W3C Proposed Edited Recommendation

OWL 2 Web Ontology Language
Structural Specification and Functional-Style Syntax

W3C Proposed Edited Recommendation 18 October 2012

This version:
http://www.w3.org/TR/2012/PER-owl2-syntax-20121018/
Latest version (series 2):
http://www.w3.org/TR/owl2-syntax/
Latest Recommendation:
http://www.w3.org/TR/owl-syntax
Previous version:
http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
Editors:
Boris Motik, University of Oxford
Peter F. Patel-Schneider, Nuance Communications
Bijan Parsia, University of Manchester
Contributors: (in alphabetical order)
Conrad Bock, National Institute of Standards and Technology (NIST)
Achille Fokoue, IBM Corporation
Peter Haase, FZI Research Center for Information Technology
Rinke Hoekstra, University of Amsterdam
Ian Horrocks, University of Oxford
Alan Ruttenberg, Science Commons (Creative Commons)
Uli Sattler, University of Manchester
Michael Smith, Clark & Parsia

A color-coded version of this document showing changes made since the previous version is also available.

This document is also available in these non-normative formats: PDF version.

Abstract

The OWL 2 Web Ontology Language, informally OWL 2, is an ontology language for the Semantic Web with formally defined meaning. OWL 2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents. OWL 2 ontologies can be used along with information written in RDF, and OWL 2 ontologies themselves are primarily exchanged as RDF documents. The OWL 2 Document Overview describes the overall state of OWL 2, and should be read before other OWL 2 documents.

The meaningful constructs provided by OWL 2 are defined in terms of their structure. As well, a functional-style syntax is defined for these constructs, with examples and informal descriptions. One can reason with OWL 2 ontologies under either the RDF-Based Semantics [*OWL 2 RDF-Based Semantics*] or the Direct Semantics [*OWL 2 Direct Semantics*]. If certain restrictions on OWL 2 ontologies are satisfied and the ontology is in OWL 2 DL, reasoning under the Direct Semantics can be implemented using techniques well known in the literature.

Status of this Document

May Be Superseded

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

Summary of Changes

There have been no substantive changes since the previous version. For details on the minor changes see the change log and color-coded diff.

W3C Members Please Review By 15 November 2012

The W3C Director seeks review and feedback from W3C Advisory Committee representatives, via their review form by 15 November 2012. This will allow the Director to assess consensus and determine whether to issue this document as a W3C Edited Recommendation.

Others are encouraged by the OWL Working Group to continue to send reports of implementation experience, and other feedback, to public-owl-comments@w3.org (public archive). Reports of any success or difficulty with the test cases are encouraged. Open discussion among developers is welcome at public-owl-dev@w3.org (public archive).

No Endorsement

Publication as a Proposed Edited Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Patents

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

W3C Proposed Edited Recommendation

Hide Diagrams | Hide Grammar | Hide Examples | Hide FSS in Examples | Show RDF in Examples

1 Introduction

This document defines the OWL 2 language. The core part of this specification — called the *structural specification* — is independent of the concrete exchange syntaxes for OWL 2 ontologies. The structural specification describes the conceptual structure of OWL 2 ontologies and thus provides a normative abstract representation for all (normative and nonnormative) syntaxes of OWL 2. This allows for a clear separation of the essential features of the language from issues related to any particular syntax. Furthermore, such a structural specification of OWL 2 provides the foundation for the implementation of OWL 2 tools such as APIs and reasoners. Each OWL 2 ontology represented as an instance of this conceptual structure can be converted into an RDF graph [*OWL 2 RDF Mapping*]; conversely, most OWL 2 ontologies represented as RDF graphs can be converted into the conceptual structure defined in this document [*OWL 2 RDF Mapping*].

This document also defines the *functional-style syntax*, which closely follows the structural specification and allows OWL 2 ontologies to be written in a compact form. This syntax is used in the definitions of the semantics of OWL 2 ontologies, the mappings from and into the RDF/XML exchange syntax, and the different profiles of OWL 2. Concrete syntaxes, such as the functional-style syntax, often provide features not found in the structural specification, such as a mechanism for abbreviating IRIs.

Finally, this document defines OWL 2 DL — the subset of OWL 2 with favorable computational properties. Each RDF graph obtained by applying the RDF mapping to an OWL 2 DL ontology can be converted back into the conceptual structure defined in this document by means of the reverse RDF mapping [*OWL 2 RDF Mapping*].

An OWL 2 ontology is a formal description of a domain of interest. OWL 2 ontologies consist of the following three different syntactic categories:

- *Entities*, such as classes, properties, and individuals, are identified by IRIs. They form the primitive *terms* of an ontology and constitute the basic elements of an ontology. For example, a class *a:Person* can be used to represent the set of all people. Similarly, the object property

a:parentOf can be used to represent the parent-child relationship. Finally, the individual *a:Peter* can be used to represent a particular person called `"Peter"`.
- *Expressions* represent complex notions in the domain being described. For example, a *class expression* describes a set of individuals in terms of the restrictions on the individuals' characteristics.
- *Axioms* are statements that are asserted to be true in the domain being described. For example, using a *subclass axiom*, one can state that the class *a:Student* is a subclass of the class *a:Person*.

These three syntactic categories are used to express the *logical* part of OWL 2 ontologies — that is, they are interpreted under a precisely defined semantics that allows useful inferences to be drawn. For example, if an individual *a:Peter* is an instance of the class *a:Student*, and *a:Student* is a subclass of *a:Person*, then from the OWL 2 semantics one can derive that *a:Peter* is also an instance of *a:Person*.

In addition, entities, axioms, and ontologies can be *annotated* in OWL 2. For example, a class can be given a human-readable label that provides a more descriptive name for the class. Annotations have no effect on the logical aspects of an ontology — that is, for the purposes of the OWL 2 semantics, annotations are treated as not being present. Instead, the use of annotations is left to the applications that use OWL 2. For example, a graphical user interface might choose to visualize a class using one of its labels.

Finally, OWL 2 provides basic support for ontology modularization. In particular, an OWL 2 ontology *O* can import another OWL 2 ontology *O*' and thus gain access to all entities, expressions, and axioms in *O*'.

This document defines the structural specification of OWL 2, the functional syntax for OWL 2, the behavior of datatype maps, and OWL 2 DL. Only the parts of the document related to these three purposes are normative. The examples in this document are informative and any part of the document that is specifically identified as informative is not normative. Further, the informal descriptions of the semantics of OWL 2 constructs in this document are informative; the Direct Semantics [*OWL 2 Direct Semantics*] and the RDF-Based [*OWL 2 RDF-Based Semantics*] are precisely specified in separate documents.

The italicized keywords *must*, *must not*, *should*, *should not*, and *may* are used to specify normative features of OWL 2 documents and tools, and are interpreted as specified in RFC 2119 [*RFC 2119*].

2 Preliminary Definitions

This section presents certain preliminary definitions that are used in the rest of this document.

2.1 Structural Specification

The structural specification of OWL 2 consists of all the figures in this document and the notion of structural equivalence given below. It is used throughout this document to precisely specify the structure of OWL 2 ontologies and the observable behavior of OWL 2 tools. An OWL 2 tool *may* base its APIs and/or internal storage model on the structural specification; however, it *may* also choose a completely different approach as long as its observable behavior conforms to the one specified in this document.

The structural specification is defined using the Unified Modeling Language (UML) [*UML*], and the notation used is compatible with the Meta-Object Facility (MOF) [*MOF*]. This document uses only a very simple form of UML class diagrams that are expected to be easily understandable by readers familiar with the basic concepts of object-oriented systems. The following list summarizes the UML notation used in this document.

- The names of the UML classes from the structural specification are written in bold font.
- The names of abstract UML classes (i.e., UML classes that are not intended to be instantiated) are written in bold and italic font.
- Instances of the UML classes of the structural specification are connected by associations, many of which are of the one-to-many type. Associations whose name is preceded by / are *derived* — that is, their value is determined based on the value of other associations and attributes. Whether the objects participating in associations are ordered and whether repetitions are allowed is made clear by the following standard UML conventions:
  - By default, all associations are sets; that is, the objects in them are unordered and repetitions are disallowed.
  - The `{ ordered,nonunique }` attribute is placed next to the association ends that are ordered and in which repetitions are allowed. Such associations have the semantics of lists.

The narrative in this document often refers to various parts of the structural specification. These references are mainly intended to be informal, but they can often be interpreted as statements about the instances of the UML classes from the structural specification. When precision is required, such statements are captured using the functional-style syntax, which is defined in Section 3.7 and other relevant parts of this document. In order to avoid confusion, the term "UML class" is used to refer to elements of the structural specification of OWL 2, whereas the term "class" is used to refer to OWL 2 classes (see Section 5.1).

Example:

The sentence "The individual `I` is an instance of the class `C`" can be understood as a statement that `I` is an instance of the UML class **Individual**, `C` is an instance of the UML class **Class**, and there is an instance of the UML class **ClassAssertion** that connects `I` with `C`. This statement can be captured precisely using the structural specification as `ClassAssertion( C I )`.

Objects o_1 and o_2 from the structural specification are *structurally equivalent* if the following conditions hold:

- If o_1 and o_2 are atomic values, such as strings or integers, they are structurally equivalent if they are equal according to the notion of equality of the respective UML type.
- If o_1 and o_2 are unordered associations without repetitions, they are structurally equivalent if each element of o_1 is structurally equivalent to some element of o_2 and vice versa.
- If o_1 and o_2 are ordered associations with repetitions, they are structurally equivalent if they contain the same number of elements and each element of o_1 is structurally equivalent to the element of o_2 with the same index.
- If o_1 and o_2 are instances of UML classes from the structural specification, they are structurally equivalent if
  - both o_1 and o_2 are instances of the same UML class, and
  - each association of o_1 is structurally equivalent to the corresponding association of o_2 and vice versa.

The notion of structural equivalence is used throughout this specification to define various conditions on the structure of OWL 2 ontologies. Note that this is a syntactic, rather than a semantic notion — that is, it compares structures, rather than their meaning under a formal semantics.

Example:

The class expression

```
ObjectUnionOf( a:Person a:Animal )
```

is structurally equivalent to the class expression

```
ObjectUnionOf( a:Animal a:Person )
```

because the order of the elements in an unordered association is not important. In contrast, the class expression

```
ObjectUnionOf( a:Person ObjectComplementOf( a:Person ) )
```

is not structurally equivalent to *owl:Thing* even though the two expressions are semantically equivalent.

Sets written in one of the exchange syntaxes (e.g., XML or RDF/XML) are not necessarily expected to be duplicate free. Duplicates *should* be eliminated when ontology documents written in such syntaxes are converted into instances of the UML classes of the structural specification.

Example:

An ontology written in functional-style syntax can contain the following class expression:

```
ObjectUnionOf( a:Person a:Animal a:Animal )
```

During parsing, this expression should be "flattened" to the following expression:

```
ObjectUnionOf( a:Person a:Animal )
```

2.2 BNF Notation

Grammars in this document are written using the BNF notation, summarized in Table 1.

Table 1. The BNF Notation

Construct	Syntax	Example
terminal symbols	enclosed in single quotes	`'PropertyRange'`
a set of terminal symbols described in English	italic	*a finite sequence of characters matching the PNAME_LN production of [SPARQL]*
nonterminal symbols	boldface	**ClassExpression**
zero or more	curly braces	{ **ClassExpression** }
zero or one	square brackets	[**ClassExpression**]
alternative	vertical bar	**Assertion** | **Declaration**

The grammar presented in this document uses the following two "special" terminal symbols, which affect the process of transforming an input sequence of characters into a sequence of regular (i.e., not "special") terminal symbols:

- *whitespace* is a nonempty sequence of space (U+20), horizontal tab (U+9), line feed (U+A), or carriage return (U+D) characters, and
- a *comment* is a sequence of characters that starts with the # (U+23) character and does not contain the line feed (U+A) or carriage return (U+D) characters.

The following characters are called *delimiters*:

- = (U+3D)
- ((U+28)
-) (U+29)
- < (U+3C)
- > (U+3E)
- @ (U+40)
- ^ (U+5E)

Given an input sequence of characters, an OWL 2 implementation *must* exhibit the same observable behavior as if it applied the BNF grammar rules to the sequence of terminal symbols obtained from the input as follows.

1. For each terminal symbol (including *whitespace* and *comment*) mentioned in this document, a regular expression is created that can recognize the symbol's characters.
2. A pointer p is initialized to point to the beginning of input.
3. All regular expressions are matched to the characters in the input starting from p. Matches are greedy — that is, if several regular expressions match a portion of the input, a regular expression with the longest match wins. The regular expressions corresponding to terminal symbols in this document ensure that there are no ties (i.e., it is not possible for two regular expressions to match a portion of the input of the same length); thus, at most one regular expression can be matched.
4. If there is no match, the input *should* be rejected. Otherwise, if the matched regular expression does not correspond to the *whitespace* or *comment* terminal symbols, the corresponding terminal symbol is emitted to the output. (In other words, the matches of *whitespace* and *comment* are ignored.)
5. Pointer p is moved to the first character after the match.
6. If the terminal symbol matched in step 3 does not end with a delimiter character and p points to a character that is not a delimiter, then the regular expressions for *whitespace* and *comment* are matched to the characters in the input starting from p. If there is no match, the input *should* be rejected; otherwise, p is moved to the first character after the match (and thus the match is discarded).
7. If p does not point past the end of input, the process is repeated from step 3.

Example:

Character sequence

```
" #comment\" " #comment "abc"
```

should be processed as follows. The first match is for the regular expression for the quoted string terminal symbol, producing a string containing a space, characters `#comment"`, and another space. Next, the regular expression for *whitespace* is matched to a single space, and the match is discarded. Finally, the *comment* regular expression is matched to characters `#comment "abc"`, and the match is discarded as well.

In similar vein, character sequence

```
<#comment>
```

should be recognized as a full IRI with value `#comment` (i.e., the occurrence of character # in this example must not be understood as a start of a comment).

Example:

All regular expressions are matched in step 3 greedily, so character sequence

```
SubClassOf:ABC
```

is parsed as abbreviated IRI with value `SubClassOf:ABC`. Furthermore, character sequence

```
pref: ABC
```

should be rejected: characters `pref:` are matched as a prefix name, but then `ABC` cannot be matched by any regular expression corresponding to a terminal symbol.

Example:

Character sequence

```
10abc
```

should be rejected: characters `10` are matched by the regular expression for nonnegative integers; however, since the match does not end with a delimiter and a is not a delimiter either, the match in step 6 fails.

Example:

Character sequences

```
"10" ^^ xsd:integer
"10"^^xsd:integer
```

are both valid should be parsed as a quoted string, terminal symbol `^^`, and an abbreviated IRI. In particular, note that the whitespace surrounding `^^` in the first line is acceptable. In similar vein, character sequences

```
"abc" @en
"abc"@en
```

are both valid and should be parsed as a quoted string and a language tag `en`. In contrast, character sequence

```
"abc"@ en
```

should be rejected: characters `@ en` do not match the regular expression for language tags.

