3. Unit: OWL

Exercise 3.1 (Win-Move Game: Draw Nodes) Consider again the Win-Move-Game. There, WinNodes and LoseNodes have been axiomatized.

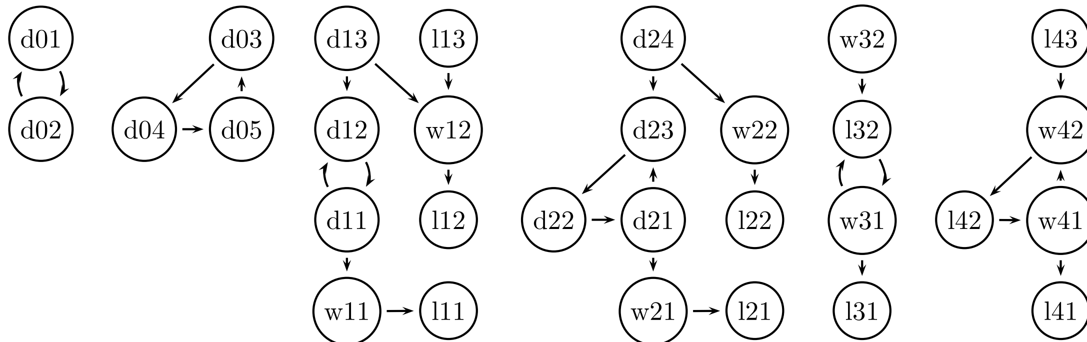
a) Is it possible to characterize DrawNodes in OWL?
   Consider two alternative variants:
   a) use the game axioms/rules to axiomatize DrawNodes explicitly.
   b) consider the possible values: win/lost/drawn.

   Test both with *typical* minimal examples and explain the results.

   Comparison with the Database Theory lecture: Interpret the results and compare them with the semantics of the well-founded model and of stable models.

b) Is it possible to obtain the drawn nodes by using SPARQL?
   If yes, give a query that does this.

Consider the following example graphs $G_0, \ldots G_4$ (G_0 contains two minimal examples with cycles, G_1, G_2 two-/three-cycles with an exit of length 2, G_3, G_4 two-/three-cycles with an exit of length 1:



```
@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
    (:d01 :d02 :d03 :d04 :d05
     :l11 :l12 :l13 :w11 :w12 :d11 :d12 :d13
     :l21 :l22 :w21 :w22 :d21 :d22 :d23 :d24
     :l31 :l32 :w31 :w32
     :l41 :l42 :w41 :w42 :l43)].
:edge a owl:ObjectProperty; rdfs:domain :Node; rdfs:range :Node.
:out a owl:DatatypeProperty.
# 2-ary cycle
:d01 a :Node; :out 1; :edge :d02 .
:d02 a :Node; :out 1; :edge :d01 .
# 3-ary cycle
:d03 a :Node; :out 1; :edge :d04 .
:d04 a :Node; :out 1; :edge :d05 .
:d05 a :Node; :out 1; :edge :d03 .
```

```
:l11 a :Node; :out 0 .
:w11 a :Node; :out 1; :edge :l11 .
:d11 a :Node; :out 2; :edge :w11, :d12 .
:d12 a :Node; :out 1; :edge :d11 .
:d13 a :Node; :out 2; :edge :d12, :w12 .
:w12 a :Node; :out 1; :edge :l12 .
:l12 a :Node; :out 0 .
:l13 a :Node; :out 1; :edge :w12 .
:l21 a :Node; :out 0 .
#
:w21 a :Node; :out 1; :edge :l21 .
:d21 a :Node; :out 1; :edge :d23 . #########
:d22 a :Node; :out 1; :edge :d21 .
:d23 a :Node; :out 1; :edge :d22 .
:d24 a :Node; :out 2; :edge :d23, :w22 .
:w22 a :Node; :out 1; :edge :l22 .
:l22 a :Node; :out 0 .
#
:l31 a :Node; :out 0 .
:w31 a :Node; :out 2; :edge :l31, :l32 .
:l32 a :Node; :out 1; :edge :w31 .
:w32 a :Node; :out 1; :edge :l32 .
#
:l41 a :Node; :out 0 .
:w41 a :Node; :out 2; :edge :l41, :w42 .
:l42 a :Node; :out 1; :edge :w41 .
:w42 a :Node; :out 1; :edge :l42 .
:l43 a :Node; :out 1; :edge :w42 .
```

[Filename: winmove-graphs-draw.n3]

Consider the following OWL axiomatizations of drawn nodes:

- implicitly: the nodes are partitioned into Win/Lose/Draw nodes, and
- by the rules: any node from which only win nodes (i.e. where the other player will win) and also some draw nodes can be reached, are drawn.

