Solutions

Solutions for Chapter 3: Resource Description Framework

Solution for Exercise 3.1

1. `"2014-12-18T18:47:00-04:00"^^xsd:dateTimeStamp`
   - `xsd:dateTime` is also reasonable but `xsd:dateTimeStamp` requires the time-zone and thus has the advantage of a totally-ordered value space
2. `"18:47:00-04:00"^^xsd:date`
3. `"1984"^^xsd:gYear`
4. `"22"^^xsd:nonNegativeInteger`
   - `xsd:unsignedByte` is also reasonable if age ≤ 255
   - `xsd:integer` would permit negative ages
   - `xsd:decimal` would permit continuous ages (e.g., 22.4)
   - `xsd:duration` could also arguably be used
   - Where possible, it would be better to model the static values of birth-date and death-date to avoid updating ages.
5. `"PT1H22M"^^xsd:duration`
   - `"PT1H22M"^^xsd:dayTimeDuration` also fine
   - `"PT82M"^^xsd:duration`, etc., also fine
   - Units with 0 value can be omitted
6. `"--18-12"^^xsd:gMonthDay`
7. `"204"`
   - Equivalent to `"204"^^xsd:string`
   - Arguably better than a numeric type since we may have apartment numbers like `"204b"` (if not now, at a later stage)

Solution for Exercise 3.2

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

[] a ex:ApartmentBlock ;
  ex:address "145, 5th Avenue, New York, US"@en ;
  ex:floors 24 ;
  ex:apartments 120 ;
  ex:constructed "1976"^^xsd:gYear ;
  ex:hasFloor [
     a ex:Floor ;
     ex:number "2" ;
     ex:apartments 5 ;
     ex:hasApartment _:a201 , _:a202 , _:a203 , _:a204 , _:a205
  ] .

_:a201 a ex:Apartment ;
  ex:number "201" ;
  ex:bedrooms 1 ;
  ex:nextDoor _:a202 .

_:jsa201 ex:occupant ex:JohnSmith ;
  ex:startDate "2013-10-12"^^xsd:date ;
  ex:apartment _:a201 .
_:msa201 ex:occupant ex:MarySmith ;
  ex:startDate "2013-10-12"^^xsd:date ;
  ex:apartment _:a201 .
ex:JohnSmith ex:name "John Smith"@en ;
  ex:wife ex:MarySmith .
ex:MarySmith ex:name "Mary Smith"@en .

_:sma201 ex:occupant ex:SimonMurphy ;
  ex:startDate "2011-12-01"^^xsd:date ;
  ex:endDate "2013-10-12"^^xsd:date ;
  ex:apartment _:a201 .
ex:SimonMurphy ex:name "Simon Murphy"@en .

_:a202 a ex:Apartment ;
  ex:number "202" ;
  ex:bedrooms 2 .

_:joba202 ex:occupant ex:JimOBrien ;
  ex:startDate "2001-05-12"^^xsd:date ;
  ex:apartment _:a202 .
_:hoba202 ex:occupant ex:HarryOBrien ;
  ex:startDate "2009-06-05"^^xsd:date ;
  ex:apartment _:a202 .
ex:JimOBrien ex:name "Jim O'Brien"@en ;
  ex:son ex:HarryOBrien .
ex:HarryOBrien ex:name "Harry O'Brien"@en .
```

- The prefix is not important.
- There are many possible variations in both syntax and structure; importantly, the Turtle syntax used should be valid.
- It is only necessary to provide information given in the question (e.g., no need to say the street, city, etc. of the address since the question does not specify that New York is the city, US is the country, etc.; also no need to say that John Smith is a person since the question doesn't state that). One can optionally add additional information/structure, however.
- IRIs are not a replacement for strings. Information encoded in IRIs is not enough (like `ex:JohnSmith`). If a name or title is given, assign a property with a literal value indicating the verbatim name.
- Datatypes are important, as are language tags. Note that year is not an integer. Regarding apartment numbers, we could also arguably use integers (but strings are probably better since we might later have cases like apartment `"201b"`). Language tags on the names of people are arguably not important in this case, though names may sometimes vary (e.g., in spelling) across languages or alphabets; hence it may be better to include such tags.
- One could replace the blank nodes above with *sensible* IRIs. Note, however, that `ex:Floor2` may not be sufficient (unless the `ex:` namespace is specific to the building) since (unlike blank nodes) IRIs are *global* identifiers and many buildings will have a floor 2. A sensible IRI would be something unique like `ex:F2_146_5thAv_NY`.
- The list of apartments could be a given as an RDF collection (aka. RDF list). But *only* use lists in the case that the list is ordered, *or* the list of values is certainly complete. In this case the latter applies, hence it is justified to use a list.
- Declaring properties such as `ex:name`, `ex:startDate`, etc., to be members of the class `rdf:Property` is optional.
- Many classes and properties could be replaced by terms in well-known vocabularies, such as `foaf:name` or `foaf:Person`; however, in the context of Exercise 3.2, using such terms is not yet important.

Solutions for Chapter 4: RDF Schema and Semantics

Solution for Exercise 4.1

The most general definitions (under an intuitive understanding of the involved terms) would be:

```
ex:wroteNovel       rdfs:domain          ex:Novelist
ex:wroteNovel       rdfs:range           ex:Novel
ex:Novelist         rdfs:subClassOf      ex:Author
ex:Author           rdfs:subClassOf      ex:Person
ex:Novel            rdfs:subClassOf      ex:LiteraryWork
ex:wroteNovel       rdfs:subPropertyOf   ex:authored
ex:authored         rdfs:subPropertyOf   ex:contributedTo
```

However, one might have considered definitions of the form:

```
ex:wroteNovel       rdfs:domain          ex:Novelist
ex:wroteNovel       rdfs:domain          ex:Author
ex:wroteNovel       rdfs:domain          ex:Person
ex:wroteNovel       rdfs:range           ex:Novel
ex:wroteNovel       rdfs:range           ex:LiteraryWork
ex:wroteNovel       rdfs:subPropertyOf   ex:authored
ex:authored         rdfs:subPropertyOf   ex:contributedTo
```

Such definitions may be sufficient to draw the conclusions needed for the example, but are (unnecessarily) weaker. For example, given the triple:

```
ex:GeorgeOrwell     rdf:type             ex:Novelist
```

The first set of definitions would permit us to infer:

```
ex:GeorgeOrwell     rdf:type             ex:Author
ex:GeorgeOrwell     rdf:type             ex:Person
```

... whilst the second set would not.

Solution for Exercise 4.2

To definitively answer this question, it would be better to see the formal semantics of RDFS as appears later in the chapter. But even with the informal definitions provided up to this point, the following should be observed:

```
rdf:Property        rdf:type             rdfs:Class
```

... is RDF(S) valid: `rdf:Property` *is* a class (the class of all properties)!

```
rdf:Property        rdfs:subClassOf      rdfs:Class
```

... is not RDF(S) valid: this would imply that all properties (members of `rdf:Property`) are also classes (members of `rdfs:Class`), which is not true!

```
rdf:Property        rdf:type             rdfs:Resource
```

... is RDF(S) valid: everything is a resource.

```
rdf:Property        rdfs:subClassOf      rdfs:Resource
```

... is RDF(S) valid: everything is a resource (so every class is a sub-class of `rdfs:Resource`, the class of everything)!

```
rdfs:subClassOf      rdfs:subPropertyOf    rdfs:subClassOf
```

... is RDF(S) valid: every property is a sub-property of itself.

```
rdfs:subClassOf      rdfs:subPropertyOf    rdf:type
```

... is not RDF(S) valid: for example, saying that `ex:Novel` is a sub-class of `ex:LiteraryWork` does not imply that the class `ex:Novel` is an *instance* of `ex:LiteraryWork`; *instances* of `ex:Novel` are instances of `ex:LiteraryWork`.

```
rdf:type             rdfs:range            rdfs:Class
```

... is RDF(S) valid: the object of a triple with predicate `rdf:type` is always a class.

```
rdf:type             rdfs:range            rdfs:Resource
```

... is RDF(S) valid: the object of a triple with predicate `rdf:type` is always a resource (*and* a class: everything is a resource). In fact, every property has domain and range `rdfs:Resource`: all this states is that the subjects and objects it takes are resources (and everything is a resource)! Of course a property may take more specific domains/ranges as well. (See Remark 4.1.)