2.3 Integers, Characters, Strings, Language Tags, and Node IDs

Nonnegative integers are defined as usual.

nonNegativeInteger := *a nonempty finite sequence of digits between 0 and 9*

Characters and strings are defined in the same way as in [*RDF:PLAINLITERAL*]. A *character* is an atomic unit of communication. The structure of characters is not further specified in this document, other than to note that each character has a Universal Character Set (UCS) code point [*ISO/IEC 10646*] (or, equivalently, a Unicode code point [*UNICODE*]). Each character *must* match the Char production from XML [*XML*]. Code points are written as U+ followed by the hexadecimal value of the code point. A *string* is a finite sequence of characters, and the *length* of a string is the number of characters in it. Two strings are identical if and only if they contain exactly the same characters in exactly the same sequence. Strings are written by enclosing them in double quotes (U+22) and using a subset of the N-triples escaping mechanism [*RDF Test Cases*] to encode strings containing quotes. Note that the definition below allows a string to span several lines of a document.

quotedString := *a finite sequence of characters in which " (U+22) and \ (U+5C) occur only in pairs of the form \" (U+5C, U+22) and \\ (U+5C, U+5C), enclosed in a pair of " (U+22) characters*

Language tags are used to identify the language in which a string has been written. They are defined in the same way as in [*RDF:PLAINLITERAL*], which follows [*BCP 47*]. Language tags are written by prepending them with the `@` (U+40) character.

languageTag := *@ (U+40) followed a nonempty sequence of characters matching the langtag production from [BCP 47]*

Node IDs are used to identify anonymous individuals (aka *blank nodes* in RDF [*RDF Concepts*]).

nodeID := *a finite sequence of characters matching the BLANK_NODE_LABEL production of [SPARQL]*

2.4 IRIs

Ontologies and their elements are identified using Internationalized Resource Identifiers (IRIs) [*RFC3987*]; thus, OWL 2 extends OWL 1, which uses Uniform Resource Identifiers (URIs). Each IRI *must* be absolute (i.e., not relative). In the structural specification, IRIs are represented by the **IRI** UML class. Two IRIs are structurally equivalent if and only if their string representations are identical.

IRIs can be written as full IRIs by enclosing them in a pair of < (U+3C) and > (U+3E) characters. These characters are not part of the IRI, but are used for quotation purposes to identify an IRI as a full IRI.

Alternatively, IRIs can be abbreviated as in SPARQL [*SPARQL*]. To this end, one can *declare* a *prefix name pn:* — that is, a possibly empty string followed by the **:** (U+3A) character — by associating it with a *prefix IRI PI*; then, an IRI *I* whose string representation consists of *PI* followed by the remaining characters *rc* can be abbreviated as *pn:rc*. By a slight abuse of terminology, a prefix name is often used to refer to the prefix IRI that is associated with the prefix name, and phrases such as "an IRI whose string representation starts with the prefix IRI associated with the prefix name *pn:*" are typically shortened to less verbose phrases such as "an IRI with prefix *pn:*".

If a concrete syntax uses this IRI abbreviation mechanism, it *should* provide a suitable mechanism for declaring prefix names. Furthermore, abbreviated IRIs are not represented in the structural specification of OWL 2, and OWL 2 implementations *must* exhibit the same observable behavior as if all abbreviated IRIs were expanded into full IRIs during parsing. Concrete syntaxes such as the RDF/XML Syntax [*RDF Syntax*] allow IRIs to be abbreviated in relation to the IRI of the document they are contained in. If used, such mechanisms are independent from the above described abbreviation mechanism. The abbreviated IRIs have the syntactic form of qualified names from the XML Namespaces specification [*XML Namespaces*]; therefore, it is common to refer to *PI* as a *namespace* and *rc* as a *local name*. This abbreviation mechanism, however, is independent from XML namespaces and can be understood as a simple macro mechanism that expands prefix names with the associated IRIs.

fullIRI := *an IRI as defined in [RFC3987], enclosed in a pair of < (U+3C) and > (U+3E) characters*
prefixName := *a finite sequence of characters matching the as PNAME_NS production of [SPARQL]*
abbreviatedIRI := *a finite sequence of characters matching the PNAME_LN production of [SPARQL]*
IRI := **fullIRI** | **abbreviatedIRI**

Table 2 declares the prefix names that are commonly used throughout this specification.

Table 2. Declarations of the Standard Prefix Names

Prefix name	Prefix IRI
rdf:	*<http://www.w3.org/1999/02/22-rdf-syntax-ns#>*
rdfs:	*<http://www.w3.org/2000/01/rdf-schema#>*
xsd:	*<http://www.w3.org/2001/XMLSchema#>*
owl:	*<http://www.w3.org/2002/07/owl#>*

IRIs with prefixes *rdf:*, *rdfs:*, *xsd:*, and *owl:* constitute the *reserved vocabulary* of OWL 2. As described in the following sections, the IRIs from the reserved vocabulary that are listed in Table 3 have special treatment in OWL 2.

Table 3. Reserved Vocabulary of OWL 2 with Special Treatment

owl:backwardCompatibleWith	*owl:bottomDataProperty*	*owl:bottomObjectProperty*	*owl:deprecated*	*owl:incompatibleWith*
owl:Nothing	*owl:priorVersion*	*owl:rational*	*owl:real*	*owl:versionInfo*
owl:Thing	*owl:topDataProperty*	*owl:topObjectProperty*	*rdf:langRange*	*rdf:PlainLiteral*
rdf:XMLLiteral	*rdfs:comment*	*rdfs:isDefinedBy*	*rdfs:label*	*rdfs:Literal*
rdfs:seeAlso	*xsd:anyURI*	*xsd:base64Binary*	*xsd:boolean*	*xsd:byte*
xsd:dateTime	*xsd:dateTimeStamp*	*xsd:decimal*	*xsd:double*	*xsd:float*
xsd:hexBinary	*xsd:int*	*xsd:integer*	*xsd:language*	*xsd:length*
xsd:long	*xsd:maxExclusive*	*xsd:maxInclusive*	*xsd:maxLength*	*xsd:minExclusive*
xsd:minInclusive	*xsd:minLength*	*xsd:Name*	*xsd:NCName*	*xsd:negativeInteger*
xsd:NMTOKEN	*xsd:nonNegativeInteger*	*xsd:nonPositiveInteger*	*xsd:normalizedString*	*xsd:pattern*
xsd:positiveInteger	*xsd:short*	*xsd:string*	*xsd:token*	*xsd:unsignedByte*
xsd:unsignedInt	*xsd:unsignedLong*	*xsd:unsignedShort*		

3 Ontologies

An OWL 2 *ontology* is an instance *O* of the **Ontology** UML class from the structural specification of OWL 2 shown in Figure 1 that satisfies certain conditions given below. The main component of an OWL 2 ontology is its set of axioms, the structure of which is described in more detail in Section 9. Because the association between an ontology and its axioms is a set, an ontology cannot contain two axioms that are structurally equivalent. Apart from axioms, ontologies can also contain ontology annotations (as described in more detail in Section 3.5), and they can also import other ontologies (as described in Section 3.4).

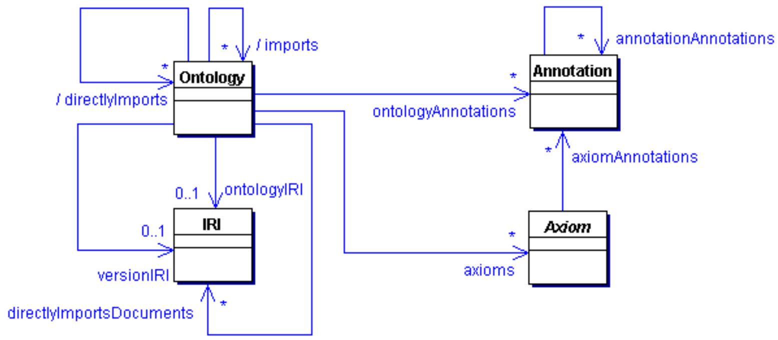

Figure 1. The Structure of OWL 2 Ontologies

The following list summarizes all the conditions that *O* is required to satisfy to be an OWL 2 ontology.

- *O must* satisfy the restrictions on the presence of the ontology IRI and version IRI from Section 3.1.
- Each **DataIntersectionOf** and **DataUnionOf** in *O must* satisfy the restrictions from Section 7.1 and Section 7.2, respectively.
- Each **DataSomeValuesFrom** and **DataAllValuesFrom** class expression in *O must* satisfy the restrictions from Section 8.4.1 and Section 8.4.2, respectively.
- Each **DataPropertyRange** axiom in *O must* satisfy the restriction from Section 9.3.5.
- Each **DatatypeDefinition** axiom in *O must* satisfy the restrictions from Section 9.4.
- Each **HasKey** axiom in *O must* satisfy the restriction from Section 9.5.
- Each *O*' directly imported into *O must* satisfy all of these restrictions as well.

The following list summarizes all the conditions that an OWL 2 ontology *O* is required to satisfy to be an OWL 2 DL ontology.

- The ontology IRI and the version IRI (if present) of *O must* satisfy the restrictions on usage of the reserved vocabulary from Section 3.1.
- Each datatype and each literal in *O must* satisfy the restrictions from Section 5.2 and Section 5.7, respectively.
- Each entity in *O must* have an IRI satisfying the restrictions on the usage of the reserved vocabulary from Sections 5.1–5.6.
- *O must* satisfy the typing constraints from Section 5.8.1.
- Each **DatatypeRestriction** in *O must* satisfy the restriction on the usage of constraining facets from Section 7.5, respectively.
- *O must* satisfy the global restriction from Section 11.
- Each *O*' directly imported into *O must* satisfy all of these restrictions as well.

An instance *O* of the **Ontology** UML class *may* have consistent declarations as specified in Section 5.8.2; however, this is not strictly necessary to make *O* an OWL 2 ontology.

3.1 Ontology IRI and Version IRI

Each ontology *may* have an *ontology IRI*, which is used to identify an ontology. If an ontology has an ontology IRI, the ontology *may* additionally have a *version IRI*, which is used to identify the version of the ontology. The version IRI *may* be, but need not be, equal to the ontology IRI. An ontology without an ontology IRI *must not* contain a version IRI.

IRIs from the reserved vocabulary *must not* be used as an ontology IRI or a version IRI of an OWL 2 DL ontology.

The following list provides conventions for choosing ontology IRIs and version IRIs in OWL 2 ontologies. This specification provides no mechanism for enforcing these constraints across the entire Web; however, OWL 2 tools *should* use them to detect problems in ontologies they process.

- If an ontology has an ontology IRI but no version IRI, then a different ontology with the same ontology IRI but no version IRI *should not* exist.
- If an ontology has both an ontology IRI and a version IRI, then a different ontology with the same ontology IRI and the same version IRI *should not* exist.
- All other combinations of the ontology IRI and version IRI are not required to be unique. Thus, two different ontologies *may* have no ontology IRI and no version IRI; similarly, an ontology containing only an ontology IRI *may* coexist with another ontology with the same ontology IRI and some other version IRI.

The ontology IRI and the version IRI together identify a particular version from an *ontology series* — the set of all the versions of a particular ontology identified using a common ontology IRI. In each ontology series, exactly one ontology version is regarded as the *current* one. Structurally, a version of a particular ontology is an instance of the **Ontology** UML class from the structural specification. Ontology series are not represented explicitly in the structural specification of OWL 2: they exist only as a side effect of the naming conventions described in this and the following sections.

3.2 Ontology Documents

An OWL 2 ontology is an abstract notion defined in terms of the structural specification. Each ontology is associated with an *ontology document*, which physically contains the ontology stored in a particular way. The name "ontology document" reflects the expectation that a large number of ontologies will be stored in physical text documents written in one of the syntaxes of OWL 2. OWL 2 tools, however, are free to devise other types of ontology documents — that is, to introduce other ways of physically storing ontologies.

Ontology documents are not represented in the structural specification of OWL 2, and the specification of OWL 2 makes only the following two assumptions about their nature:

- Each ontology document can be accessed via an IRI by means of an appropriate protocol.
- Each ontology document can be converted in some well-defined way into an ontology (i.e., into an instance of the **Ontology** UML class from the structural specification).

Example:

An OWL 2 tool might publish an ontology as a text document written in the functional-style syntax (see Section 3.7) and accessible via the IRI *<http://www.example.com/ontology>*. An OWL 2 tool could also devise a scheme for storing OWL 2 ontologies in a relational database. In such a case, each subset of the database representing the information about one ontology corresponds to one ontology document. To provide a mechanism for accessing these ontology documents, the OWL 2 tool should identify different database subsets with distinct IRIs.

The ontology document of an ontology *O* *should* be accessible via the IRIs determined by the following rules:

- If *O* does not contain an ontology IRI (and, consequently, it does not contain a version IRI either), then the ontology document of *O* *may* be accessible via any IRI.
- If *O* contains an ontology IRI *OI* but no version IRI, then the ontology document of *O* *should* be accessible via the IRI *OI*.
- If *O* contains an ontology IRI *OI* and a version IRI *VI*, then the ontology document of *O* *should* be accessible via the IRI *VI*; furthermore, if *O* is the current version of the ontology series with the IRI *OI*, then the ontology document of *O* *should* also be accessible via the IRI *OI*.

Thus, the document containing the current version of an ontology series with some IRI *OI* *should* be accessible via *OI*. To access a particular version of *OI*, one needs to know that version's version IRI *VI*; the ontology document of the version *should* then be accessible via *VI*.

Example:

An ontology document of an ontology that contains an ontology IRI *<http://www.example.com/my>* but no version IRI should be accessible via the IRI *<http://www.example.com/my>*. In contrast, an ontology document of an ontology that contains an ontology IRI *<http://www.example.com/my>* and a version IRI *<http://www.example.com/my/2.0>* should be accessible via the IRI *<http://www.example.com/my/2.0>*. In both cases, the ontology document should be accessible via the respective IRIs using the HTTP protocol.

OWL 2 tools will often need to implement functionality such as caching or off-line processing, where ontology documents may be stored at addresses different from the ones dictated by their ontology IRIs and version IRIs. OWL 2 tools *may* implement a *redirection* mechanism: when a tool is used to access an ontology document at IRI *I*, the tool *may* redirect *I* to a different IRI *DI* and access the ontology document via *DI* instead. The result of accessing the ontology document via *DI* *must* be the same as if the ontology were accessed via *I*. Furthermore, once the ontology document is converted into an ontology, the ontology *should* satisfy the three conditions from the beginning of this section in the same way as if it the ontology document were accessed via *I*. No particular redirection mechanism is specified — this is assumed to be implementation dependent.

Example:

To enable off-line processing, an ontology document that — according to the above rules — should be accessible via *<http://www.example.com/my>* might be stored in a file accessible via *<file:///usr/local/ontologies/example.owl>*. To access this ontology document, an OWL 2 tool might redirect the IRI *<http://www.example.com/my>* and actually access the ontology document via *<file:///usr/local/ontologies/example.owl>*. The ontology obtained after accessing the ontology document should satisfy the usual accessibility constraints: if the ontology contains only the ontology IRI, then the ontology IRI should be equal to *<http://www.example.com/my>*, and if the ontology contains both the ontology IRI and the version IRI, then one of them should be equal to *<http://www.example.com/my>*.

3.3 Versioning of OWL 2 Ontologies

The conventions from Section 3.2 provide a simple mechanism for versioning OWL 2 ontologies. An ontology series is identified using an ontology IRI, and each version in the series is assigned a different version IRI. The ontology document of the ontology representing the current version of the series *should* be accessible via the ontology IRI and, if present, via its version IRI as well; the ontology documents of the previous versions *should* be accessible solely via their respective version IRIs. When a new version *O* in the ontology series is created, the ontology document of *O* *should* replace the one accessible via the ontology IRI (and it *should* also be accessible via its version IRI).

