Semantic Technology Tutorial

Part 3: Languages
SemTech Languages

1. Standardised language for data
   - W3C standard for data exchange is RDF
   - RDF is a simple language consisting of <S P O> triples
     - for example <eg:Ian eg:worksAt eg:Oxford>
     - all S, P, O are URIs or literals (data values)
   - URIs provides a flexible naming scheme
   - Set of triples can be viewed as a graph
Standardised language for data

- W3C standard for data exchange is RDF
- RDF is a simple language consisting of S,P,O triples
  - for example, `<eg:Ian eg:worksAt eg:Oxford>`
  - all S,P,O are URIs or literals (data values)
- URIs provide a flexible naming scheme
- Set of triples can be viewed as a graph

SemTech Languages

eg:organisation
http://...rdf-syntax-ns/#type

eg:w3c
eg:worksfor
http://www.w3.org/People/EM/contact#me
http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

eg:organisation

eg:Boston
http://...fullName

eg:organisation

eg:w3c
eg:worksfor
http://www.w3.org/People/EM/contact#me
http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

eg:w3c
eg:organisation
http://...rdf-syntax-ns/#type

eg:Boston
http://...fullName

eg:Boston
http://...fullName

Eric Miller
http://www.w3.org/2000/10/swap/pim/contact#mailbox
mailto:em@w3.org

Dr.
http://www.w3.org/2000/10/swap/pim/contact#personalTitle
http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
SemTech Languages

1. Standardised language for **data**

- W3C standard for data exchange is **RDF**
- RDF is a simple language consisting of <S,P,O> triples
  - for example <em1234 rdf:type Person>
  - all S,P,O are URIs or literals (data values)
- URIs provide a flexible **naming** scheme
- Set of triples can be viewed as a **graph**

<table>
<thead>
<tr>
<th>Triple</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>P</td>
<td>O</td>
</tr>
<tr>
<td>em1234</td>
<td>rdf:type</td>
<td>Person</td>
</tr>
<tr>
<td>em1234</td>
<td>name</td>
<td>“Eric Miller”</td>
</tr>
<tr>
<td>em1234</td>
<td>title</td>
<td>“Dr”</td>
</tr>
<tr>
<td>em1234</td>
<td>mailbox</td>
<td><a href="mailto:em@w3.org">mailto:em@w3.org</a></td>
</tr>
<tr>
<td>em1234</td>
<td>worksfor</td>
<td>w3c</td>
</tr>
<tr>
<td>w3c</td>
<td>rdf:type</td>
<td>organisation</td>
</tr>
<tr>
<td>w3c</td>
<td>hq</td>
<td>Boston</td>
</tr>
<tr>
<td>w3c</td>
<td>name</td>
<td>“W3C”</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SemTech Languages

1. Standardised language for data

- W3C standard for data exchange is RDF
- RDF is a simple language consisting of <S,P,O> triples
  - for example <em1234 eg:worksAt eg:Oxford>
  - all S,P,O are URIs or literals (data values)
- URIs provide a flexible naming scheme
- Set of triples can be viewed as a graph

<table>
<thead>
<tr>
<th>PERSON</th>
<th>ID</th>
<th>NAME</th>
<th>TITLE</th>
<th>MAILBOX</th>
<th>WORKSFOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>em1234</td>
<td>“Eric Miller”</td>
<td>“Dr”</td>
<td><a href="mailto:em@w3.org">mailto:em@w3.org</a></td>
<td>w3c</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORGANISATION</th>
<th>ID</th>
<th>NAME</th>
<th>HQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w3c</td>
<td>“W3C”</td>
<td>Boston</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SemTech Languages

2 Standardised language for vocabularies/schemas

- W3C standard for vocabulary/schema exchange is OWL
- OWL provides for rich conceptual schemas, aka ONTOLOGIES

```
Heart ⊆ MuscularOrgan ⊆
   ∃isPartOf.CirculatorySystem
HeartDisease ≡ Disease ⊆
   ∃affects.Heart
VascularDisease ≡ Disease ⊆
   ∃affects.(∃isPartOf.CirculatorySystem)
```
SemTech Languages

3. Standardised language for queries
   - W3C standard for queries is **SPARQL**
   - SPARQL provides a rich query language comparable to **SQL**

```sql
SELECT ?x
WHERE
{
  ?x rdf:type Patient .
  ?x suffersFrom ?y .
  ?y rdf:type VascularDisease }
```
How Does it Work?

\( \text{John rdf:type Patient} \)
\( \text{John suffersFrom d1} \)
\( \text{d1 rdf:type HeartDisease} \)
How Does it Work?

Patients suffering from heart disease

\[ \langle \text{John} \text{ rdf:type Patient} \rangle \]
\[ \langle \text{John suffersFrom d1} \rangle \]
\[ \langle d1 \text{ rdf:type HeartDisease} \rangle \]
How Does it Work?

Patients suffering from heart disease

<John rdf:type Patient>
<John suffersFrom d1>
<d1 rdf:type HeartDisease>
How Does it Work?

Patients suffering from heart disease

\[
\langle \text{John rdf:type Patient} \rangle \\
\langle \text{John suffersFrom d1} \rangle \\
\langle \text{d1 rdf:type HeartDisease} \rangle
\]
How Does it Work?