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

[Filename: File: winmove-draw.n3]

- Simple query file:

```
prefix : <foo://bla/>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?W ?L ?D
#from <file:winmove-graph.n3>
from <file:winmove-graph-draws.n3>
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
from <file:winmove-draw.n3>
where {{?W a :WinNode} UNION
       {?L a :LoseNode} UNION
       {?D a :DrawNode}}
order by ?D ?L ?W
```

[Filename: winmove-draw.sparql]

Drawn nodes:

Only the three-node-cycles (d03,d04,d05) and (d21,d22,d23, together with d24) in subgraphs 0 and 2 are detected. The two-node-cycles are missing. Note also that if one of the characterizations in `winmove-draw.n3` is commented out, no drawn nodes are found at all!

This corresponds to the existence of (total) stable models:

- consider the two-node cycle (d01,d02): there are two total stable models where (i) d01 is won and d02 is lost, and (ii) d02 is lost and d01 is won.
  Correspondingly, in the tableau calculus underlying OWL, there are the above two models. Thus, d01 cannot be refuted to be non-won or non-lost! The tableau branch cannot be closed, thus no answer is possible.
- consider the three-node cycle (d03,d04,d05): there is no total stable model. Thus the tableau can prove that d03 is not won, and not lost, thus it can only be drawn.

In contrast, the negation-by-default in SPARQL's `not exists` easily classifies all nodes that cannot be proven to be won or lost as drawn.

```
prefix : <foo://bla/>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?W ?L ?D
#from <file:winmove-graph.n3>
from <file:winmove-graph-draws.n3>
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
where {{?W a :WinNode} UNION
       {?L a :LoseNode} UNION
       {?D a :Node . not exists {?D a :WinNode} . not exists {?D a :LoseNode}}}
order by ?D ?L ?W
```

[Filename: winmove-draw-cwa-neg.sparql]

- negation-by-default covers the "unknown" semantics of the well-founded semantics and of stable models: if it cannot be proven that a node *is* won or *is* lost (in *all* models), it is drawn.

  Note that also in the well-founded/stable models, it is not *derived in the model* that nodes are drawn, but only the external interpretation of the models/results yields the conclusion that the nodes are drawn.

Exercise 3.2 (Gods) Give an OWL specification of the following situation:

The Jewish belief is a monotheistic belief – the only god is Jehova. The old nothern European belief was polytheistic, the set of gods consists of Odin, Thor, and Freya. Moshe is a Jew, Haegar is a northern European.

State a SPARQL query that tells you who believes in whom.

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

:JewishGod rdfs:subClassOf :God;
   owl:equivalentClass [ owl:oneOf (:jehova)].
:GermanGod rdfs:subClassOf :God;
   owl:equivalentClass [ owl:oneOf (:odin :thor :freya)].
:Jew owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :believesIn; owl:someValuesFrom :JewishGod].
:German owl:equivalentClass [ a owl:Restriction;
    owl:onProperty :believesIn; owl:someValuesFrom :GermanGod].
:moshe a :Jew.
:haegar a :German.
```

[Filename: musterlsgn/gods.n3]

```
prefix :<http://example.org#>
SELECT ?P ?G
FROM <file:gods.n3>
WHERE { ?P :believesIn ?G }
```

[Filename: musterlsgn/gods.sparql]

The query tells only that Moshe believes in Jehova: as it is specified that he must believe in some Jewish God, this must be Jehova. For Haegar someValuesFrom is not sufficient, since it only states that some/all of the things he believes in are german gods (note that "allValuesFrom" would even allow that he does not believe in anything).

One possibility is to assert the cardinality "3" to make clear that germans must believe in the three german gods (but such a statement must be made then for each religion, and the number must be adapted if there are more/less fillers of such a role):

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

:German rdfs:subClassOf [ a owl:Restriction;
    owl:onProperty :believesIn; owl:allValuesFrom :GermanGod];
  rdfs:subClassOf [ a owl:Restriction;
    owl:onProperty :believesIn; owl:cardinality 3].
```

[Filename: germangods.n3]

```
prefix :<http://example.org#>
SELECT ?P ?G
FROM <file:gods.n3>
FROM <file:germangods.n3>
WHERE { ?P :believesIn ?G }
```

[Filename: germangods.sparql]

It is not possible to state in general that Germans believe in all German gods. For Haegar, one can alternatively do the following:

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

:believedBy owl:inverseOf :believesIn.
:GermanGod owl:equivalentClass
  [a owl:Restriction; owl:onProperty :believedBy;
    owl:hasValue :haegar].
```

[Filename: haegargods.n3]