Example:

The ontology document containing the current version of an ontology series might be accessible via the IRI *<http://www.example.com/my>*, as well as via the version-specific IRI *<http://www.example.com/my/2.0>*. When a new version is created, the ontology document of the previous version should remain accessible via *<http://www.example.com/my/2.0>*; the ontology document of the new version, called, say, *<http://www.example.com/my/3.0>*, should be made accessible via both *<http://www.example.com/my>* and *<http://www.example.com/my/3.0>*.

3.4 Imports

An OWL 2 ontology can import other ontologies in order to gain access to their entities, expressions, and axioms, thus providing the basic facility for ontology modularization.

Example:

Assume that one wants to describe research projects about diseases. Managing information about the projects and the diseases in the same ontology might be cumbersome. Therefore, one might create a separate ontology *O* about diseases and a separate ontology *O*' about projects. The ontology *O*' would import *O* in order to gain access to the classes representing diseases; this allows one to use the diseases from *O* when writing the axioms of *O*'.

From a physical point of view, an ontology contains a set of IRIs, shown in Figure 1 as the **directlyImportsDocuments** association; these IRIs identify the ontology documents of the directly imported ontologies as specified in Section 3.2. The logical *directly imports* relation between ontologies, shown in Figure 1 as the **directlyImports** association, is obtained by accessing the directly imported ontology documents and converting them into OWL 2 ontologies. The logical *imports* relation between ontologies, shown in Figure 1 as the **imports** association, is the transitive closure of directly imports. In Figure 1, associations **directlyImports** and **imports** are shown as derived associations, since their values are derived from the value of the **directlyImportsDocuments** association. Ontology documents usually store the **directlyImportsDocuments** association. In contrast, the **directlyImports** and **imports** associations are typically not stored in ontology documents, but are determined during parsing as specified in Section 3.6.

Example:

The following ontology document contains an ontology that directly imports an ontology contained in the ontology document accessible via the IRI *<http://www.example.com/my/2.0>*.

```
Ontology( <http://www.example.com/importing-ontology>
    Import( <http://www.example.com/my/2.0> )

...
)
```

The IRIs identifying the ontology documents of the directly imported ontologies can be redirected as described in Section 3.2. For example, in order to access the above mentioned ontology document from a local cache, the IRI *<http://www.example.com/my/2.0>* might be redirected to *<file:///usr/local/ontologies/imported.v20.owl>*. Note that this can be done without changing the ontology document of the importing ontology.

The *import closure* of an ontology *O* is a set containing *O* and all the ontologies that *O* imports. The import closure of *O* *should not* contain ontologies O_1 and O_2 such that

- O_1 and O_2 are different ontology versions from the same ontology series, or
- O_1 contains an ontology annotation *owl:incompatibleWith* with the value equal to either the ontology IRI or the version IRI of O_2.

The *axiom closure* of an ontology *O* is the smallest set that contains all the axioms from each ontology *O'* in the import closure of *O* with all anonymous individuals *standardized apart* — that is, the anonymous individuals from different ontologies in the import closure of *O* are treated as being different; see Section 5.6.2 for further details.

3.5 Ontology Annotations

An OWL 2 ontology contains a set of annotations. These can be used to associate information with an ontology — for example the ontology creator's name. As discussed in more detail in Section 10, each annotation consists of an annotation property and an annotation value, and the latter can be a literal, an IRI, or an anonymous individual.

ontologyAnnotations := { **Annotation** }

OWL 2 provides several built-in annotation properties for ontology annotations. The usage of these annotation properties on entities other than ontologies is discouraged.

- The *owl:priorVersion* annotation property specifies the IRI of a prior version of the containing ontology.
- The *owl:backwardCompatibleWith* annotation property specifies the IRI of a prior version of the containing ontology that is compatible with the current version of the containing ontology.
- The *owl:incompatibleWith* annotation property specifies the IRI of a prior version of the containing ontology that is incompatible with the current version of the containing ontology.

3.6 Canonical Parsing of OWL 2 Ontologies

Many OWL 2 tools need to support *ontology parsing* — the process of converting an ontology document written in a particular syntax into an OWL 2 ontology. Depending on the syntax used, the ontology parser may need to know which IRIs are used in the ontology as entities of which type. This typing information is extracted from declarations — axioms that associate IRIs with entity types. Please refer to Section 5.8 for more information about declarations.

Example:

An ontology parser for the ontology documents written in the RDF syntax might encounter the following triples:

```
a:Father rdfs:subClassOf _:x .
_:x owl:someValuesFrom a:Child .
_:x owl:onProperty a:parentOf.
```

From this axiom alone, it is not clear whether *a:parentOf* is an object or a data property, and whether *a:Child* is a class or a datatype. In order to disambiguate the types of these IRIs, the parser needs to look at the declarations in the ontology document being parsed, as well as those in the directly or indirectly imported ontology documents.

In OWL 2 there is no requirement for a declaration of an entity to physically precede the entity's usage in ontology documents; furthermore, declarations for entities can be placed in imported ontology documents and imports are allowed to be cyclic. In order to precisely define the result of ontology parsing, this specification defines the notion of *canonical parsing*. An OWL 2 parser *may* implement parsing in any way it chooses, as long as it produces a result that is structurally equivalent to the result of canonical parsing.

An OWL 2 ontology corresponding to an ontology document D_{GI} accessible via a given IRI *GI* can be obtained using the following *canonical parsing* process. All steps of this process *must* be successfully completed.

CP 1 Make *AllDoc* and *Processed* equal to the empty set, and make *ToProcess* equal to the set containing only the IRI *GI*.

CP 2 While *ToProcess* is not empty, remove an arbitrary IRI *I* from it and, if *I* is not contained in *Processed*, perform the following steps:

CP 2.1 Retrieve the ontology document D_I from *I* as specified in Section 3.2.

CP 2.2 Using the rules of the relevant syntax, analyze D_I and compute the set $Decl(D_I)$ of declarations explicitly present in D_I and the set $Imp(D_I)$ of IRIs of ontology documents directly imported in D_I.

CP 2.3 Add D_I to *AllDoc*, add *I* to *Processed*, and add each IRI from $Imp(D_I)$ to *ToProcess*.

CP 3 For each ontology document *D* in *AllDoc*, perform the following steps:

CP 3.1 Compute the set *AllDecl(D)* as the union of the set *Decl(D)*, the sets *Decl(D')* for each ontology document *D*' that is (directly or indirectly) imported into *D*, and the set of all declarations listed in Table 5. For an OWL 2 DL ontology, the set *AllDecl(D) must* satisfy the typing constraints from Section 5.8.1.

CP 3.2 Create an instance O_D of the **Ontology** UML class from the structural specification.

CP 3.3 Using the rules of the relevant syntax, analyze *D* and populate O_D by instantiating appropriate classes from the structural specification. Use the declarations in *AllDecl(D)* to disambiguate IRIs if needed; it *must* be possible to disambiguate all IRIs.

CP 4 For each pair of ontology documents *DS* and *DT* in *AllDoc* such that the latter is directly imported into the former, add O_{DT} to the **directlyImports** association of O_{DS}.

CP 5 For each ontology document *D* in *AllDoc*, set the **imports** association of O_D to the transitive closure of the **directlyImports** association of O_D.

CP 6 For each ontology document *D* in *AllDoc*, ensure that O_D is an OWL 2 ontology — that is, O_D *must* satisfy all the restrictions listed in Section 3.

It is important to understand that canonical parsing merely defines the result of the parsing process, and that an implementation of OWL 2 *may* optimize this process in numerous ways. In order to enable efficient parsing, OWL 2 implementations are encouraged to write ontologies into documents by placing all IRI declarations before the axioms that use these IRIs; however, this is not required for conformance.

3.7 Functional-Style Syntax

A *functional-style syntax ontology document* is a sequence of Unicode characters [*UNICODE*] accessible via some IRI by means of the standard protocols such that its text matches the **ontologyDocument** production of the grammar defined in this specification document, and it can be converted into an ontology by means of the canonical parsing process described in Section 3.6 and other parts of this specification document. A functional-style syntax ontology document *should* use the UTF-8 encoding [*RFC 3629*].

```
ontologyDocument := { prefixDeclaration } Ontology
prefixDeclaration := 'Prefix' '(' prefixName '=' fullIRI ')'
Ontology :=
    'Ontology' '(' [ ontologyIRI [ versionIRI ] ]
       directlyImportsDocuments
       ontologyAnnotations
       axioms
    ')'
ontologyIRI := IRI
versionIRI := IRI
directlyImportsDocuments := { 'Import' '(' IRI ')' }
axioms := { Axiom }
```

Each part of the ontology document matching the **prefixDeclaration** production declares a prefix name and associates it with a prefix IRI. An ontology document *must* contain at most one such declaration per prefix name, and it *must not* declare a prefix name listed in Table 2. Prefix declarations are used during parsing to expand abbreviated IRIs in the ontology document — that is, parts of the ontology document matching the **abbreviatedIRI** production — into full IRIs. This is done as follows:

- The abbreviated IRI is split into a prefix name *pn:* — the part up to and including the **:** (U+3A) character — and the remaining part *rp* following the **:** (U+3A) character.
- If *pn:* is not one of the standard prefix names listed in Table 2, then the prefix declarations of the ontology document being parsed *must* contain a declaration for *pn:* associating it with a prefix IRI *PI*.
- The resulting full IRI is obtained by concatenating the string representation of *PI* with *rp*. The resulting IRI *must* be a valid IRI.

Example:

The following is a functional-style syntax ontology document containing an ontology with the ontology IRI *<http://www.example.com/ontology1>*. The IRI *<http://www.example.com/ontology1#>* is associated with the prefix name *:* (that is, the prefix name consisting only of a colon character); this prefix is often called "empty" or "default". This ontology imports an ontology whose ontology document should be accessed via *<http://www.example.com/ontology2>*, and it contains an ontology annotation providing a label for the ontology and a single subclass axiom. The abbreviated IRI *:Child* is expanded into the full IRI *<http://www.example.com/ontology1#Child>* during parsing. The prefix name *owl:* occurs in Table 2 and therefore does not need to be explicitly declared in the ontology document.

```
Prefix(:=<http://www.example.com/ontology1#>)
Ontology( <http://www.example.com/ontology1>
    Import( <http://www.example.com/ontology2> )
    Annotation( rdfs:label "An example" )

    SubClassOf( :Child owl:Thing )
)
```

4 Datatype Maps

OWL 2 ontologies can refer to data values such as strings or integers. Each kind of such values is called a *datatype*. Datatypes can be used in OWL 2 ontologies as described in Section 5.2. Each datatype is identified by an IRI and is defined by the following components:

- The *value space* is the set of values of the datatype. Elements of the value space are called *data values*.
- The *lexical space* is a set of strings that can be used to refer to data values. Each member of the lexical space is called a *lexical form*, and it is mapped to a particular data value.
- The *facet space* is a set of pairs of the form (*F* , *v*) where *F* is an IRI called a *constraining facet*, and *v* is an arbitrary data value called the *constraining value*. Each such pair is mapped to a subset of the value space of the datatype.

A set of datatypes supported by a reasoner is called a *datatype map*. This is not a syntactic construct — that is, it is not used directly to construct OWL 2 ontologies in a way that, say, classes and datatypes are. Because of that, a datatype map is not represented in the structural specification of OWL 2.

The rest of this section defines a particular datatype map called the *OWL 2 datatype map*, which lists the datatypes that can be used in OWL 2 ontologies. Most datatypes are taken from the set of XML Schema Datatypes [XML Schema Datatypes], the RDF specification [RDF Concepts], or the specification for plain literals [RDF:PLAINLITERAL]. The normative definitions of these datatypes are provided by the respective specifications, and this document merely provides guidance on how to interpret these definitions properly in the context of OWL 2. For all these datatypes, this section lists the *normative constraining facets* that OWL 2 implementations *must* support. This section also contains the complete normative definitions of the datatypes *owl:real* and *owl:rational*, as these datatypes have not been taken from other specifications.

4.1 Real Numbers, Decimal Numbers, and Integers

The OWL 2 datatype map provides the following datatypes for the representation of real numbers, decimal numbers, and integers:

- *owl:real*
- *owl:rational*
- *xsd:decimal*
- *xsd:integer*
- *xsd:nonNegativeInteger*
- *xsd:nonPositiveInteger*
- *xsd:positiveInteger*
- *xsd:negativeInteger*
- *xsd:long*
- *xsd:int*
- *xsd:short*
- *xsd:byte*
- *xsd:unsignedLong*
- *xsd:unsignedInt*
- *xsd:unsignedShort*
- *xsd:unsignedByte*

For each datatype from the above list that is identified by an IRI with the *xsd:* prefix, the definitions of the value space, the lexical space, and the facet space are provided by XML Schema [XML Schema Datatypes]; furthermore, the normative constraining facets for the datatype are *xsd:minInclusive*, *xsd:maxInclusive*, *xsd:minExclusive*, and *xsd:maxExclusive*. An OWL 2 implementation *may* support all lexical forms of these datatypes; however, it *must* support at least the lexical forms listed in Section 5.4 of XML Schema Datatypes [XML Schema Datatypes], which can be mapped to the primitive values commonly found in modern implementation platforms.

The datatypes *owl:real* and *owl:rational* are defined as follows.

Value Spaces.

- The value space of *owl:real* is the set of all real numbers.
- The value space of *owl:rational* is the set of all rational numbers. It is a subset of the value space of *owl:real*, and it contains the value space of *xsd:decimal* (and thus of all *xsd:* numeric datatypes listed above as well).

Lexical Spaces.

- The *owl:real* datatype does not directly provide any lexical forms.
- The *owl:rational* datatype supports lexical forms defined by the following grammar (whitespace within the grammar *must* be ignored and *must not* be included in the lexical forms of *owl:rational*, and single quotes are used to introduce terminal symbols):

  ```
  numerator '/' denominator
  ```

  Here, `numerator` is an integer with the syntax as specified for the *xsd:integer* datatype, and `denominator` is a positive, nonzero integer with the syntax as specified for the *xsd:integer* datatype, not containing the plus sign. Each such lexical form of *owl:rational* is mapped to the rational number obtained by dividing the value of `numerator` by the value of `denominator`. An OWL 2 implementation *may* support all such lexical forms; however, it *must* support at least the lexical forms where the numerator and the denominator are in the value space of *xsd:long*.

Facet Spaces. The facet spaces of *owl:real* and *owl:rational* are defined in Table 4.

Table 4. The Facet Spaces of *owl:real* and *owl:rational*

Each pair of the form...	...is mapped to...
(*xsd:minInclusive* , v) where v is from the value space of *owl:real*	the set of all numbers x from the value space of *DT* such that $x = v$ or $x > v$
(*xsd:maxInclusive* , v) where v is from the value space of *owl:real*	the set of all numbers x from the value space of *DT* such that $x = v$ or $x < v$
(*xsd:minExclusive* , v) where v is from the value space of *owl:real*	the set of all numbers x from the value space of *DT* such that $x > v$
(*xsd:maxExclusive* , v) where v is from the value space of *owl:real*	the set of all numbers x from the value space of *DT* such that $x < v$
Note. *DT* is either *owl:real* or *owl:rational*.	