Patients suffering from vascular disease

\( \langle \text{John} \text{ rdf:type Patient} \rangle \)
\( \langle \text{John} \text{ suffersFrom d1} \rangle \)
\( \langle \text{d1 rdf:type HeartDisease} \rangle \)
How Does it Work?

Patients suffering from vascular disease

\{John rdf:type Patient\}
\{John suffersFrom d1\}
\{d1 rdf:type HeartDisease\}
How Does it Work?

Patients suffering from vascular disease

\(<John \text{ rdf:type Patient}>\>
\(<John \text{ suffersFrom d1}>\>
\(<d1 \text{ rdf:type HeartDisease}>\)
How Does it Work?

Patients suffering from vascular disease

\[
\langle \text{John} \ rdf:\text{type} \ Patient \rangle \\
\langle \text{John} \ suffers\text{From} \ d1 \rangle \\
\langle d1 \ rdf:\text{type} \ Heart\text{Disease} \rangle \\
\]

\text{Heart} \sqsubseteq \text{MuscularOrgan} \\
\text{HeartDisease} \equiv \text{Disease} \\
\text{VascularDisease} \equiv \text{Disease}
How Does it Work?

Patients suffering from vascular disease

\[
\langle \text{John} \ rdf:\text{type} \ Patient \rangle \\
\langle \text{John} \ suffers\text{From} \ d1 \rangle \\
\langle d1 \ rdf:\text{type} \ Heart\text{Disease} \rangle
\]
How Does it Work?

Patients suffering from vascular disease

\[
\langle \text{John rdf:type Patient}\rangle \\
\langle \text{John suffersFrom d1}\rangle \\
\langle \text{d1 rdf:type HeartDisease}\rangle \\
\text{Heart} \sqsubseteq \text{MuscularOrgan} \\
\exists \text{isPartOf.CirculatorySystem} \\
\text{HeartDisease} \equiv \text{Disease} \\
\exists \text{affects.Heart} \\
\text{VascularDisease} \equiv \text{Disease} \\
\exists \text{affects.}(\exists \text{isPartOf.CirculatorySystem})
\]
How Does it Work?

Is heart disease a kind of vascular disease?

\[
\langle \text{John} \text{ rdf:type Patient} \rangle \\
\langle \text{John suffersFrom d1} \rangle \\
\langle \text{d1 rdf:type HeartDisease} \rangle
\]

\[
\text{Heart} \sqsubseteq \text{MuscularOrgan} \\
\text{isPartOf. CirculatorySystem} \\
\text{HeartDisease} \sqsubseteq \text{Disease} \\
\text{affects. Heart} \\
\text{VascularDisease} \sqsubseteq \text{Disease} \\
\text{affects.(isPartOf. CirculatorySystem)}
\]
How Does it Work?

Is heart disease a kind of vascular disease?

YES

\[
\langle \text{John \ rdf:type \ Patient} \rangle \\
\langle \text{John \ suffersFrom \ d1} \rangle \\
\langle \text{d1 \ rdf:type \ HeartDisease} \rangle
\]

\[
\text{Heart} \sqsubseteq \text{MuscularOrgan} \\
\text{HeartDisease} \sqsubseteq \text{Disease} \\
\text{VascularDisease} \sqsubseteq \text{Disease}
\]

\[
\text{isPartOf. CirculatorySystem} \\
\text{affects. Heart} \\
\text{affects. (isPartOf. CirculatorySystem)}
\]
How Does it Work?

\[
\langle \text{John} \ rdf:\text{type} \ Patient \rangle \\
\langle \text{John} \ suffersFrom \ d1 \rangle \\
\langle d1 \ rdf:\text{type} \ HeartDisease \rangle
\]

\[
\text{Heart} \sqsubseteq \text{MuscularOrgan} \sqsubseteq \\
\exists \text{isPartOf. CirculatorySystem}
\]

\[
\text{HeartDisease} \sqsubseteq \text{Disease} \sqsubseteq \\
\exists \text{affects. Heart}
\]

\[
\text{VascularDisease} \sqsubseteq \text{Disease} \sqsubseteq \\
\exists \text{affects. (isPartOf. CirculatorySystem)}
\]

Why?
How Does it Work?

\[
\begin{align*}
&\langle \text{John} \ rdf\text{:type} \ Patient \rangle \\
&\langle \text{John} \ suffersFrom \ d1 \rangle \\
&\langle d1 \ rdf\text{:type} \ Heart\text{Disease} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Heart} &\in \text{MuscularOrgan} \\
&\exists \text{isPartOf}.\text{CirculatorySystem} \\
\text{Heart\text{Disease}} &\equiv \text{Disease} \\
&\exists \text{affects}.\text{Heart} \\
\text{Vascular\text{Disease}} &\equiv \text{Disease} \\
&\exists \text{affects}.(\exists \text{isPartOf}.\text{CirculatorySystem})
\end{align*}
\]
An ontology language defines constructs available to modellers
- E.g., kinds of statements about concepts (conjunction, negation, ...)
- Formal semantics specifies mathematically the constructs’ meaning
- Semantics determines the inferences one can draw

Standard languages facilitate interoperability

Semantic Web language stack:

