. Man\" Olson" .
  ```

- Turtle specification: `http://www.w3.org/TR/turtle/`

2 SPARQL Protocol And RDF Query Language

As the title says, SPARQL is a *query language* for RDF. Documentation can be found at these locations

- Queries `http://www.w3.org/TR/rdf-sparql-query/`
- Protocol `http://www.w3.org/TR/rdf-sparql-protocol/`
- Results `http://www.w3.org/TR/rdf-sparql-XMLres/`

2.1 SPARQL fundamentals

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
WHERE  { ?book  ex:publishedBy  <http://crc-press.com/uri> .
         ?book  ex:title        ?title .
         ?book  ex:author       ?author . }
```

The actual query is initiated by the key word `WHERE`. It is followed by a *simple graph pattern*, enclosed in curly braces.

Exam exercise Consider the following RDF [1]

```
@prefix nasa: <http://www.nasa.gov/vocab#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

nasa:voyager a nasa:SpaceProbe;
             nasa:objective nasa:jupiter, nasa:saturn,
                            nasa:uranus, nasa:neptune .

nasa:mariner9 a nasa:SpaceProbe;
              nasa:objective nasa:mars .

nasa:jupiter  a nasa:Planet;
              nasa:moon nasa:io, nasa:ganumede, nasa:callisto;
              nasa:distanceFromSun "778"^^xsd:double .

nasa:saturn   a nasa:Planet;
              nasa:distanceFromSun "1426"^^xsd:double .

nasa:neptune  a nasa:Planet;
              nasa:moon nasa:triton, nasa:proteus, nasa:naiad;
              nasa:distanceFromSun "4503"^^xsd:double .
```

List all spaceprobes.

```
SELECT ?probe WHERE {
  ?probe a nasa:SpaceProbe .
}
```

For each planet that has one or more moons, list the planet and its moons.

[1] Example taken from exam exercise 2, 2010.

```
SELECT ?planet ?moon WHERE {
  ?planet a nasa:Planet;
          nasa:moon ?moon .
}
```

List all planets that lie further than 2000 million kilometers from the sun.

```
SELECT ?planet WHERE {
  ?planet a nasa:Planet;
          nasa:distanceFromSun ?distance .

  FILTER(?distance > 2000)
}
```

All planets that have been visited b more than one space probe.

```
SELECT DISTINCT ?planet WHERE {
  ?planet a nasa:Planet .
  ?probe1 a nasa:SpaceProbe ;
          nasa:objective ?planet .
  ?probe2 a nasa:SpaceProbe ;
          nasa:objective ?planet .

  FILTER(probe1 != ?probe2)
}
```

2.2 Groups and options

Group patterns can be used to restrict the scope of query conditions to certain parts of the pattern. It is possible to define sub-patterns as being optional, or to provide multiple alternative patterns.

```
SELECT ?title ?author
WHERE  {{ ?book  ex:publishedBy  <http://crc-press.com/uri>
          ?book  ex:title        ?title }
          {}
          ?book  ex:author    ?author
        }
```

Optional patterns are not required to occur in all retrieved results, but if they are found, may produce bindings of variables and thus extend the query result.

```
{  ?book  ex:publishedBy  <http://crc-press.com/uri>
   ?book  ex:title        ?title .
   OPTIONAL { ?book  ex:author  ?author }
}
```

Every occurrence of `OPTIONAL` must be followed by a group pattern.

2.2.1 SPARQL keywords

`PREFIX` Lets you use a prefix in the query instead of the whole URI.

`SELECT` Return clause. Returns data that fit some condition.

`FROM` Defines the RDF dataset which is being queried.

`WHERE` Specifies the query graph pattern to be matched.

`ORDER BY` One solution modifier. Used to rearrange query results. Other solution modifiers are `LIMIT` and `OFFSET`.

`ASK` Return clause. Checks if there is at least one result for a given pattern. Returns true or false.

`DESCRIBE` Returns a RDF graph that describes a resource.

`CONSTRUCT` Returns a RDF graph that is created from a template specified as a part of the query itself.

`DISTINCT` Lists equal entries once.

