[SYNTAX] OWL:ALLDIFFERENT IN RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:f="foo://bla/"  xml:base="foo://bla/">
 <owl:Class rdf:about="Foo">
  <owl:equivalentClass> <owl:Class>
    <owl:oneOf rdf:parseType="Collection">
     <owl:Thing rdf:about="a"/>  <owl:Thing rdf:about="b"/>
     <owl:Thing rdf:about="c"/>  <owl:Thing rdf:about="d"/>
    </owl:oneOf>
   </owl:Class> </owl:equivalentClass>
 </owl:Class>
 <owl:AllDifferent>  <!-- use like a class, but is only a shorthand -->
    <owl:members rdf:parseType="Collection">
     <owl:Thing rdf:about="a"/>  <owl:Thing rdf:about="b"/>
     <owl:Thing rdf:about="c"/>  <owl:Thing rdf:about="d"/>
    </owl:members>
 </owl:AllDifferent>
 <owl:Thing rdf:about="a"> <owl:sameAs rdf:resource="b"/> </owl:Thing>
</rdf:RDF>
```

[Filename: RDF/alldiff.rdf]

```
prefix : <foo://bla/>
prefix owl:
  <http://www.w3.org/2002/07/owl#>
select ?X ?P ?P2 ?V
from <file:alldiff.rdf>
where {?X a owl:AllDifferent ;
       ?P [?P2 ?V]}
```

[Filename: RDF/alldiffxml.sparql]

- AllDifferent is only intended as a kind of command to the application to add all pairwise "different-from" statements, it does not actually introduce itself as triples:
- querying {?X a owl:AllDifferent} is actually not intended.

[SYNTAX] OWL:ALLDIFFERENT IN N3

Example:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.
:Foo owl:equivalentClass [ owl:oneOf (:a :b :c :d) ].
# both the following syntaxes are equivalent and correct:
[ a owl:AllDifferent; owl:members (:a :b)].
[] a owl:AllDifferent; owl:members (:c :d).
:a owl:sameAs :b.
# :b owl:sameAs :d.
```

[Filename: RDF/alldiff.n3]

```
prefix : <foo://bla/>
select ?X ?Y
from <file:alldiff.n3>
where {?X a owl:AllDifferent ; ?P [?P2 ?V]}
```

[Filename: RDF/alldiff.sparql]

ONEOF: A TEST

- owl:oneOf defines a “closed set” (use with anonymous class; see below):
- note that in owl:oneOf ($x_1, \ldots, x_n$), two items may be the same (open world),
- optional owl:AllDifferent to guarantee that ($x_1, \ldots, x_n$) are pairwise distinct.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.
:Person owl:equivalentClass [ owl:oneOf (:john :alice :bob) ].
# :john owl:sameAs :alice.  # to show that it is consistent that they are the same
[] a owl:AllDifferent; owl:members (:john :alice :bob).  # to guarantee distinctness
# :name a owl:FunctionalProperty.  # this also guarantees distinctness ;)
:john :name "John".
:alice :name "Alice".
:bob :name "Bob".
:d a :Person.
:d owl:differentFrom :john, :alice.
# :d owl:differentFrom :bob.   ### adding this makes the ontology inconsistent
```

[Filename: RDF/three.n3]

- Who is :d?

oneOf: a Test (cont’d)

Who is :d?

- check the class tree:
  bla:Person - (bla:bob, bla:alice, bla:d, bla:john)
- and ask it:

```
prefix : <foo://bla/>
select ?N
from <file:three.n3>
where {:d :name ?N}
```

[Filename: RDF/three.sparql]
The answer is ?N/“Bob”.

7.5 Closing Parts of the Open World

- "forall items" is only applicable if additional items can be excluded ($\Rightarrow$ locally closed predicate/property),
- often, RDF data is generated from a database,
- certain predicates can be closed by defining restriction classes with maxCardinality.

OWL:ALLVALUESFROM

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#>.
[ a :Male; a :ThreeChildrenParent; :name "John";
   :child [a :Female; :name "Alice"], [a :Male; :name "Bob"],
          [a :Female; :name "Carol"]].
[ a :Female; a :TwoChildrenParent; :name "Sue";
   :child [a :Female; :name "Anne";], [a :Female; :name "Barbara"]].
:name a owl:FunctionalProperty.
:OneChildParent owl:equivalentClass [a owl:Re
  owl:onProperty :child; owl:cardinality 1].
:TwoChildrenParent owl:equivalentClass [a owl
  owl:onProperty :child; owl:cardinality 2].
:ThreeChildrenParent owl:equivalentClass [a owl:Restriction;
  owl:onProperty :child; owl:cardinality 3].
:OnlyFemaleChildrenParent owl:equivalentClass [a owl:Restriction;
  owl:onProperty :child; owl:allValuesFrom :Female].
```

[Filename: RDF/allvaluesfrom.n3]

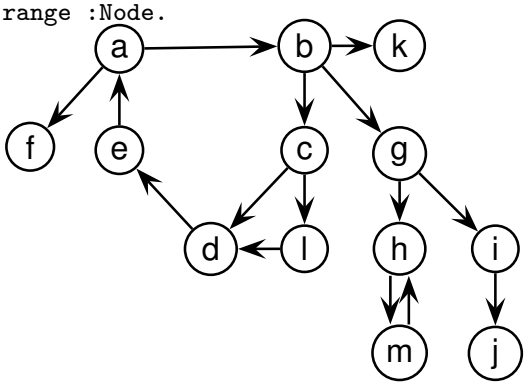
```
prefix : <foo://bla/names#>
select ?N
from <file:allvaluesfrom.n3>
where {?X :name ?N .
  ?X a :OnlyFemaleChildrenParent}
```

[Filename: RDF/allvaluesfrom.sparql]

EXAMPLE: WIN-MOVE-GAME IN OWL

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.

:Node a owl:Class; owl:equivalentClass
  [ a owl:Class; owl:oneOf (:a :b :c :d :e :f :g :h :i :j :k :l :m)].
:edge a owl:ObjectProperty; rdfs:domain :Node; rdfs:range :Node.
:out a owl:DatatypeProperty.
:a a :Node; :out 2; :edge :b, :f.
:b a :Node; :out 3; :edge :c, :g, :k.
:c a :Node; :out 2; :edge :d, :l.
:d a :Node; :out 1; :edge :e.
:e a :Node; :out 1; :edge :a.
:f a :Node; :out 0 .
:g a :Node; :out 2; :edge :i, :h.
:h a :Node; :out 1; :edge :m.
:i a :Node; :out 1; :edge :j.
:j a :Node; :out 0 .
:k a :Node; :out 0 .
:l a :Node; :out 1; :edge :d.
:m a :Node; :out 1; :edge :h.
```



[Filename: RDF/winmove-graph.n3]

Win-Move-Game in OWL – the Game Axioms

"If a player cannot move, he loses."

Which nodes are WinNodes, which one are LoseNodes (i.e., the player who has to move wins/loses)?

- if a player can move to some LoseNode (for the other), he will win.
- if a player can move only to WinNodes (for the other), he will lose.
- recall that there can be nodes that are neither WinNodes nor LoseNodes.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.

:WinNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:someValuesFrom :LoseNode]).
:LoseNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:allValuesFrom :WinNode]).
```

[Filename: RDF/winmove-axioms.n3]

Win-Move-Game in OWL – Closure

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.
:DeadEndNode a owl:Class;  rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 0],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 0].
:OneExitNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 1],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 1].
:TwoExitsNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 2],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 2].
:ThreeExitsNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 3],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 3].
```

[Filename: RDF/winmove-closure.n3]

Win-Move-Game in OWL: DeadEndNodes

Prove that DeadEndNodes are LoseNodes:

- obvious: Player cannot move from there
- exercise: give a formal (Tableau) proof
- The OWL Reasoner does it:

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <foo://bla/>
select ?X
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
where {:DeadEndNode rdfs:subClassOf :LoseNode}
```

[Filename: RDF/deadendnodes.sparql]

The answer contains an (empty) tuple which means “yes”.

Win-Move-Game in OWL

```
prefix : <foo://bla/>
select ?W ?L ?DE
from <file:winmove-graph.n3>
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
where {{?W a :WinNode} UNION
       {?L a :LoseNode} UNION
       {?DE a :DeadEndNode}}
```

[Filename: RDF/winmove.sparql]

lose: f, k, j, e, l
win: c, a, i, b, d

Exercise

- Is it possible to characterize DrawNodes in OWL?
  - 2 alternative variants:
    - using the game axioms/rules,
    - consider the possible values: win/lost/drawn,
  - test with *typical* minimal examples,
  - explain the results [DB Theory: compare also with well-founded and stable models].

7.6 OWL 2 (W3C Recommendation since October 2009)

- OWL2 notions belong to the OWL namespace
  (aside: development proposal owl11 used a separate namespace)
- Syntactic Sugar: owl:disjointUnionOf and negative assertions: ObjectPropertyAssertion vs. NegativeObjectPropertyAssertion
- User-defined datatypes (like XML Schema simple types).
- $\mathcal{SROIQ}$: Qualified cardinality restrictions (only for non-complex properties), local reflexivity restrictions (individuals that are related to themselves via the given property), reflexive, irreflexive, symmetric, and anti-symmetric properties (only for non-complex properties), disjoint properties (only for non-complex properties), Property chain inclusion axioms (e.g., SubPropertyOf(PropertyChain(owns hasPart) owns) asserts that if x owns y and y has a part z, then x owns z.
- $\mathcal{SROIQ}(D)$ is decidable.
  The Even More Irresistible $\mathcal{SROIQ}$. Ian Horrocks, Oliver Kutz, and Ulrike Sattler. In Principles of Knowledge Representation and Reasoning (KR 2006). AAAI Press, 2006. Available at `www.cs.man.ac.uk/~sattler/publications/sroiq-tr.pdf`.