<table>
<thead>
<tr>
<th>Basic Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Description Framework (RDF)</td>
<td>basic semistructured data model</td>
</tr>
<tr>
<td>RDF Schema (RDFS)</td>
<td>a simple ontology language over RDF</td>
</tr>
<tr>
<td>Web Ontology Language (OWL) 2</td>
<td>extends RDFS to an expressive language undecidable</td>
</tr>
<tr>
<td>OWL 2 Full</td>
<td>decidable, based on description logics</td>
</tr>
<tr>
<td>OWL 2 DL</td>
<td></td>
</tr>
<tr>
<td>OWL 2 EL</td>
<td></td>
</tr>
<tr>
<td>OWL 2 QL</td>
<td></td>
</tr>
<tr>
<td>OWL 2 RL</td>
<td></td>
</tr>
<tr>
<td>Semantic Web Rule Language (SWRL)</td>
<td>unofficial rule standard</td>
</tr>
<tr>
<td>Rule Interchange Format (RIF)</td>
<td>(mainly production) rule standard</td>
</tr>
</tbody>
</table>
Node — an object one can make statements about (often called resource)
- IRI — well-known identifier for an object
  - E.g., ⟨http://skyscanner.net/Savoy⟩, often abbreviated as sky:Savoy
- Blank node — an object with an unknown identity (aka labelled null)
  - E.g., _:x
- Literal — concrete value such as a string or an integer
  - E.g., “abc”^^xsd:string, “1”^^xsd:integer, “+01”^^xsd:byte

Triple — the simplest statement about objects
- ⟨s, p, o⟩ with s, p, and o nodes: object o is the value of property p on subject s
- E.g., ⟨:Savoy, :locatedIn, :London⟩, ⟨:Savoy, rdf:type, :Hotel⟩

RDF graph — a finite set of RDF triples
- Can be understood as a three-column relation over nodes

RDF dataset — a finite set of RDF graphs, each associated with a node
  - rdf:type states that a node is an instance of a class

More details at http://www.w3.org/TR/rdf11-concepts/
RDF graphs can be represented graphically

- Properties are nodes, so one can make statements about them
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://skyscanner.net/">
    <rdf:Description rdf:about="http://skyscanner.net/Hotel">
        <rdfs:subClassOf rdf:resource="http://skyscanner.net/Accommodation"/>
    </rdf:Description>

    <rdf:Description rdf:about="http://skyscanner.net/Savoy">
        <rdf:type rdf:resource="http://skyscanner.net/Accommodation"/>
        <locatedIn rdf:resource="http://skyscanner.net/London"/>
    </rdf:Description>

    <rdf:Description rdf:about="http://skyscanner.net/London">
        <rdf:type rdf:resource="http://skyscanner.net/City"/>
    </rdf:Description>

    <rdf:Description rdf:about="http://skyscanner.net/locatedIn">
        <rdfs:domain rdf:resource="http://skyscanner.net/Accommodation"/>
        <rdfs:range rdf:resource="http://skyscanner.net/City"/>
        <rdfs:subPropertyOf rdf:resource="http://skyscanner.net/containedIn"/>
    </rdf:Description>
</rdf:RDF>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://skyscanner.net/>

:Hotel rdfs:subClassOf :Accommodation .

:Savoy rdf:type :Accommodation ;
   :locatedIn :London" .

:London rdf:type :City .

:locatedIn rdfs:domain :Accommodation ;
   rdfs:range :City ;
   rdfs:subPropertyOf :containedIn .

- Much more readable and compact!
- http://www.w3.org/TR/turtle/
RDF data can be stored in a relational database in (at least) two ways

- **Dictionary encoding** commonly used to map nodes to integers

- **Triple table** approach
  - Store triples in a three-column table
  - Exhaustive indexing can be achieved using only six indexes
  - Often extended to quads → triples with additional graph membership node
  - Main benefit: flexibility to support any kind of query
  - Main problem: queries involve many self-joins on the triple table

- **Vertical partitioning** approach
  - Use binary relations for properties, unary relations for classes
  - Store \(\langle s, p, o \rangle\) with \(p \neq rdf\text{-}type\) as tuple \(\langle s, o \rangle\) in relation \(p\)
  - Store \(\langle s, rdf\text{-}type, o \rangle\) as tuple \(\langle s \rangle\) in relation \(o\)
  - Use exhaustive indexing
  - Main benefit: avoids self-joins → easier for DBMSs
  - Main problem: does not support queries with variables in predicate position
RDF supports only binary relations → often very restrictive in practice
- E.g., ‘British Airways operates flight BA1452 from LHR to EDI’

Reification represents a statement as an object

Can be used to make statements about triples
- E.g., ‘(:Savoy, :locatedIn, :London) was obtained from Expedia’
“lexicalValue”\~datatypeIRI — datatypeIRI identifies a datatype that specifies how to map “lexicalValue” to a concrete value
- Many datatypes come from XML Schema 1.1
- http://www.w3.org/TR/xmlschema11-2/

E.g., “abc”\~xsd:string, “1”\~xsd:integer, “+01”\~xsd:byte

Syntactic shortcuts:
- xsd:string can be omitted: “abc”\~xsd:string → “abc”
- “abc”@en supports localisation → equivalent to “abc@en”\~rdf:PlainLiteral

Literal equality and equivalence are different concepts:
- Equal if lexical values and datatypes are the same
- Equivalent if mapped to the same value
- E.g., “1”\~xsd:integer and “+01”\~xsd:byte are not equal, but are equivalent

RDF systems often normalise literals on import
- E.g., “+01”\~xsd:byte is stored as “1”\~xsd:integer
RDF and RDFS