`FILTER` Filters the result with numerical functions, regular expressions, string operations etc.

`REDUCE` Allows to remove some or all duplicate solutions.

`UNION` Matches if any or some alternative matches. Combining graph patterns.

`OPTIONAL` Optional parts of a pattern.

2.2.2 Functions and operators

- Usual binary operators: `||`, `&&`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`.
- Usual unary operators: `!`, `+`, `-`.
- Unary tests: `BOUND(?var)`, `isURI(?var)`, `isBLANK(?var)`, `isLITERAL(?var)`.
- Accessors: `STR(?var)`, `LANG(?var)`, `DATATYPE(?var)`.

Some other tests are among others `sameTerm(?var)` which is used with unsupported data types. `langMatches` is used with `lang` to test for a language, like `langMatches(lang(?title), "no")`. Another one is `regex` which is used to match a variable with a regular expression. Always use `regex` with `str(?var)`!

2.3 SPARQL 1.1

Became a recommendation in March 2013.

2.3.1 Graph operations

`LOAD (SILENT)? IRIref_from (INTO GRAPH IRIref_to)?` Loads the graph at `IRIref_from` into the specified graph, or the default graph if graph not given.

`CLEAR (SILENT)? (GRAPH IRIref | DEFAULT | NAMED | ALL)` Removes the triples from the specified graph, the default graph, all named graphs or all graphs respectively.

`CREATE (SILENT)? GRAPH IRIref` Creates a new graph in stores that record empty graphs.

`DROP (SILENT)? (GRAPH IRIref | DEFAULT | NAMED | ALL)` Same as `CLEAR`, but removes all triples as well.

2.3.2 Inserting and deleting triples

Inserting

```
INSERT DATA {
  GRAPH </graph/courses/> {
    <course/inf3580>   :takenBy <student/veronika> .
    <student/veronika> foaf:name "Veronika Heimsbakk> ;
                       owl:sameAs <http:// ... > .
}}
```

Inserting conditionally

```
INSERT {
  <http:// ... /geo/inndeling/03> a gd:Fylke ;
                       gn:name "Oslo" ;
                       ?p ?o .
}
WHERE {
  <http:// ... /geo/inndeling/03/0301> a gd:Kommune ;
```

```
                    ?p ?o .
}
```

Deleting

```
DELETE DATA {
  GRAPH </graph/courses/> {
    <course/inf3580>  ex:oblig <exercise/oblig6> .
    <exercise/oblig6> rdfs:label "Mandatory Exercise 6" .
}}
```

Deleting conditionally

```
DELETE {
  ?book ?p ?v .
}
WHERE {
  ?book dc:date ?date .
  FILTER (?date < "2000-01-01T00:00:00"^^xsd:datetime)
  ?book ?p ?v .
}
```

2.3.3 Aggregation

Solutions can optionally be grouped according to one or more expressions. To specify the group, use `GROUP BY`. And to filter solutions resulting from grouping, use `HAVING`.

Example using `HAVING`

```
SELECT ?name (count(?kommune) AS ?kcount)
WHERE {
  ?fylke a gd:Fylke ;
         gn:officialName ?name ;
         gn:childrenFeatures ?kommune .
  ?kommune a gd:Kommune .
  FILTER (langMatches(lang(?name), 'no'))
} GROUP BY ?name HAVING (?kcount < 15)
```

Returns counties of Norway with less than 15 municipalities.

3 RDFS Reasoning

Interpretations need in no way comply with actual reality. One usually choose a certain mathemathical structure as interpretation in order to work in a formally correct way.

3.1 Syntactic reasoning with deduction rules

One option for describing a syntactic method consist of providing *deduction rules*.

$$\frac{p_1 \dots p_n}{p}$$

Given the validity of the propositions $p_1, \dots, p_n$, we can deduce that p must be valid.

3.1.1 Soundness and completeness

A deduction calculus is *sound* if every proposition set P' that can be derived from a set of propositions P by means of the deduction rules in a semantic consequence; $P \vdash P'$ implies $P \models P'$.