OWL: Disjoint Union

... syntactic sugar for owl:unionOf and owl:disjointWith:

(only a simple test and syntax example)

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:f="foo://bla/"
   xml:base="foo://bla/">
 <owl:Class rdf:about="Person">
  <owl:disjointUnionOf rdf:parseType="Collection">
   <owl:Class rdf:about="Male"/>
   <owl:Class rdf:about="Female"/>
  </owl:disjointUnionOf>
 </owl:Class>
 <f:Male rdf:about="John"/>
 <f:Female rdf:about="Mary"/>
 <!--<f:Female rdf:about="John"/>-->
</rdf:RDF>
```

[Filename: RDF/disjointunion.xml]

```
prefix f: <foo://bla/>
select ?X
from <file:disjointunion.xml>
where {?X a f:Person}
```

[Filename: RDF/disjointunion.sparql]

Example: Parricides in Greek Mythodology

(from ESWC'07 SPARQL tutorial by Marcelo Arenas et al)

A parricide is a person who killed his/her father.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:greek#>.
:Person owl:disjointUnionOf (:Parricide :Non-Parricide).
  :iokaste a :Person; :child :oedipus.
  :oedipus a :Person, :Parricide; :married-to :iokaste; :child :perineikes.
  :perineikes a :Person; :child :thesandros.
  :thesandros a :Person; a :Non-Parricide.
:Parent-of-Parricide owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :child; owl:someValuesFrom :Parricide ].
:Parent-of-Non-Parricide owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :child; owl:someValuesFrom :Non-Parricide ].
:Parent-of-Parricide-Grandparent-of-Non-Parricide owl:intersectionOf
  ([a owl:Restriction; owl:onProperty :child; owl:someValuesFrom :Parricide]
   [a owl:Restriction;
     owl:onProperty :child; owl:someValuesFrom :Parent-of-Non-Parricide]).
```

[Filename: RDF/parricide.n3]

EXAMPLE (CONT'D)

- have a short look on the results:

```
prefix : <foo:greek#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?P ?NP ?PP ?PNP ?X
from <file:parricide.n3>
where {{?P a :Parricide} UNION
       {?NP a :Non-Parricide} UNION
       {?PP a :Parent-of-Parricide} UNION
       {?PNP a :Parent-of-Non-Parricide} UNION
       {?X a :Parent-of-Parricide-Grandparent-of-Non-Parricide}}
```

[Filename: RDF/parricide.sparql]

- No X reported.

Example (Cont'd)

- ask Zeus whether Parent-of-Parricide-Grandparent-of-Non-Parricide is really non-empty:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:greek#>.
:zeus :knows :iokaste, :oedipus, :perineikes, :thesandros.
:KnowsPoPGonP  owl:equivalentClass [ a owl:Restriction; owl:onProperty :knows;
  owl:someValuesFrom :Parent-of-Parricide-Grandparent-of-Non-Parricide ].
```

[Filename: RDF/parricide2.n3]

```
prefix : <foo:greek#>
select ?K ?X
from <file:parricide.n3>
from <file:parricide2.n3>
where {{?K a :KnowsPoPGonP} UNION
       {?X a :Parent-of-Parricide-Grandparent-of-Non-Parricide}}
```

[Filename: RDF/parricide2.sparql]

- Zeus is in K, i.e., he knows such a person (explicitly: he knows a person who must be a P.o.p.G.o.n.p),
- but neither SPARQL, nor Zeus know who that person is,
- it can be either Iokaste or Oedipus (depending on whether Perineikes is a parricide, which nobody knows).

Qualified Role Restrictions

- extends owl:Restriction, owl:onProperty, owl:{min/max}QualifiedCardinality (int value) with owl:on{Class/DataRange} as result class/type.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
:alice :name "Alice"; :hasAnimal :pluto, :struppi.
:john :name "John"; :hasAnimal :garfield, :odie.
:pluto a :Dog; :name "Pluto".
:struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".
:odie a :Dog; :name "Odie".
:name a owl:FunctionalProperty.
:Dog a owl:Class.   :Cat a owl:Class.
:Cat owl:disjointWith :Dog.
:HasTwoAnimals owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:minCardinality 2].
:HasTwoCats owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:onClass :Cat; owl:minQualifiedCardinality 2].
:HasTwoDogs owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:onClass :Dog; owl:minQualifiedCardinality 2].
```

[Filename: RDF/cats-and-dogs.n3]

```
prefix : <foo://bla/names#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
select ?X ?Y ?Z ?C
from <file:cats-and-dogs.n3>
where {{?X a :HasTwoCats} UNION
       {?Y a :HasTwoDogs} UNION
       {?Z a :HasTwoAnimals} UNION
       {?C rdfs:subClassOf :HasTwoAnimals}}
```

[Filename: RDF/cats-and-dogs.sparql]

Qualified Role Restrictions – Another Test

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
:alice :name "Alice"; :hasAnimal :pluto, :struppi.
:john :name "John"; :hasAnimal :garfield, :nermal, :odie.
:sue :hasAnimal :grizabella.              :grizabella :name "Grizabella".
:pluto a :Dog; :name "Pluto".             :struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".       :nermal a :Cat; :name "Nermal".
:odie a :Dog; :name "Odie".
:name a owl:FunctionalProperty.
:Dog a owl:Class.   :Cat a owl:Class.     :Cat owl:disjointWith :Dog.
:HasAnimal owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:minCardinality 1].
:HasCat owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:onClass :Cat; owl:minQualifiedCardinality 1].
:HasDog owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:someValuesFrom :Dog].
```

[Filename: RDF/hasanimals.n3]

- export class tree:
  HasCat and HasDog are (non-disjoint) subclasses of HasAnimal.
- “owl:onClass X & owl:minQualifiedCardinality 1” is equivalent to “owl:someValuesFrom X”.
- “owl:minCardinality 1” alone is equivalent to “owl:someValuesFrom owl:Thing”.

Qualified Role Restrictions – Another Test

```
@prefix : <foo://bla/names#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:TwoChildren owl:equivalentClass [a owl:Restriction;
  owl:onProperty :child; owl:cardinality 2].
:ThreeMaleChildren owl:equivalentClass [a owl:Restriction;
  owl:onProperty :child; owl:onClass :Male; owl:minQualifiedCardinality 3].
:TCTMC owl:equivalentClass [ owl:intersectionOf (:TwoChildren :ThreeMaleChildren) ].
```

[Filename: RDF/twochildren-threemale.n3

- export class tree:
- note that the ontology is not inconsistent, but that simply TCTMC is derived to be equivalent to owl:Nothing.

Negative Assertions

- Assert that something is known *not* to hold: NegativeObjectPropertyAssertion and NegativeDataPropertyAssertion
- with owl:sourceIndividual, owl:assertionProperty, and owl:targetIndividual or owl:targetValue.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .
family:john a :Person.
[ rdf:type owl:NegativePropertyAssertion;
  owl:sourceIndividual family:john;
  owl:assertionProperty :lives;
  owl:targetIndividual :germany].
:German owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :lives; owl:hasValue :germany ].
:NonGerman owl:complementOf :German.
```

[Filename: RDF/nongerman.n3]

```
prefix : <foo://bla/names#>
select ?P
from <file:nongerman.n3>
where {?P a :NonGerman}
```

[Filename: RDF/nongerman.sparql]

- John is derived to be a Non-German.

Comment on Negative Assertions

... are just syntactic sugar for a construct using complement classes (and actually implemented in the reasoner by this):

Any owl:NegativeObjectPropertyAssertion $\neg(x\ r\ y)$ is encoded as

- a restriction $R(r, y)$ based on owl:hasValue:
  $R(r, y) = \{x|(x\ r\ y)\}$
  (above: R(lives,germany) = :German)
- its complement $CompR(r, y) := \top \setminus R(r, y)$
  (above: CompR(lives,germany) = :NonGerman)
- and the assertion that $x \in CompR(r, y)$.
  (above: assert (:john a :NonGerman))

DATATYPES: HASVALUE WITH LITERAL VALUE

Characterize a class as the set of all things where a given property has a given value:

- all things in Mondial that have the name “Berlin”:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix : <foo:bla#>.
:Berlin owl:equivalentClass [ a owl:Restriction;
   owl:onProperty mon:name; owl:hasValue "Berlin"].
```

[Filename: RDF/has-literal-value.n3]

```
prefix : <foo:bla#>
select ?X
from <file:has-literal-value.n3>
from <file:mondial-europe.n3>
where {?X a :Berlin}
```

[Filename: RDF/has-literal-value.sparql]

- Often preferable: define an owl:DataRange (unary or enumeration), give it a url, and use some/allValuesFrom.

Enumerated Datatypes

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix uni: <foo://uni/>.
uni:graded a owl:FunctionalProperty;
 a owl:DatatypeProperty; rdfs:range uni:Grades.
uni:Grades a rdfs:Datatype;
 owl:equivalentClass [ a rdfs:Datatype;
  owl:oneOf ("1.0" "1.3" "1.7" "2.0" "2.3" "2.7" "3.0" "3.3" "3.7" "4.0") ] .
[ a uni:Thesis; uni:author <foo://bla/john>;
  uni:graded "2.5"].
```

[Filename: RDF/grades-one-of-namedset.n3]

- inconsistent: “2.5” does not belong to the allowed grades,
- note: “3” is also not allowed since “3” and “3.0” are different strings,
- see alternative next slide.