RDF Schema (RDFS)

- RDFS: a simple ontology language for RDF data
- Introduces special vocabulary
  - E.g., `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, ...
  - Schema not separate from data ⇒ schema is data
- RDF(S) semantics specifies consequences of the special vocabulary
  - [http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/](http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/)
  - Can be captured using entailment rules
  - E.g., ‘If ?X is an instance of ?Y, and ?Y is a subclass of ?Z, then ?X is an instance of ?Z’
RDF Schema (RDFS)

- RDFS: a simple ontology language for RDF data
- Introduces special vocabulary
  - E.g., `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, ...
  - Schema not separate from data ⇒ schema is data
- RDF(S) semantics specifies consequences of the special vocabulary
  - [http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/](http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/)
  - Can be captured using entailment rules
    - E.g., ‘If ?X is an instance of ?Y, and ?Y is a subclass of ?Z, then ?X is an instance of ?Z’
Datalog captures entailment rules in a formal way

Related to Prolog, widely used in databases and Semantic Web

**Term** — a node or a variable
- E.g., ?X, sometimes also written as #X

**(RDF) atom** — a triple in which s, p, and o are terms (not just nodes)
- E.g., ⟨?X, rdf:type, :City⟩, ⟨?X, :locatedIn, ?Y⟩
- General atoms have form R(t₁, . . . , tₙ) for R an n-ary relation
  - In RDF, there is just one ‘triple’ relation so we omit it
- Equivalent logical notation:
  - Classes → unary relations: ⟨?X, rdf:type, :City⟩ ≡ :City(?X)
  - Properties → binary relations: ⟨?X, :locatedIn, ?Y⟩ ≡ :locatedIn(?X, ?Y)
  - Works if triples do not contain variables in class/property positions

**(Datalog) rule** — implication of the form \( H \leftarrow B₁ \land \ldots \land Bₙ \)
- Also written as \( H :− B₁, \ldots, Bₙ \).
- \( H \) is the **head** atom
- \( B₁, \ldots, Bₙ \) are **body** atoms
- Each rule must be **safe**: each variable in the rule must occur in some body atom

**(Datalog) program** — a finite set of rules
Capturing Entailment Rules of RDFS in Datalog

- Entailments about schema:

\[
\begin{align*}
\langle ?X, \text{rdfs:subClassOf}, ?Z \rangle & \leftarrow \langle ?X, \text{rdfs:subClassOf}, ?Y \rangle \land \langle ?Y, \text{rdfs:subClassOf}, ?Z \rangle \\
\langle ?X, \text{rdfs:subPropertyOf}, ?Z \rangle & \leftarrow \langle ?X, \text{rdfs:subPropertyOf}, ?Y \rangle \land \langle ?Y, \text{rdfs:subPropertyOf}, ?Z \rangle \\
\langle ?X, \text{rdfs:domain}, ?Z \rangle & \leftarrow \langle ?X, \text{rdfs:domain}, ?Y \rangle \land \langle ?Y, \text{rdfs:subPropertyOf}, ?Z \rangle \\
\langle ?X, \text{rdfs:range}, ?Z \rangle & \leftarrow \langle ?X, \text{rdfs:range}, ?Y \rangle \land \langle ?Y, \text{rdfs:subPropertyOf}, ?Z \rangle 
\end{align*}
\]

- Rules in red are not mentioned in standards, but should be
- This part of the standard is, IMHO, poorly designed

- Entailments about data:

\[
\begin{align*}
\langle ?X, \text{rdf:type}, ?Z \rangle & \leftarrow \langle ?X, \text{rdf:type}, ?Y \rangle \land \langle ?Y, \text{rdfs:subClassOf}, ?Z \rangle \\
\langle ?X, \text{rdf:type}, ?Z \rangle & \leftarrow \langle ?X, ?W, ?Y \rangle \land \langle ?W, \text{rdfs:domain}, ?Z \rangle \\
\langle ?Y, \text{rdf:type}, ?Z \rangle & \leftarrow \langle ?X, ?W, ?Y \rangle \land \langle ?W, \text{rdfs:range}, ?Z \rangle 
\end{align*}
\]

- Rules are fixed ⇒ do not depend on the ontology
ALTERNATIVE: ONTOLOGY-SPECIFIC ENTAILMENT RULES

One can use rules created for each ontology separately:

\[
\langle ?X, \text{rdf:type, } \text{:Accommodation}\rangle \leftarrow \langle ?X, \text{rdf:type, } \text{:Hotel}\rangle \\
\langle ?X, \text{rdf:type, } \text{:Accommodation}\rangle \leftarrow \langle ?X, \text{locatedIn, } ?Y\rangle \\
\quad \langle ?Y, \text{rdf:type, } \text{:City}\rangle \leftarrow \langle ?X, \text{locatedIn, } ?Y\rangle \\
\langle ?X, \text{containedIn, } ?Y\rangle \leftarrow \langle ?X, \text{locatedIn, } ?Y\rangle
\]

Often written using logical syntax:

\[
:\text{Accommodation}(?X) \leftarrow :\text{Hotel}(?X) \\
:\text{Accommodation}(?X) \leftarrow :\text{locatedIn}(?X, ?Y) \\
\quad :\text{City}(?X) \leftarrow :\text{locatedIn}(?X, ?Y) \\
\quad :\text{containedIn}(?X, ?Y) \leftarrow :\text{locatedIn}(?X, ?Y)
\]