```
rdfs:Datatype        rdfs:subClassOf       rdfs:Literal
```

... is not RDF(S) valid. `rdfs:Datatype` contains members like `xsd:int` (which *in turn* contains members like `"2"^^xsd:int`) while `rdfs:Literal` directly contains members like `"2"^^xsd:int`, not datatype classes like `xsd:int`. Instead, *members* of `rdfs:Datatype` (like `xsd:int`) are sub-classes of the class `rdfs:Literal`. There is no direct way to express this relation in RDFS: that `rdfs:Datatype` is a class of sub-classes of `rdfs:Literal`.

Solution for Exercise 4.3

The definition:

```
ex:Bonobo            rdf:type              ex:Species
ex:Chimpanzee        rdf:type              ex:Genus
```

... would seem more appropriate. If we instead selected the sub-class relation, then we would infer that `ex:Kanzi` is an instance of the class `ex:Species`!

Solution for Exercise 4.4

We could complete the triples as follows:

```
ex:inGenus           rdfs:subPropertyOf    rdfs:subClassOf
ex:inOrder           rdfs:subPropertyOf    rdfs:subClassOf
```

This would then entail from the given data:

`ex:Bonobo`	`rdfs:subClassOf`	`ex:Chimpanzee`
`ex:Chimpanzee`	`rdfs:subClassOf`	`ex:Primate`

... which would give us the desired result. We have essentially defined two domain-specific specialisations of the general `rdfs:subClassOf` relation (which make perfect sense in this context).

Finally, the triple:

`ex:inGenus`	`rdfs:subPropertyOf`	`ex:inOrder`

... would probably not make sense! We would not like to infer:

`ex:Bonobo`	`ex:inOrder`	`ex:Chimpanzee`

... since `ex:Chimpanzee` is not an order, but rather a genus.

Solution for Exercise 4.5

First we can note that everything in RDFS is of type resource, so for every term x in the data, we would have:

x	`rdf:type`	`rdfs:Resource`

If now we add the triple from the question:

`rdf:type`	`rdfs:subPropertyOf`	`rdfs:subPropertyOf`

for every term x in the data, we will entail the nonsensical triple:

x	`rdfs:subPropertyOf`	`rdfs:Resource`

Now note that x covers all properties, so any triple:

w	x	y

... will infer:

w	`rdfs:Resource`	y

... resulting in nonsense. We could make matters even worse by defining further such triples that further "redefine" the core RDFS terms using the RDFS terms themselves.

Solution for Exercise 4.6

From set theory, we already have that $A \subseteq A$ (reflexivity) and that if $A \subseteq B \subseteq C$ (transitivity), so we could, for example, simply say that if `ex:X` refers to the class X and `ex:Y` refers to the class Y, then we could assert that the triple (`ex:X`,`rdfs:subClassOf`,`ex:Y`) holds *if and only if* $X \subseteq Y$. (However, as we will see in Section 4.3.4, RDFS does not define such an *if and only if* semantics for sub-class relations for reasons that will be explained later.)

Solution for Exercise 4.7

Much like a class can be interpreted as a set of elements, a property can be interpreted as a set of pairs of elements. Thereafter, we can define the sub-property relation analogously to the sub-class relation: to state that p is a sub-property of q, we can again say that the set of pairs associated with p is a subset ($\subseteq$) of the set of pairs associated with q. We will formalise this idea in Section 4.3.1.

Solution for Exercise 4.8

We could, for example, map `ex:A` to 👸 ($I_s($`ex:A`$) =$ 👸), `ex:B` to 👦 ($I_s($`ex:B`$) =$ 👦), `ex:C` to ⚽ ($I_s($`ex:C`$) =$ ⚽), and `ex:D` also to ⚽ ($I_s($`ex:D`$) =$ ⚽). Another solution would be, for example, to map `ex:A` to 👦 and `ex:B`, `ex:C` and `ex:D` all to 👸. What is most important to realise is that we can map multiple names to one element of the universe in the interpretation. This means that RDF does *not* have a **Unique Name Assumption (UNA)**: we can use multiple identifiers to refer to the same thing. This is clearly an important semantic feature for the Web since it is likely that different websites will create different identifiers for the same resources, or, put in reverse, it seems unlikely that all websites will agree perfectly on a set of identifiers for all possible resources.

Solution for Exercise 4.9

No: there is no such mapping; e.g., if we map `ex:X` to 👸 and `ex:Y` to ♫, the first triple would be assigned `true` but not the second; if we assigned `ex:Y` to 👸 and `ex:X` to ♫, the second triple would be assigned `true` but not the first.

Solution for Exercise 4.10

No: under simple semantics, an empty RDF graph says nothing about a possible world and hence (recalling the **Open World Assumption (OWA)**, which intuitively states that the RDF graph does not need to offer a "complete description" of the world), all simple interpretations assign true to an empty RDF graph. Formally this is shown in Definition 10, which states that a graph is assigned true by an interpretation if none of the triples it contains is assigned false, which trivially holds for the empty RDF graph given any interpretation.

Solution for Exercise 4.11

Yes: a simple interpretation with no properties ($I_P = \emptyset$) will assign false to any non-empty RDF graph: Definition 10 does not specify that I_P has to be non-empty, and if it is empty, the interpretation cannot specify any relations. On the other hand, a non-empty RDF graph must encode at least one such relation, which will have no correspondence in the interpretation. More formally, letting (s, p, o) denote any triple in the RDF graph, if $I_P = \emptyset$, then $I_S(p) \in I_P$ cannot hold and I assigns the triple (and graph) false.

Solution for Exercise 4.12

We can create an interpretation that satisfies every RDF graph such that $|I_R \cup I_P| = 1$; in other words, the interpretation only requires one element (let's call it Ω). More specifically, $I_R = I_P = \Omega$, $I_{EXT}(\Omega) = (\Omega, \Omega)$, I_S maps all IRIs to Ω, and I_L maps all literals to Ω. Essentially this interpretation will map all ground triples in the RDF graph to a binary relation of the form $\Omega(\Omega, \Omega)$. If the RDF graph contains blank nodes – as you may have guessed – those will also be mapped to Ω by the extension function A.

Solution for Exercise 4.13

$G_1 \not\models G_2'$: There are models of G_1 without reflexive relations that are not models of G_2'.

Solution for Exercise 4.14

Recall that an interpretation I is a model of G if and only if it satisfies G, meaning that it satisfies all of the triples of G. Then if $G' \subseteq G$, clearly I must satisfy all triples of G' and thus it must also be a model of G'. Hence we see that any model of G is a model of G', satisfying the definition of entailment.

Solution for Exercise 4.15

Lemma 2 states that if $G' \subseteq G$, then $G \models G'$. Hence we are left to prove that if $G \models G'$ and G' is ground, then $G' \subseteq G$. For the purposes of a proof by contradiction, assume that the result does not hold: that $G \models G'$ and G' is ground and $G' \not\subseteq G$. In this case there must be a ground triple (s,p,o) such that $(s,p,o) \in G'$ and $(s,p,o) \notin G$. Now let us consider the Herbrand interpretation of G; this is a model for G but cannot be for G' since the relation corresponding to (s,p,o) does not appear; hence we have a model for G that is not a model for G', resulting in the contradiction $G \not\models G'$!

Solution for Exercise 4.16

First we will prove that if there exists a blank node mapping μ such that $\mu(G') \subseteq G$, then $G \models G'$ (every model of G is a model of G'). For the purposes of a proof by contradiction, let us assume that $\mu(G') \subseteq G$ holds and that there exists a model I of G that is not a model of G'. Let us assume (without loss of generality) that A is the function of I that maps blank nodes in G (if any) to elements of the universe, and let us define $AI_{LS}(x) = I_{LS}(x)$ if x is a literal or IRI, or $AI_{LS}(x) = A(x)$ otherwise; per Definition 11, we know that for I to be a model of G, it must hold that for every triple $(s,p,o) \in G$, $I_S(p) \in I_P$ and $(AI_{LS}(s), AI_{LS}(o)) \in I_{EXT}(I_S(p))$. To derive the contradiction, we will show that I must be a model of G' given that $\mu(G') \subseteq G$. First observe that if $\mu(G') \subseteq G$, it must hold that every predicate in G' must occur in G (since predicates cannot be blank nodes), and hence for every triple $(s',p',o') \in G'$, $I_S(p') \in I_P$. We are left to show that there exists a mapping A' such that for every triple $(s',p,o') \in G'$, it holds that $(A'I_{LS}(s), A'I_{LS}(o)) \in I_{EXT}(I_S(p))$ (defining $A'I_{LS}$ analogously as before). We define such an A' mapping as follows: $A' \coloneqq AI_{LS} \circ \mu$ (i.e., for a blank node b, $A'(b) = AI_{LS}(\mu(b))$). Observe that A' is a mapping from blank node terms in G' to the universe of I; furthermore, since for every triple $(s',p,o') \in G'$ it holds that $(\mu(s'),p,\mu(o')) \in G$ (assuming μ to be the identity for IRIs and literals), it holds that $(AI_{LS}(\mu(s')), AI_{LS}(\mu(o'))) \in I_{EXT}(I_S(p))$, which im-

plies that $(A'I_{LS}(s'), A'I_{LS}(o')) \in I_{EXT}(I_S(p))$, from which follows that I is a model of G'; hence we arrive at a contradiction!

Second, we are left to prove that if $G \models G'$, then there exists a blank node mapping μ such that $\mu(G') \subseteq G$. The proof is similar to that for Lemma 2 (which already covers the case where G' is ground, leaving us with the case that G' contains at least one blank node). Again relying on proof by contradiction, assume that the result does not hold: that $G \models G'$ and there exists no blank node mapping μ such that $\mu(G') \subseteq G$. Let us say that I is the Herbrand interpretation of G; from $G \models G'$, this implies that I is also a model for G', which implies that the function A of I must map the blank nodes of G' to the relations of G represented in the universe of I; observe then that $A(G') \subseteq G$ as it maps G' to the structure of G. Hence we arrive at a contradiction: the existence of A implies the existence of μ such that $\mu(G') \subseteq G$!

Solution for Exercise 4.17

An RDF graph G is unsatisfiable if and only if it has no models. Recall that $G \models G'$ if and only if the models of G are a subset of the models for G'. But if G has no models, then it must entail every possible RDF graph G'! This is known as the *principle of explosion* in logic: that anything follows from a contradiction, where for G to be unsatisfiable, it must contain some form of contradiction or other ill-defined characteristic.

What about $G' \models G$ where G is unsatisfiable? The models of G' can only be a subset of those for G (the empty set) if G' also has no models. Hence only unsatisfiable graphs can imply unsatisfiable graphs (thankfully, otherwise we could entail contradictions from consistent graphs, which would be rather problematic for the semantics to say the least).

Solution for Exercise 4.18

A simple model of G that is not a model of G' could define, for example, $I_R := \{$, $\}$, $I_P := \{\heartsuit\}$, $I_{EXT}(\heartsuit) := ($, $)$, $I_S($`ex:Adam`$) :=$, $I_S($`ex:loves`$) := \heartsuit$ and $I_S($`ex:Eve`$) :=$ (with an arbitrary choice for other IRIs). This cannot be a simple model for G' (no matter how I_S is further defined for `rdf:type` or `rdf:Property`) since in that RDF graph, the term `ex:loves` appears in the subject and predicate position, meaning that at least one element of the universe (in this case $\heartsuit$) would have to be in the set I_S and I_P.

However, this simple model would no longer be an RDF model of G – nor even an RDF interpretation – since it does not meet the first additional condition of Definition 19: that for every element of $x \in I_P$, it must hold that $(x, I_S($`rdf:Property`$)) \in I_{EXT}(I_S($`rdf:type`$))$. If we extend the original

simple model to meet all such conditions, then we would see that whatever the resulting RDF model of G, it would also have to RDF-satisfy G'.

(Of course many other such counter-examples for the simple-entailment in question could be constructed beyond the one presented here.)

Solution for Exercise 4.19

Every property in I_P is stated to be a sub-property of itself in the following:

- If $p \in I_P$, then $(p,p) \in I_{EXT}(I_S(\texttt{rdfs:subPropertyOf}))$.

Since I_{EXT} maps from I_P to pairs from I_R, each such p must be in I_R for it to hold that $(p,p) \in I_{EXT}(I_S(\texttt{rdfs:subPropertyOf}))$. Hence if I is an RDFS interpretation, then every property must also be considered a resource in I_R; in other words, under RDFS semantics, $I_P \subseteq I_R$.

As for literal values, these are defined to be in I_R (though we define I_{LV}, these are by definition a subset of I_R).

Solution for Exercise 4.20

One example would be to define the domain of a property as a datatype and then use that property with an incompatible datatype; for example, making minimal assumptions on the datatypes supported by D, the following graph would be RDF_D satisfiable but RDFS_D unsatisfiable:

```
ex:name          rdfs:range       rdf:langString
ex:Chile         ex:name          "Chile"
```

If we assume, e.g., `xsd:boolean` to be recognised in the set of supported datatypes D, a clearer example might be:

```
ex:alive         rdfs:range       xsd:boolean
ex:GeorgeOrwell  ex:alive         "no"
```

Note that such examples are RDF_D satisfiable since under RDF semantics, the term `rdfs:range` has no special meaning. There are also other such examples definable using domain, range, sub-class, sub-property, etc., in combination with incompatible datatypes supported by D.

Solution for Exercise 4.21

Yes! IRIs can be mapped to literal values (from I_{LV}), no problem. So for example, we could have a model of this RDF graph where $I_S(\texttt{ex:A}) = I_L(\texttt{"A"\^\^xsd:string})$ without any issue.

Solution for Exercise 4.22

If we apply rdfs2, we could add to G:

```
ex:LemonPie          rdf:type            ex:Recipe
```

Applying rdfD2, we could add:

```
rdfs:domain          rdf:type            rdf:Property
rdfs:range           rdf:type            rdf:Property
rdfs:subPropertyOf   rdf:type            rdf:Property
rdfs:subClassOf      rdf:type            rdf:Property
ex:hasIngredient     rdf:type            rdf:Property
```

Solution for Exercise 4.23

We can add:

```
rdfs:subPropertyOf   rdfs:domain         rdf:Property
rdfs:subPropertyOf   rdfs:range          rdf:Property
```

Combined with rdfs2 and rdfs3, add these two triples to the graph will achieve the desired inferences without needing new rules.

Solution for Exercise 4.24

The set of possible terms always remains finite: note that point 1 in Figure 4.3 restricts the inference process to only consider properties of the form `rdf:_`n appearing in the input graph (which must be finite), and that no rule allows for creating new terms (assuming GrdfD1 replaces rdfD1).

More importantly, given that the set of possible terms remains finite, and that RDF graphs have fixed arity, we know that the set of all possible triples using these terms must be finite: if there are n terms available, then there is a limit of n^3 possible triples. Even in the worst case, the inference procedure would have to terminate once G contains all possible triples: no new conclusions beyond that can be generated.

Solution for Exercise 4.25

We could consider that the following two triples should hold:

```
ex:hasTopping         rdfs:range          ex:Ingredient
ex:hasMeatTopping     rdfs:domain         ex:Recipe
```

Solutions for Chapter 5: Web Ontology Language

Solution for Exercise 5.1

There are variations on this possible, but the following would suffice:

- A zebroid has precisely two parents: one parent is a zebra and one parent is a non-zebra Equus.
- Sire and dam are both parent relations.
- Something cannot be a zebra and non-zebra Equus at the same time.
- The same animal cannot be a dam and a mare. This tells us that Marty and Lea are different animals rather than two names for the same parent; otherwise if it were possible that they were the same parent, Zia could be any Equus since we only know one parent (which could be a zebra, or a non-zebra Equus) and we know nothing about the other parent.

Even though we don't know which parent is the zebra and which the non-zebra, we now know enough to know that Zia is a zebroid.

Solution for Exercise 5.2

With respect to the following point in Example 5.1:

- `ex1:WilliamGolding` and `ex2:WGGolding` refer to the same resource, while `ex2:NormandyInvasion` and `ex3:NormandyInvasion` also refer to the same resource.

  We can now state this using OWL's features for equality as:

```
ex1:WilliamGolding    owl:sameAs          ex2:WGGolding
ex2:NormandyInvasion  owl:sameAs          ex3:NormandyInvasion
```

Solution for Exercise 5.3

With respect to the following points in Example 5.1:

- `ex3:partOf` is transitive, meaning that if x `ex3:partOf` y and y `ex3:partOf` z, then x `ex3:partOf` z;
- `ex1:hasAward` is the inverse of `ex1:awardedTo`, meaning if x `ex1:hasAward` y, then y `ex1:awardedTo` x, and vice versa;
- given a path where x `ex2:foughtIn` y and y `ex3:partOf` z, then x `ex2:foughtIn` z.

We can state:

```
ex3:partOf    rdf:type               owl:TransitiveProperty
ex1:hasAward  owl:inverseOf          ex1:awardedTo
ex2:foughtIn  owl:propertyChainAxiom ( ex2:foughtIn ex3:partOf )
```

Solution for Exercise 5.4

1. Does not hold (only holds for classes, not all resources)
2. Does not hold
3. Holds
4. Does not hold (only holds for classes, not all resources)
5. Holds
6. Holds
7. Does not hold: a contradiction! (... since I_C is not empty)
8. Holds
9. Does not hold

Solution for Exercise 5.5

We can conclude that both A and B are unsatisfiable classes:

```
:A owl:equivalentClass owl:Nothing .
:B owl:equivalentClass owl:Nothing .
```

Note that there is no inconsistency *unless* either A or B has a member (just because a class is unsatisfiable does not mean that the graph will have no models: in this case, models where A and B are both empty can exist).

Solution for Exercise 5.6

The disjointness axiom allows for resources that are neither a member of living nor deceased. On the other hand, the complement axiom defines that every resource must be a member of either living or deceased. Thus, if we know that x is not living, in the disjointness case we cannot infer that it is deceased, whereas in the complement case we can. Note that the complement case entails the disjointness case, being a strictly stronger condition.

Solution for Exercise 5.7

- ```
  :Club owl_equivalentClass
     [ owl_oneOf ( :Ac ... :2c ) ] .
  ```

- ```
  :Eight owl:equivalentClass
     [ owl:oneOf ( :8c :8d :8h :8s ) ] .
  ```

- ```
  :StandardCard owl:equivalentClass
     [ owl:disjointUnionOf ( :Club :Diamond :Heart :Spade ) ] .
  ```

  (Plain union is fine though disjoint-union offers more detail.)

- ```
  :FaceCard owl:equivalentClass
     [ owl:disjointUnionOf ( :Jack :Queen :King :Ace ) ] .
  ```

- ```
  :BlackCard owl:equivalentClass
     [ owl:disjointUnionOf ( :Club :Spade ) ] .
  ```

- ```
  :RedCard owl:equivalentClass
     [ owl:intersectionOf ( :StandardCard [ owl:complementOf :BlackCard ] ) ] .
  ```

  . . . or equivalently . . .

  ```
  :StandardCard owl:equivalentClass
     [ owl:disjointUnionOf ( :BlackCard :RedCard ) ] .
  ```

- ```
  :RedFaceCard owl:equivalentClass
     [ owl:intersectionOf ( :RedCard :FaceCard ) ] .
  ```

Solution for Exercise 5.8

We could state:

```
:Lawnmower rdfs:subClassOf
   [ a owl:Restriction ;
     owl:allValuesFrom owl:Nothing ;
     owl:onProperty :hasChild ] .
```
```

Thereafter, if we stated:

```
:OldRusty a :Lawnmower ;
   :hasChild :YoungSpark .
```

We would infer:

```
:YoungSpark a owl:Nothing .
```

A contradiction!

Solution for Exercise 5.9

Some values from c on property p can be alternatively expressed as min-qualified-cardinality 1 on property p and on class c:

```
:EUCitizen owl:equivalentClass
   [ a owl:Restriction ;
     owl:minQualifiedCardinality 1 ;
     owl:onProperty :citizenOf ;
     owl:onClass :EUState ] .
```

Solution for Exercise 5.10

Neither definition is weaker/stronger than the other. The definition of Examples 5.16 uniquely tells us that the other parent of Zebroid must be a non-Zebra Equus (a Horse, Donkey, etc.). On the other hand, the definition of Example 5.16 uniquely tells us that any resource with exactly one parent who is a Zebra must be a Zebroid. Stating both together is perfectly valid (and implies that the two restriction classes are equivalent by transitivity, meaning that if something has *precisely* one parent that's a Zebra, it must have precisely one parent that's a non-Zebra Equus).

Solution for Exercise 5.11

We can cover the following points:

- that the class `ex4:WWIIVeteran` is the class of all resources with the value `ex3:WorldWarII` for the property `ex2:foughtIn`;
- that the class `ex4:NobelLaureateLit` is the class of all resources with a value from the class `ex1:NobelPrizeLit` for `ex1:hasAward`;

```
ex4:WWIIVeteran owl:equivalentClass
   [ a owl:Restriction ;
     owl:hasValue ex3:WorldWarII ;
     owl:onProperty ex2:foughtIn ] .

ex4:NobelLaureateLit owl:equivalentClass
   [ a owl:Restriction ;
     owl:someValuesFrom ex1:NobelPrizeLit ;
     owl:onProperty ex1:hasAward ] .
```

Solution for Exercise 5.12

- *Define that all values for the property* HAS-PLAYER *are in the class* POKER-PLAYER.

  ```
  :hasPlayer rdfs:range :PokerPlayer .
  ```

  The wording may have been a bit deceptive here. One *can* also use all-values-from as follows:

  ```
  owl:Thing owl:equivalentClass
     [ a owl:Restriction ;
       owl:allValuesFrom :Player ;
       owl:onProperty :hasPlayer ] .
  ```

  But range is much more concise given that it does not matter what the type of the subject is.

- *Define that the property* HAS-CARD *for members of the class* POKER-HAND *takes values only from the class* STANDARD-CARD.

  ```
  :PokerHand rdfs:subClassOf
     [ a owl:Restriction ;
       owl:allValuesFrom :StandardCard ;
       owl:onProperty :hasCard ] .
  ```

  Using `rdfs:subClassOf` states that instances of POKER-HAND (only) use standard cards. It is not appropriate to use `owl:equivalentClass` since other card games may also use cards only from the standard deck; furthermore, we do not wish to entail that resources without values for HAS-CARD (e.g., LAWNMOWERS) are members of POKER-HAND.

- *Define that each* POKER-DEAL *has exactly one value on the property* HAS-PLAYER *and exactly one value on the property* HAS-HAND.

  ```
  :PokerDeal rdfs:subClassOf
     [ a owl:Restriction ;
       owl:cardinality 1 ;
       owl:onProperty :hasPlayer ] .
  ```

```
:PokerDeal rdfs:subClassOf
   [ a owl:Restriction ;
     owl:cardinality 1 ;
     owl:onProperty :hasHand ] .
```

- *Define that each* POKER-HAND *has exactly five values on the property* HAS-CARD.

```
:PokerHand rdfs:subClassOf
   [ a owl:Restriction ;
     owl:cardinality 5 ;
     owl:onProperty :hasCard ] .
```

- *Define that each member of* POKER-DEAL *is the value of exactly one* HAS-DEAL *relation (i.e., it is part of one and only one round).*

```
:PokerDeal rdfs:subClassOf
   [ a owl:Restriction ;
     owl:cardinality 1 ;
     owl:onProperty [ owl:inverseOf :hasDeal ] ] .
```

We could also name the inverse relation for HAS-DEAL. A qualified cardinality on-class POKER-ROUND may also be understood from the question.

- *Define that any* POKER-ROUND *has at least two and at most ten values (inclusive) for the property* HAS-DEAL.

```
:PokerRound rdfs:subClassOf
   [ a owl:Restriction ;
     owl:minCardinality 2 ;
     owl:onProperty :hasDeal ] .

:PokerRound rdfs:subClassOf
   [ a owl:Restriction ;
     owl:maxCardinality 10 ;
     owl:onProperty :hasDeal ] .
```

Could also be defined using an intersection of the restriction classes.

- *Define any* POKER-HAND *having all* CLUB *cards as a* CLUB-FLUSH.

```
:ClubFlush owl:equivalentClass
   [ owl:intersectionOf (
        :PokerHand
        [ a owl:Restriction ;
          owl:allValuesFrom :Club ;
          owl:onProperty :hasCard ] ) ] .
```

The all-values-from class could be replaced with a qualified-cardinality of 5 on property HAS-CARD and class CLUB *assuming* the previous max-cardinality 5 definition on HAS-CARD (if a poker hand can only have 5 cards, and it has 5 clubs, then all the cards must be clubs!).