Enumerated Datatypes

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix uni: <foo://uni/>.
uni:graded a owl:FunctionalProperty;
 a owl:DatatypeProperty; rdfs:range [ a rdfs:Datatype;
  owl:oneOf (1 1.3 1.7 2.0 2.3 2.7 3 3.3 3.7 4) ] .
[ a uni:Thesis; uni:author <foo://bla/john>;
  uni:graded 2].
```

[Filename: RDF/grades-one-of-anonymous.n3]

```
prefix : <foo://uni/>
select ?X ?G
from <file:grades-one-of-anonymous.n3>
where {?X :graded ?G}
```

[Filename: RDF/grades-one-of-anonymous.sparql]

- grade 2.5 results in an inconsistency,
- internally (in case of an error message e.g.), the values are represented/handled as “2.3”^^xsd:decimal,
- parsing and output uses the default representation,
- both representations 2 and 2.0 are allowed.

OneOf on DataRange

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/names#>.
:Male a owl:Class.    :Female a owl:Class.
:Person owl:disjointUnionOf (:Male :Female).
:MaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
     owl:oneOf ("John"^^xsd:string "Bob"^^xsd:string) ] .
:FemaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
    owl:oneOf ("Alice"^^xsd:string "Carol"^^xsd:string) ].
:Male a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames])].
:Female a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :FemaleNames])].
:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John"^^xsd:string.
:alice a :Person; :name "Alice"^^xsd:string.
```

[Filename: RDF/names.n3]

```
prefix : <foo://bla/names#>
select ?C ?N
from <file:names.n3>
where {:john a ?C ; :name ?N}
```

[Filename: RDF/names.sparql]

Exercise

Consider again the ontology from the previous slide

- The name “Maria” is a female first name, but (mainly by catholics) also used as an additional first name for males, e.g. Rainer Maria Rilke (German poet, 1875-1926), José Maria Aznar (*1956, Spanish Prime Minister 1996-2004). Discuss the consequences on the ontology.
- Check what happens with names like “Kim” that can be both male and Female names.

REIFICATION

Reification: treat a class (or a property or a statement) as a thing:

- Male and Female are both classes and instances of class Sex

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix : <foo://bla/names#>.
:Person owl:disjointUnionOf (:Male :Female).
:Male a :Sex.
:Female a :Sex.
:MaleNames owl:equivalentClass [ a rdfs:Datatype; owl:oneOf ("John" "Bob") ] .
:FemaleNames owl:equivalentClass [ a rdfs:Datatype; owl:oneOf ("Mary" "Alice") ].
:Male a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames])].
:Female a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :FemaleNames])].
:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John".
:mary a :Person; :name "Mary".
```

[Filename: RDF/reification-class.n3]

```
prefix : <foo://bla/names#>
select ?P ?N ?S
from <file:reification-class.n3>
where {{?S a :Sex .
        ?P a :Person ; a ?S ; :name ?N}}
```

[Filename: RDF/reification-class.sparql]

DATATYPES

- common built-ins from XML Schema: int, decimal, ..., date, time, datetime.
- “2”^^xsd:decimal is different from “2”^^xsd:int

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo:bla#>.
:value a owl:DatatypeProperty; rdfs:range xsd:decimal.
:foo :value "2"^^xsd:decimal; :value "1.0"^^xsd:decimal.
:foo :value "2.0"^^xsd:decimal; :value "2.3"^^xsd:decimal.
:foo :value "2"^^xsd:integer; :value "1"^^xsd:integer.
```

[Filename: RDF/decimal.n3]

```
prefix : <foo:bla#>
select ?X ?Y
from <file:decimal.n3>
where {?X :value ?Y}
```

[Filename: RDF/decimal.sparql]

- jena: returns 6 results: “2”^^xsd:decimal, 1.0, 2.0, 2.3, 1, 2
- pellet: returns 5 results: 1, 2, 2.3, 2.0, 1.0

Defining own Datatypes

Two possibilities:

- use XML Schema xsd:simpleType definitions on the Web:
  - OWL reasoners parse+understand XML Schema simpleType declarations
  - adopt the DAML+OIL solution: datatype URI is constructed from the URI of the XML schema document and the local name of the simple type.
- OWL vocabulary to do the same as in XML Schema simpleTypes.

Datatypes in OWL

- use the XML Schema built-in types as resources (int and string must be supported; Pellet does also support decimal)
- rdfs:Datatype: cf. simple Types in XML schema; derived from the basic ones (e.g. xsd:int is an rdfs:Datatype)
- specified by
  - owl:onDatatype: from what datatype they are derived,
  - owl:withRestrictions is a list of restricting facets
  - facets as in XML Schema: xsd:{max/min}{In/Ex}clusive etc.
- similar to owl:Restrictions: define by *myDatatypeName* owl:equivalentClass [*datatypeSpec*].

Data Ranges: Adults

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/names#> .
:kate :name "Kate"; :age 62; :child :john.
:john :name "John"; :age 35; :child [:name "Alice"], [:name "Bob"; :age 8].
:child rdfs:domain :Person; rdfs:range :Person.
:age a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:int.
:name a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:string.
:atLeast18T owl:equivalentClass
  [a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions ( _:x1 )].
_:x1 xsd:minInclusive 18 .
:Adult owl:intersectionOf (:Person
  [ a owl:Restriction;
    owl:onProperty :age;
    owl:someValuesFrom :atLeast18T]).
:Child owl:intersectionOf (:Person
  [ owl:complementOf :Adult ]).
```

[Filename: RDF/adult.n3]

```
prefix : <foo://bla/names#>
select ?AN ?CN ?X ?Y
from <file:adult.n3>
where {{?A a :Adult; :name ?AN} UNION
       {?C a :Child; :name ?CN} UNION
       {?X :age ?Y}}
```

[Filename: RDF/adult.sparql]

An Example with Two QRRs

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/names#> .
:kate :name "Kate"; :age 62; :child :john, :sue.
:sue :name "Sue";  :age 32; :child [:name "Barbara"].
:john :name "John";  :age 35;
      :child :alice, [:name "Bob"; :age 8], [:name "Alice"; :age 10].
:frank :name "Frank"; :age 40; :child [:age 18], [:age 13].
:child rdfs:domain :Person; rdfs:range :Person.
:age a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:int.
:name a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:string.
:atLeast18T owl:equivalentClass [a rdfs:Datatype;
  owl:onDatatype xsd:int; owl:withRestrictions ( [ xsd:minInclusive 18 ] ) ].
:Adult owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :age; owl:someValuesFrom :atLeast18T]).
:HasTwoAdultChildren  owl:equivalentClass [ a owl:Restriction;
    owl:onProperty :child; owl:onClass :Adult; owl:minCardinality 2 ].
```

[Filename: RDF/adultchildren.n3]

```
prefix : <foo://bla/names#>
select ?AN ?N
from <file:adultchildren.n3>
where {{?A a :Adult; :name ?AN} UNION
  {?X a :HasTwoAdultChildren; :name ?N}}
```

[Filename: RDF/adultchildren.sparql]

DataRange Restriction for Geographical Coordinates

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:LongitudeT owl:equivalentClass [ a rdfs:Datatype; owl:onDatatype xsd:decimal;
   owl:withRestrictions ( [xsd:minExclusive -180] [xsd:maxInclusive 180] ) ] .
:LatitudeT owl:equivalentClass [ a rdfs:Datatype; owl:onDatatype xsd:decimal;
  owl:withRestrictions ( [ xsd:minInclusive -90] [xsd:maxInclusive 90] ) ] .
:EasternLongitudeT  owl:equivalentClass [a rdfs:Datatype;
  owl:onDatatype :LongitudeT; owl:withRestrictions ( [xsd:minInclusive 0] ) ] .
:EasternHemispherePlace owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:longitude; owl:someValuesFrom :EasternLongitudeT].
mon:longitude rdfs:range :LongitudeT.
mon:latitude rdfs:range :LatitudeT.
:Berlin a mon:City; :name "Berlin"; mon:longitude 13.3; mon:latitude 52.45 .
#:Atlantis a mon:City; :name "Atlantis"; mon:longitude -200; mon:latitude 100 .
:Lisbon a mon:City; :name "Lisbon"; mon:longitude -9.1; mon:latitude 38.7 .
```

[Filename: RDF/coordinates.n3]

```
prefix : <foo://bla/>
select ?N
from <file:coordinates.n3>
where {?X :name ?N .
  ?X a :EasternHemispherePlace}
```

[Filename: RDF/coordinates.sparql]

Example: Using XSD Datatypes

- [Does not work completely ...] Define simple dataypes in an XML Schema file:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="file:coordinates2.xsd">
 <xs:simpleType name="longitudeT">
  <xs:restriction base="xs:decimal">
   <xs:minExclusive value="-180"/>
   <xs:maxInclusive value="180"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="easternLongitude">
  <xs:restriction base="xs:decimal">
    <!-- note: base="longitudeT" would be nicer, but is not allowed when parsing from RDF -->
   <xs:minInclusive value="10"/>
   <xs:maxInclusive value="180"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="latitudeT">
  <xs:restriction base="xs:decimal">
   <xs:minInclusive value="-90"/>
   <xs:maxInclusive value="90"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

[Filename: RDF/coordinates2.xsd]

... and now use the datatypes ...