A deduction calculus is called *complete* if every proposition set P' that is semantically entailed by a proposition set P can also be deduced by means of the provided deduction rules. $P \models P'$ implies $P \vdash P'$.

3.2 Interpretations

A DL-interpretation $\mathcal{I}$ consists of:

- A set $\Delta^{\mathcal{I}}$, called the *domain* of $\mathcal{I}$.
- For each individual URI i, and element $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.
- For each **class** URI C, a subset $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$.
- For each **property** URI r, a relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.
- For each **datatype property** URI, a relation $a^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Lambda$, where Λ is the set of all literal values.

`individual`	`object-prop`	`individual .`	$r(i_1, i_2)$
`individual`	`data-prop`	`"literal" .`	$a(i, l)$
`individual`	`rdf:type`	`class .`	$C(i_1)$
`class`	`rdfs:subClassOf`	`class .`	$C \sqsubseteq D$
`object-prop`	`rdfs:subPropOf`	`object-prop`	$r \sqsubseteq s$
`data-prop`	`rdfs:subPropOf`	`data-prop`	$a \sqsubseteq b$
`object-prop`	`rdfs:domain`	`class`	$dom(r, C)$
`object-prop`	`rdfs:range`	`class`	$rg(r, C)$

Table 1: Allowed triples

3.2.1 Blank node semantics

Given an interpretation $\mathcal{I}$ with domain $\Delta^{\mathcal{I}}$, a blank node valuation β

- gives a domain element or literal value $\beta(b) \in \Delta^{\mathcal{I}} \cup \Delta$.
- for every blank node ID b.

Interpretation $\mathcal{I}, \beta \models r(x, y)$ iff $\langle x^{\mathcal{I},\beta}, y^{\mathcal{I},\beta} \rangle \in r^{\mathcal{I}}$, for any legal combination of URIs, literals or blank nodes x, y and object or datatype propery r.

3.2.2 Example on an interpretation

Let Γ be following RDF graph.

```
:Tweety  rdf:type      :Bird .
:Nixon   rdf:type      :Republican .
:Nixon   rdf:type      :Quacker .
:Nixon   :listensTo    :Tweety .
:Nixon   :likes        [ a :Bird ] .
[]       :likes        :Nixon .
:Nixon   :hasNickname  "Ric" .
:Tweety  :hasNickname  "Mr. Man" .
:Tweety  :likes        :Tux .
```

Create an interpretation $\mathcal{I}_1$, such that $\mathcal{I}_1 \models \Gamma$.

- $\Delta^{\mathcal{I}} = \{1, 2, 3, 4_b, 5_b\}$
- $\texttt{:Tweety}^{\mathcal{I}} = 1$, $\texttt{:Nixon}^{\mathcal{I}} = 2$, $\texttt{:Tux}^{\mathcal{I}} = 3$
- $\texttt{:Bird}^{\mathcal{I}} = \{1, 4_b\}$, $\texttt{:Republican}^{\mathcal{I}} = \{2\}$, $\texttt{:Quacker}^{\mathcal{I}} = \{2\}$
- $\texttt{:listensTo}^{\mathcal{I}} = \{\langle 2, 1\rangle\}$, $\texttt{:likes}^{\mathcal{I}} = \{\langle 2, 4_b\rangle, \langle 5_b, 2\rangle, \langle 1, 3\rangle\}$, $\texttt{:hasNickname}^{\mathcal{I}} = \{\langle 2, \texttt{"Ric"}\rangle, \langle 1, \texttt{"Mr. Man"}\rangle\}$

Let $\beta(b_1) = 4_b$ and $\beta(b_2) = 5_b$, then $\mathcal{I}, \beta \models \Gamma$, so we have that $\mathcal{I} \models \Gamma$.

Create an interpretation $\mathcal{I}_2$ such that $\mathcal{I}_2 \not\models \Gamma$. Let $\mathcal{I}$ be as above, but let $\texttt{:Bird}^{\mathcal{I}} = \emptyset$. Then there is nothing we can send b_1 to have (`:Nixon :likes [a :Bird] .`). So this interpretation satisfy $\mathcal{I} \not\models \Gamma$.