4.2 Floating-Point Numbers

The OWL 2 datatype map supports the following datatypes for the representation of floating-point numbers:

- *xsd:double*
- *xsd:float*

As specified in XML Schema [XML Schema Datatypes], the value spaces of *xsd:double*, *xsd:float*, and *xsd:decimal* are pairwise disjoint. In accordance with this principle, the value space of *owl:real* is defined as being disjoint with the value spaces of *xsd:double* and *xsd:float* as well. The normative constraining facets for these datatypes are *xsd:minInclusive*, *xsd:maxInclusive*, *xsd:minExclusive*, and *xsd:maxExclusive*.

Example:

Although floating-point values are numbers, they are not contained in the value space of *owl:real*. Thus, the value spaces of *xsd:double* and *xsd:float* can be understood as containing "fresh copies" of the appropriate subsets of the value space of *owl:real*. To understand how this impacts the consequences of OWL 2 ontologies, consider the following example.

`DataPropertyRange( a:hasAge xsd:integer )`	The range of the *a:hasAge* property is *xsd:integer*.
`DataPropertyAssertion( a:hasAge a:Meg "17"^^xsd:double )`	Meg is seventeen years old.

The first axiom states that all values of the *a:hasAge* property must be in the value space of *xsd:integer*, but the second axiom provides a value for *a:hasAge* that is equal to the floating-point number 17. Since floating-point numbers are not contained in the value space of *xsd:integer*, the mentioned ontology is inconsistent.

Example:

According to XML Schema, the value spaces of *xsd:double* and *xsd:float* contain positive and negative zeros. These two objects are equal, but not identical. To understand this distinction, consider the following example ontology:

`DataPropertyAssertion( a:numberOfChildren a:Meg "+0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *+0*.
`DataPropertyAssertion( a:numberOfChildren a:Meg "-0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *-0*.
`FunctionalDataProperty( a:numberOfChildren )`	An individual can have at most one value for *a:numberOfChildren*.

The last axiom states that no individual should have more than one distinct value for *a:numberOfChildren*. Since positive and negative zero are not identical, the first two axioms violate the restriction of the last axiom, which makes the ontology inconsistent. In other words, equality of values from the value space of *xsd:double* and *xsd:float* has no effect on the semantics of cardinality restrictions of OWL 2; in fact, equality is used only in the definition of facets.

Example:

According to XML Schema, the semantics of facets is defined with respect to equality, and positive and negative zeros are equal. Therefore, the subset of the value space of *xsd:double* between *-1.0* and *1.0* contains both *+0* and *-0*.

4.3 Strings

The OWL 2 datatype map provides the *rdf:PlainLiteral* datatype for the representation of strings in a particular language. The definitions of the value space, the lexical space, the facet space, and the necessary mappings are given in [*RDF:PLAINLITERAL*]. The normative constraining facets for *rdf:PlainLiteral* are *xsd:length*, *xsd:minLength*, *xsd:maxLength*, *xsd:pattern*, and *rdf:langRange*; furthermore, only *basic language ranges* [*BCP 47*] are supported in the *rdf:langRange* constraining facet.

In addition, OWL 2 supports the following datatypes defined in XML Schema [*XML Schema Datatypes*]:

- *xsd:string*
- *xsd:normalizedString*
- *xsd:token*
- *xsd:language*
- *xsd:Name*
- *xsd:NCName*
- *xsd:NMTOKEN*

As explained in [*RDF:PLAINLITERAL*], the value spaces of all of these datatypes are contained in the value space of *rdf:PlainLiteral*. Furthermore, for each datatype from the above list, the normative constraining facets are *xsd:length*, *xsd:minLength*, *xsd:maxLength*, and *xsd:pattern*.

4.4 Boolean Values

The OWL 2 datatype map provides the *xsd:boolean* XML Schema datatype [*XML Schema Datatypes*] for the representation of Boolean values. No constraining facet is normative for this datatype.

4.5 Binary Data

The OWL 2 datatype map provides the following XML Schema datatypes [*XML Schema Datatypes*] for the representation of binary data:

- *xsd:hexBinary*
- *xsd:base64Binary*

As specified in XML Schema [*XML Schema Datatypes*], the value spaces of these two datatypes are disjoint. For each datatype from the above list, the normative constraining facets are *xsd:minLength*, *xsd:maxLength*, and *xsd:length*.

Example:

According to XML Schema, the value spaces of *xsd:hexBinary* and *xsd:base64Binary* are isomorphic copies of the set of all finite sequences of *octets* — integers between 0 and 255, inclusive. To understand the effect that the disjointness requirement has on the semantics of OWL 2, consider the following example ontology:

`DataPropertyRange( a:personID xsd:base64Binary )`	The range of the *a:personID* property is *xsd:base64Binary*.
`DataPropertyAssertion( a:personID a:Meg "0203"^^xsd:hexBinary )`	The ID of Meg is the octet sequence consisting of the octets 2 and 3.

The first axiom states that all values of the *a:personID* property must be in the value space of *xsd:base64Binary*, but the second axiom provides a value for *a:personID* that is in the value space of *xsd:hexBinary*. Since the value spaces of *xsd:hexBinary* and *xsd:base64Binary* are disjoint, the above ontology is inconsistent.

4.6 IRIs

The OWL 2 datatype map provides the *xsd:anyURI* XML Schema datatype [*XML Schema Datatypes*] for the representation of IRIs. As specified in XML Schema [*XML Schema Datatypes*], the value spaces of *xsd:anyURI* and *xsd:string* are disjoint. The normative constraining facets are *xsd:minLength*, *xsd:maxLength*, *xsd:length*, and *xsd:pattern*.

Example:

According to XML Schema, the value space of *xsd:anyURI* is the set of all IRIs. Although each IRI has a string representation, IRIs are not strings. The value space of *xsd:anyURI* can therefore be seen as an "isomorphic copy" of a subset of the value space of *xsd:string*.

The lexical forms of *xsd:anyURI* include relative IRIs. If an OWL 2 syntax employs rules for the resolution of relative IRIs (e.g., the OWL 2 XML Syntax [*OWL 2 XML Serialization*] uses *xml:base* for that purpose), such rules do not apply to *xsd:anyURI* lexical forms that represent relative IRIs; that is, the lexical forms representing relative IRIs *must* be parsed as they are.

4.7 Time Instants

The OWL 2 datatype map provides the following XML Schema datatypes [*XML Schema Datatypes*] for the representation of time instants with and without time zone offsets:

- *xsd:dateTime*
- *xsd:dateTimeStamp*

For each datatype from the above list, the normative constraining facets are *xsd:minInclusive*, *xsd:maxInclusive*, *xsd:minExclusive*, and *xsd:maxExclusive*. An OWL 2 implementation *may* support all lexical forms of these datatypes; however, it *must* support at least the lexical forms listed in Section 5.4 of XML Schema Datatypes [*XML Schema Datatypes*].

Example:

According to XML Schema, two *xsd:dateTime* values representing the same time instant but with different time zone offsets are equal, but not identical. The consequences of this definition are demonstrated by the following example ontology:

`FunctionalDataProperty( a:birthDate )`	Each object can have at most one birth date.
`DataPropertyAssertion( a:birthDate a:Peter "1956-06-25T04:00:00-05:00"^^xsd:dateTime )`	Peter was born on June 25th, 1956, at 4am EST.
`DataPropertyAssertion( a:birthDate a:Peter "1956-06-25T10:00:00+01:00"^^xsd:dateTime )`	Peter was born on June 25th, 1956, at 10am CET.

June 25th, 1956, 4am EST and June 25th, 1956, 10am CET denote the same time instants, but have different time zone offsets. Consequently, the two *xsd:dateTime* literals are mapped to two equal, but nonidentical data values. Consequently, *a:Peter* is connected by the property *a:birthDate* to two distinct data values, which violates the functionality requirement on *a:birthDate* and makes the ontology inconsistent.

Example:

The semantics of constraining facets on *xsd:dateTime* is defined with respect to equality and ordering on time instants. For example, the following datatype restriction contains all time instants that are larger than or equal to the time instant corresponding to the lexical form `"1956-01-01T04:00:00-05:00"`.

```
DatatypeRestriction( xsd:dateTime xsd:minInclusive "1956-01-01T04:00:00-05:00"^^xsd:dateTime )
```

According to XML Schema datatypes [*XML Schema Datatypes*], time instants are compared with respect to their *timeOnTimeline* value, which roughly corresponds to the number of seconds elapsed from the origin of the proleptic Gregorian calendar. Thus, the above data range contains the time instants corresponding to the lexical forms `"1956-06-25T04:00:00-05:00"` and `"1956-06-25T10:00:00+01:00"` despite the fact that the time zone offset of the latter does not match the one used in the datatype restriction.

A time instant might not contain a time zone offset, in which case comparisons are slightly more involved. Let T_1 and T_2 be time instants with and without time zone offsets, respectively. Then, T_1 is not equal to T_2, and comparisons are defined as follows:

- T_1 is smaller than T_2 if the *timeOnTimeline* value of T_1 is smaller than the *timeOnTimeline* value of T_2^{low}, where T_2^{low} is the time instant equal to T_2 but with the time zone offset equal to `"+14:00"`.
- T_1 is greater than T_2 if the *timeOnTimeline* value of T_1 is greater than the *timeOnTimeline* value of T_2^{high}, where T_2^{high} is the time instant equal to T_2 but with the time zone offset equal to `"-14:00"`.

Thus, for T_1 to be smaller than T_2, the *timeOnTimeline* value of T_1 should be smaller than the *timeOnTimeline* value of T_2 even if we substitute the largest positive time zone offset in T_2; the definition of "greater than" is analogous. Note that, for certain T_1 and T_2, it is possible that neither condition holds, in which case T_1 and T_2 are incomparable.

According to this definition, the datatype restriction mentioned earlier in this example contains the time instant corresponding to the lexical form `"1956-01-01T10:00:00Z"`, but not the one corresponding to `"1956-01-01T10:00:00"`; the latter is the case because the time instant corresponding to `"1956-01-01T10:00:00+14:00"` is not greater than or equal to the one corresponding to `"1956-01-01T04:00:00-05:00"`.

4.8 XML Literals

The OWL 2 datatype map provides the *rdf:XMLLiteral* datatype for the representation of XML content in OWL 2 ontologies. The datatype is defined in Section 5.1 of the RDF specification [*RDF Concepts*]. It has no normative constraining facets.

5 Entities, Literals, and Anonymous Individuals

Entities are the fundamental building blocks of OWL 2 ontologies, and they define the vocabulary — the named terms — of an ontology. In logic, the set of entities is usually said to constitute the *signature* of an ontology. Apart from entities, OWL 2 ontologies typically also contain literals, such as strings or integers.

The structure of entities and literals in OWL 2 is shown in Figure 2. Classes, datatypes, object properties, data properties, annotation properties, and named individuals are entities, and they are all uniquely identified by an IRI. Classes represent sets of individuals; datatypes are sets of literals such as strings or integers; object and data properties can be used to represent relationships in the domain; annotation properties can be used to associate nonlogical information with ontologies, axioms, and entities; and named individuals can be used to represent actual objects from the domain. Apart from named individuals, OWL 2 also provides for anonymous individuals — that is, individuals that are analogous to blank nodes in RDF [*RDF Concepts*] and that are accessible only from within the ontology they are used in. Finally, OWL 2 provides for literals, which consist of a string called a *lexical form* and a datatype specifying how to interpret this string.

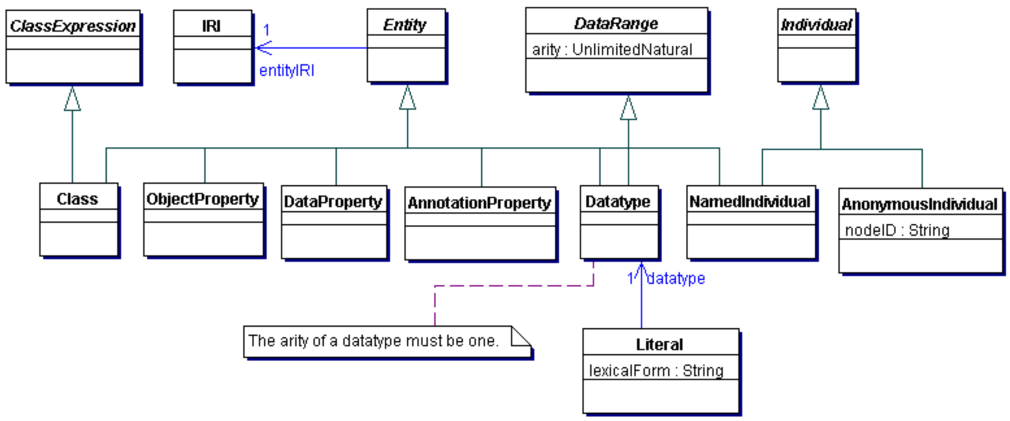

Figure 2. Entities, Literals, and Anonymous Individuals in OWL 2

5.1 Classes

Classes can be understood as sets of individuals.

Class := **IRI**

The classes with the IRIs *owl:Thing* and *owl:Nothing* are available in OWL 2 as built-in classes with a predefined semantics:

- The class with IRI *owl:Thing* represents the set of all individuals. (In the DL literature this is often called the top concept.)
- The class with IRI *owl:Nothing* represents the empty set. (In the DL literature this is often called the bottom concept.)

IRIs from the reserved vocabulary other than *owl:Thing* and *owl:Nothing must not* be used to identify classes in an OWL 2 DL ontology.

Example:

Classes *a:Child* and *a:Person* can be used to represent the set of all children and persons, respectively, in the application domain, and they can be used in an axiom such as the following one:

SubClassOf(*a:Child a:Person*)	Each child is a person.

5.2 Datatypes

Datatypes are entities that refer to sets of data values. Thus, datatypes are analogous to classes, the main difference being that the former contain data values such as strings and numbers, rather than individuals. Datatypes are a kind of data range, which allows them to be used in restrictions. As explained in Section 7, each data range is associated with an arity; for datatypes, the arity is always one. The built-in datatype *rdfs:Literal* denotes any set of data values that contains the union of the value spaces of all datatypes.

An IRI used to identify a datatype in an OWL 2 DL ontology *must*

- be *rdfs:Literal*, or
- identify a datatype in the OWL 2 datatype map (see Section 4), or
- not be in the reserved vocabulary of OWL 2 (see Section 2.4).

The conditions from the previous paragraph and the restrictions on datatypes in Section 11.2 require each datatype in an OWL 2 DL ontology to be *rdfs:Literal*, one of the datatypes from Section 4, or a datatype defined by means of a datatype definition (see Section 9.4).

Datatype := **IRI**

Example:

The datatype *xsd:integer* denotes the set of all integers. It can be used in axioms such as the following one:

`DataPropertyRange( a:hasAge xsd:integer )`	The range of the *a:hasAge* data property is *xsd:integer*.

5.3 Object Properties

Object properties connect pairs of individuals.

ObjectProperty := **IRI**

The object properties with the IRIs *owl:topObjectProperty* and *owl:bottomObjectProperty* are available in OWL 2 as built-in object properties with a predefined semantics:

- The object property with IRI *owl:topObjectProperty* connects all possible pairs of individuals.
- The object property with IRI *owl:bottomObjectProperty* does not connect any pair of individuals.

IRIs from the reserved vocabulary other than *owl:topObjectProperty* and *owl:bottomObjectProperty* *must not* be used to identify object properties in an OWL 2 DL ontology.

Example:

The object property *a:parentOf* can be used to represent the parenthood relationship between individuals. It can be used in axioms such as the following one:

`ObjectPropertyAssertion( a:parentOf a:Peter a:Chris )`	Peter is a parent of Chris.