```
<!DOCTYPE rdf:RDF [ <!ENTITY mon "http://www.semwebtech.org/mondial/10/meta#">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY Coords "file:coordinates2.xsd"> ]>
<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#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#">

 <!-- ****** IMPORTANT: ALL DATATYPES MUST BE MENTIONED TO BE PARSED ****** -->
 <rdfs:Datatype rdf:about="&Coords;#longitudeT"/>
 <rdfs:Datatype rdf:about="&Coords;#easternLongitude"/>
 <rdfs:Datatype rdf:about="&Coords;#latitudeT"/>
 <owl:Class rdf:about="&mon;EasternHemispherePlace">
 <owl:equivalentClass> <!-- again: don't give a uri to an owl:Restriction! -->
  <owl:Restriction>
   <owl:onProperty rdf:resource="&mon;longitude"/>
   <owl:someValuesFrom rdf:resource="&Coords;#easternLongitude"/>
  </owl:Restriction>
 </owl:equivalentClass>
 </owl:Class>

 <mon:City mon:name="Berlin">
  <mon:longitude rdf:datatype="&Coords;#longitudeT">13.3</mon:longitude>
  <mon:latitude rdf:datatype="&Coords;#latitudeT">52.45</mon:latitude> </mon:City>
 <mon:City mon:name="Lisbon">
  <mon:longitude rdf:datatype="&Coords;#longitudeT">-9.1</mon:longitude>
  <mon:latitude rdf:datatype="&Coords;#latitudeT">38.7</mon:latitude> </mon:City>
</rdf:RDF>
```

[Filename: RDF/coordinates2.rdf]

... and now to the query:

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?N
from <file:coordinates2.rdf>
where {?X :name ?N . ?X a :EasternHemispherePlace}
```

[Filename: RDF/coordinates2.sparql]

Comments

- the RDF file must “define” all used rdf:Datatypes to be parsed from the XML Schema file. (if <rdfs:Datatype rdf:about=“&Coords;#easternLongitude”/> is omitted, the result is empty)
- if a prohibited value, e.g. longitude=200 is given in the RDF file, it is rejected.
- the rdf:Datatype for mon:longitude and mon:latitude must be given, otherwise it is not recognized as a number (but it does not matter if xsd:int or coords:longitude is used).
- specifying rdfs:range for longitude and latitude *without* rdf:Datatype for mon:longitude and mon:latitude is even inconsistent!

QUALIFIED ROLE RESTRICTIONS: EXAMPLE

Example: Country with at least two cities with more than a million inhabitants.

- define “more than a million” as a rdfs:Datatype
- search for all BigCities (= more than 1000000 inhabitants)
- check -via Provinces- which countries have two such cities.

Example: Cont’d

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.

mon:population  rdfs:range xsd:int; a owl:FunctionalProperty. ## all cities are different.
_:Million a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions (_:m1).
_:m1 xsd:minInclusive 1000000 .
:HasBigPopulation owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:population; owl:someValuesFrom _:Million].
:BigCity owl:intersectionOf (mon:City :HasBigPopulation).
:ProvinceWithBigCity owl:intersectionOf (mon:Province
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:someValuesFrom :BigCity]).
:ProvinceWithTwoBigCities owl:intersectionOf (mon:Province ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:onClass :BigCity; owl:minCardinality 2]).
[owl:intersectionOf (mon:Country ## with 2 big cities, no provinces ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:onClass :BigCity; owl:minCardinality 2]);
  rdfs:subClassOf :CountryWithTwoBigCities].
[owl:intersectionOf (mon:Country ## with 2 provs with big cities ## TR,GB,E,R,UA,D,I,NL
  [a owl:Restriction; owl:onProperty mon:hasProvince; owl:onClass :ProvinceWithBigCity; owl:minCardinality 2]);
  rdfs:subClassOf :CountryWithTwoBigCities].
[owl:intersectionOf (mon:Country ## with a prov with 2 big cities ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasProvince; owl:someValuesFrom :ProvinceWithTwoBigCities]);
   rdfs:subClassOf :CountryWithTwoBigCities].
```

[Filename: RDF/bigcities.n3]

Example: Cont'd

```
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.
:grmny a mon:Country; mon:hasCity :bln, :mch .
:bln a :BigCity; mon:population 3500000 .
:mch a :BigCity; mon:population 1500000 .
:frc a mon:Country; mon:hasProvince :ile, :prov .
:ile owl:differentFrom :prov.
:prs a mon:City; mon:cityIn :ile; mon:population 2000000 .
:mrs a mon:City; mon:cityIn :prov; mon:population 1500000 .
```

[Filename: RDF/dummy-cities.n3]

```
prefix : <foo://bla/>
select ?BC ?P1 ?P2 ?X
from <file:bigcities.n3>
#from <file:dummy-cities.n3>
from <file:mondial-europe.n3>
from <file:mondial-meta.n3>
where {# {?BC a :BigCity} UNION
       # {?P1 a :ProvinceWithBigCity} UNION
       # {?P2 a :ProvinceWithTwoBigCities} UNION
       {?X a :CountryWithTwoBigCities}}
```

[Filename: RDF/bigcities.sparql]

7.7 OWL 2: More about Properties

- $\mathcal{SHIQ}$/OWL-DL concentrate on *concept* definitions ($\mathcal{SQ}$ portion),
  - The $\mathcal{H}$ allows for a hierarchy of *properties* as already provided by RDFS, the $\mathcal{I}$ allows for inverse.
- $\mathcal{SHOIQ}/\mathcal{SHOIQ}(D)$ add nominals and datatypes
  (i.e., provide database-oriented functionality for handling *instances*),
- $\mathcal{SROIQ}$ provides more expressiveness around *properties*.

Transitive and Symmetric Properties

- transitive: descendants (cf. Slide 217), train connections etc.
- symmetric: married

```
@prefix : <foo://bla/names#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
  [ :name "John";  :married [ :name "Mary" ]] .
  :married rdf:type owl:SymmetricProperty.
```

[Filename: RDF/symmetric-married.n3]

```
prefix : <foo://bla/names#>
select ?X ?Y
from <file:symmetric-married.n3>
where { [ :name ?X ; :married [ :name ?Y]] }
```

[Filename: RDF/symmetric-married.sparql]

Symmetric Properties

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
:germany :borders :austria, :switzerland.
:borders a owl:SymmetricProperty.
```

[Filename: RDF/symmetricborders.n3]

```
prefix : <foo://bla/names#>
select ?X ?Y
from <file:symmetricborders.n3>
where {?X :borders ?Y}
```

[Filename: RDF/symmetricborders.sparql]

Reflexive Properties (OWL 2)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
:john a :Person; :knows :mary; :child :alice.
:knows a owl:ReflexiveProperty.
:germany a :Country.
```

[Filename: RDF/reflexive.n3]

```
prefix : <foo://bla/names#>
select ?X ?Y
from <file:reflexive.n3>
where {?X :knows ?Y}
```

[Filename: RDF/reflexive.sparql]

- only applied to individuals, but ... to all of them:
  John knows John, Alice knows Alice, and Germany knows Germany.

IRREFLEXIVE PROPERTIES

- irreflexive(rel): $\forall x : \neg rel(x, x)$.
- acts as constraint,
- but can also induce that two things must be different:
  $\forall x, y : rel(x, y) \rightarrow x \neq y$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
:john :hasAnimal :pluto, :garfield.
:pluto :bites :garfield.
# we exclude neurotic animals:
:bites a owl:IrreflexiveProperty.
:HasTwoAnimals  owl:equivalentClass
  [ a owl:Restriction;
    owl:onProperty :hasAnimal; owl:minCardinality 2 ].
```

[Filename: RDF/irreflexive.n3]

```
prefix : <foo://bla/names#>
select ?X ?Y ?Z
from <file:irreflexive.n3>
where {{?X :bites ?Y} UNION
       {?X :bites ?X} UNION
       {?Z a :HasTwoAnimals}}
```

[Filename: RDF/irreflexive.sparql]

- Pluto cannot be the same as Garfield.

ASYMMETRY

- asymmetric(rel): $\forall x, y : \neg rel(x, y) \vee \neg rel(y, x))$.
- acts as a constraint.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#>.
:rel a owl:AsymmetricProperty.
:a a :Node; :rel :b.
:b a :Node; :rel :c.
:c a :Node.
# :a owl:sameAs :b.
```

[Filename: RDF/asymmetry.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla/names#>
select ?X ?Y
from <file:asymmetry.n3>
where {?X a :Node; owl:differentFrom ?Y . ?Y a :Node}
```

[Filename: RDF/asymmetry.sparql]

- a,b,c, are not identified to be different, buy any owl:sameAs makes the ontology inconsistent.