- *Assuming* Diamond-Flush, Heart-Flush *and* Spade-Flush *are defined likewise, define any* Poker-Hand *where all five cards have the same suit as a* Flush.

```
:Flush owl:equivalentClass
   [ owl:unionOf ( :ClubFlush :DiamondFlush :HeartFlush :SpadeFlush ) ] .
```

  We don't need an intersection with Poker-Hand since this was already defined for each of the constituent union classes. We could define disjoint-union here, but this is already entailed by previous definitions.

- *Define any* Poker-Hand *having four cards of a particular value as a* Four-Of-A-Kind.

```
:FourOfAKind owl:equivalentClass
   [ owl:intersectionOf (
        :PokerHand
        [ owl:unionOf (
             [ a owl:Restriction ;
               owl:qualifiedCardinality 4 ;
               owl:onProperty :hasCard ;
               owl:onClass :Two ]
             ...
             [ a owl:Restriction ;
               owl:qualifiedCardinality 4 ;
               owl:onProperty :hasCard ;
               owl:onClass :Ace ] ) ] ) ] .
```

- *Define any poker hand having three cards of one value and two cards of a different value as a* Full-House.

```
:FullHouse owl:equivalentClass
   [ owl:intersectionOf (
        :PokerHand
        [ owl:unionOf (
             [ a owl:Restriction ;
               owl:qualifiedCardinality 2 ;
               owl:onProperty :hasCard ;
               owl:onClass :Two ]
             ...
             [ a owl:Restriction ;
               owl:qualifiedCardinality 2 ;
               owl:onProperty :hasCard ;
               owl:onClass :Ace ] ) ]
        [ owl:unionOf (
             [ a owl:Restriction ;
               owl:qualifiedCardinality 3 ;
               owl:onProperty :hasCard ;
               owl:onClass :Two ]
             ...
             [ a owl:Restriction ;
               owl:qualifiedCardinality 3 ;
               owl:onProperty :hasCard ;
               owl:onClass :Ace ] ) ] ) ] .
```

⚠ *Define that the* Poker-Hands *in different* Poker-Deals *of the same* Poker-Round *cannot share a card.*

```
:cardDealt owl:propertyChainAxion ( :hasHand :hasCard ) .
:PokerDeal owl:hasKey ( [ owl:inverseOf :hasDeal ] :cardDealt ) .
```

This states that any Poker-Deals in the same round (inverse of has-deal) and with the same card-dealt must be same-as. If we infer such a same-as between two or more Poker-Deals, an inconsistency will arise if these hands together have more than 1 player, or more than 5 cards (which would break previously defined cardinality restrictions since cards and players are defined to be pairwise different in the exercise).

- *Define that no two* Poker-Deals *in the same* Poker-Round *can have the same player.*

```
:PokerDeal owl:hasKey ( [ owl:inverseOf :hasDeal ] :hasPlayer ) .
```

This is similar to the previous definition. As before, we highlight that without a UNA, it is possible to have multiple IRIs for the same Poker-Deal. Hence to state that only one Poker-Deal can have a given Player in a given Poker-Round, without a UNA we should rather think of it as any Poker-Deals with the same Player in the same Poker-Round are same-as (and if that same-as breaks another restriction elsewhere, then a contradiction will arise).

⚠ *Define that any two* Poker-Hands *that do not share all five values for* has-card *are different.*
A trick question: this is already covered by the previous definitions. Since the cards are pairwise different, and since Poker-Hands have precisely five cards, if two Poker-Hands not sharing all five cards were to be the same, then a Poker-Hand would end up with more than five cards and an inconsistency would arise! Hence we know that such Poker-Hands must be different.

- *Define that* beaten-by *is transitive, irreflexive, and has the domain and range* Poker-Hand.

```
:beatenBy a owl:TransitiveProperty , owl:IrreflexiveProperty ;
  rdfs:domain :PokerHand ; rdfs:range :PokerHand .
```

⚠ *Define that any member of* Flush *is* beaten-by *any member of* Full-House *and that any member of* Full-House *is* beaten-by *any member of* Four-Of-A-Kind.

Perhaps the most direct way to do this is as follows:

```
:Flush :classBeatenBy :FullHouse .
:FullHouse :classBeatenBy :FourOfAKind .
:beatenBy owl:propertyChainAxiom
   ( rdf:type :classBeatenBy [ owl:inverseOf rdf:type ] ) .
```

However, here we use the built-in property `rdf:type` as if it were a domain term; this is allowed in unrestricted versions of OWL, but such definitions can cause a lot of problems when left unrestricted and are thus not allowed in the restricted flavours of OWL discussed later.

There are other options that would be valid under restricted versions of OWL. One other trick we can consider for this case is to use *rolification* and the top object property in the following manner [**?**]:

```
:Flush owl:equivalentClass
   [ a owl:Restriction ;
     owl:hasSelf true ;
     owl:onProperty :fl ] .

:FullHouse owl:equivalentClass
   [ a owl:Restriction ;
     owl:hasSelf true ;
     owl:onProperty :fh ] .

:FourOfAKind owl:equivalentClass
   [ a owl:Restriction ;
     owl:hasSelf true ;
     owl:onProperty :fk ] .

:beatenBy owl:propertyChainAxiom ( :fl owl:topObjectProperty :fh ) .
:beatenBy owl:propertyChainAxiom ( :fh owl:topObjectProperty :fk ) .
```

Such a definition is valid in a restricted version of OWL and does not require us to know the precise cardinality of the classes in question.

⚠ *Define that the player of any* Poker-Deal *in a given round that has a hand* beaten-by *by the hand of any other* Poker-Deal *in that round* loses *that round (i.e., that in the example above Frank and Julie lose round 42).*

We will break this up into parts. First we define a Losing-Poker-Deal as one where there exists a path from that deal x_1 to the round x_2, from that round x_2 to a(nother) deal of that same round x_3, from the second deal x_3 to its hand x_4, from the hand x_4 to a hand it beats x_5, and from a hand it beats x_5 to a deal x_6 with the beaten hand (note that, thus far, x_5 and x_6 may be from a different round). If such a path exists where $x_1 = x_6$ (a cycle), then deal x_1 is beaten by deal x_3. We can use has-self to detect such a cycle to classify a losing hand.

```
:beatenByLoop owl:propertyChainAxiom
  ( [ owl:inverseOf :hasDeal ] :hasDeal :hasHand
       [ owl:inverseOf :beatenBy ] [ owl:inverseOf :hasHand ] ) .

:LosingPokerDeal owl:equivalentClass
  [ a owl:Restriction ;
    owl:hasSelf true ;
    owl:onProperty :beatenByLoop ] .
```

Next we can define that a player loses a round if they have a Losing-Poker-Deal in that round. For this, we will use rolification again:

```
:LosingPokerDeal owl:equivalentClass
   [ a owl:Restriction ;
     owl:hasSelf true ;
     owl:onProperty :lpd ] .

:loses owl:propertyChainAxiom
   ( [ owl:inverseOf :hasPlayer ] :lpd [ owl:inverseOf :hasHand ] ) .
```

☢ *Define that any two Poker-Hands with the same five values for has-card are the same.*

We first define the class of all pairs of hands:

```
:PokerHandPair owl:equivalentClass
  [ a owl:Restriction ;
    owl:cardinality 2 ;
    owl:onProperty :hasMember ] .

:hasMember rdfs:range :PokerHand .
```

Next we define the cards in a pair:

```
:hasMemberWithCard owl:propertyChainAxiom ( :hasMember :hasCard ) .
```

Finally, we define that pairs of hands must have at least six cards:

```
:PokerHandPair rdfs:subClassOf
  [ a owl:Restriction ;
    owl:minCardinality 6 ;
    owl:onProperty :hasMemberWithCard ] .
```

Now, if there were to exist two distinct Poker-Hands with the same five cards, then there would exist a Poker-Hand-Pair with those two hands but only five distinct cards, which would contradict the final axiom. Hence, given two Poker-Hands with the same five cards, we must conclude that they are the same to avoid the contradiction.
(Note, however, that the latter axiom will not be permitted in restricted flavours of OWL as it requires counting property chains.)

Solution for Exercise 5.13

We assume O and O' to be as defined in Remark 5.36. Having defined these ontologies, rather than using entailment to decide the Domino Problem, we will show how the following tasks can be used equivalently:

Ontology Consistency: Add a triple "`[] a :D .`" – stating that there exists something that is a member of the class `:D` – to O. There is an infinite tiling if and only if the resulting ontology is consistent.