5.4 Data Properties

Data properties connect individuals with literals. In some knowledge representation systems, functional data properties are called *attributes*.

DataProperty := **IRI**

The data properties with the IRIs *owl:topDataProperty* and *owl:bottomDataProperty* are available in OWL 2 as built-in data properties with a predefined semantics:

- The data property with IRI *owl:topDataProperty* connects all possible individuals with all literals.
- The data property with IRI *owl:bottomDataProperty* does not connect any individual with a literal.

IRIs from the reserved vocabulary other than *owl:topDataProperty* and *owl:bottomDataProperty* *must not* be used to identify data properties in an OWL 2 DL ontology.

Example:

The data property *a:hasName* can be used to associate a name with each person. It can be used in axioms such as the following one:

`DataPropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is `"Peter Griffin"`.

5.5 Annotation Properties

Annotation properties can be used to provide an annotation for an ontology, axiom, or an IRI. The structure of annotations is further described in Section 10.

AnnotationProperty := **IRI**

The annotation properties with the IRIs listed below are available in OWL 2 as built-in annotation properties with a predefined semantics:

- The *rdfs:label* annotation property can be used to provide an IRI with a human-readable label.
- The *rdfs:comment* annotation property can be used to provide an IRI with a human-readable comment.
- The *rdfs:seeAlso* annotation property can be used to provide an IRI with another IRI such that the latter provides additional information about the former.
- The *rdfs:isDefinedBy* annotation property can be used to provide an IRI with another IRI such that the latter provides information about the definition of the former; the way in which this information is provided is not described by this specification.
- An annotation with the *owl:deprecated* annotation property and the value equal to `"true"^^xsd:boolean` can be used to specify that an IRI is deprecated.
- The *owl:versionInfo* annotation property can be used to provide an IRI with a string that describes the IRI's version.
- The *owl:priorVersion* annotation property is described in more detail in Section 3.5.
- The *owl:backwardCompatibleWith* annotation property is described in more detail in Section 3.5.
- The *owl:incompatibleWith* annotation property is described in more detail in Section 3.5.

IRIs from the reserved vocabulary other than the ones listed above *must not* be used to identify annotation properties in an OWL 2 DL ontology.

Example:

The comment provided by the following annotation assertion axiom might, for example, be used by an OWL 2 tool to display additional information about the IRI *a:Peter*.

`AnnotationAssertion( rdfs:comment a:Peter "The father of the Griffin family from Quahog." )`	This axiom provides a comment for the IRI *a:Peter*.

5.6 Individuals

Individuals in the OWL 2 syntax represent actual objects from the domain. There are two types of individuals in the syntax of OWL 2. *Named individuals* are given an explicit name that can be used in any ontology to refer to the same object. *Anonymous individuals* do not have a global name and are thus local to the ontology they are contained in.

Individual := **NamedIndividual** | **AnonymousIndividual**

5.6.1 Named Individuals

Named individuals are identified using an IRI. Since they are given an IRI, named individuals are entities.

IRIs from the reserved vocabulary *must not* be used to identify named individuals in an OWL 2 DL ontology.

NamedIndividual := **IRI**

Example:

The individual *a:Peter* can be used to represent a particular person. It can be used in axioms such as the following one:

`ClassAssertion( a:Person a:Peter )`	Peter is a person.

5.6.2 Anonymous Individuals

If an individual is not expected to be used outside a particular ontology, one can use an *anonymous individual*, which is identified by a local node ID rather than a global IRI. Anonymous individuals are analogous to blank nodes in RDF [*RDF Concepts*].

AnonymousIndividual := **nodeID**

Example:

Anonymous individuals can be used, for example, to represent objects whose identity is of no relevance, such as the address of a person.

`ObjectPropertyAssertion( a:livesAt a:Peter _:a1 )`	Peter lives at some (unknown) address.
`ObjectPropertyAssertion( a:city _:a1 a:Quahog )`	This unknown address is in the city of Quahog and...
`ObjectPropertyAssertion( a:state _:a1 a:RI )`	...in the state of Rhode Island.

Special treatment is required in case anonymous individuals with the same node ID occur in two different ontologies. In particular, these two individuals are structurally equivalent (because they have the same node ID); however, they are not treated as identical in the semantics of OWL 2 (because anonymous individuals are local to an ontology they are used in). The latter is achieved by *standardizing anonymous individuals apart* when constructing the axiom closure of an ontology *O*: if anonymous individuals with the same node ID occur in two different ontologies in the import closure of *O*, then one of these individuals *must* be replaced in the axiom closure of *O* with a fresh anonymous individual (i.e., an anonymous individual whose node ID is unique in the import closure of *O*).

Example:

Assume that ontologies O_1 and O_2 both use *_:a5*, and that O_1 imports O_2. Although they both use the same local node ID, the individual *_:a5* in O_1 may be different from the individual *_:a5* in O_2.

At the level of the structural specification, individual *_:a5* in O_1 is structurally equivalent to individual *_:a5* in O_2. This might be important, for example, for tools that use structural equivalence to define the semantics of axiom retraction.

In order to ensure that these individuals are treated differently by the semantics they are standardized apart when computing the axiom closure of O_1 — either *_:a5* in O_1 is replaced with a fresh anonymous individual, or this is done for *_:a5* in O_2.

5.7 Literals

Literals represent data values such as particular strings or integers. They are analogous to typed RDF literals [*RDF Concepts*] and can also be understood as individuals denoting data values. Each literal consists of a lexical form, which is a string, and a datatype; the datatypes supported in OWL 2 are described in more detail in Section 4. A literal consisting of a lexical form `"abc"` and a datatype identified by the IRI `datatypeIRI` is written as `"abc"^^datatypeIRI`. Furthermore, literals whose datatype is *rdf:PlainLiteral* can be abbreviated in functional-style syntax ontology documents as plain RDF literals [*RDF Concepts*]. These abbreviations are purely syntactic shortcuts and are thus not reflected in the structural specification of OWL 2. The observable behavior of OWL 2 implementation *must* be as if these shortcuts were expanded during parsing.

- Literals of the form `"abc@"^^`*`rdf:PlainLiteral`* *should* be abbreviated in functional-style syntax ontology documents to `"abc"` whenever possible.
- Literals of the form `"abc@langTag"^^`*`rdf:PlainLiteral`* where `"langTag"` is not empty *should* be abbreviated in functional-style syntax documents to `"abc"@langTag` whenever possible.

The lexical form of each literal occurring in an OWL 2 DL ontology *must* belong to the lexical space of the literal's datatype.

```
Literal := typedLiteral | stringLiteralNoLanguage | stringLiteralWithLanguage
typedLiteral := lexicalForm '^^' Datatype
lexicalForm := quotedString
stringLiteralNoLanguage := quotedString
stringLiteralWithLanguage := quotedString languageTag
```

Example:

`"1"^^`*`xsd:integer`* is a literal that represents the integer 1.

Example:

`"Family Guy"` is an abbreviation for `"Family Guy@"^^`*`rdf:PlainLiteral`* — a literal with the lexical form `"Family Guy@"` and the datatype *rdf:PlainLiteral* — which denotes a string `"Family Guy"` without a language tag.

Furthermore, `"Padre de familia"@es` is an abbreviation for the literal `"Padre de familia@es"^^`*`rdf:PlainLiteral`*, which denotes a pair consisting of the string `"Padre de familia"` and the language tag `es`.

Two literals are structurally equivalent if and only if both the lexical form and the datatype are structurally equivalent; that is, literals denoting the same data value are structurally different if either their lexical form or the datatype is different.

Example:

Even though literals `"1"^^`*`xsd:integer`* and `"+1"^^`*`xsd:integer`* are interpreted as the integer 1, these two literals are not structurally equivalent because their lexical forms are not identical. Similarly, `"1"^^`*`xsd:integer`* and `"1"^^xsd:positiveInteger` are not structurally equivalent because their datatypes are not identical.

5.8 Entity Declarations and Typing

Each IRI *I* used in an OWL 2 ontology *O* can be, and sometimes even needs to be, declared in *O*; roughly speaking, this means that the axiom closure of *O* must contain an appropriate declaration for *I*. A declaration for *I* in *O* serves two purposes:

- A declaration says that *I* exists — that is, it says that *I* is part of the vocabulary of *O*.
- A declaration associates with *I* an entity type — that is, it says whether *I* is used in *O* as a class, datatype, object property, data property, annotation property, an individual, or a combination thereof.

Example:

An ontology might contain a class declaration for the IRI *a:Person*. Such a declaration introduces the class *a:Person* into the ontology, and it states that the IRI *a:Person* is used to name a class in the ontology. An ontology editor might use declarations to implement functions such as "Add New Class".

In OWL 2, declarations are a type of axiom; thus, to declare an entity in an ontology, one can simply include the appropriate axiom in the ontology. These axioms are nonlogical in the sense that they do not affect the consequences of an OWL 2 ontology. The structure of entity declarations is shown in Figure 3.

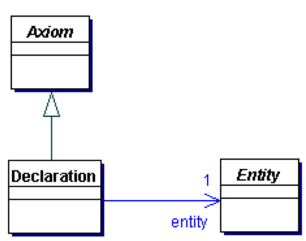

Figure 3. Entity Declarations in OWL 2

```
Declaration := 'Declaration' '(' axiomAnnotations Entity ')'
Entity :=
    'Class' '(' Class ')' |
    'Datatype' '(' Datatype ')' |
    'ObjectProperty' '(' ObjectProperty ')' |
```

```
    'DataProperty' '(' DataProperty ')' |
    'AnnotationProperty' '(' AnnotationProperty ')' |
    'NamedIndividual' '(' NamedIndividual ')'
```

Example:

The following axioms state that the IRI *a:Person* is used as a class and that the IRI *a:Peter* is used as an individual.

```
Declaration( Class( a:Person ) )
Declaration( NamedIndividual( a:Peter ) )
```

Declarations for the built-in entities of OWL 2, listed in Table 5, are implicitly present in every OWL 2 ontology.

Table 5. Declarations of Built-In Entities

Declaration(Class(*owl:Thing*))	
Declaration(Class(*owl:Nothing*))	
Declaration(ObjectProperty(*owl:topObjectProperty*))	
Declaration(ObjectProperty(*owl:bottomObjectProperty*))	
Declaration(DataProperty(*owl:topDataProperty*))	
Declaration(DataProperty(*owl:bottomDataProperty*))	
Declaration(Datatype(*rdfs:Literal*))	
Declaration(Datatype(*I*))	for each IRI *I* of a datatype in the OWL 2 datatype map (see Section 4)
Declaration(AnnotationProperty(*I*))	for each IRI *I* of a built-in annotation property listed in Section 5.5

5.8.1 Typing Constraints of OWL 2 DL

Let *Ax* be a set of axioms. An IRI *I* is *declared* to be of type *T* in *Ax* if a declaration axiom of type *T* for *I* is contained in *Ax* or in the set of built-in declarations listed in Table 5. The set *Ax* satisfies the *typing constraints* of OWL 2 DL if all of the following conditions are satisfied:

- Property typing constraints:
  - If an object property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as an object property.
  - If a data property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a data property.
  - If an annotation property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as an annotation property.
  - No IRI *I* is declared in *Ax* as being of more than one type of property; that is, no *I* is declared in *Ax* to be both object and data, object and annotation, or data and annotation property.
- Class/datatype typing constraints:
  - If a class with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a class.
  - If a datatype with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a datatype.
  - No IRI *I* is declared in *ax* to be both a class and a datatype.

The axiom closure *Ax* of each OWL 2 DL ontology *O* *must* satisfy the typing constraints of OWL 2 DL.

The typing constraints thus ensure that the sets of IRIs used as object, data, and annotation properties in *O* are disjoint and that, similarly, the sets of IRIs used as classes and datatypes in *O* are disjoint as well. These constraints are used for disambiguating the types of IRIs when reading ontologies from external transfer syntaxes. All other declarations are optional.

Example:

An IRI *I* can be used as an individual in *O* even if *I* is not declared as an individual in *O*.

Declarations are often omitted in the examples in this document in cases where the types of entities are clear.

5.8.2 Declaration Consistency

Although declarations are not always required, they can be used to catch obvious errors in ontologies.

Example:

The following ontology erroneously refers to the individual *a:Petre* instead of the individual *a:Peter*.

```
Ontology( <http://www.my.example.com/example>
    Declaration( Class( a:Person ) )
    ClassAssertion( a:Person a:Petre )
)
```

There is no way of telling whether *a:Petre* was used by mistake. If, in contrast, all individuals in an ontology were by convention required to be declared, this error could be caught by a simple tool.

An ontology *O* is said to have *consistent declarations* if each IRI *I* occurring in the axiom closure of *O* in position of an entity with a type *T* is declared in *O* as having type *T*. OWL 2 ontologies are not required to have consistent declarations: an ontology *may* be used even if its declarations are not consistent.

Example:

The ontology from the previous example fails this check: *a:Petre* is used as an individual but the ontology does not declare *a:Petre* to be an individual. In contrast, the following ontology satisfies this condition.

```
Ontology( <http://www.my.example.com/example>
    Declaration( Class( a:Person ) )
    Declaration( NamedIndividual( a:Peter ) )
    ClassAssertion( a:Person a:Peter )
)
```

5.9 Metamodeling

An IRI *I* can be used in an OWL 2 ontology to refer to more than one type of entity. Such usage of *I* is often called *metamodeling*, because it can be used to state facts about classes and properties themselves. In such cases, the entities that share the same IRI *I* should be understood as different "views" of the same underlying notion identified by the IRI *I*.

Example:

Consider the following ontology.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( a:Species a:Dog )`	Dog is a species.

In the first axiom, the IRI *a:Dog* is used as a class, while in the second axiom, it is used as an individual; thus, the class *a:Species* acts as a metaclass for the class *a:Dog*. The individual *a:Dog* and the class *a:Dog* should be understood as two "views" of one and the same IRI — *a:Dog*. Under the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*], these two views are interpreted independently: the class view of *a:Dog* is interpreted as a unary predicate, while the individual view of *a:Dog* is interpreted as a constant.

Both metamodeling and annotations provide means to associate additional information with classes and properties. The following rule-of-the-thumb can be used to determine when to use which construct:

- Metamodeling should be used when the information attached to entities should be considered a part of the domain.
- Annotations should be used when the information attached to entities should not be considered a part of the domain and when it should not contribute to the logical consequences of an ontology.

Example:

Consider the following ontology.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( a:PetAnimals a:Dog )`	Dogs are pet animals.
`AnnotationAssertion( a:addedBy a:Dog "Seth MacFarlane" )`	The IRI *a:Dog* has been added to the ontology by Seth MacFarlane.

The facts that Brian is a dog and that dogs are pet animals are statements about the domain. Therefore, these facts are represented in the above ontology via metamodeling. In contrast, the information about who added the IRI *a:Dog* to the ontology does not describe the actual domain, but might be interesting from a management point of view. Therefore, this information is represented using an annotation.

6 Property Expressions

Properties can be used in OWL 2 to form property expressions.

6.1 Object Property Expressions

Object properties can by used in OWL 2 to form object property expressions, which represent relationships between pairs of individuals. They are represented in the structural specification of OWL 2 by **ObjectPropertyExpression**, and their structure is shown in Figure 4.