IRREFLEXIVE AND ASYMMETRIC PROPERTIES

- Motivated by the "Ascending, Descending" graphics by M.C.Escher
  `http://en.wikipedia.org/wiki/Ascending_and_Descending`

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/names#>.

:Corner owl:oneOf (:a :b :c); rdfs:subClassOf
  [a owl:Restriction; owl:onProperty :higher; owl:cardinality 1].
:higher rdfs:domain :Corner; rdfs:range :Corner.
#:higher a owl:FunctionalProperty. ## redundant, note cardinality 1
#:higher a owl:InverseFunctionalProperty. ## also redundant
:higher a owl:AsymmetricProperty.
:higher a owl:IrreflexiveProperty.
:a :higher :b.
```

[Filename: RDF/escherstairs.n3]

```
prefix : <foo://bla/names#>
select ?X ?Y
from <file:escherstairs.n3>
where {?X :higher ?Y}
```

[Filename: RDF/escherstairs.sparql]

- Solution: $a > b$, $b > c$, $c > a$ is a valid model.

DISJOINT PROPERTIES

- Syntax: (prop$_1$ owl:propertyDisjointWith prop$_2$)
- for more than 2 properties (similar to owl:AllDifferent):
  [a owl:AllDisjointProperties; owl:members (...)]

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/names#>.

:alice :name "Alice"; :hasDog :pluto, :struppi.
:john :name "John"; :hasCat :garfield, :nermal; :hasDog :odie.
:sue :hasCat :grizabella.
#:sue :hasDog :grizabella.   ### test ##########
:pluto a :Dog; :name "Pluto".
:struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".
:nermal a :Cat; :name "Nermal".
:odie a :Dog; :name "Odie".
:grizabella :name "Grizabella".
:name a owl:FunctionalProperty.
:hasCat rdfs:subPropertyOf :hasAnimal.
:hasDog rdfs:subPropertyOf :hasAnimal.
:hasCat owl:propertyDisjointWith :hasDog.
```

[Filename: RDF/disjointproperties.n3]

```
prefix : <foo://bla/names#>
select ?A ?B ?C ?D ?E ?F
from <file:disjointproperties.n3>
where {{?X :name ?A; :hasCat/:name ?B} UNION
       {?X :name ?C; :hasDog/:name ?D} UNION
       {?X :name ?E; :hasAnimal/:name ?F}}
```

[Filename: RDF/disjointproperties.sparql]

At the Decidability Border

Some combinations of advanced constructs in DL that are part of OWL 2 are not even decidable:

- $\mathcal{ALC}_{reg}$ with transitivity, composition and union is ExpTime-complete
- the same when inverse roles and even cardinalities for *atomic* roles ($\mathcal{ALCQI}_{reg}$) are added (recall that inverse and transitive closure are important concepts in ontologies).
- The combination of *non-atomic* roles with cardinalities is in general undecidable.
- The same holds for Role-Value-Maps. Decidability is obtained only for Role-Value-Maps over *functional* roles.

Cardinalities on Atomic Roles

- a city can be the capital of at most one country (but also of one or more provinces)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/>.

:City a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :isCapitalOf;
    owl:onClass :Country; owl:maxCardinality 1 ].

:name a owl:FunctionalProperty.
mon:C-Oslo a :City;
  :isCapitalOf mon:Norway, mon:P-Akershus, mon:P-Oslo.
mon:P-Akershus a :Province; :name "Akershus".
mon:P-Oslo a :Province; :name "Oslo".
mon:Norway a :Country; :name "Norway".
# mon:C-Oslo :isCapitalOf :foo.  :foo a :Country; :name "Foo".
```

[Filename: RDF/one-capital.n3]

- use jena -e to export class/instance tree

ACROSS THE DECIDABILITY BORDER

- Cardinality restrictions on complex (e.g. transitive) properties are not allowed (undecidable) ⇒ rejected by the reasoner

Every city can be located in several provinces, but these must belong to the same country.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/>.

# Countries, Provinces, Cities:
:cityIn rdfs:subPropertyOf :belongsTo; rdfs:range :Province.
:isProvinceOf a owl:FunctionalProperty; rdfs:range :Country; rdfs:subPropertyOf :belongsTo.
:belongsTo a owl:TransitiveProperty; owl:inverseOf :hasProvOrCity.    # << trans.Prop <<<

:City a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :belongsTo;                       # << cardinality <<
    owl:onClass :Country; owl:maxCardinality 1].

:name a owl:FunctionalProperty.
mon:C-Oslo a :City; :cityIn mon:P-Akershus, mon:P-Oslo.
mon:Norway a :Country; :name "Norway".
mon:P-Akershus a :Province; :isProvinceOf mon:Norway; :name "Akershus".
mon:P-Oslo a :Province; :isProvinceOf mon:Norway; :name "Oslo".
# mon:C-Oslo :isCapitalOf :foo.  :foo a :Country; :name "Foo".
```

[Filename: RDF/one-country.n3]

Detection of Potentially Undecidable Situations

Pellet does not accept combinations that can potentially be undecidable

The ontology is rejected by Pellet:

- Unsupported axiom: Ignoring transitivity axiom due to an existing cardinality restriction for property `http://www.semwebtech.org/mondial/10/meta#belongsTo`
- It is also rejected if

  ```
  :cityIn a owl:FunctionalProperty.
  :isProvinceOf a owl:FunctionalProperty.
  ```

  is added (which guarantees decidability).

Further Features of OWL 2

- Role Chains/Property Chains: SubPropertyOf(PropertyChain(owns hasPart) owns)
  asserts that if x owns y and y has a part z, then x owns z.

  SubPropertyOf(PropertyChain(parent brother) uncle)
  asserts that the relationship "uncle" is a superset of "parent $\circ$ brother", i.e., the brothers of my parents are my uncles.
- Cross-property restrictions/role-value maps:
  (cf. draft at `http://www.w3.org/Submission/owl11-overview/`)
  - ObjectAllValuesFrom(likes knows =) describes the class of individuals who like all people they know (in DL syntax: the concept defined by the role value map (X.knows $\sqsubseteq$ X.likes)).
  - DataSomeValuesFrom(shoeSize IQ greaterThan)
    describes the class of individuals whose shoeSize is greater than their IQ (in DL syntax: the concept defined by the role value map (X.shoeSize$>$X.IQ)).

Role Chains

- (brotherOf $\circ$ child) $\sqsubseteq$ uncle

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .

[ owl:propertyChain (:brotherOf :child)]
  rdfs:subPropertyOf :uncleOf.
family:john a :Person; :brotherOf family:sue.
family:sue a :Person; :child family:anne, family:barbara.
:name a owl:FunctionalProperty.
family:anne :name "Anne".   family:barbara :name "Barbara".
```

[Filename: RDF/uncle.n3]

```
prefix : <foo://bla/names#>
select ?U ?X
from <file:uncle.n3>
where {?U :uncleOf ?X}
```

[Filename: RDF/uncle.sparql]

Exercise

- Extend the above example: the husbands of sisters of parents of x are also x's uncles.

Syntax: Role Chains in XML/RDF

... as expected: a blank node that refers to an rdf:List which is an owl:subPropertyOf another property.

```
<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#"
   xmlns="foo://bla/names#"
   xml:base="foo://bla/names">
 <rdf:Description>
   <rdfs:subPropertyOf rdf:resource="#uncleOf"/>
   <owl:propertyChain>
     <rdf:List>
       <rdf:rest rdf:parseType="Collection">
         <owl:ObjectProperty rdf:about="#child"/>
       </rdf:rest>
       <rdf:first rdf:resource="#brotherOf"/>
     </rdf:List>
   </owl:propertyChain>
 </rdf:Description>
 <Person rdf:ID="sue">
  <child rdf:resource="#anne"/>
  <child rdf:resource="#barbara"/>
  <brother rdf:resource="#john"/>
 </Person>
 <Person rdf:ID="john">
  <brotherOf rdf:resource="#sue"/>
 </Person>
</rdf:RDF>
```

[Filename: RDF/uncle.rdf]

```
prefix : <foo://bla/names#>
select ?U ?X
from <file:uncle.rdf>
where {?U :uncleOf ?X}
```

[Filename: RDF/uncle2.sparql]

Role Chains

- propertyChains with 3 or more elements are allowed:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .

[ owl:propertyChain (:brotherOf :child)]
  rdfs:subPropertyOf :uncleOf.
[ owl:propertyChain (:parent :brotherOf :child)]
  rdfs:subPropertyOf :cousinOf.
# [ owl:propertyChain (:father)] rdfs:subPropertyOf :parent.  ## complains
# [ :uncleOf rdfs:subPropertyOf owl:propertyChain (:brotherOf :child)]
#      is also not allowed (nullpointer error from inside pellet!)
family:john a :Person; :brotherOf family:sue.
family:bob :parent family:john.
family:sue a :Person; :child family:anne, family:barbara.
:name a owl:FunctionalProperty.
family:anne :name "Anne".   family:barbara :name "Barbara".
```

[Filename: RDF/propchain3-family.n3]

```
prefix : <foo://bla/names#>
select ?U ?X ?C
from <file:propchain3-family.n3>
where {{?U :uncleOf ?X}
       union
       {?C :cousinOf ?X}}
```

[Filename: RDF/propchain3-family.sparql]

Undecidable: Role Chains and Cardinalities

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/names#> .
@prefix family: <foo://bla/persons/> .