More rules, but fewer body atoms
- More efficient due to shorted rules
- Can capture only data entailments

Semantic Web Rule Language (SWRL)

- De facto standard for rules on the Web
- [http://www.w3.org/Submission/SWRL/](http://www.w3.org/Submission/SWRL/)
- Several syntaxes, one of them encodes rules into RDF

```
<ruleml:imp>
  <ruleml:_body>
    <owlx:Class owlx:name="Hotel" />
    <ruleml:var>X</ruleml:var>
  </swrlx:classAtom>
</ruleml:_body>
<ruleml:_head>
  <swrlx:classAtom>
    <owlx:Class owlx:name="Accommodation" />
    <ruleml:var>X</ruleml:var>
  </swrlx:classAtom>
</ruleml:_head>
</ruleml:imp>
```
Basics of Datalog

**Rule Interchange Format (RIF)**

- A standard for rules on the Web
- http://www.w3.org/standards/techs/rif#w3c_all
- IMHO, mostly used in production rule systems, not the Semantic Web

```xml
Document(
  Prefix(sky <http://skyscanner.net/>)

  Group (
    Forall ?X (sky:Accommodation(?X) :- sky:Hotel(?X))
    Forall ?X (sky:Accommodation(?X) :- sky:locatedIn(?X ?Y))
    Forall ?Y (sky:City(?X) :- sky:locatedIn(?X ?Y))
    Forall ?Y (sky:containedIn(?X ?Y) :- sky:locatedIn(?X ?Y))
  )
)
```

B. Motik
An Introduction to Semantic Technologies 34/83
Rules can express recursive queries!
Significantly more expressive than relational databases
- WITH clause in SQL-1999 supports limited recursion
- Not widely (efficiently) implemented

Reachability:

\[
: \text{Reachable}(\text{?}Y) \leftarrow : \text{Reachable}(\text{X}) \land : \text{connected}(\text{?X}, \text{?Y}) \\
: \text{Reachable}(\text{:source})
\]

Transitivity:

\[
: \text{connected}(\text{?X}, \text{?Z}) \leftarrow : \text{connected}(\text{?X}, \text{?Y}) \land : \text{connected}(\text{?Y}, \text{?Z})
\]
A program is **recursive** if its rule-goal graph contains a cycle.
Iterative semantics: apply rules as long as new facts are derived

Example rule: $\langle ?Y, rdf: type, :A \rangle \leftarrow \langle ?X, rdf: type, :A \rangle \land \langle ?X, :R, ?Y \rangle$

The number of iterative steps depends on the program and the data. Cannot be determined in advance by just looking at the program.

Crucial aspect of recursion: semantics just specifies the meaning; implementation can be different.
Iterative semantics: apply rules as long as new facts are derived

Example rule: $\langle ?Y, rdf:type, :A \rangle \leftarrow \langle ?X, rdf:type, :A \rangle \land \langle ?X, :R, ?Y \rangle$
- **Iterative** semantics: apply rules as long as new facts are derived
- Example rule: \( \langle ?Y, \text{rdf:type}, :A \rangle \leftarrow \langle ?X, \text{rdf:type}, :A \rangle \land \langle ?X, :R, ?Y \rangle \)
Iterative semantics: apply rules as long as new facts are derived

Example rule: \( \langle ?Y, rdf: type, :A \rangle \leftarrow \langle ?X, rdf: type, :A \rangle \land \langle ?X, :R, ?Y \rangle \)
Iterative semantics: apply rules as long as new facts are derived

Example rule: \(<? Y, \text{rdf:type, :A} \rangle \leftarrow \langle ?X, \text{rdf:type, :A} \rangle \land \langle ?X, :R, ?Y \rangle \rangle
Iterative semantics: apply rules as long as new facts are derived

Example rule: \( \langle ?Y, \text{rdf:type}, :A \rangle \leftarrow \langle ?X, \text{rdf:type}, :A \rangle \land \langle ?X, :R, ?Y \rangle \)

The number of iterative steps depends on the program and the data

- Cannot be determined in advance by just looking at the program
- Crucial aspect of recursion

Semantics just specifies the meaning: implementation can be different
Benefits of OWL at a glance:
- Decidable, but yet very expressive fragment of datalog
- More user-friendly representation style (no variables)
- W3C standard (http://www.w3.org/TR/owl2-overview/)

Can describe complex concepts using **class expressions**
- E.g., ‘Hotel located at some beach’, ‘Hotel with exactly two swimming pools’, ‘Not a hotel’, ‘Hotel with only non-smoking rooms’, ‘Hotel or B&B’
- Features: conjunction, disjunction, negation, existential and universal quantification, and cardinality restrictions

Can describe class expression hierarchies
- E.g., ‘Each country is headed by a king or a president’, ‘A kingdom is a country headed only by a king’, ‘Nobody is both a king and a president’, ‘A king is a monarch’, ‘A country headed by a monarch is a monarchy’

Can express complex role properties
- ‘A friend of a friend is a friend’, ‘An enemy of an enemy is a friend’, ‘A father’s brother is an uncle’, ‘If A is reachable from B, then B is reachable from A’
SubClassOf(
    :Country
    ObjectSomeValuesFrom( :headedBy ObjectUnionOf( :King :President ) )
)