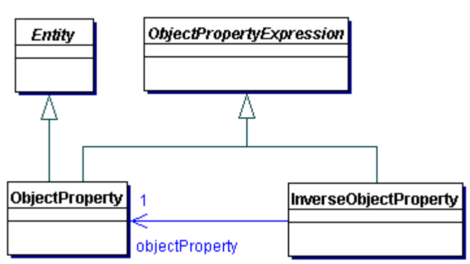

Figure 4. Object Property Expressions in OWL 2

As one can see from the figure, OWL 2 supports only two kinds of object property expressions. Object properties are the simplest form of object property expressions, and inverse object properties allow for bidirectional navigation in class expressions and axioms.

ObjectPropertyExpression := **ObjectProperty** | **InverseObjectProperty**

6.1.1 Inverse Object Properties

An inverse object property expression `ObjectInverseOf( P )` connects an individual I_1 with I_2 if and only if the object property P connects I_2 with I_1.

InverseObjectProperty := 'ObjectInverseOf' '(' **ObjectProperty** ')'

Example:

Consider the ontology consisting of the following assertion.

ObjectPropertyAssertion(*a:fatherOf a:Peter a:Stewie*)	Peter is Stewie's father.

This ontology entails that *a:Stewie* is connected by the following object property expression to *a:Peter*:

ObjectInverseOf(*a:fatherOf*)

6.2 Data Property Expressions

For symmetry with object property expressions, the structural specification of OWL 2 also introduces data property expressions, which represent relationships between an individual and a literal. The structure of data property expressions is shown in Figure 5. The only allowed data property expression is a data property; thus, **DataPropertyExpression** in the structural specification of OWL 2 can be seen as a place-holder for possible future extensions.

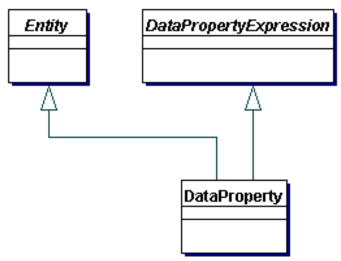

Figure 5. Data Property Expressions in OWL 2

DataPropertyExpression := **DataProperty**

7 Data Ranges

Datatypes, such as *xsd:string* or *xsd:integer*, and literals such as "1"^^*xsd:integer*, can be used to express data ranges — sets of tuples of literals, where tuples consisting of only one literal are identified with the literal itself. Each data range is associated with a positive arity, which determines the size of the tuples in the data range. All datatypes have arity one. This specification currently does not define data ranges of arity more than one; however, by allowing for *n*-ary data ranges, the syntax of OWL 2 provides a "hook" allowing implementations to introduce extensions such as comparisons and arithmetic.

Data ranges can be used in restrictions on data properties, as discussed in Sections 8.4 and 8.5. The structure of data ranges in OWL 2 is shown in Figure 6. The simplest data ranges are datatypes. The **DataIntersectionOf**, **DataUnionOf**, and **DataComplementOf** data ranges provide for the standard set-theoretic operations on data ranges; in logical languages these are usually called conjunction, disjunction, and negation, respectively. The **DataOneOf** data range consists of exactly the specified set of literals. Finally, the **DatatypeRestriction** data range restricts the value space of a datatype by a constraining facet.

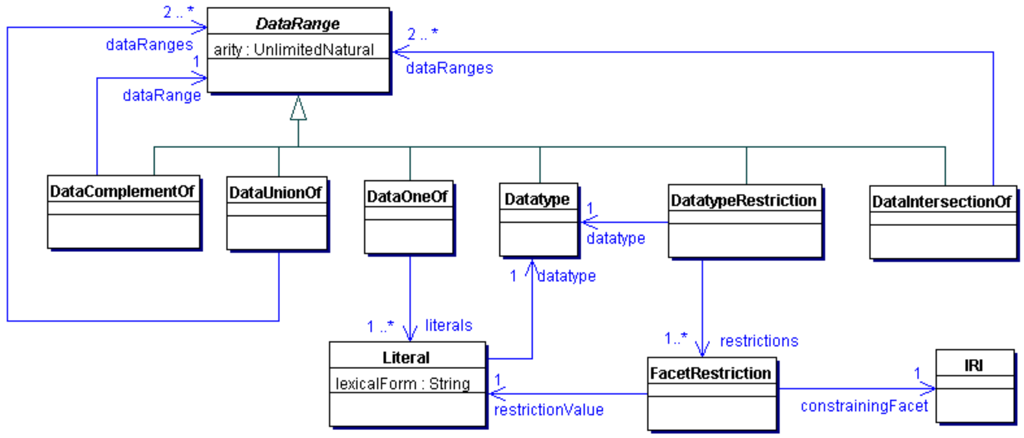

Figure 6. Data Ranges in OWL 2

DataRange :=
    Datatype |
    DataIntersectionOf |
    DataUnionOf |
    DataComplementOf |
    DataOneOf |
    DatatypeRestriction

7.1 Intersection of Data Ranges

An intersection data range DataIntersectionOf(DR_1 ... DR_n) contains all tuples of literals that are contained in each data range DR_i for $1 \le i \le n$. All data ranges DR_i *must* be of the same arity, and the resulting data range is of that arity as well.

DataIntersectionOf := 'DataIntersectionOf' '(' **DataRange** **DataRange** { **DataRange** } ')'

Example:

The following data range contains exactly the integer 0:

```
DataIntersectionOf( xsd:nonNegativeInteger xsd:nonPositiveInteger )
```

7.2 Union of Data Ranges

A union data range DataUnionOf(DR_1 ... DR_n) contains all tuples of literals that are contained in the at least one data range DR_i for $1 \le i \le n$. All data ranges DR_i *must* be of the same arity, and the resulting data range is of that arity as well.

DataUnionOf := 'DataUnionOf' '(' **DataRange** **DataRange** { **DataRange** } ')'

Example:

The following data range contains all strings and all integers:

```
DataUnionOf( xsd:string xsd:integer )
```

7.3 Complement of Data Ranges

A complement data range DataComplementOf(DR) contains all tuples of literals that are not contained in the data range DR. The resulting data range has the arity equal to the arity of DR.

DataComplementOf := 'DataComplementOf' '(' **DataRange** ')'

Example:

The following complement data range contains literals that are not positive integers:

```
DataComplementOf( xsd:positiveInteger )
```

In particular, this data range contains the integer zero and all negative integers; however, it also contains all strings (since strings are not positive integers).

7.4 Enumeration of Literals

An enumeration of literals DataOneOf(lt_1 ... lt_n) contains exactly the explicitly specified literals lt_i with 1 ≤ i ≤ n. The resulting data range has arity one.

DataOneOf := 'DataOneOf' '(' **Literal** { **Literal** } ')'

Example:

The following data range contains exactly two literals: the string "Peter" and the integer one.

```
DataOneOf( "Peter" "1"^^xsd:integer )
```

7.5 Datatype Restrictions

A datatype restriction DatatypeRestriction(DT F_1 lt_1 ... F_n lt_n) consists of a unary datatype DT and n pairs (F_i , lt_i). The resulting data range is unary and is obtained by restricting the value space of DT according to the semantics of all (F_i , v_i) (multiple pairs are interpreted conjunctively), where v_i are the data values of the literals lt_i.

In an OWL 2 DL ontology, each pair (F_i , v_i) *must* be contained in the facet space of DT (see Section 4).

DatatypeRestriction := 'DatatypeRestriction' '(' **Datatype** **constrainingFacet** **restrictionValue** { **constrainingFacet** **restrictionValue** } ')'
constrainingFacet := **IRI**
restrictionValue := **Literal**

Example:

The following data range contains exactly the integers 5, 6, 7, 8, and 9:

```
DatatypeRestriction( xsd:integer xsd:minInclusive "5"^^xsd:integer xsd:maxExclusive "10"^^xsd:integer )
```

8 Class Expressions

In OWL 2, classes and property expressions are used to construct *class expressions*, sometimes also called *descriptions*, and, in the description logic literature, *complex concepts*. Class expressions represent sets of individuals by formally specifying conditions on the individuals' properties; individuals satisfying these conditions are said to be *instances* of the respective class expressions. In the structural specification of OWL 2, class expressions are represented by **ClassExpression**.

Example:

A class expression can be used to represent the set of "people that have at least one child". If an ontology additionally contains statements that "Peter is a person" and that "Peter has child Chris", then Peter can be classified as an instance of the mentioned class expression.

OWL 2 provides a rich set of primitives that can be used to construct class expressions. In particular, it provides the well known Boolean connectives *and*, *or*, and *not*; a restricted form of universal and existential quantification; number restrictions; enumeration of individuals; and a special *self*-restriction.

As shown in Figure 2, classes are the simplest form of class expressions. The other, complex, class expressions, are described in the following sections.

ClassExpression :=
Class |
ObjectIntersectionOf | **ObjectUnionOf** | **ObjectComplementOf** | **ObjectOneOf** |
ObjectSomeValuesFrom | **ObjectAllValuesFrom** | **ObjectHasValue** | **ObjectHasSelf** |
ObjectMinCardinality | **ObjectMaxCardinality** | **ObjectExactCardinality** |
DataSomeValuesFrom | **DataAllValuesFrom** | **DataHasValue** |
DataMinCardinality | **DataMaxCardinality** | **DataExactCardinality**

8.1 Propositional Connectives and Enumeration of Individuals

OWL 2 provides for enumeration of individuals and all standard Boolean connectives, as shown in Figure 7. The **ObjectIntersectionOf**, **ObjectUnionOf**, and **ObjectComplementOf** class expressions provide for the standard set-theoretic operations on class expressions; in logical languages these are usually called conjunction, disjunction, and negation, respectively. The **ObjectOneOf** class expression contains exactly the specified individuals.

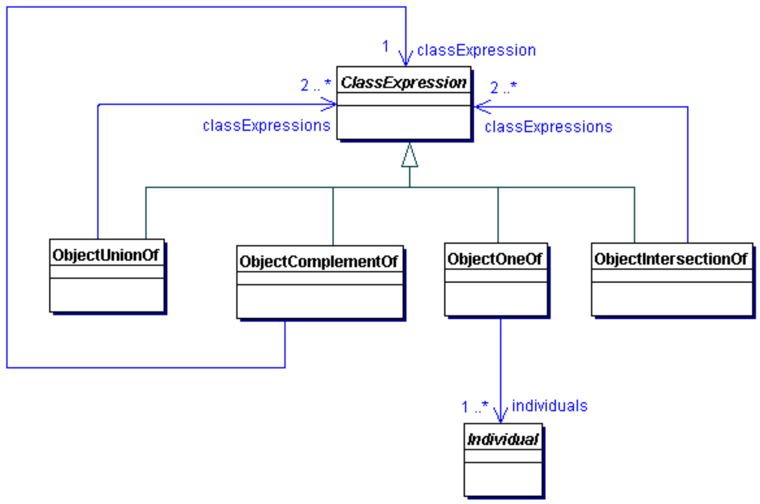

Figure 7. Propositional Connectives and Enumeration of Individuals in OWL 2

8.1.1 Intersection of Class Expressions

An intersection class expression ObjectIntersectionOf(CE_1 ... CE_n) contains all individuals that are instances of all class expressions CE_i for $1 \leq i \leq n$.

ObjectIntersectionOf := 'ObjectIntersectionOf' '(' **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

ClassAssertion(*a:Dog a:Brian*)	Brian is a dog.
ClassAssertion(*a:CanTalk a:Brian*)	Brian can talk.

The following class expression describes all dogs that can talk; furthermore, *a:Brian* is classified as its instance.

ObjectIntersectionOf(*a:Dog a:CanTalk*)

8.1.2 Union of Class Expressions

A union class expression ObjectUnionOf(CE_1 ... CE_n) contains all individuals that are instances of at least one class expression CE_i for $1 \leq i \leq n$.

ObjectUnionOf := 'ObjectUnionOf' '(' **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

ClassAssertion(*a:Man a:Peter*)	Peter is a man.
ClassAssertion(*a:Woman a:Lois*)	Lois is a woman.

The following class expression describes all individuals that are instances of either *a:Man* or *a:Woman*; furthermore, both *a:Peter* and *a:Lois* are classified as its instances:

ObjectUnionOf(*a:Man a:Woman*)

8.1.3 Complement of Class Expressions

A complement class expression ObjectComplementOf(CE) contains all individuals that are not instances of the class expression CE.

ObjectComplementOf := 'ObjectComplementOf' '(' **ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

```
DisjointClasses( a:Man a:Woman )          Nothing can be both a man and a woman.
ClassAssertion( a:Woman a:Lois )          Lois is a woman.
```

The following class expression describes all things that are not instances of *a:Man*:

```
ObjectComplementOf( a:Man )
```

Since *a:Lois* is known to be a woman and nothing can be both a man and a woman, then *a:Lois* is necessarily not a *a:Man*; therefore, *a:Lois* is classified as an instance of this complement class expression.

Example:

OWL 2 has *open-world* semantics, so negation in OWL 2 is the same as in classical (first-order) logic. To understand open-world semantics, consider the ontology consisting of the following assertion.

```
ClassAssertion( a:Dog a:Brian )          Brian is a dog.
```

One might expect *a:Brian* to be classified as an instance of the following class expression:

```
ObjectComplementOf( a:Bird )
```

Intuitively, the ontology does not explicitly state that *a:Brian* is an instance of *a:Bird*, so this statement seems to be false. In OWL 2, however, this is not the case: it is true that the ontology does not state that *a:Brian* is an instance of *a:Bird*; however, the ontology does not state the opposite either. In other words, this ontology simply does not contain enough information to answer the question whether *a:Brian* is an instance of *a:Bird* or not: it is perfectly possible that the information to that effect is actually true but it has not been included in the ontology.

The ontology from the previous example (in which *a:Lois* has been classified as *a:Man*), however, contains sufficient information to draw the expected conclusion. In particular, we know for sure that *a:Lois* is an instance of *a:Woman* and that *a:Man* and *a:Woman* do not share instances. Therefore, any additional information that does not lead to inconsistency cannot lead to a conclusion that *a:Lois* is an instance of *a:Man*; furthermore, if one were to explicitly state that *a:Lois* is an instance of *a:Man*, the ontology would be inconsistent and, by definition, it then entails all possible conclusions.

8.1.4 Enumeration of Individuals

An enumeration of individuals ObjectOneOf(a_1 ... a_n) contains exactly the individuals a_i with $1 \le i \le n$.

ObjectOneOf := 'ObjectOneOf' '(' **Individual** { **Individual** }')'

Example:

Consider the ontology consisting of the following axioms.

```
EquivalentClasses( a:GriffinFamilyMember
    ObjectOneOf( a:Peter a:Lois a:Stewie a:Meg a:Chris a:Brian )     The Griffin family consists exactly of Peter, Lois, Stewie, Meg, Chris, and Brian.
)
DifferentIndividuals( a:Quagmire a:Peter a:Lois a:Stewie a:Meg      Quagmire, Peter, Lois, Stewie, Meg, Chris, and Brian are
a:Chris a:Brian )                                                   all different from each other.
```

The class *a:GriffinFamilyMember* now contains exactly the six explicitly listed individuals. Since we also know that *a:Quagmire* is different from these six individuals, this individual is classified as an instance of the following class expression:

```
ObjectComplementOf( a:GriffinFamilyMember )
```

The last axiom in the ontology is necessary to derive the mentioned conclusion; without it, the open-world semantics of OWL 2 would allow for situations where *a:Quagmire* is the same as *a:Peter*, *a:Lois*, *a:Stewie*, *a:Meg*, *a:Chris*, or *a:Brian*.