:uncleOf a owl:ObjectProperty.   ### required !!!!!!!!!!!
[ ] rdfs:subPropertyOf :uncleOf;
    owl:propertyChain (:brotherOf :child).
family:john a :Person; :brotherOf family:sue.
family:sue a :Person; :child family:anne, family:barbara.

:name a owl:FunctionalProperty.
:anne :name "Anne".   :barbara :name "Barbara".
:UncleOfMore a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :uncleOf; owl:minCardinality 2].
```

[Filename: RDF/uncleOfMore.n3]

```
prefix : <foo://bla/names#>
select ?U ?X
from <file:uncleOfMore.n3>
where {{?U :uncleOf ?X} UNION
       {?U a :uncleOfMore}}
```

[Filename: RDF/uncleOfMore.sparql]

- pellet: Definition of uncle is ignored; result empty.
  WARNING - Unsupported axiom: Ignoring transitivity and/or complex subproperty axioms for uncleOf

SELF RESTRICIONS: $\{x \mid x\ r\ x\}$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:Cyclic a owl:Class;
  owl:equivalentClass [ owl:intersectionOf
    (:Node [a owl:Restriction; owl:onProperty :to;
            owl:hasSelf "true"^^xsd:boolean ])].
:b a :Cyclic.
:a a :Node; :to :a, :b.
# :a a [ owl:complementOf :Cyclic ].
```

[Filename: RDF/cyclic.n3]

```
prefix : <foo://bla/>
select ?N ?N2
from <file:cyclic.n3>
where {{?N a :Cyclic} UNION
       {:a a :Cyclic} UNION
       {?N :to ?N2}}
```

[Filename: RDF/cyclic.sparql]

SELF RESTRICTIONS (CONT'D)

... just another example:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:NeuroticAnimal a owl:Class;
  owl:equivalentClass [ owl:intersectionOf
   ( :Animal
     [a owl:Restriction; owl:onProperty :bites; owl:hasSelf "true"^^xsd:boolean])].
:pluto a :Animal; :bites :pluto, :garfield.
:garfield a :NeuroticAnimal.
```

[Filename: RDF/neurotic.n3]

```
prefix : <foo://bla/>
select ?N ?N2
from <file:neurotic.n3>
where {{?N a :NeuroticAnimal} UNION
       {?N :bites ?N2}}
```

[Filename: RDF/neurotic.sparql]

7.8 DL and OWL Proving and Query Answering

- Tableau provers use refutation techniques:
  Given an ontology formalization Φ,
  prove $\Phi \models \varphi$ by starting a tableau over $\Phi \wedge \neg\varphi$ and trying to close it.

For that, it is well-suited for *testing* if something holds:

- consistency of a concept definition:
  $KB \models C \equiv \bot \;\Leftrightarrow\; KB \cup \{C(a)\}$ for a new constant a is unsatisfiable.
- concept containment:
  $KB \models C \sqsubseteq D \;\Leftrightarrow\; KB \models (C \sqcap \neg D) \equiv \bot$.
- concept equivalence:
  $KB \models C \equiv D \;\Leftrightarrow\; KB \models C \sqsubseteq D$ and $KB \models D \sqsubseteq C$.
- concept membership (for a given individual a):
  $KB \models C(a) \;\Leftrightarrow\; KB \cup \{\neg C(a)\}$ is unsatisfiable.

TABLEAU EXPANSION RULES FOR DL

- DL: use tableau without free variables. Expansion of universally quantified formulas takes only place for constants that are actually introduced.
- makes it more similar to Model Checking
- actually, not the tableau is generated completely, but branches are investigated by backtracking.

$(C \sqcap D)(s)$	Add $C(s)$ and $D(s)$ to the branch.
$(C \sqcup D)(s)$	Add two branches, one with $C(s)$, the other with $D(s)$.
$\exists R.C(s)$	Add $R(s,x)$ and $C(x)$ where x is new.
$\forall R.C(s)$	Add $C(t)$ whenever $R(s,t)$ is on the tableau (requires bookkeeping).
$\geq nR.C(s)$	Add $R(s,x_1),\ldots,R(s,x_n),C(x_1),\ldots,C(x_n)$ and $x_i \neq x_j$ where x_i are new.
$\leq nR.C(s)$	Bookkeeping about $\{x \mid R(s,x)\}$. Whenever more than n, then add branches with all combinations $x_i = x_j$. Continue bookkeeping.
$C \sqsubseteq D$	For each s recursively add two branches with $\neg C(s)$ and $D(s)$.
Closure	Close a branch whenever $A(s)$ and $\neg A(s)$ occur.

QUERY ANSWERING IN DL AND OWL

Query answering requires to find all answer bindings to variables.

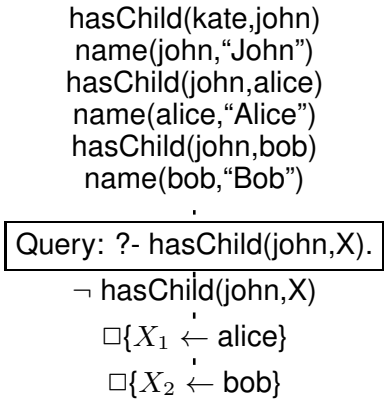
- find all X such that $KB \models C(X)$.
- find all D such that $KB \models D \sqsubseteq C$.

Start a tableau and collect substitutions that close branches:

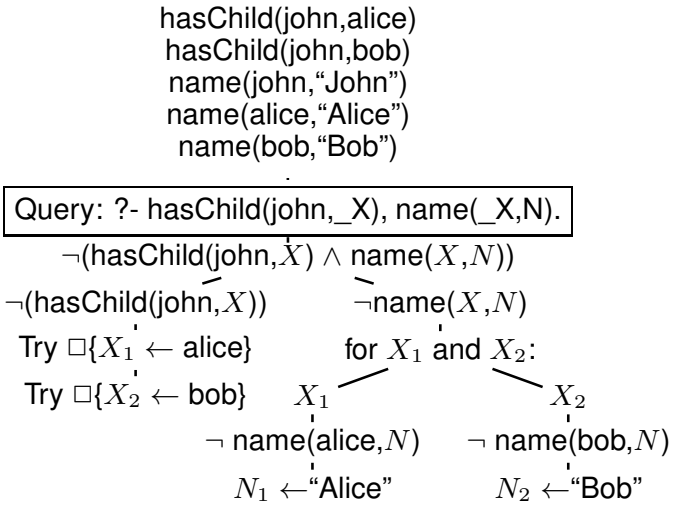
- start with $KB \cup \{\neg C(X)\}$.
- collect substitutions for X for which the tableau closes.
- without free variables: generate a new $\neg C(s)$ whenever any rule introduces a constant s. (= check if that s is an answer)
- harder to implement.
  Not always all answers are found by the current implementations.
- help the system by not only asking “{?X :age ?Y}”, but pruning the search space by “{?X a :Person; :age ?Y}”.

DL TABLEAUX: EXAMPLES

Who are John's children?



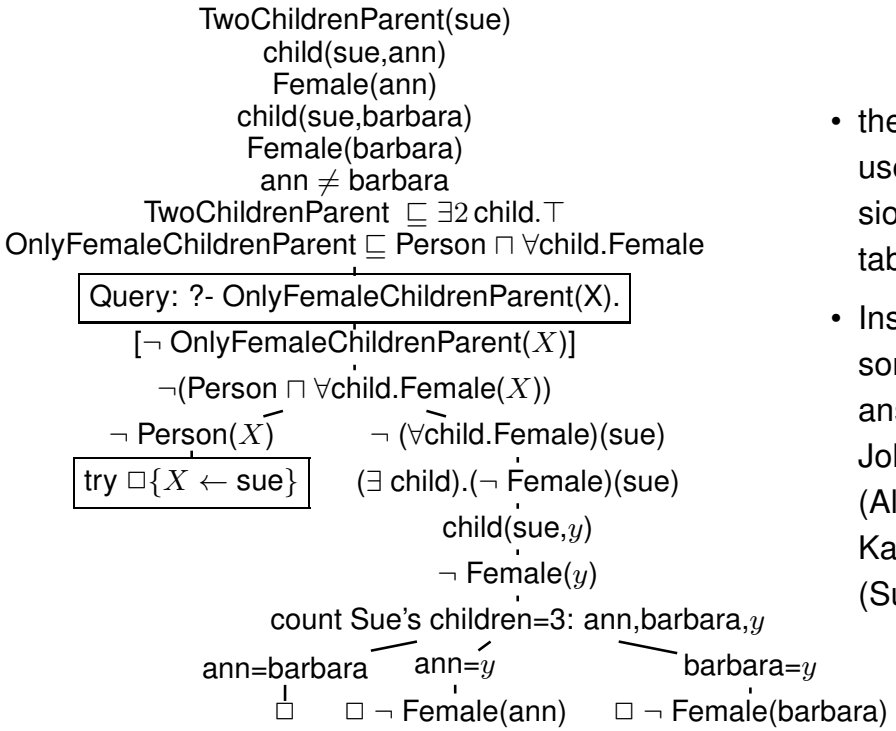
What are the names of John's children?



- Note: one could try close the right branch with X_0 ←john and N_0 ←"John", but for that, the left branch will not close.

DL TABLEAUX: EXAMPLES

Consider the "Only female children" example from Slide 365.



- the negated query can be used for leading the expansion, but not for closing the tableau.
- Instead of X, all other persons are also tried to derive answers:
  John: tableau does not close (Alice)
  Kate: tableau does not close (Sue)

DL TABLEAUX: A MORE INVOLVED EXAMPLE

Consider again the Escher Stairs example (Slide 404).

(1) Corner = AllDifferent(a,b,c)
(2) cardinality: Corner $\sqsubseteq \exists 1$ higher.$\top$
(3) domain: Corner $\sqsupseteq \exists$ higher.$\top$
(4) range: $\top \sqsubseteq \forall$ higher.Corner
(5) AntiSymmetric(higher)
(6) Irreflexive(higher)
(7) higher(a,b)

Query: ?- higher(X,Y).

[¬ higher(X,Y)]

First Answer Candidate:
with (7) X←a, Y← b
Try further answers ...

- The negated query can be used for leading the expansion, but not for closing the tableau. The first answer candidate is higher(a,b) – which was given in the input.
- Show that the developing model is consistent,
- and try to find additional answer candidates.

(2) can be applied for any constant occurring in the branch.

- Choose “b” since it is already used in another fact and search for further answers in this model.

Escher stairs tableau: continue with (2) for b

(1) Corner = AllDifferent(a,b,c)
(2) cardinality: Corner $\sqsubseteq \exists 1$ higher.$\top$
(4) range: $\top \sqsubseteq \forall$ higher.Corner
(5) AntiSymmetric(higher)
(6) Irreflexive(higher)
(7) higher(a,b)

[¬ higher(X,Y)]

apply (2) for b:

¬ Corner(b) □ | (8) $\exists 1$ higher.$\top$(b)

from (8): (9) higher(b,e)

next: range of “higher” derives that e is a corner:

(4) $\top \sqsubseteq \forall$ higher.Corner

¬$\top$(b) □ | (10) $\forall$ higher.Corner(b)

(11) Corner(e)

e must be one of a,b,c

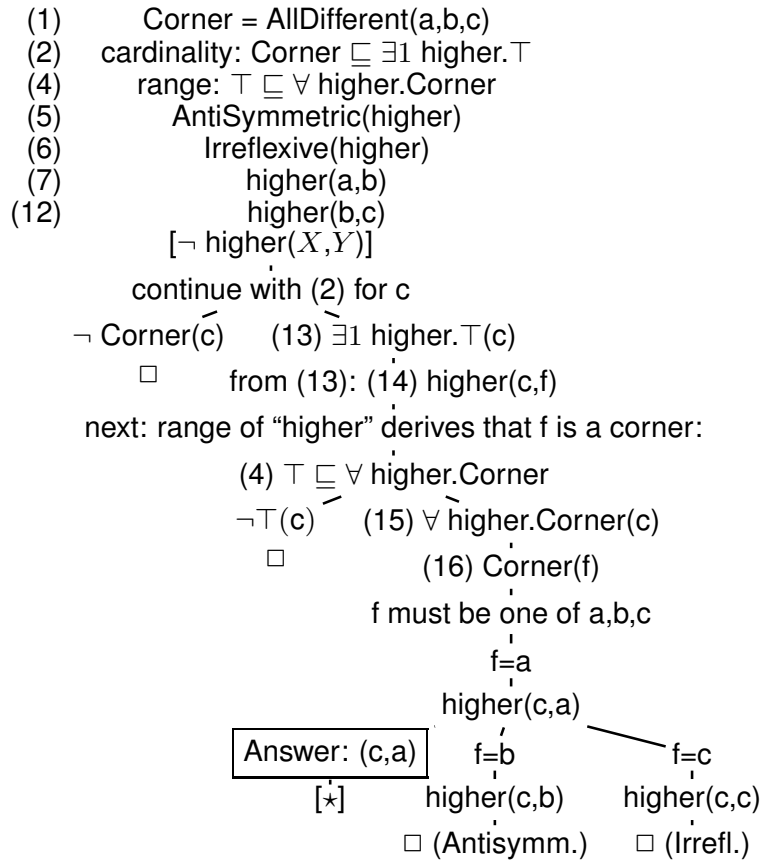
e=a: higher(b,a) □ (Antisymm.)

e=b: higher(b,b) □ (Irrefl.)

e=c: (12) higher(b,c) Answer candidate: (b,c)

- Expand the branch (=model) by investigating b.
- This yields another answer candidate.
- Still, it must be checked that the branch is not inconsistent.
- Further answers will be found.

Escher stairs tableau: continue with (2) for b



- The branch [⋆] cannot be closed.
- The set of formulas on this branch is consistent and describes a model.
- The answers to ?- higher(X,Y) in this model are (a,b), (b,c), and (c,a).

REQUIREMENTS ON (NOT ONLY DL) TABLEAU STRATEGIES

- select most promising formula to be expanded next
  - based on coincident constants,
  - "selectivity" of conditions,
  - α-rules non-branching before β-rules (branching).
- non-closing branches: know when to stop and return answer matches
  - "saturated" branches: expansion does not add new formulas,
  - do not expand irrelevant formulas at all.

DL TABLEAUX: SO FAR, SO GOOD ...

Consider the axiom

$$\text{Person} \sqsubseteq \exists \text{hasParent.Person}$$

The tableau generation does not terminate.

Blocking

- a constant s_2 is introduced as an existential filler from expanding a fact about constant s_1,
- the knowledge about s_1 and s_2 is *saturated* (i.e., nothing new about them can be derived),
- and the same facts are known about s_1 and s_2 except the above existential chain,
- then *block* s_2 from application of the existential formula (which would just create another same thing).
- Such blocking can be done for every existentially introduced thing, and it has only to be dropped if differences between it and its "predecessor" are derived.
- Such ontologies can be used. Queries only return instances in the "relevant" finite portion.

BLOCKING

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/names#>.
:kate a :Person; :name "Kate"; :child :john.
:john a :Person; :name "John"; :child :alice.
:alice a :Person; :name "Alice".
:child rdfs:domain :Parent;
       owl:inverseOf :parent.
:Person rdfs:subClassOf
 [a owl:Restriction;
    owl:onProperty :parent;
    owl:cardinality 2].
:Parent owl:equivalentClass
 [a owl:Restriction; owl:onProper
:Grandparent owl:equivalentClass
 [a owl:Restriction; owl:onProper
```