Ontology Equivalence: Check if O and $O \cup O'$ are equivalent. There is an infinite tiling if and only if both ontologies are not equivalent.

Class Satisfiability: Check if the class `:D` in O is satisfiable. There is an infinite tiling if and only if the class is satisfiable.

Class Subsubmption: Check if the class `:D` in O is a sub-class of `owl:Nothing`. There is an infinite tiling if and only if the subsumption is not entailed.

Instance Checking: Add to O the following:

```
:DE owl:equivalentClass
  [ owl:disjointUnionOf ( :D :E ) ] .
:e a :DE .
```

Check in the extended ontology if `:e` is an instance of `:E`. There is an infinite tiling if and only if `:e` is not known to be an instance of `:E`.

Boolean Conjunctive Query Answering: Check if O' is an answer to O. There is an infinite tiling if and only if it is not an answer.

Solution for Exercise 5.14

R-Box:

$$\textsc{ingred} \sqsubseteq \textsc{contains} \qquad \mathsf{Trans}(\textsc{contains})$$
$$\textsc{containedIn} \equiv \textsc{contains}^{-} \qquad \textsc{allergy} \circ \textsc{containedIn} \sqsubseteq \textsc{unsuitableFor}$$

T-Box:

$$\textsc{CitrusFruit} \sqsubseteq \exists \textsc{contains}.\{\textsc{Citrus}\}$$
$$\textsc{Coeliac} \equiv \exists \textsc{allergy}.\{\textsc{Gluten}\}$$

Solutions for Chapter 6: SPARQL Query Language

Solution for Exercise 6.1

```
SELECT DISTINCT ?war
WHERE {
  ?npl a :NobelPrizeLiterature ; :winner ?winner .
  ?war :combatant ?winner .
}
```

Though not specified in the question (or needed for this example), we add `DISTINCT` to remove duplicates. There are a number syntactic variations pos-

sible, where, for example, the order of triple patterns is not important. The query will return a solution mapping `?war` to `:FrancoPrussianWar`.

Solution for Exercise 6.2

We could write this several ways. First we could write:

```
SELECT DISTINCT ?w
WHERE {
  {  ?nplBB a :NobelPrizeLiterature ; :winner :BBjørnson .
     ?nplBB :prev ?nplPrev . ?nplPrev :winner ?w . }
  UNION
  {  ?nplBB a :NobelPrizeLiterature ; :winner :BBjørnson .
     ?nplBB :next ?nplNext . ?nplNext :winner ?w . }
}
```

More succinctly we could write (for example):

```
SELECT DISTINCT ?w
WHERE {
  ?nplBB a :NobelPrizeLiterature ; :winner :BBjørnson .
  ?npl :winner ?w .
  { ?nplBB :prev ?npl }
  UNION
  { ?nplBB :next ?npl }
}
```

Either query will return `:TMommsen`, `:FMistral` and `:JEchegaray` for `?w`. (We could also consider using two different variables – one for winners before and one for winners after – as discussed in Remark 6.6.)

Solution for Exercise 6.3

?predecessor	?winner
	:SPrudhomme
:SPrudhomme	:TMommsen
:SPrudhomme	:TMommsen
:TMommsen	:BBjørnson
:TMommsen	:BBjørnson
:BBjørnson	:FMistral
:BBjørnson	:FMistral
:BBjørnson	:JEchegaray
:BBjørnson	:JEchegaray

Solution for Exercise 6.4

- ```
  SELECT DISTINCT ?w1 ?w2
  WHERE {
    ?npl1 a :NobelPrizeLiterature ; :winner ?w1 .
    ?npl2 a :NobelPrizeLiterature ; :winner ?w2 .
    ?w1 :country ?c1 .
    ?w2 :country ?c2 .
    ?c1 :war ?war .
    ?c2 :war ?war .
    FILTER (?c1 != ?c2 && ?w1 != ?w2)
  }
  ```

  This could also be expressed as two `FILTER` clauses – one for `?c1!=?c2` and one for `?w1!=?w2` – where multiple such clauses in the same scope act as a conjunction. Four results would be returned:

?w1	?w2
:SPrudhomme	:TMommsen
:TMommsen	:SPrudhomme
:TMommsen	:FMistral
:FMistral	:TMommsen

  (One may argue that `?w1!=?w2` is redundant if we have `?c1!=?c2`; this argument assumes a winner can come from at most one country.)

- ```
  SELECT DISTINCT ?w1 ?w2 ?gap
  WHERE {
    ?npl1 a :NobelPrizeLiterature ; :winner ?w1 ; :year ?y1 .
    ?npl2 a :NobelPrizeLiterature ; :winner ?w2 ; :year ?y2 .
    ?w1 :country ?c .
    ?w2 :country ?c .
    BIND ((xsd:integer(str(?y1)) - xsd:integer(str(?y2))) AS ?gap)
    FILTER (?gap >= 0 && ?w1 != ?w2)
  }
  ```

  The query will return:

?w1	?w2	?gap
:SPrudhomme	:FMistral	"3"^^xsd:integer

Solution for Exercise 6.5

In relation to Remark 6.11 and what happens when no join variable is present, consider the query:

```
SELECT ?npl
WHERE {
  ?npl a :NobelPrizeLiterature .
  OPTIONAL { :Germany :war ?war . }
  FILTER (!bound(?war))
}
```

No matter what the solution for the outer query, the inner query will have a solution; hence `?war` will always be bound and empty results will be given. This illustrates that `OPTIONAL/!bound` acts like `FILTER NOT EXISTS`.

In relation to the second issue of variable scope, it depends on where the additional filter condition is added, but if it is inside the `OPTIONAL`, then the situation is similar as for `MINUS`.

Solution for Exercise 6.6

In the following solutions, we use `MINUS` to express negation, but we could alternatively use `FILTER NOT EXISTS` or `OPTIONAL/!bound`.

1.
```
SELECT DISTINCT ?npl
WHERE {
  ?npl a :NobelPrizeLiterature ; :winner ?w .
  ?w :country ?c .
  MINUS { ?c :war ?war . }
}
```

Returns `:NPL1903` and `:NPL1904`.

2.
```
SELECT DISTINCT ?npl
WHERE {
  ?npl a :NobelPrizeLiterature .
  MINUS {
    ?npl :winner ?w .
    ?w :country ?c .
    ?c :war ?war .
  }
}
```

Returns `:NPL1903`.

3.
```
SELECT DISTINCT ?npl
WHERE {
  ?npl a :NobelPrizeLiterature .
  MINUS {
    ?npl :winner ?w .
    ?w :country ?c .
    MINUS {
      ?c :war ?war .
    }
  }
}
```

Returns `:NPL1901` and `:NPL1902`.

Solution for Exercise 6.7

The first two queries will return `:NPL1904` (since that prize has a winner not from `:France`), while the third query will not (since that prize has a winner from `:France`). Comparing the second and third queries, the fact that `?winner` is bound before/inside the `FILTER NOT EXISTS` clause is thus important: if bound before, we keep solutions involving winners not from `:France`; if bound inside, we keep solutions involving prizes without winners from `:France`.

Solution for Exercise 6.8

A possible solution would be:

```
SELECT ?w1 ?w2
WHERE {
  ?np1 a/rdfs:subClassOf* :NobelPrize  ; ?winner ?w1 .
  ?np2 a/rdfs:subClassOf* :NobelPrize  ; ?winner ?w2 .
  ?np1 :next* ?np2 .
  ?w1 :country ?c .
  ?w2 :country ?c .
  FILTER (?w1 != ?w2)
  FILTER NOT EXISTS {
    ?np a/rdfs:subClassOf* :NobelPrize  ; ?winner ?w .
    ?w :country ?c .
    ?np :prev+ ?np1 .
    ?np :next+ ?np2 .
  }
}
```

Here, the variable `?w` looks for the existence of a compatriot winner between `?w1` and `?w2`; we check that `?w` is a compatriot of both `?w1` and `?w2` in case

a winner may be associated with more than one country, and we check that the corresponding prize is between `?np1` and `?np2`.