Example:

To understand how the open-world semantics affects enumerations of individuals, consider the ontology consisting of the following axioms.

```
ClassAssertion( a:GriffinFamilyMember a:Peter )       Peter is a member of the Griffin Family.
ClassAssertion( a:GriffinFamilyMember a:Lois )        Lois is a member of the Griffin Family.
ClassAssertion( a:GriffinFamilyMember a:Stewie )      Stewie is a member of the Griffin Family.
ClassAssertion( a:GriffinFamilyMember a:Meg )         Meg is a member of the Griffin Family.
ClassAssertion( a:GriffinFamilyMember a:Chris )       Chris is a member of the Griffin Family.
ClassAssertion( a:GriffinFamilyMember a:Brian )       Brian is a member of the Griffin Family.
```

The class *a:GriffinFamilyMember* now also contains the mentioned six individuals, just as in the previous example. The main difference to the previous example, however, is that the extension of *a:GriffinFamilyMember* is not closed: the semantics of OWL 2 assumes that information about a potential instance of *a:GriffinFamilyMember* may be missing. Therefore, *a:Quagmire* is now not classified as an instance of the following class expression, and this does not change even if we add the axiom stating that all of these six individuals are different from each other:

```
ObjectComplementOf( a:GriffinFamilyMember )
```

8.2 Object Property Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on object property expressions, as shown in Figure 8. The **ObjectSomeValuesFrom** class expression allows for existential quantification over an object property expression, and it contains those individuals that are connected through an object property expression to at least one instance of a given class expression. The **ObjectAllValuesFrom** class expression allows for universal quantification over an object property expression, and it contains those individuals that are connected through an object property expression only to instances of a given class expression. The **ObjectHasValue** class expression contains those individuals that are connected by an object property expression to a particular individual. Finally, the **ObjectHasSelf** class expression contains those individuals that are connected by an object property expression to themselves.

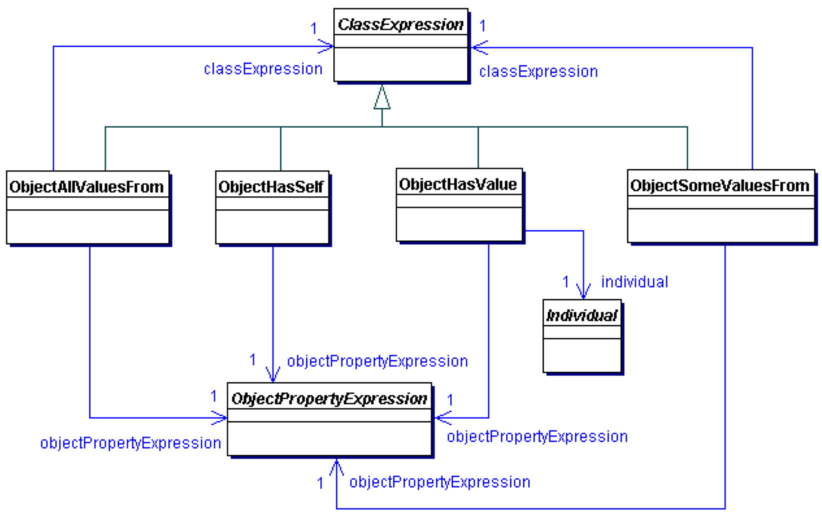

Figure 8. Restricting Object Property Expressions in OWL 2

8.2.1 Existential Quantification

An existential class expression `ObjectSomeValuesFrom( OPE CE )` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to an individual that is an instance of `CE`. Provided that `OPE` is *simple* according to the definition in Section 11, such a class expression can be seen as a syntactic shortcut for the class expression `ObjectMinCardinality( 1 OPE CE )`.

ObjectSomeValuesFrom := 'ObjectSomeValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is Stewie's father.
`ClassAssertion( a:Man a:Stewie )`	Stewie is a man.

The following existential expression contains those individuals that are connected by the *a:fatherOf* property to individuals that are instances of *a:Man*; furthermore, *a:Peter* is classified as its instance:

```
ObjectSomeValuesFrom( a:fatherOf a:Man )
```

8.2.2 Universal Quantification

A universal class expression `ObjectAllValuesFrom( OPE CE )` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` only to individuals that are instances of `CE`. Provided that `OPE` is *simple* according to the definition in Section 11, such a class expression can be seen as a syntactic shortcut for the class expression `ObjectMaxCardinality( 0 OPE ObjectComplementOf( CE ) )`.

ObjectAllValuesFrom := 'ObjectAllValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyAssertion( a:hasPet a:Peter a:Brian )`	Brian is a pet of Peter.
`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( ObjectMaxCardinality( 1 a:hasPet ) a:Peter )`	Peter has at most one pet.

The following universal expression contains those individuals that are connected through the *a:hasPet* property only with individuals that are instances of *a:Dog* — that is, it contains individuals that have only dogs as pets:

```
ObjectAllValuesFrom( a:hasPet a:Dog )
```

The ontology axioms clearly state that *a:Peter* is connected by *a:hasPet* only to instances of *a:Dog*: it is impossible to connect *a:Peter* by *a:hasPet* to an individual different from *a:Brian* without making the ontology inconsistent. Therefore, *a:Peter* is classified as an instance of the mentioned class expression.

The last axiom — that is, the one stating that *a:Peter* has at most one pet — is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology might not have listed all the individuals to which *a:Peter* is connected by *a:hasPet*. In such a case *a:Peter* would not be classified as an instance of the mentioned class expression.

8.2.3 Individual Value Restriction

A has-value class expression `ObjectHasValue( OPE a )` consists of an object property expression `OPE` and an individual a, and it contains all those individuals that are connected by `OPE` to a. Each such class expression can be seen as a syntactic shortcut for the class expression `ObjectSomeValuesFrom( OPE ObjectOneOf( a ) )`.

ObjectHasValue := 'ObjectHasValue' '(' **ObjectPropertyExpression** **Individual** ')'

Example:

Consider the ontology consisting of the following axiom.

`ObjectPropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is Stewie's father.

The following has-value class expression contains those individuals that are connected through the *a:fatherOf* property with the individual *a:Stewie*; furthermore, *a:Peter* is classified as its instance:

```
ObjectHasValue( a:fatherOf a:Stewie )
```

8.2.4 Self-Restriction

A self-restriction `ObjectHasSelf( OPE )` consists of an object property expression `OPE`, and it contains all those individuals that are connected by `OPE` to themselves.

ObjectHasSelf := 'ObjectHasSelf' '(' **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axiom.

`ObjectPropertyAssertion( a:likes a:Peter a:Peter )`	Peter likes Peter.

The following self-restriction contains those individuals that like themselves; furthermore, *a:Peter* is classified as its instance:

```
ObjectHasSelf( a:likes )
```

8.3 Object Property Cardinality Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on the cardinality of object property expressions, as shown in Figure 9. All cardinality restrictions can be qualified or unqualified: in the former case, the cardinality restriction only applies to individuals that are connected by the object property expression and are instances of the qualifying class expression; in the latter case the restriction applies to all individuals that are connected by the object property expression (this is equivalent to the qualified case with the qualifying class expression equal to *owl:Thing*). The class expressions **ObjectMinCardinality**, **ObjectMaxCardinality**, and **ObjectExactCardinality** contain those individuals that are connected by an object property expression to at least, at most, and exactly a given number of instances of a specified class expression, respectively.

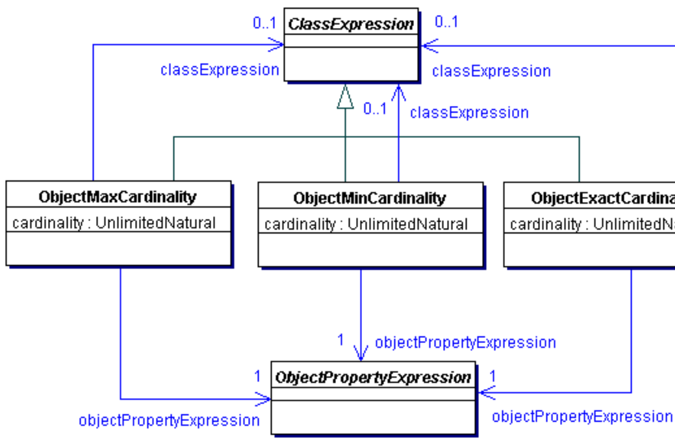

Figure 9. Restricting the Cardinality of Object Property Expressions in OWL 2

8.3.1 Minimum Cardinality

A minimum cardinality expression `ObjectMinCardinality( n OPE CE )` consists of a nonnegative integer n, an object property expression OPE, and a class expression CE, and it contains all those individuals that are connected by OPE to at least n different individuals that are instances of CE. If CE is missing, it is taken to be *owl:Thing*.

ObjectMinCardinality := 'ObjectMinCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

Example:

Consider the ontology consisting of the following axioms.

ObjectPropertyAssertion(*a:fatherOf a:Peter a:Stewie*)	Peter is Stewie's father.
ClassAssertion(*a:Man a:Stewie*)	Stewie is a man.
ObjectPropertyAssertion(*a:fatherOf a:Peter a:Chris*)	Peter is Chris's father.
ClassAssertion(*a:Man a:Chris*)	Chris is a man.
DifferentIndividuals(*a:Chris a:Stewie*)	Chris and Stewie are different from each other.

The following minimum cardinality expression contains those individuals that are connected by *a:fatherOf* to at least two different instances of *a:Man*:

```
ObjectMinCardinality( 2 a:fatherOf a:Man )
```

Since *a:Stewie* and *a:Chris* are both instances of *a:Man* and are different from each other, *a:Peter* is classified as an instance of this class expression.

Due to the open-world semantics, the last axiom — the one stating that *a:Chris* and *a:Stewie* are different from each other — is necessary for this inference: without this axiom, it is possible that *a:Chris* and *a:Stewie* are actually the same individual.

8.3.2 Maximum Cardinality

A maximum cardinality expression `ObjectMaxCardinality( n OPE CE )` consists of a nonnegative integer n, an object property expression OPE, and a class expression CE, and it contains all those individuals that are connected by OPE to at most n different individuals that are instances of CE. If CE is missing, it is taken to be *owl:Thing*.

ObjectMaxCardinality := 'ObjectMaxCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

Example:

Consider the ontology consisting of the following axioms.

ObjectPropertyAssertion(*a:hasPet a:Peter a:Brian*)	Brian is a pet of Peter.
ClassAssertion(ObjectMaxCardinality(1 *a:hasPet*) *a:Peter*)	Peter has at most one pet.

The following maximum cardinality expression contains those individuals that are connected by *a:hasPet* to at most two individuals:

```
ObjectMaxCardinality( 2 a:hasPet )
```

Since *a:Peter* is known to be connected by *a:hasPet* to at most one individual, it is certainly also connected by *a:hasPet* to at most two individuals so, consequently, *a:Peter* is classified as an instance of this class expression.

The example ontology explicitly names only *a:Brian* as being connected by *a:hasPet* from *a:Peter*, so one might expect *a:Peter* to be classified as an instance of the mentioned class expression even without the second axiom. This, however, is not the case due to the open-world semantics. Without the last axiom, it is possible that *a:Peter* is connected by *a:hasPet* to other individuals. The second axiom closes the set of individuals that *a:Peter* is connected to by *a:hasPet*.

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyAssertion( a:hasDaughter a:Peter a:Meg )`	Meg is a daughter of Peter.
`ObjectPropertyAssertion( a:hasDaughter a:Peter a:Megan )`	Megan is a daughter of Peter.
`ClassAssertion( ObjectMaxCardinality( 1 a:hasDaughter ) a:Peter )`	Peter has at most one daughter.

One might expect this ontology to be inconsistent: on the one hand, it says that *a:Meg* and *a:Megan* are connected to *a:Peter* by *a:hasDaughter*, but, on the other hand, it says that *a:Peter* is connected by *a:hasDaughter* to at most one individual. This ontology, however, is not inconsistent because the semantics of OWL 2 does not make the *unique name assumption* — that is, it does not assume distinct individuals to be necessarily different. For example, the ontology does not explicitly say that *a:Meg* and *a:Megan* are different individuals; therefore, since *a:Peter* can be connected by *a:hasDaughter* to at most one distinct individual, *a:Meg* and *a:Megan* must be the same. This example ontology thus entails the following assertion:

```
SameIndividual( a:Meg a:Megan )
```

One can axiomatize the unique name assumption in OWL 2 by explicitly stating that all individuals are different from each other. This can be done by adding the following axiom, which makes the example ontology inconsistent.

`DifferentIndividuals( a:Peter a:Meg a:Megan )`	Peter, Meg, and Megan are all different from each other.

8.3.3 Exact Cardinality

An exact cardinality expression `ObjectExactCardinality( n OPE CE )` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to exactly `n` different individuals that are instances of `CE`. If `CE` is missing, it is taken to be *owl:Thing*. Such an expression is actually equivalent to the expression

```
ObjectIntersectionOf( ObjectMinCardinality( n OPE CE ) ObjectMaxCardinality( n OPE CE ) ).
```

ObjectExactCardinality := 'ObjectExactCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

Example:

Consider the ontology consisting of the following axioms.

```
ObjectPropertyAssertion( a:hasPet a:Peter a:Brian )          Brian is a pet of Peter.
ClassAssertion( a:Dog a:Brian )                              Brian is a dog.
ClassAssertion(
    ObjectAllValuesFrom( a:hasPet
       ObjectUnionOf(
          ObjectOneOf( a:Brian )
          ObjectComplementOf( a:Dog )                        Each pet of Peter is either Brian or it is not a dog.
        )
    )
    a:Peter
)
```

The following exact cardinality expression contains those individuals that are connected by *a:hasPet* to exactly one instance of *a:Dog*; furthermore, *a:Peter* is classified as its instance:

```
ObjectExactCardinality( 1 a:hasPet a:Dog )
```

This is because the first two axioms say that *a:Peter* is connected to *a:Brian* by *a:hasPet* and that *a:Brian* is an instance of *a:Dog*, and the last axiom says that any individual different from *a:Brian* that is connected to *a:Peter* by *a:hasPet* is not an instance of *a:Dog*; hence, *a:Peter* is connected to exactly one instance of *a:Dog* by *a:hasPet*.

8.4 Data Property Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on data property expressions, as shown in Figure 10. These are similar to the restrictions on object property expressions, the main difference being that the expressions for existential and universal quantification allow for *n*-ary data ranges. All data ranges explicitly supported by this specification are unary; however, the provision of *n*-ary data ranges in existential and universal quantification allows OWL 2 tools to support extensions such as value comparisons and, consequently, class expressions such as "individuals whose width is greater than their height". Thus, the **DataSomeValuesFrom** class expression allows for a restricted existential quantification over a list of data property expressions, and it contains those individuals that are connected through the data property expressions to at least one literal in the given data range. The **DataAllValuesFrom** class expression allows for a restricted universal quantification over a list of data property expressions, and it contains those individuals that are connected through the data property expressions only to literals in the given data range. Finally, the **DataHasValue** class expression contains those individuals that are connected by a data property expression to a particular literal.