SubClassOf(
    :Kingdom
    ObjectIntersectionOf(
        :Country
        ObjectAllValuesFrom( :headedBy :King )
    )
)

DisjointClasses( :King :President )

SubClassOf( :King :Monarch )

SubClassOf(
    ObjectIntersectionOf(
        :Country ObjectSomeValuesFrom( :headedBy :Monarch )
    )
    :Monarchy
)
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
  <rdf:Class rdf:about="http://skyscanner.net/Country">
    <rdfs:subClassOf>
      <rdf:Restriction>
        <owl:onProperty rdf:resource="http://skyscanner.net/headedBy"/>
        <owl:someValuesFrom>
          <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
              <owl:Class rdf:about="http://skyscanner.net/King"/>
              <owl:Class rdf:about="http://skyscanner.net/President"/>
            </owl:unionOf>
          </owl:Class>
          <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
              <owl:Class rdf:about="http://skyscanner.net/King"/>
              <owl:Class rdf:about="http://skyscanner.net/President"/>
            </owl:unionOf>
          </owl:Class>
        </owl:someValuesFrom>
      </rdf:Restriction>
    </rdfs:subClassOf>
  </rdf:Class>
</rdf:RDF>

- Widely used, but awkward and unreadable ⇒ mostly machine-generated!
Prefix: : <http://skyscanner.net/>

:Country
    SubClassOf: :headedBy some ( :King or :President )

:Kingdom
    SubClassOf: :Country and ( :headedBy all :King )

:King
    SubClassOf: :Monarch
    DisjointWith: :President

:Auxiliary
    EquivalentTo: :Country and ( :headedBy some :Monarch )
    SubClassOf: :Monarchy

- Compact and readable
- Does not cover OWL 2 faithfully → hence the :Auxiliary class!
Description logics (DLs) provide the formal underpinning of OWL

- Studied in-depth in theory
- Tradeoff between complexity and expressivity is well understood
- Extensive body of research in practical reasoning

More compact syntax, used mostly by theoreticians in academic publications:

```ontop
:Country ⊑ ∃:headedBy.(:King ⊔ :President)
:Kingdom ⊑ :Country ⊓ ∀:headedBy.:King
:King ⊓ :President ⊑ ⊥
:King ⊑ :Monarch
:Country ⊓ ∃:headedBy.:Monarch ⊑ :Monarchy
```
Other OWL Features

- XSD datatypes and (in OWL 2) facets, e.g.,
  - integer, string and (in OWL 2) real, float, decimal, datetime, ...
  - minExclusive, maxExclusive, length, ...
  - PropertyAssertion( hasAge Meg "17"^^xsd:integer )
  - DatatypeRestriction( xsd:integer xsd:minInclusive "5"^^xsd:integer xsd:maxExclusive "10"^^xsd:integer )

These are equivalent to (a limited form of) DL concrete domains

- Keys
  - E.g., HasKey(Vehicle Country LicensePlate)
    - Country + License Plate is a unique identifier for vehicles

This is equivalent to (a limited form of) DL safe rules
Other OWL Features

Keys

- HasKey axioms provide functionality similar to keys in relational databases.

- A HasKey axiom is of the form:

  \[
  \text{HasKey}(C(p_1 \ldots p_n)(d_1 \ldots d_m))
  \]

  where \( C \) is a class, \( p_i \) is an object property and \( d_j \) is a data property.

- Axiom states that no two distinct named instances of class \( C \) can be related to the same set of individuals and literals via the given properties.
Other OWL Features

Keys

More formally, if ontology $\mathcal{O}$ includes an axiom:

$$\text{HasKey}(C(p_1 \ldots p_n)(d_1 \ldots d_m))$$

then a model $\mathcal{I}$ of $\mathcal{O}$ has to satisfy the following condition:

For each pair $a, b$ of individuals occurring in $\mathcal{O}$, with \{a$^\mathcal{I}$, b$^\mathcal{I}$\} $\subseteq C^\mathcal{I}$, and for each $e \in \Delta^\mathcal{I}$, $v \in \Delta^D$, $1 \leq i \leq n$ and $1 \leq j \leq m$, if:

- $(a^\mathcal{I}, e) \in p^\mathcal{I}_i \iff (b^\mathcal{I}, e) \in p^\mathcal{I}_i$ and
- $(a^\mathcal{I}, v) \in d^\mathcal{I}_j \iff (b^\mathcal{I}, v) \in d^\mathcal{I}_j$

then $a^\mathcal{I} = b^\mathcal{I}$. 
Other OWL Features

For example, if an ontology $\mathcal{O}$ includes the following axiom and assertions:

\[
\text{HasKey}(\text{:Person}, \text{:hasChild}, \text{:hasGender}) \\
\text{ClassAssertion}(\text{:Person}, \text{:Elizabeth}) \\
\text{ObjectPropertyAssertion}(\text{:hasChild}, \text{:Elizabeth}, \text{:Mary}) \\
\text{DataPropertyAssertion}(\text{:hasGender}, \text{:Elizabeth}, \text{"F"}) \\
\text{ClassAssertion}(\text{:Person}, \text{:Liz}) \\
\text{ObjectPropertyAssertion}(\text{:hasChild}, \text{:Liz}, \text{:Mary}) \\
\text{DataPropertyAssertion}(\text{:hasGender}, \text{:Liz}, \text{"F"})
\]

then $\mathcal{O}$ entails $\text{SameIndividual}(\text{:Elizabeth}, \text{:Liz})$. If $\mathcal{O}$ additionally includes the following axioms and assertions...
Other OWL Features