```
prefix :<http://example.org#>
SELECT ?P ?G
FROM <file:gods.n3>
FROM <file:haegargods.n3>
WHERE { ?P :believesIn ?G }
```

[Filename: haegargods.sparql]

Alternatively, the other way round, one can fix each god explicitly (gods change less frequently than people):

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

:German rdfs:subClassOf
  [ a owl:Restriction; owl:onProperty :believesIn; owl:hasValue :odin],
  [ a owl:Restriction; owl:onProperty :believesIn; owl:hasValue :thor],
  [ a owl:Restriction; owl:onProperty :believesIn; owl:hasValue :freya].
```

[Filename: germangods2.n3]

```
prefix :<http://example.org#>
SELECT ?P ?G
FROM <file:gods.n3>
FROM <file:germangods2.n3>
WHERE { ?P :believesIn ?G }
```

[Filename: germangods2.sparql]

Exercise 3.3 (Male and Female Names)

Consider again the *Male and Female Names* Example from the lecture:

The name commonly female name “Maria” is (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), cf. also Jean-Marie Le Pen (*1928, French Politician).

Discuss the consequences on the ontology.

- adding “Maria” as both male and female names would classify *Rainer Maria* as both male and female, which has to be disjoint.

⇒ The ontology is inconsistent:

```
@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 ("Rainer"^^xsd:string "Maria"^^xsd:string) ] .
:FemaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
    owl:oneOf ("Anna"^^xsd:string "Maria"^^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])].
:rainermaria a :Person; :name "Rainer"^^xsd:string, "Maria"^^xsd:string.
:maria a :Person; :name "Maria"^^xsd:string.
:annamaria a :Person; :name "Anna"^^xsd:string, "Maria"^^xsd:string.
```

[File: maria-male-and-female.n3]

- ignore “Maria” if another male/female name is given. Then, if (as by German law) it is required that each person also carries another name hat unambiguously identifies its sex, every person is classified correctly. But, a person only called “Maria” cannot be classified.

  Add “Maria” as “potentially female name”, and use the heuristics that if a person has no male name, but a potentially female name, is female:

⇒ *Rainer Maria* is classified as male, *Anna Maria* is classified as female. But *Maria* is still not classified at all!

  The reason is the Open World Assumption: it cannot be proven that *Maria* is in the complement of *Male*, since she **may have** a male name that is not stored.

  Thus, it is necessary to close the name property (PersonWithOneName, PersonWithTwoNames, etc.).

- **Recall that DL can only do *monotonic* reasoning:** Adding another, male, name for *Maria* would require to withdraw the default assumption that this person is female.

```
@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 ("Rainer"^^xsd:string) ] .
:FemaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
    owl:oneOf ("Anna"^^xsd:string) ].
:PotFemaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
    owl:oneOf ("Maria"^^xsd:string) ].
[owl:intersectionOf ( :Person
    [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames])]
  rdfs:subClassOf :Male.
[owl:intersectionOf ( :Person
    [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :FemaleNames])]
  rdfs:subClassOf :Female.
[owl:intersectionOf ( :Person
    [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :PotFemaleNames])]
  rdfs:subClassOf :PotFemale.
[owl:intersectionOf ( :Person
    [owl:complementOf
      [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames]]
    [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :PotFemaleNames])]
  rdfs:subClassOf :Female.
:rainermaria a :Person; :name "Rainer"^^xsd:string, "Maria"^^xsd:string.
:maria a :Person; :name "Maria"^^xsd:string.
:annamaria a :Person; :name "Anna"^^xsd:string, "Maria"^^xsd:string.
```

[File: maria-potentially-female.n3]

- With a SPARQL query, using its negation-by-default via not bound/not exists, closed world can be simulated:

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla/names#>
select ?F ?M
from <file:maria-potentially-female.n3>
where
 {{ ?M a :Male }
  union
  { ?F a :Female }
  union
  { ?F a :PotFemale . not exists { ?F a :Male}}}
```

[File: maria-potentially-female.sparql]

- Note: with SPARQL's `CONSTRUCT` clause, the result can be turned into a graph (and it can be written to a file):

```
## call
##   jena -q -pellet -qf maria-potentially-female2.sparql -of bla
## for output to file.
```

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla/names#>
construct {?F a :Female . ?M a :Male}
from <file:maria-potentially-female.n3>
where
 {{ ?M a :Male }
  union
  { ?F a :Female }
  union
  { ?F a :PotFemale . not exists { ?F a :Male}}}
```

[File: maria-potentially-female2.sparql]

- generates
  ```
  @prefix : <foo://bla/names#> .
  :rainermaria a :Male .
  :annamaria a :Female .
  :maria a :Female .
  ```

⇒ By scripting SPARQL chains, complex tasks can be carried out.