[Filename: RDF/infinite-parents.n3]

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla/names#>
select ?A ?B ?C ?X
from <file:infinite-parents.n3>
where {{?A a :Parent} UNION
       {?B a :Grandparent} UNION
       {:parent rdfs:range ?C} UNION
       {:kate :parent ?X}} # kate has no parent?!
```

[Filename: RDF/infinite-parents.sparql]

EXERCISE

Write RDF/OWL instances:

- John has two children in school, they are in the 3nd and 5th year. Children in the first year are 6 years old, those in the 2nd year are 7 years old, and so on. There are 12 years in school.
- Alice is a daugther of John. She is 8 years old.
- an “ideal family” consists of a father, a mother, and they have 2 children, a son and a daughter, and a dog.
- John’s family is an “ideal family”.
- Bob is John’s son.

Feed them into the Jena tool, activate the reasoner.

- How old is Bob?
- which of the above information can be omitted without losing information how old Bob is?

7.9 Rules in DL: Hybrid Reasoning

- Early Approaches: Donini, Lenzerini et al 1991; Levy, Rousset 1996 (CARIN):
  rather disappointing safety and decidability results:
  roughly, due to objects implicitly (existentially) assured by DL specifications.
- Newer investigations in Semantic Web context:
  DLV (Eiter et al 2004), DL+log (Rosati 2006); Motik, Sattler, Studer 2005; Lukasiewicz 2007: more detailed syntactical and structural constraints.
- SWRL (Semantic Web Rule Language; 2004):
  - Full Power of OWL-DL, allows for specifying undecidable settings, high computational complexity,
  - building upon the basic RULE-ML ontology for describing rules (rule; head, body; different kinds of atoms),
  - DL-safe rules (decidable) supported by Pellet: restriction in syntax and in semantics variables only applied to named resources (prunes the tableau; roughly ignoring all only existentially known objects).
- recall that SPARQL also returns only answers bound to explicitly known nodes (cf. Slide 337).

Simple Rule Example: Uncle

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/names#>.
:sue :child :barbara; :sibling :john.
:john :name "John"; :child :alice, :bob;  :sibling :sue.
:x a swrl:Variable.
:y a swrl:Variable.
:z a swrl:Variable.
 :uncleAuntRule a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :uncleAunt;
                swrl:argument1 :y ;   swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :child;
                swrl:argument1 :x ;   swrl:argument2 :y ]
              [ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :sibling;
                swrl:argument1 :x ;   swrl:argument2 :z ]).
```

[Filename: RDF/uncle-rule.n3]

```
prefix : <foo://bla/names#>
select ?X ?U
from <file:uncle-rule.n3>
where {?X :uncleAunt ?U}
```

[Filename: RDF/uncle-rule.sparql]

DL-Safe Rules consider only named Resources

- analogous to SPARQL queries and owl:hasKey (cf. Slide 329)

⇒ work only on a finite instantiated subgraph of the whole DL model

⇒ does not interfere with the blocking, and

⇒ does not break decidability.