ClassAssertion( ObjectSomeValuesFrom( hasFriend :P ) :John )
SubClassOf( :P ObjectHasValue( hasChild :Mary ) )
SubClassOf( :P DataHasValue( hasGender "F" ) )
SubClassOf( :P :Person ), SubClassOf( :P :Happy ),
ClassAssertion( ObjectComplementOf( :Happy ) :Liz )

then is $\mathcal{O}$ inconsistent?
Other OWL Features

Anonymous Individuals

- Recall that ABox assertions in OWL directly correspond to RDF triples of the form \( \langle a, rdf\text{:type}, C \rangle \) and \( \langle a, p, b \rangle \), where \( C \) is a class, \( p \) is a property, and \( a, b \) are IRIs.

- Unlike standard DLs, \( a \) and \( b \) do not have to be named individuals, but can also be RDF blank nodes.

- Blank nodes are denoted by the use of \( _{\text{x}} \) as an IRI prefix (e.g., \( _{\text{x}}: x \)), and are treated as variables that are existentially quantified at the outer level of the ABox.

- In OWL, blank nodes used in ABox assertions are called anonymous individuals.
Relationship to Description Logics (DLs)

Other OWL Features

For example, the assertions

\[
\text{ObjectPropertyAssertion(} \: \text{hasFriend} :Liz \_:_x \) \\
\text{ObjectPropertyAssertion(} \: \text{livesIn} \_:_x \_:_y \) \\
\text{ObjectPropertyAssertion(} \: \text{livesIn} :Mary \_:_y \)
\]

assert that :Liz has a friend who lives in the same place as :Mary without explicitly naming the friend or the place where they live; they are semantically equivalent to a first-order logic sentence of the form

\[
\exists x \exists y (\text{hasFriend}(Liz, x) \land \text{livesIn}(x, y) \land \text{livesIn}(Mary, y)).
\]

These assertions can also be written as a semantically equivalent SROIQ concept assertion

\[
Liz : \exists \text{hasFriend}.(\exists \text{livesIn}.(\exists \text{livesIn}^\neg .\{Mary\})),
\]
Other OWL Features

Metamodelling

In some applications it may be desirable to use the same name for both a class (or property) and an individual. For example, we might want to state that `:Harry` is an instance of `:Eagle`

```owl
ClassAssertion( :Eagle :Harry )
```

and that `:Eagle` is an instance of `:EndangeredSpecies`

```owl
ClassAssertion( :EndangeredSpecies :Eagle )
```

We could then extend our modelling of the domain to describe classes of classes, e.g., by stating that it is illegal to hunt any class of animal that is an instance of `:EndangeredSpecies`; this is often called metamodelling. Metamodelling is not possible in a standard DL, where it is usually assumed that the sets C, R and I (of, respectively, concept, role and individual names) are pairwise disjoint, and where class assertions can only be used to describe individual names; i.e., in an assertion `a : C`, a must be an individual name.
Other OWL Features

- OWL 2 uses a mechanism known as *punning* to provide a simple form of metamodelling while still retaining the correspondence between OWL ontologies and *SROIQ* KBs.

- Punning allows for the same IRI to be used as an individual, a class and a property, but IRIs used in the individual, class and property contexts are semantically unrelated.

- This is equivalent to rewriting the ontology by adding unique prefixes such as \(i:\), \(c:\) and \(p:\) to IRIs according to the context in which they occur. For example:

\[
\text{ClassAssertion( } c:\text{Eagle } i:\text{Harry } ) \\
\text{ClassAssertion( } c:\text{EndangeredSpecies } i:\text{Eagle } )
\]
Other OWL Features

Annotations

- OWL includes a flexible annotation mechanism that allows for comments and other “non-logical” information to be included in the ontology.

- An OWL annotation consists of an annotation property and a literal, and zero or more annotations can be attached to class, property and individual names, to axioms and assertions, to datatypes, to the ontology as a whole, and even to annotations themselves; for example:

```owl
ClassAssertion( Annotation( rdfs:comment "Liz is a person" ) :Person :Liz )
```
Other OWL Features

Imports

- The OWL Import statement provides a mechanism for “importing” the contents of one ontology document into another.

- For example, if :ont1 includes the statement:

  \[
  \text{Import( :ont2 )}
  \]

  then :ont1 is treated as though it also includes all of the contents of :ont2 and, recursively, any ontology documents imported by :ont2.

- The OWL specification defines a parsing procedure that extracts ontological content from the current ontology document and all those that it (possibly recursively) imports, while ensuring termination even if ontology documents (directly or indirectly) import each other cyclically.
**Example**

- Known fact: ‘Mary is a woman’
- Question: ‘Does Mary have a daughter?’
  - Database/datalog answer: ‘No’ → intuitive!

- Question: ‘Does Mary not have a daughter?’
  - Intuitive answer: ‘Don’t know’ → not enough information!
  - Database/datalog answer: ‘No’ → not in the database, so ‘No’
Databases/datalog assume complete knowledge
- Everything that is not provable is false → closed-world assumption
- Appropriate in some cases: flight schedules, corporate profits, ...
- Inappropriate in others: mathematics, certain common-sense reasoning, ...