Further comments:

- Further language-based reasoning: in Italian, "Andrea" and "Nicola" are male names, in German, both are female names.

Exercise 3.4 Consider again the "Escher Stairs" example from the lecture:

```
@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; a owl:IrreflexiveProperty.
:a :higher :b.
## query: { ?X :higher ?Y }
```

[Filename: escherstairs.n3]

- what happens when the above program is the extended to four corners (:a :b :c :d)?
  - What is the result of the SPARQL query?
  - What are the possible models? – Analyze the result also from the logical point of view.
- Characterize the models when the program is extended to five and six nodes.

- The answer to the SPARQL query is only $\{X/a, Y/b\}$.
- There are two models: $a > b > c > d > a$ and $a > b > d > c > a$

- The logical semantics of the SPARQL query is model-theoretic. Let φ denote the DL formula corresponding to the above specification:

  $$\begin{array}{l}\forall x : (\text{Corner}(x) \iff x = a \vee x = b \vee x = c) \wedge a \neq b \wedge a \neq c \wedge b \neq c \wedge \\ \text{Corner} \sqsupseteq 1\text{higher}.\top \wedge \\ \text{Corner} \sqsupseteq \exists\text{higher}.\top \wedge \top \subseteq \forall\text{higher}.\text{Corner} \wedge \forall x, y : (\text{higher}(x, y) \rightarrow \neg\text{higher}(y, x)) \wedge \forall x : \neg\text{higher}(x, x)\end{array}$$

  Then, a pair (i, j) is an answer of the SPARQL query pattern { `?X :higher ?Y`} if and only if

  $$\varphi \models \text{higher}(i, j) ,$$

  i.e., if higher(i, j) holds in *every* model of φ.
  In the lecture's example, there was only one such model – thus, the answer gave a complete account of the model. Now, there are two models, and only the (guaranteed) intersection of their individual "answers" is listed.

  Aside: Note that the Answer Set Programming Approach/Stable Models (Database Theory Lecture) would list both models as possible answers.

- For five nodes a, b, c, d, e, all five-cycles containing $a > b$ are models, yielding 6 results ($abcdea$, $abceda$, $abdcea$, $abdeca$
  For six nodes, not only the 6-cycles (there are $4 \cdot 3 \cdot 2 = 24$ of them) are models, but there are also eight 3-cycle models
  $abca+defd$, $abda+cefc$, $abea+cdfc$, $abfa+cdec$ and $abca+dfed$, $abda+cfec$, $abea+cfdc$, $abfa+cedc$.

Exercise 3.5 (Role Chains: Uncles)

Characterize the uncle relationship as a role chain:

- x's uncles are the brothers of x's parents, and
- x's uncles are husbands of the sisters of x's parents.

```
@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 ( :married :sisterOf :child )]
  rdfs:subPropertyOf :uncleOf.
:married a owl:SymmetricProperty.

family:john a :Person; :brotherOf family:sue.
family:maggie a :Person; :sisterOf family:sue; :married family:george.
family:sue a :Person; :child family:anne, family:barbara.
:name a owl:FunctionalProperty.
family:anne :name "Anne".   family:barbara :name "Barbara".
```

[File: uncles.n3]

Note that the above example also uses the symmetry of married.

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

[File: uncles.sparql]

Exercise 3.6 (Gods) Solve Exercise 3.2 using OWL2 features.

The classes "Jew" and "German" are reified assign the gods by a property to the classes. Then, a property chain is defined as a "rule" that a believer of a religion believes in the gods of this religion. Note that it is not possible to use rdf:type in the chain, but another property, here hasBelief must be defined:

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

:Jew owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :believesIn; owl:someValuesFrom :JewishGod];
  :haveGods :jehova.
:German owl:equivalentClass [ a owl:Restriction;
    owl:onProperty :believesIn; owl:someValuesFrom :GermanGod];
  :haveGods :odin, :thor, :freya.
:moshe a :Jew; :hasBelief :Jew.
:haegar a :German; :hasBelief :German.

:believesIn a owl:ObjectProperty.
[ owl:propertyChain (:hasBelief :haveGods);
  rdfs:subPropertyOf :believesIn].
```

[Filename: gods-reified-owl2.n3]

```
prefix :<http://example.org#>
SELECT ?P ?G
FROM <file:gods-reified-owl2.n3>
WHERE { ?P :believesIn ?G }
```

[Filename: gods-reified-owl2.sparql]