Comparison: SWRL Rule, Property Chain, SPARQL, DL

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/names#>.
:john :child :bob; :sibling :paul, [].
:sibling a owl:SymmetricProperty.
:paul a [a owl:Restriction; owl:onProperty :child; owl:minCardinality 1].

:uncleRule a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :uncle1;
                swrl:argument1 :y ; swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :child;
                swrl:argument1 :x ; swrl:argument2 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :sibling;
                swrl:argument1 :x ; swrl:argument2 :z ]).
   :x a swrl:Variable.    :y a swrl:Variable.   :z a swrl:Variable.
[ owl:propertyChain ([owl:inverseOf :child] :sibling) ] rdfs:subPropertyOf :uncle2.
:Uncle owl:equivalentClass [a owl:Restriction; owl:onProperty :sibling;
  owl:someValuesFrom [a owl:Restriction; owl:onProperty :child;
                      owl:minCardinality 1]].
```

[Filename: RDF/uncle-comparison.n3]

DL-Safe Rules consider only named Resources (cont'd)

```
prefix : <foo://bla/names#>
select ?N ?U1 ?U2 ?isU
from <file:uncle-comparison.n3>
where {{?N :uncle1 ?U1} union {?N :uncle2 ?U2}
       union {?isU a :Uncle}}
```

[Filename: RDF/uncle-comparison.sparql]

- blank nodes are considered. Paul and a bnode (John's other brother) are Bob's uncles.
- implicitly known nodes are not considered: John's brother Paul has a child (so John is also an uncle) which is only implicitly known.

BUILT-IN SWRL ATOMS

SWRL provides some built-in atoms for owl:sameAs, owl:differentFrom etc.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla#> .

:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John"; :child :alice, :bob.
:alice a :Person; :name "Alice".
:bob a :Person; :name "Bob".
  :x a swrl:Variable.   :y a swrl:Variable.  :z a swrl:Variable.
:r a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :sibling;
                swrl:argument1 :y ; swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :child;
                swrl:argument1 :x ; swrl:argument2 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :child;
                swrl:argument1 :x ; swrl:argument2 :z ]
              [ a swrl:DifferentIndividualsAtom;
                swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/sibling-rule.n3]

Built-In SWRL atoms (Cont'd)

```
prefix : <foo://bla#>
select ?X ?CH ?SIB
from <file:sibling-rule.n3>
where {{?X :child ?CH} union {?X :sibling ?SIB}}
```

[Filename: RDF/sibling.sparql]

EVALUATION OF DL REASONING VS SWRL RULES

- DL Reasoning considers implicitly known resources (= graph nodes) and handles them in the tableau via blocking:
  - Structurally identical graph fragments are not further explored. Due to DL's locality principle and tree structure of the model, the model can be kept finite.
- Rules are also incorporated into the tableau, but since they do not have the tree property, blocking would not be sufficient for keeping the model finite when implicitly known resources are considered.

$\Rightarrow$ if something "important" about an implicitly known node can only be derived by a rule, it is not discovered (cf. next example).

DL-SAFETY: SWRL RULES DO NOT CONSIDER IMPLICIT RESOURCES

(see N3 fragment next slide)

- Rule: all persons believe in God,
- jack has a blank node child _:b0 who is a parent,
- _:b0 child is a believer (by the rule),
- as the grandchild is a person, application of the rule would result in the fact that it believes in God, i.e. it is a believer, which makes _:b0 a ParentOfBeliever.
- how to show that the grandchild is not considered by the rule: add a statement that _:b0 is not parent of any believer.
- run "classify" for the n3:
  - the ontology is consistent,
  - _:b0 is accepted to be a :NotParentOfBeliever.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/names#> .
:jack a :Person; :child [a :Person; a :Parent; a :NotParentOfBeliever].
:Parent owl:equivalentClass
  [a owl:Restriction; owl:onProperty :child; owl:someValuesFrom :Person].
:Believer owl:equivalentClass
  [a owl:Restriction; owl:onProperty :believes; owl:minCardinality 1].
:ParentOfBeliever owl:equivalentClass
  [a owl:Restriction; owl:onProperty :child; owl:someValuesFrom :Believer].
:NotParentOfBeliever owl:equivalentClass
  [a owl:Restriction; owl:onProperty :child; owl:onClass :Believer; owl:cardinality 0].

:x a swrl:Variable.
:r a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :believes;
                swrl:argument1 :x ; swrl:argument2 :god ]);
   swrl:body ([ a swrl:ClassAtom; swrl:classPredicate :Person;
                swrl:argument1 :x ]).
```

[Filename: RDF/hidden-prop.n3]


```
prefix : <foo://bla/names#>
select ?B ?PB ?NPB
from <file:hidden-prop.n3>
where {{?B a :Believer} union {?PB a :ParentOfBeliever}
                         union {?NPB a :NotParentOfBeliever}}
```

[Filename: RDF/hidden-prop.sparql]

RULE EXAMPLE: BIG CITIES

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

mon:population  rdfs:range xsd:int; a owl:FunctionalProperty. ## all cities are different.
_:Million a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions (_:m1).
_:m1 xsd:minInclusive 1000000 .

:ProvinceWithBigCity a owl:Class.   # otherwise sparql answer empty.
:ProvinceWithTwoBigCities a owl:Class.   # otherwise sparql answer empty.
:CountryWithTwoBigCities a owl:Class.   # otherwise sparql answer empty.

:HasBigPopulation owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:population; owl:someValuesFrom _:Million].
:BigCity owl:intersectionOf (mon:City :HasBigPopulation).
```

[Filename: RDF/bigcities-base.n3]

First: the simplest rule: a country where no provinces are contained in the database:

$\mathsf{Cw2BCs}(X) :- \mathsf{Country}(Y) \wedge \mathsf{BigCity}(Y) \wedge \mathsf{BigCity}(Z) \wedge \mathsf{hasCity}(X, Y) \wedge \mathsf{hasCity}(X, Z) \wedge Y \neq Z.$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:CountryNoProvsRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities;
              swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/bigcities-country-noprovs-rule.n3]

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:x a swrl:Variable.   :y a swrl:Variable.     :z a swrl:Variable.
:ProvBigCityRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Province; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]).

:TwoProvsRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/bigcities-2provs-rules.n3]


```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:Prov2BigCitiesRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithTwoBigCities; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Province; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).

:Prov2CRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities; swrl:argument1 :x]);
   swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
              [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithTwoBigCities; swrl:argument1 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
                swrl:argument1 :x ; swrl:argument2 :y ]).
```

[Filename: RDF/bigcities-prov-2bigcities-rules.n3]

Rule Example: Big Cities (Cont'd)

```
prefix : <foo://bla/>
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?C ?BC ?P1 ?P2 ?X
from <file:bigcities-base.n3>
from <file:bigcities-2provs-rules.n3>
from <file:bigcities-prov-2bigcities-rules.n3>
from <file:bigcities-country-noprovs-rule.n3>
#from <file:dummy-cities.n3>     ## a small test setting
from <file:mondial-europe.n3>    ## europe is more than sufficient =:(
from <file:mondial-meta.n3>
where {# {?BC a :BigCity} UNION
       {?X a mon:Country; mon:carCode ?C; mon:hasCity ?BC . ?BC a :BigCity} UNION
       {?P1 a :ProvinceWithBigCity} UNION
       {?P2 a :ProvinceWithTwoBigCities} UNION
       {?X a :CountryWithTwoBigCities}}
```

[Filename: RDF/bigcities-by-rules.sparql]

Chapter 8
Conclusion and Outlook

What should have been learnt:

- Formal Logic: interpretations, model theory, first-order logic
- Deductive systems: Datalog, minimal model semantics
- reasoning: tableau calculi
- RDF as a special, simple data model; URIs
  representations: N3 and RDF/XML
- DL as another logic, Open World
- "database" vs. "knowledge base"
- OWL as "DL alive"

SEMANTIC WEB DATA: XML; RDF AND OWL

In contrast to XPath/XQuery, XSLT, XML Schema, XLink etc., RDF and OWL are *not* languages *"inside"* the XML world, but are concepts of their own that have - incidentally- also an XML syntax.

The combination of XML data and RDF/RDFS/OWL concepts is the base for the *Semantic Web*.

A Semantic Web application e.g. exists of

- a "central" portal that uses the following things:
- a set of ontological (OWL, RDFS) sources,
- a set of RDF sources,
- reasoning (using OWL and RDFS information),
- a semantical description of itself for allowing others to use it.

SEMANTIC WEB SERVICES

- Ontologies for describing Web Services
  (lifting the WSDL, UDDI stuff to a semantic level)
- different current proposals
  OWL-S, WSMO (Web Services Modeling Ontology)
- semantic matchmaking between tasks and services

Other Issues

- trust, recommender systems, personalization
  "Web 2.0": semantic wikis, semantic blogs
- dynamics:
  DBIS: 2004-2008 REWERSE Eu NoE Working Group I5,
  continued with the MARS (Modular Active Rules in the Semantic Web) and SWAN (Semantic Web Application Node) projects
- policies
- verification

Application Areas

- Bioinformatics