Many situations have incomplete knowledge
- Negative information must be explicitly provable

**Example**

Known facts: ‘Every man is a person’, ‘Garfield is not a person’

Can deduce ‘Garfield is not a man’ → proof by contradiction
1. Assume the opposite: ‘Garfield is a man and not a person’
2. By ‘Every man is a person’, we have ‘Garfield is a man, a person, and not a person’
3. This is a contradiction, so ‘Garfield is a man’ cannot be true
4. But ‘Either Garfield is a man, or Garfield is not a man’ (aka law of excluded middle)
5. Hence, ‘Garfield is not a man’ is true
Classical negation $\neg$ works under incomplete knowledge

- Comes from propositional and first-order predicate logic
- Very different from database-style *not* from datalog
- Used in OWL 2 as ObjectComplementOf

**Example**

\[
:Man(:\text{garfield}) \quad \neg :Person(:\text{garfield}) \quad \forall X.[:Person(\,X) \iff :Man(\,X)]
\]

- Can use $\neg$ in front of facts or rule heads (e.g., $\neg :Person(:\text{garfield})$)
- Material implication $\iff$ is different from datalog implication $\leftarrow$

\[
\begin{align*}
A & \iff B \\
A \lor \neg B \\
\bot & \iff \neg A \land B \\
\neg B & \iff \neg A
\end{align*}
\]

all equivalent to each other
## Comparing Two Kinds of Implication

<table>
<thead>
<tr>
<th>Material implication</th>
<th>Datalog implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ontology:</strong></td>
<td></td>
</tr>
<tr>
<td>$\forall ?X. [\text{:Person}(?X) \iff \text{:Man}(?X)]$</td>
<td>$\text{:Person}(?X) \iff \text{:Man}(?X)$</td>
</tr>
<tr>
<td><strong>Facts:</strong></td>
<td></td>
</tr>
<tr>
<td>$\text{:Man}(\text{peter})$</td>
<td>$\text{:Man}(\text{peter})$</td>
</tr>
<tr>
<td>$\text{:Man}(\text{paul})$</td>
<td>$\text{:Man}(\text{paul})$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td><strong>Conclusions:</strong></td>
<td></td>
</tr>
<tr>
<td>$\text{:Person}(\text{peter})$</td>
<td>$\text{:Person}(\text{peter})$</td>
</tr>
<tr>
<td>$\text{:Person}(\text{paul})$</td>
<td>$\text{:Person}(\text{paul})$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

⇒ No observable difference on negation-free rules and positive facts.

More facts: $\neg \text{:Person}(\text{garfield})$ Syntax error!

More conclusions: $\neg \text{:Man}(\text{garfield})$

⇒ Difference observable if facts or rules contain negation.

- Lots of theoretical work on integrating the two → very hard problem!
OWL 2 Profiles

- Reasoning in OWL 2 is of high worst-case computational complexity
  - Undecidable for the RDF version of OWL 2
  - \( \text{N2ExpTime} \) for the DL version of OWL 2

- OWL 2 profiles trade some expressivity for lower complexity
  - [http://www.w3.org/TR/owl2-profiles/](http://www.w3.org/TR/owl2-profiles/)

- OWL 2 RL
  - No support for incomplete information
  - Can be implemented fully using datalog (without negation)
  - Targets mainly database-like warehousing-style applications

- OWL 2 QL
  - Incompleteness via existential quantification, but not disjunction
  - No support for recursion
  - Can be implemented using query rewriting
  - Targets virtual information integration

- OWL 2 EL
  - Incompleteness via existential quantification, but not disjunction
  - Supports recursion
  - Tractable query answering
  - Targets applications that rely on expressive taxonomies
Current version 1.1

http://www.w3.org/TR/sparql11-query/

Used to query RDF and OWL systems

Uses a familiar SELECT-WHERE paradigm

Two parts:
- **Basic SPARQL** → roughly as expressive as SQL
  - No recursive queries
- **Property paths** in 1.1 version → expressivity beyond SQL
  - Supports property paths → a form of recursion
Matching of graph patterns

- Entailment regimes determine semantics of matches

Relational algebra over answers to graph patterns

- Union, subtraction, subqueries, built-in expressions, aggregate functions
- No NULL-values, but variables can be unbound

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://skyscanner.net/>

SELECT ?H ?N WHERE {
  ?H rdf:type :Hotel ; :hasName ?N ; :hasAmenity :Wifi .
}

SELECT ?H ?N ?D WHERE {
}

SELECT ?A WHERE {
  { ?A rdf:type :Hotel } UNION { ?A rdf:type :Hostel }
}

SELECT ?H WHERE {
  ?H rdf:type :Hotel } MINUS { ?H :locatedIn :Prague }
```
Terms can be connected by regular expressions over properties

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://skyscanner.net/>
SELECT ?H WHERE {
    ?H rdf:type :Hotel ; :inCity/:inCountry :Germany .
}

SELECT ?C1 ?C2 WHERE {
    ?C1 rdf:type :Country (:hasLandBorderWith/:hasLandBorderWith?) ?C2 .
}

SELECT ?C WHERE {
    ?C rdf:type :Country ; :hasLandBorderWith+ :Germany .
}
```

- Regular expressions support a form of recursion
- Blurs the distinction between reasoning and querying
- Such queries are common in graph databases (e.g., Neo4j)