	If S contains:	then S RDFS entails recognizing D:
rdfs1	any IRI aaa in D	aaa `rdf:type rdfs:Datatype` .
rdfs2	aaa `rdfs:domain` xxx . yyy aaa zzz .	yyy `rdf:type` xxx .
rdfs3	aaa `rdfs:range` xxx . yyy aaa zzz .	zzz `rdf:type` xxx .
rdfs4a	xxx aaa yyy .	xxx `rdf:type rdfs:Resource` .
rdfs4b	xxx aaa yyy .	yyy `rdf:type rdfs:Resource` .
rdfs5	xxx `rdfs:subPropertyOf` yyy . yyy `rdfs:subPropertyOf` zzz .	xxx `rdfs:subPropertyOf` zzz .
rdfs6	xxx `rdf:type rdf:Property` .	xxx `rdfs:subPropertyOf` xxx .
rdfs7	aaa `rdfs:subPropertyOf` bbb . xxx aaa yyy	xxx bbb yyy .
rdfs8	xxx `rdf:type rdfs:Class` .	xxx `rdfs:subClassOf rdfs:Resource` .
rdfs9	xxx `rdfs:subClassOf` yyy . zzz `rdf:type` xxx	zzz `rdf:type` yyy .
rdfs10	xxx `rdf:type rdfs:Class` .	xxx `rdfs:subClassOf` xxx .
rdfs11	xxx `rdfs:subClassOf` yyy . yyy `rdfs:subClassOf` zzz .	xxx `rdfs:subClassOf` zzz .
rdfs12	xxx `rdf:type rdfs:ContainerMemebershipProperty` .	xxx `rdfs:subPropertyOf rdfs:member` .
rdfs13	xxx `rdf:type rdfs:Datatype` .	xxx `rdfs:subClassOf rdfs:Literal` .

Table 2: The RDFS deduction rules.

4 Description Logic ($\mathcal{ALC}$)

The basic building blocks of $\mathcal{ALC}$ are *classes*, *roles* and *individuals*, which can be put into relationships with each other.

`Student(veronahe)`

This describes that the individual `veronahe` belongs to the class `Student`.

`hasAfflication(veronahe, ifi)`

Describes that `veronahe` is affiliated with `ifi`. Subclass relations are expressed by using $\sqsubseteq$.

`Professor` $\sqsubseteq$ `FacultyMember`

Equivalence between classes is expressed by using $\equiv$.

`Professor` $\equiv$ `Prof.`

$\mathcal{ALC}$ provides logical class constructors.

Conjunction	$\sqcap$
Disjunction	$\sqcup$
Negation	$\neg$

Complex classes can also be defined by using quantifiers. If R is a role and C a class expression, then $\forall R.C$ and $\exists R.C$ are also class expressions.

The empty class `owl:Nothing`, denoted $\bot$, can be expressed by

$$\bot \equiv C \sqcap \neg C$$

$\top$ corresponds to `owl:Thing`, can be expressed by

$$\top \equiv C \sqcup \neg C$$

, or equivalently by

$$\top \equiv \neg \bot$$

Disjointness of class (`owl:disjointWith`) C and D, expressed by

$$C \sqcap D \sqsubseteq \bot \Rightarrow C \sqsubseteq \neg D$$

`rdfs:range` can be expressed by

$$\top \sqsubseteq \forall R.C$$

and `rdfs:domain` by

$$\exists R.\top \sqsubseteq C$$

Class expressions of $\mathcal{ALC}$ $C, D ::= A|\top|\bot|\neg C|C \sqcap D|C \sqcup D|\forall R.C|\exists R.C$

$\mathcal{ALC}$ semantics An interpretation $\mathcal{I}$ fixes a set $\Delta^{\mathcal{I}}$, the domain, $\mathrm{A}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ for each atomic concept A, $\mathrm{R}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ for each role R, and $\mathrm{a}^{\mathcal{I}} \in \Delta$ for each individual a.