Solution for Exercise 6.9

Here we use a property path on `:prev` to find previous winners, though one may also consider finding previous winners by year.

```
SELECT ?w1 (count(distinct ?w2) AS ?prev)
WHERE {
  ?npl1 a :NobelPrizeLiterature ; :winner ?w1 .
  OPTIONAL {
    ?npl2 a :NobelPrizeLiterature ; :winner ?w2 .
    ?npl1 :prev+ ?npl2 .
    ?w1 :country/^:country ?w2 .
  }
}
GROUP BY ?w1
```

The results of this query are then as follows:

?w1	?prev
:SPrudhomme	"0"^^xsd:integer
:TMommsen	"0"^^xsd:integer
:BBjørnson	"0"^^xsd:integer
:FMistral	"1"^^xsd:integer
:JEchegaray	"0"^^xsd:integer

Noting that the `count` function ignores unbound values, the optional graph pattern then ensures that winners with zero previous winners from the same country are returned (as opposed to the solution being omitted).

Solution for Exercise 6.10

This is (arguably) a trick question. We can use a plain filter as follows.

```
SELECT (count(distinct ?winner) AS ?num)
WHERE {
  ?npl a :NobelPrizeLiterature ; :winner ?winner ; :year ?year .
  FILTER (xsd:integer(str(?year)) > 1902)
}
```

The query returns `3` as a solution for `?num`.

Solution for Exercise 6.11

We use a sub-query to count the distinct Nobel Prizes in Literature per country, then take the average:

```
SELECT (avg(?num) AS ?average)
WHERE {
  {
    SELECT ?country (count(distinct ?npl) AS ?num)
    WHERE {
      ?npl a :NobelPrizeLiterature ; :winner ?winner .
      ?winner :country ?country .
    }
    GROUP BY ?country
  }
}
```

(Another option would be, for example, to count the distinct Nobel Prizes per country, count the number of countries, and divide the two.)

Solution for Exercise 6.12

Unfortunately this query gets a bit messy:

```
SELECT ?country
WHERE {
  ?npl a :NobelPrizeLiterature ; :winner ?winner .
  ?winner :country ?country .
  {
    SELECT (avg(?num) AS ?average)
    WHERE {
      {
        SELECT ?country (count(distinct ?npl) AS ?num)
        WHERE {
          ?npl a :NobelPrizeLiterature ; :winner ?winner .
          ?winner :country ?country .
        }
        GROUP BY ?country
      }
    }
  }
}
GROUP BY ?country ?average
HAVING (count(distinct ?npl) > ?average)
```

There are two aspects of note: first the variables used in the outer query (e.g. `?npl`) have no correspondence with the synonymous variables in the innermost query that are projected away; second, we need to group on `?average` as

well as on `?country` in the outer query: though it does not affect the grouping of countries (there is only one global average), we need `?average` in the group key to enable its use in the `HAVING` condition.

Solution for Exercise 6.13

We use `OPTIONAL` to support the case where laureate has zero successors and zero predecessors (without `OPTIONAL`, `:BBjørnson` will still be returned).

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/>

SELECT ?winner
WHERE {
 ?npl a :NobelPrizeLiterature ; :winner ?winner .
 OPTIONAL { ?npl :prev+ ?nplp . ?nplp :winner ?prev . }
 OPTIONAL { ?npl :next+ ?npln . ?npln :winner ?next . }
}
GROUP BY ?winner
HAVING (count(distinct ?prev) = count(distinct ?next))
```

Solution for Exercise 6.14

The most general query (not knowing which data are in named graphs and which are in the default graph) would be as follows:

```
SELECT DISTINCT ?pw
WHERE {
  { ?w :combatant ?pw } UNION { GRAPH ?g { ?w :combatant ?pw } }
  { ?p :winner ?pw } UNION { GRAPH ?g { ?p :winner ?pw } }
  { ?p a :NobelPrizeLiterature } UNION
    { GRAPH ?g { ?p a :NobelPrizeLiterature } }
}
```

Solution for Exercise 6.15

Without `GRAPH`, we must use `FROM`:

```
SELECT ?prize
FROM :G-JE
FROM :G-FM
WHERE {
  ?prize :winner ?winner1 . ?winner1 :country :France .
  ?prize :winner ?winner2 . ?winner2 :country :Spain .
}
```

Solution for Exercise 6.16

```
CONSTRUCT {
  ?w1 :succ ?w2 .
  ?w2 :prev ?w1 .
  ?w1a :shared ?w1b .
}
WHERE {
  { ?npl1 a :NobelPrizeLiterature ; ?winner ?w1 .
    ?npl2 a :NobelPrizeLiterature ; ?winner ?w2 .
    ?w1 :country ?c .
    ?w2 :country ?c .
    ?np1 :next+ ?np2 .
    FILTER (?w1 != ?w2)
    FILTER NOT EXISTS {
      ?npl a :NobelPrizeLiterature ; ?winner ?w .
      ?w :country ?c .
      ?npl :prev+ ?npl1 .
      ?npl :next+ ?npl2 .
      FILTER (?w NOT IN (?w1,?w2))
    }
  } UNION {
    ?npl a :NobelPrizeLiterature ; ?winner ?w1a , ?w1b .
    ?w1a :country/^:country ?w1b .
    FILTER (?w1a != ?w1b)
  }
}
```

We use separate variables for the predecessor/successor laureates and the sharing laureates to bind the correct triple in the `CONSTRUCT` clause; in each solution, either `?w1` and `?w2` will be unbound, generating a `:shared` relation, or `?w1a` and `w1b` will be unbound, generating `:succ` and `:prev` relations.

Solution for Exercise 6.17

```
DELETE { GRAPH ?g { ?s ?p ?o } }
INSERT { ?s ?p ?o }
WHERE  { GRAPH ?g { ?s ?p ?o } }
```

Solution for Exercise 6.18

```
INSERT { GRAPH ?c { ?p :winner ?w . ?w :country ?c } }
WHERE  {
  GRAPH ?g { ?p :winner ?w . ?w :country ?c }
  ?p a :NobelPrizeLiterature .
}
```

Solution for Exercise 6.19

Without the triple that Bill is in year 2, both queries return no solutions! Though the query solution that maps `?student` to `:Jane` does not contain any "blocked" literals, since no literal for the value 2 appears in the graph nor the basic graph pattern, the intermediate solution for the basic graph pattern that maps `?student` to `:Jane` and `?year` to `"2"^^xsd:decimal` will be blocked. Note that in the second query, though 2 does appear in the query, it does not appear in the basic graph pattern (for which entailment is defined).

When the triple that Bill is in year 2 is added to the graph, both queries will return `:Jane` and `:Bill` as solutions.

Solutions for Chapter 7: Shape Constraints and Expressions

Solution for Exercise 7.1

There are multiple possible solutions (and also perhaps multiple possible interpretations) for defining the constraints mentioned in each part of the question. Here we provide example solutions.

(1) Any entity can only have one value for year: a value in `xsd:gYear`

```
o:year a owl:FunctionalProperty .
o:year rdfs:range xsd:gYear .
```

(Throwing an inconsistency would require that the OWL reasoner support the `xsd:gYear` datatype, which may not be the case.)

(6) Winners should be people (and cannot be countries):

```
o:winner rdfs:range o:Person .
o:Person owl:disjointWith o:Country .
```

(7) Nobel Prizes can only have one number: a value in `xsd:integer`

```
o:NobelPrizePhysics rdfs:subClassOf o:NobelPrize . # given

o:NobelPrize rdfs:subClassOf
  [ a owl:Restriction ;
    owl:onProperty o:number ;
    owl:cardinality 1 ] .

o:number rdfs:range xsd:integer .
```

(A qualified cardinality would not work here as it would imply that the number property can have multiple values, only one of which is a number.)

(8) No entity can have itself as previous

```
o:prev a owl:IrreflexiveProperty .
```

(10) Name must take a non-blank string as value

```
o:BlankString owl:equivalentClass
  [ owl:oneOf ( "" ) ] .

o:NonBlankString rdfs:subClassOf xsd:string ;
  owl:disjointWith o:BlankString .

o:name rdfs:range o:NonBlankString .
```

String-length facets would also work, as would defining `o:NonBlankString` to be the intersection of `xsd:string` and `o:BlankString`'s complement.

Solution for Exercise 7.2

```
ASK {
  {
    ?npp a o:NobelPrizePhysics ; o:winner ?w .
    ?w a ?wt . FILTER(?wt != o:Person)
  }
  UNION
  {
    {
      SELECT ?npp {
        ?npp a o:NobelPrizePhysics .
        OPTIONAL { ?npp o:winner ?w }
      }
      GROUP BY ?npp
      HAVING (count(?w)=0 || count(?w)>3)
    }
  }
}
```

Solution for Exercise 7.3