Statements in $\mathcal{ALC}$ are divided into *TBox statements* and *ABox statements*. The TBox is considered to contain terminological (schema) knowledge, while the ABox contains knowledge about instances (i.e. individuals). An $\mathcal{ALC}$ knowledge base consists of an ABox and a TBox.

5 The Web Ontology Language (OWL)

Became a W3C recommendation in 2004 and is now the undisputed standard ontology language. It is built on Description Logics (DL) and combines DL expressiveness with RDF technology (URIs, namespaces ...). OWL extends RDFS with boolean operations, universal/existential restrictions and more.

OWL is defined as set of things that can be said about classes, properties and instances.

5.1 OWL vocabulary in OWL/RDF

New	From RDFS	From RDF
`owl:Ontology`	~~`rdfs:Class`~~	`rdf:type`
`owl:Class`	`rdfs:subClassOf`	~~`rdf:Property`~~
`owl:Thing`	`rdfs:subPropertyOf`	
Properties	`rdfs:domain`	
Restrictions	`rdfs:range`	
Annotations		

Table 3: Vocabulary of OWL.

Table 3, including XSD datatypes.

5.2 Properties in OWL

There are three kinds of *mutually disjoint properties in OWL.*

- `owl:DatatypeProperty`
  - Link individuals to data values as `xsd:string`.
  - Example: `:hasAge`, `:hasSurname`
- `owl:ObjectProperty`

  - Link individuals to individuals.
  - Example: `:hasFather`, `:driveAxle`

- `owl:AnnotationProperty`
  - Has no logical implication, ignored by reasoners.
  - Anything can be annotated.
  - Use for human readable-only data.
  - Example: `rdfs:label`, `dc:creator`

5.2.1 Characteristics of OWL properties

Object properties link individuals to individuals, so all characteristics and operations are defined for them.

Datatype properties link individuals to data values, so they cannot be

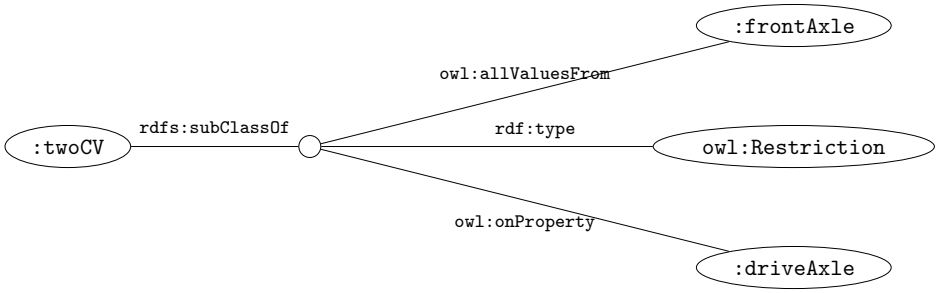
- reflexive, or they would not be datatypes properties
- transitive, since no property takes data values in first position
- symmetric, as above
- inverses, as above
- inverse functional, for computational reasons
- part of chains, as above

So what remains is functionality, subsumption, equivalence and disjointness.

Annotation properties have no logical implication, so nothing can be said about them.

Example: Universal Restrictions in OWL/RDF

- TwoCV $\sqsubseteq \forall$ driveAxle.FrontAxle



- In Turtle syntax:

```
:TwoCV rdfs:subClassOf [ rdf:type owl:Restrictions ;
                         owl:onProperty :driveAxle ;
                         owl:allValuesFrom :FrontAxle
                       ] .
```

- In OWL/XML syntax:

```
<SubClassOf>
  <Class URI=":TwoCV"/>
  <ObjectAllValuesFrom>
    <ObjectProperty URI=":driveAxle"/>
    <Class URI=":FrontAxle"/>
  </ObjectAllValuesFrom>
</SubClassOf>
```

- In OWL Functional syntax:

```
SubClassOf(TwoCV ObjectAllValuesFrom(driveAxle FrontAxle))
```

5.3 Manchester OWL Syntax

Manchester OWL syntax is used in Protégé for concept descriptions. It also has syntax for axioms, but this is less used. Correspondence to DL constructs.

DL	**Manchester**
C ⊓ D	C **AND** D
C ⊔ D	C **OR** D
¬D	**NOT** C
∀R.C	R **ONLY** C
∃R.C	R **SOME** C

With Protégé, everything can be mapped to DL axioms.

5.4 Assumptions

5.4.1 World assumptions

Closed World Assumption (CWA)

- Complete knowledge.
- Any statement that is now known to be true is false.
- Typical semantics for database systems.

Open World Assumption (OWA)

- Potential incomplete knowledge.
- Any statement that is not known to be true is false, does not hold.
- Typical semantics for logic-based systems.

5.4.2 Name assumptions

Under any assumptions, equal names (individual URIs, DB constants) always denote the same "thing". Cannot have $a^{\mathcal{I}} \neq a^{\mathcal{I}}$.

Unique Name Assumption (UNA) Under UNA, different names always denote different things. E.g., $a^{\mathcal{I}} \neq b^{\mathcal{I}}$. This is common in relational databases.

Non-unique Name Assumption (NUNA) Under NUNA, different named *need not* denote different things. We can have $a^{\mathcal{I}} = a^{\mathcal{I}}$, or `dpedia:Oslo`$^{\mathcal{I}}$ =`geo:34521`$^{\mathcal{I}}$.

5.5 OWL 2

OWL 2 is based on the DL $\mathcal{SHOIN}(D)$.

5.5.1 ABox axioms: equality

This axiom express equality and non-equality between individuals.

RDF/OWL	DL	Manchester
`:a owl:sameAs :b`	$a = b$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$	**SameAs**
`:a owl:differentFrom :b`		**DifferentFrom**

Semantics

- $\mathcal{I} \models a = b$ iff $a^{\mathcal{I}} = b^{\mathcal{I}}$
- $\mathcal{I} \models a \neq b$ iff $a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Remember that with non-unique name assumptions in semantic web, you must sometimes use $=$ and $\neq$ to get expected results.

5.5.2 Creating concepts using individuals

Create anonymous concepts by explicitly listing all members. This is called *closed classes* in OWL.

RDF/OWL	DL	Manchester
`owl:oneOf`	$\{a, b, \dots\}$	{a, b, ...}
`rdf:List`		

Example SimpsonFamilyMember $\equiv$ {Homer, Marge, Bart, Lisa, Maggie}

Note that the individuals does not necessarily represent different objects. We still need $=$ and $\neq$ to say that members are the same or different. "Closed classes of data values" are *datatypes*.

5.5.3 Axioms involving individuals

Using closed classes we can exclude individuals from classes.

Example {NedFlanders} $\sqsubseteq \neg$SimpsonFamilyMember

- Ned Flanders is not a family member of the Simpsons.
- Better solution: FlandersFamilyMember $\equiv$ {Ned Flanders, ...} and FlandersFamilyMember $\sqsubseteq \neg$SimpsonFamilyMember

Closed properties does not exist in OWL. However, it can be done with closed classes, but there is *negated role assignment* to exclude relationships from relations and roles.

5.5.4 ABox axioms: negative

RDF/OWL	DL	Manchester
`NegativePropertyAssertion`	$\neg R(a, b)$	a **NOT** R b

Semantics

- $\mathcal{I} \models \neg R(a, b)$ iff $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \notin R^{\mathcal{I}}$

This works both for object properties and datatype properties.

Example `:Bart not :hasFather :NedFlanders`

5.5.5 Cardinality restrictions

Cardinality restrictions are introduced through $\mathcal{SHOIN}(D)$.

RDF/OWL	OWL	Manchester
`owl:minCardinality`	$\geq_n R.D$	**MIN**
`owl:maxCardinality`	$\leq_n R.D$	**MAX**
`owl:cardinality`	$=_n R.D$	**EXACTLY**

Semantics

- $(\leq_n R.D)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \{b : \langle a, b\rangle \in R^{\mathcal{I}} \wedge b \in D^{\mathcal{I}}\}\# \leq n\}$
- $(\geq_n R.D)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \{b : \langle a, b\rangle \in R^{\mathcal{I}} \wedge b \in D^{\mathcal{I}}\}\# \geq n\}$

Restricts the number of relations a type of object can/must have.