```
v:LorentzAndZeemanShape a sh:NodeShape ;
  sh:targetNode d:HLorentz , d:PZeeman .
```

Solution for Exercise 7.4

```
v:WinnerObjectShape a sh:PropertyShape ;
  sh:targetObjectsOf o:winner ;
  sh:path o:country ;
  sh:minCount 1 .
```

Note that the following would not work:

```
v:PrizeShape a sh:PropertyShape ;
  sh:targetClass o:Prize ;
  sh:path ( o:winner o:country ) ;
  sh:minCount 1 .
```

The latter property shape requires that a prize have at least one country defined on at least one winner, not that *each* winner have a country defined.

Solution for Exercise 7.5

```
v:CountryOfPersonShape a sh:PropertyShape ;
  sh:targetClass o:Country ;
  sh:targetObjectsOf o:country ;
  sh:path [ sh:inversePath o:country ] ;
  sh:class o:Person ;
  sh:minCount 1 ;
  sh:property v:PersonOfCountryShape .

v:PersonOfCountryShape a sh:PropertyShape ;
  sh:path [ sh:inversePath o:winner ] ;
  sh:minCount 1 .
```

Solution for Exercise 7.6

```
v:NobelPrizeShape a sh:NodeShape ;
  sh:targetClass o:NobelPrize ;
  sh:and (
    v:NobelPrizeAtMostOneNext
    v:NobelPrizeAtMostOnePrev
    [
      sh:or (
        v:NobelPrizeAtLeastOneNext
        v:NobelPrizeAtLeastOnePrev
      )
    ]
  ) .

v:NobelPrizeAtLeastOneNext a sh:PropertyShape ;
  sh:path o:next ;
  sh:minCount 1 .

v:NobelPrizeAtMostOneNext a sh:PropertyShape ;
  sh:path o:next ;
  sh:maxCount 1 .

v:NobelPrizeAtLeastOnePrev a sh:PropertyShape ;
  sh:path o:prev ;
  sh:minCount 1 .

v:NobelPrizeAtMostOnePrev a sh:PropertyShape ;
  sh:path o:prev ;
  sh:maxCount 1 .
```

Solution for Exercise 7.7

```
PREFIX o: <http://nobel.org/ont#>
PREFIX v: <http://nobel.org/val#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

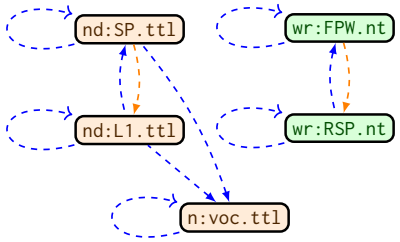
v:NobelLaureate IRI {
  ( o:name xsd:string |
    o:firstname xsd:string ;
    o:lastname xsd:string ) ;
  o:country @v:Country + ;
  a [ o:Person o:Organisation ] ;
  ^o:winner { a [ o:NobelPrize ] ; a . * } +
}

v:Country {
  o:name [ @en~ ] ;
  o:name . * ;
  a [ o:Country ]
}
```

Solutions for Chapter 8: Linked Data

Solution for Exercise 8.1

Links are added from `nd:SP.ttl` (data about Sully Prudhomme on the Nobel site) to `nd:L1.ttl` (data about his Nobel Prize), and from `wr:FPW.nt` (data about the Franco Prussian War) to `wr:RSP.nt` (data about the Sully Prudhomme on the Wiki site). Note that for a large dataset, adding links from `wr:FPW.nt` to all people who participated in the war may lead to a prohibitively large document; to avoid such a case, relations with high degree or a limit on incoming relations can be defined when dereferencing documents.



Solution for Exercise 8.2

- In the document `nd:SP.ttl`, a same-as link can be added between `ne:SPrudhomme` and `w:RSPrudhomme`.
- In the document `wr:RSP.nt`, a same-as link can be added from between `w:RSPrudhomme` and `ne:SPrudhomme`.

With both links, an agent starting on one site can find the related data on the other. Of course, other external IRIs can also be added to datasets such as Wikidata; here we only discuss links for the data shown in the figure.

Solution for Exercise 8.3

Here we show the triples produced considering only the first row of each table.

```
@base <http://nobel.org/db/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<Laureate/winner=Sully%20Prudhomme;year=1901;A_name=Literature>
  a <Laureate> ;
  <Laureate#winner> "Sully Prudhomme" ;
  <Laureate#year> "1901"^^xsd:integer ;
  <Laureate#A_name> "Literature" ;
  <Laureate#ref-A_name> <Award/name=Literature> .
    # genre is omitted since the value is NULL

···
<Award/name=Literature>
  a <Award> ;
  <Award#name> "Literature" ;
  <Award#since> "1901"^^xsd:integer .

···
```

Solution for Exercise 8.4

```
@base <http://nobel.org/db/> .
@prefix nv: <http://nobel.org/voc#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<AwardMap> a rr:TriplesMap ;

  rr:logicalTable [ rr:tableName "Award" ] ;

  rr:subjectMap [
    rr:template "http://nobel.org/voc#NobelPrize{name}" ;
  ] ;

  rr:predicateObjectMap [
    rr:predicate rdfs:subClassOf ;
    rr:object nv:NobelPrize
  ] .

<LaureateMap> a rr:TriplesMap ;

  rr:logicalTable [ rr:tableName "Laureate" ] ;

  rr:subjectMap [
    rr:template "http://nobel.org/db/p/{A_name}#{year}" ;
    rr:class nv:NobelPrize
  ] ;

  rr:predicateObjectMap [
    rr:predicate rdf:type ;
    rr:objectMap [
      rr:template "http://nobel.org/voc#NobelPrize{A_name}"
    ]
  ] ;

  rr:predicateObjectMap [
    rr:predicate nv:winner ;
    rr:objectMap [
      rr:template "http://nobel.org/db/l/{winner}"
    ]
  ] ;

  rr:predicateObjectMap [
    rr:predicate nv:year ;
    rr:objectMap [
      rr:column "year" ;
      rr:datatype xsd:gYear
    ]
  ] .
```

To model the genres, we should add the winner to the identifier in the subject map of `LaureateMap` (and the additional predicate–object map to represent the genre); if we did not extend the identifier, in the case of two winners in the same year with different genres (see the Literature prize in 1904), we would not know which genre corresponded to which year. (An alternative solution if the resulting subject IRI appeared too convoluted would be to analogously use a blank node for each row of the **`Laureate`** `table`.)

Solution for Exercise 8.5

Here we highlight only the changes required to `ShowMap`.

```
...
  rr:predicateObjectMap [
    rr:predicate schema:workPresented ;
    rr:objectMap [
      rr:template "http://mov.ie/db/m/{M_id}"
    ]
  ] .
```

Solution for Exercise 8.6

```
@base <http://nobel.org/db/> .
@prefix nv: <http://nobel.org/voc#> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<LaureateMap> a rr:TriplesMap ;
  rr:logicalTable [ rr:tableName "Laureate" ] ;
  rr:subjectMap [
    rr:template "http://nobel.org/db/l/{winner}" ;
    rr:class nv:Laureate
  ] .

<AwardMap> a rr:TriplesMap ;
  rr:logicalTable [ rr:tableName "Award" ] ;
  rr:subjectMap [
    rr:template "http://nobel.org/db/p/{name}" ;
    rr:class nv:NobelPrize
  ] ;
  rr:predicateObjectMap [
    rr:predicate nv:inaugural ;
    rr:objectMap [
      rr:parentTriplesMap <LaureateMap> ;
      rr:joinCondition [
        rr:child "name" ; rr:parent "A_name"
      ] ;
      rr:joinCondition [
        rr:child "since" ; rr:parent "year"
      ]
    ]
  ] .
```

Solution for Exercise 8.7

```
@prefix dct: <http://purl.org/dc/terms/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ldp: <http://www.w3.org/ns/ldp#> .
@prefix schema: <http://schema.org/> .

<> a ldp:IndirectContainer ;
  dct:title "hoyt's Offerings"@en ;
  ldp:membershipResource <../profile#cine> ;
  ldp:isMemberRelationOf schema:availableAtOrFrom ;
  ldp:insertedContentRelation foaf:primaryTopic .
```

(We assume the container is currently empty.)