Example Car $\sqsubseteq \leq_2$ driveAxle.$\top$ — "A car has at most two drive axles."

5.5.6 Value restrictions

Existential and universal restrictions are called *value restrictions*. Restrictions of the form $\forall R.D$, $\exists R.D$, $\leq_n R.D$, $\geq_n R.D$ are called *qualified* when D is not $\top$. Can also qualify with a closed class.

RDF/OWL	DL	Manchester
`owl:hasValue`	$\exists R\{a\}$	R **VALUE** a

Example Norwegian $\equiv$ Person $\sqcup \exists$citizenOf.{Norway}

5.5.7 Self restriction

Local reflexivity restriction. Restricts to objects which are related to themselves.

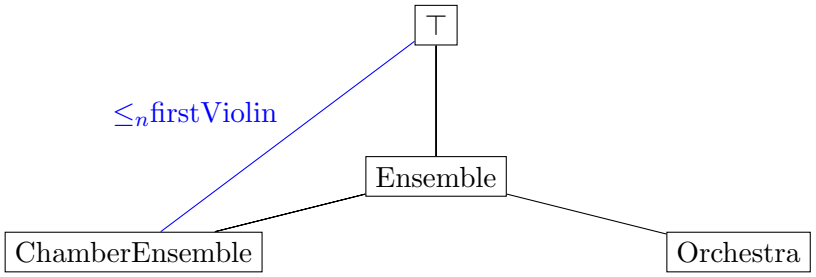
RDF/OWL	DL	Manchester
`owl:hasSelf`	$\exists R.Self$	**SELF**

Semantics

- $(\exists R.Self)^{\mathcal{I}} = \{x | \langle x, x\rangle \in R^{\mathcal{I}}\}$

Example $\exists$hasBoss.Self $\sqsubseteq$ SelfEmployed

5.5.8 Restrictions, NUNA and OWA



TBox

`Orchestra` $\sqsubseteq$ `Ensemble`

`ChamberEnsemble` $\sqsubseteq$ `Ensemble`

`ChamberEnsemble` $\sqsubseteq \leq_1$ `firstViolin`.$\top$

ABox

`Ensemble(oslo)`

`firstViolin(oslo, skolem)`

`firstViolin(oslo, lie)`

Orchestras and Chamber ensembles are Ensembles. Chamber ensembles have onlye one instrument on each voice, in particular, only one first violin.

5.5.9 Unexpected results

It does not follow from TBox + ABox that `oslo` is an Orchestra. An ensemble need neither be an orchestra nor a chamber ensemble, its "just" an ensemble. Add a "covering axiom" Ensemble $\sqsubseteq$ Orchestra $\sqcup$ ChamberEnsemble — an ensemble is an orchestra or a chamber ensemble.

It still does not follow that `oslo` is an Orchestra. This is due to NUNA. We cannot assume that `skolem` and `lie` are distinct. The statement `skolem owl:differentFrom lie`, makes `oslo` an orchestra.

5.5.10 Role modeling in OWL 2

This can get excessively complicated. For instance, transitive roles cannot be irreflexive or asymmetric. Role inclusions are not allowed to cycle.

`hasParent` $\circ$ `hasHusband` $\sqsubseteq$ `hasFather`

`hasFather` $\sqsubseteq$ `hasParent`

Transitive roles R and S cannot be declared disjoint. These restrictions can be hard to keep track of, the reason they exist are computational, not logical.

5.6 Mathematical properties and operations

A relation R over a set $X (R \subseteq X \times X)$ is

Reflexive	if $\langle a, a\rangle \in R$ for all $a \in X$
Irreflexive	if $\langle a, a\rangle \notin R$ for all $a \in X$
Symmetric	if $\langle a, b\rangle \in R$ implies $\langle b, a\rangle \in R$
Asymmetric	if $\langle a, b\rangle \in R$ implies $\langle b, a\rangle \notin R$
Transitive	if $\langle a, b\rangle, \langle b, c\rangle \in R$ implies $\langle a, c\rangle \in R$
Functional	if $\langle a, b\rangle, \langle a, c\rangle \in R$ implies $b = c$
Functional	if $\langle a, b\rangle, \langle c, b\rangle \in R$ implies $a = c$

Relations from ordinary language

- Symmetric: `hasSibling`, `differentFrom`
- Non-symmetric: `hasBrother`
- Asymmetric: `olderThan`, `memberOf`
- Transitive: `olderThan`, `hasSibling`
- Functional: `hasBiologicalMother`
- Inverse functional: `gaveBirthTo`

Handy tables

OWL constructor	**DL syntax**	**Manchester**
`intersectionOf`	$C \sqcap D$	C **AND** D
`unionOf`	$C \sqcup D$	C **OR** D
`complementOf`	$\neg C$	**NOT** C
`oneOf`	$\{a\} \sqcap \{b\} \ldots$	{a b ...}
`someValuesFrom`	$\exists RC$	R **SOME** C
`allValuesFrom`	$\forall RC$	R **ONLY** C
`minCardinality`	$\geq NR$	R **MIN** 3
`maxCardinality`	$\leq NR$	R **MAX** 3
`cardinality`	$= NR$	R **EXACTLY** 3
`hasValue`	$\exists R\{a\}$	R **VALUE** a
`NegativePropertyAssertion`	$\neg R(a, b)$	a **NOT** R b
`hasSelf`	$\exists R.Self$	**SELF**

Table 4: Class constructions in OWL

Class	`Class(individual)`
Relations	`relation(subject, object)`
Subclasses	`Class` $\sqsubseteq$ `Class`
Equivalence	`Class` $\equiv$ `Class`
`owl:Nothing`	$\bot \equiv C \sqcap \neq C$
`owl:Thing`	$\top \equiv C \sqcup \neq C$
Disjointness	$C \sqcap D \sqsubseteq \bot \Rightarrow C \sqsubseteq \neg D$
`rdfs:range`	$\top \sqsubseteq \forall R.C$
`rdfs:domain`	$\exists R.\top \sqsubseteq C$

Table 5: Description logics

	If S contains:	**then S RDFS entails recognizing D:**
rdfs1	any IRI aaa in D	aaa `rdf:type rdfs:Datatype` .
rdfs2	aaa `rdfs:domain` xxx . yyy aaa zzz .	yyy `rdf:type` xxx .
rdfs3	aaa `rdfs:range` xxx . yyy aaa zzz .	zzz `rdf:type` xxx .
rdfs4a	xxx aaa yyy .	xxx `rdf:type rdfs:Resource` .
rdfs4b	xxx aaa yyy .	yyy `rdf:type rdfs:Resource` .
rdfs5	xxx `rdfs:subPropertyOf` yyy . yyy `rdfs:subPropertyOf` zzz .	xxx `rdfs:subPropertyOf` zzz .
rdfs6	xxx `rdf:type rdf:Property` .	xxx `rdfs:subPropertyOf` xxx .
rdfs7	aaa `rdfs:subPropertyOf` bbb . xxx aaa yyy	xxx bbb yyy .
rdfs8	xxx `rdf:type rdfs:Class` .	xxx `rdfs:subClassOf rdfs:Resource` .
rdfs9	xxx `rdfs:subClassOf` yyy . zzz `rdf:type` xxx	zzz `rdf:type` yyy .
rdfs10	xxx `rdf:type rdfs:Class` .	xxx `rdfs:subClassOf` xxx .
rdfs11	xxx `rdfs:subClassOf` yyy . yyy `rdfs:subClassOf` zzz .	xxx `rdfs:subClassOf` zzz .
rdfs12	xxx `rdf:type rdfs:ContainerMemebershipProperty` .	xxx `rdfs:subPropertyOf rdfs:member` .
rdfs13	xxx `rdf:type rdfs:Datatype` .	xxx `rdfs:subClassOf rdfs:Literal` .

Table 6: The RDFS deduction rules.

se1	xxx aaa yyy .	xxx aaa _:n .
se2	xxx aaa yyy .	_:n aaa yyy .

Table 7: Simple entailment deduction rules.