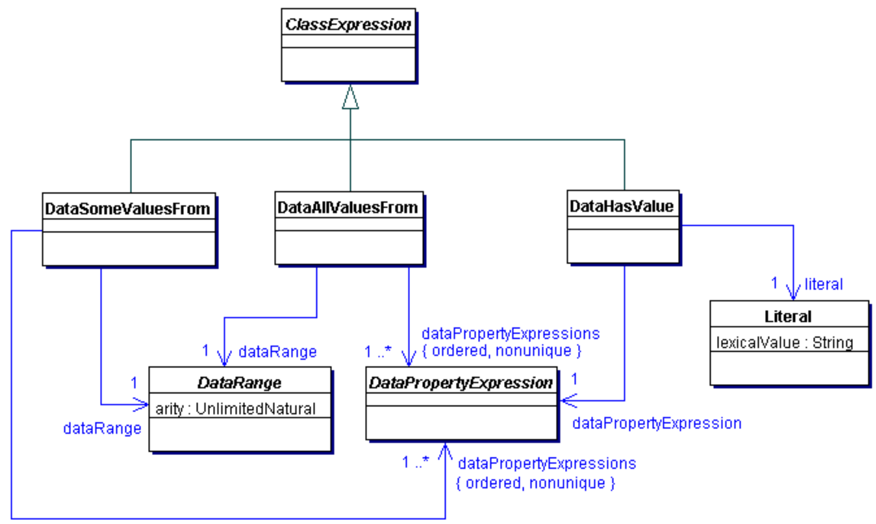

Figure 10. Restricting Data Property Expressions in OWL 2

8.4.1 Existential Quantification

An existential class expression DataSomeValuesFrom(DPE_1 ... DPE_n DR) consists of n data property expressions DPE_i, 1 ≤ i ≤ n, and a data range DR whose arity *must* be n. Such a class expression contains all those individuals that are connected by DPE_i to literals lt_i, 1 ≤ i ≤ n, such that the tuple (lt_1 , ..., lt_n) is in DR. A class expression of the form DataSomeValuesFrom(DPE DR) can be seen as a syntactic shortcut for the class expression DataMinCardinality(1 DPE DR).

DataSomeValuesFrom := 'DataSomeValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

Example:

Consider the ontology consisting of the following axiom.

DataPropertyAssertion(*a:hasAge a:Meg* "17"^^*xsd:integer*)	Meg is seventeen years old.

The following existential class expression contains all individuals that are connected by *a:hasAge* to an integer strictly less than 20 so; furthermore, *a:Meg* is classified as its instance:

```
DataSomeValuesFrom( a:hasAge DatatypeRestriction( xsd:integer xsd:maxExclusive "20"^^xsd:integer ) )
```

8.4.2 Universal Quantification

A universal class expression DataAllValuesFrom(DPE_1 ... DPE_n DR) consists of n data property expressions DPE_i, 1 ≤ i ≤ n, and a data range DR whose arity *must* be n. Such a class expression contains all those individuals that are connected by DPE_i only to literals lt_i, 1 ≤ i ≤ n, such that each tuple (lt_1 , ..., lt_n) is in DR. A class expression of the form DataAllValuesFrom(DPE DR) can be seen as a syntactic shortcut for the class expression DataMaxCardinality(0 DPE DataComplementOf(DR)).

DataAllValuesFrom := 'DataAllValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

Example:

Consider the ontology consisting of the following axioms.

DataPropertyAssertion(*a:hasZIP* _:a1 "02903"^^*xsd:integer*)	The ZIP code of _:a1 is the integer 02903.
FunctionalDataProperty(*a:hasZIP*)	Each object can have at most one ZIP code.

In United Kingdom and Canada, ZIP codes are strings (i.e., they can contain characters and not just numbers). Hence, one might use the following universal expression to identify those individuals that have only integer ZIP codes (and therefore have non-UK and non-Canadian addresses):

```
DataAllValuesFrom( a:hasZIP xsd:integer )
```

The anonymous individual _:a1 is by the first axiom connected by *a:hasZIP* to an integer, and the second axiom ensures that _:a1 is not connected by *a:hasZIP* to other literals; therefore, _:a1 is classified as an instance of the mentioned class expression.

The last axiom — the one stating that *a:hasZIP* is functional — is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology is not guaranteed to list all literals that _:a1 is connected to by *a:hasZIP*; hence, without this axiom _:a1 would not be classified as an instance of the mentioned class expression.

8.4.3 Literal Value Restriction

A has-value class expression `DataHasValue( DPE lt )` consists of a data property expression `DPE` and a literal `lt`, and it contains all those individuals that are connected by `DPE` to `lt`. Each such class expression can be seen as a syntactic shortcut for the class expression `DataSomeValuesFrom( DPE DataOneOf( lt ) )`.

DataHasValue := 'DataHasValue' '(' **DataPropertyExpression** **Literal** ')'

Example:

Consider the ontology consisting of the following axiom.

`DataPropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )`	Meg is seventeen years old.

The following has-value expression contains all individuals that are connected by *a:hasAge* to the integer 17; furthermore, *a:Meg* is classified as its instance:

```
DataHasValue( a:hasAge "17"^^xsd:integer )
```

8.5 Data Property Cardinality Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on the cardinality of data property expressions, as shown in Figure 11. These are similar to the restrictions on the cardinality of object property expressions. All cardinality restrictions can be qualified or unqualified: in the former case, the cardinality restriction only applies to literals that are connected by the data property expression and are in the qualifying data range; in the latter case it applies to all literals that are connected by the data property expression (this is equivalent to the qualified case with the qualifying data range equal to *rdfs:Literal*). The class expressions **DataMinCardinality**, **DataMaxCardinality**, and **DataExactCardinality** contain those individuals that are connected by a data property expression to at least, at most, and exactly a given number of literals in the specified data range, respectively.

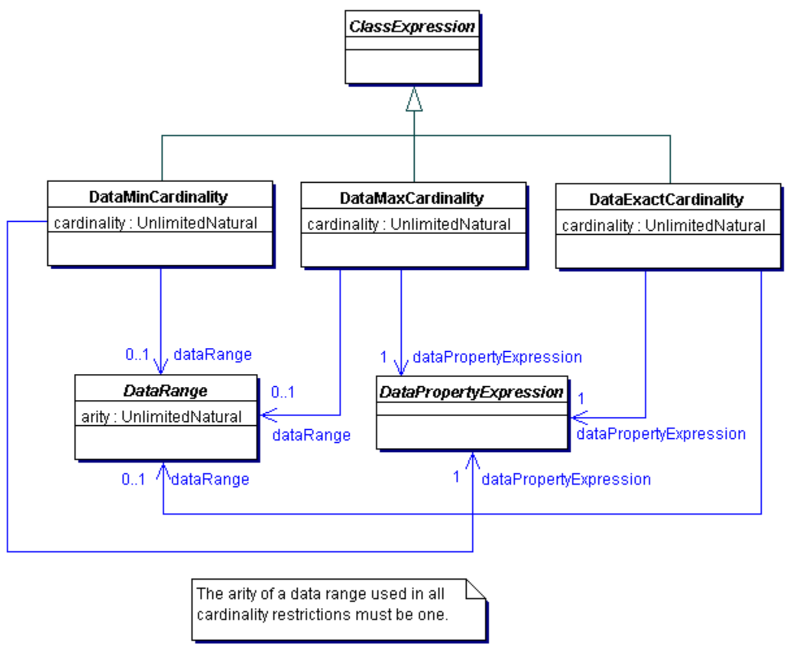

Figure 11. Restricting the Cardinality of Data Property Expressions in OWL 2

8.5.1 Minimum Cardinality

A minimum cardinality expression `DataMinCardinality( n DPE DR )` consists of a nonnegative integer `n`, a data property expression `DPE`, and a unary data range `DR`, and it contains all those individuals that are connected by `DPE` to at least `n` different literals in `DR`. If `DR` is not present, it is taken to be *rdfs:Literal*.

DataMinCardinality := 'DataMinCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Example:

Consider the ontology consisting of the following axioms.

`DataPropertyAssertion( a:hasName a:Meg "Meg Griffin" )`	Meg's name is "Meg Griffin".
`DataPropertyAssertion( a:hasName a:Meg "Megan Griffin" )`	Meg's name is "Megan Griffin".

The following minimum cardinality expression contains those individuals that are connected by *a:hasName* to at least two different literals:

```
DataMinCardinality( 2 a:hasName )
```

Different string literals are distinct, so "Meg Griffin" and "Megan Griffin" are different; thus, the individual *a:Meg* is classified as an instance of the mentioned class expression.

Note that some datatypes from the OWL 2 datatype map distinguish between equal and identical data values, and that the semantics of cardinality restrictions in OWL 2 is defined with respect to the latter. For an example demonstrating the effects such such a definition, please refer to Section 9.3.6.

8.5.2 Maximum Cardinality

A maximum cardinality expression `DataMaxCardinality( n DPE DR )` consists of a nonnegative integer n, a data property expression DPE, and a unary data range DR, and it contains all those individuals that are connected by DPE to at most n different literals in DR. If DR is not present, it is taken to be *rdfs:Literal*.

DataMaxCardinality := 'DataMaxCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Example:

Consider the ontology consisting of the following axiom.

`FunctionalDataProperty( a:hasName )`	Each object can have at most one name.

The following maximum cardinality expression contains those individuals that are connected by *a:hasName* to at most two different literals:

```
DataMaxCardinality( 2 a:hasName )
```

Since the ontology axiom restricts *a:hasName* to be functional, all individuals in the ontology are instances of this class expression.

Note that some datatypes from the OWL 2 datatype map distinguish between equal and identical data values, and that the semantics of cardinality restrictions in OWL 2 is defined with respect to the latter. For an example demonstrating the effects such such a definition, please refer to Section 9.3.6.

8.5.3 Exact Cardinality

An exact cardinality expression `DataExactCardinality( n DPE DR )` consists of a nonnegative integer n, a data property expression DPE, and a unary data range DR, and it contains all those individuals that are connected by DPE to exactly n different literals in DR. If DR is not present, it is taken to be *rdfs:Literal*.

DataExactCardinality := 'DataExactCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Example:

Consider the ontology consisting of the following axioms.

`DataPropertyAssertion( a:hasName a:Brian "Brian Griffin" )`	Brian's name is "Brian Griffin".
`FunctionalDataProperty( a:hasName )`	Each object can have at most one name.

The following exact cardinality expression contains those individuals that are connected by *a:hasName* to exactly one literal:

```
DataExactCardinality( 1 a:hasName )
```

Since the ontology axiom restricts *a:hasName* to be functional and *a:Brian* is connected by *a:hasName* to "Brian Griffin", it is classified as an instance of this class expression.

Note that some datatypes from the OWL 2 datatype map distinguish between equal and identical data values, and that the semantics of cardinality restrictions in OWL 2 is defined with respect to the latter. For an example demonstrating the effects such such a definition, please refer to Section 9.3.6.

9 Axioms

The main component of an OWL 2 ontology is a set of *axioms* — statements that say what is true in the domain. OWL 2 provides an extensive set of axioms, all of which extend the **Axiom** class in the structural specification. As shown in Figure 12, axioms in OWL 2 can be declarations, axioms about classes, axioms about object or data properties, datatype definitions, keys, assertions (sometimes also called *facts*), and axioms about annotations.

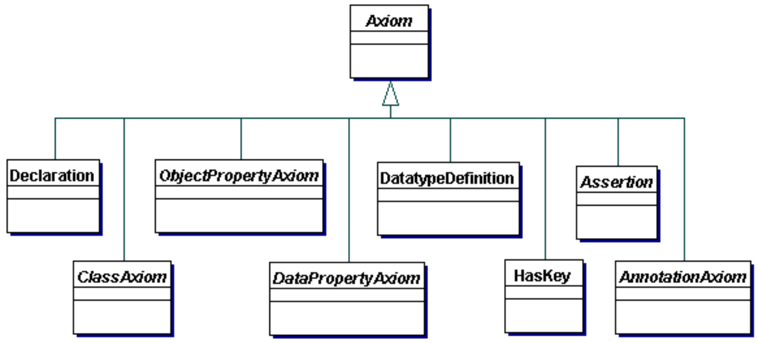

Figure 12. The Axioms of OWL 2

Axiom := **Declaration** | **ClassAxiom** | **ObjectPropertyAxiom** | **DataPropertyAxiom** | **DatatypeDefinition** | **HasKey** | **Assertion** | **AnnotationAxiom**

axiomAnnotations := { **Annotation** }

As shown in Figure 1, OWL 2 axioms can contain axiom annotations, the structure of which is defined in Section 10. Axiom annotations have no effect on the semantics of axioms — that is, they do not affect the logical consequences of OWL 2 ontologies. In contrast, axiom annotations do affect structural equivalence: axioms will not be structurally equivalent if their axiom annotations are not structurally equivalent.

Example:

The following axiom contains a comment that explains the purpose of the axiom.

```
SubClassOf( Annotation( rdfs:comment "Male people are people." ) a:Man a:Person )
```

Since annotations affect structural equivalence between axioms, the previous axiom is not structurally equivalent with the following axiom, even though these two axioms are semantically equivalent.

```
SubClassOf( a:Man a:Person )
```

9.1 Class Expression Axioms

OWL 2 provides axioms that allow relationships to be established between class expressions, as shown in Figure 13. The **SubClassOf** axiom allows one to state that each instance of one class expression is also an instance of another class expression, and thus to construct a hierarchy of classes. The **EquivalentClasses** axiom allows one to state that several class expressions are equivalent to each other. The **DisjointClasses** axiom allows one to state that several class expressions are pairwise disjoint — that is, that they have no instances in common. Finally, the **DisjointUnion** class expression allows one to define a class as a disjoint union of several class expressions and thus to express covering constraints.

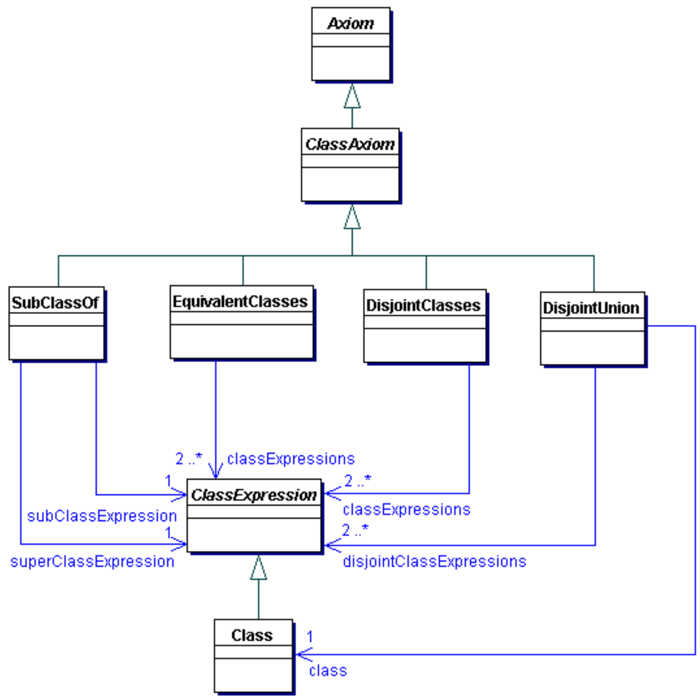

Figure 13. The Class Axioms of OWL 2

ClassAxiom := **SubClassOf** | **EquivalentClasses** | **DisjointClasses** | **DisjointUnion**

9.1.1 Subclass Axioms

A subclass axiom SubClassOf(CE_1 CE_2) states that the class expression CE_1 is a subclass of the class expression CE_2. Roughly speaking, this states that CE_1 is more specific than CE_2. Subclass axioms are a fundamental type of axioms in OWL 2 and can be used to construct a class hierarchy. Other kinds of class expression axiom can be seen as syntactic shortcuts for one or more subclass axioms.

SubClassOf := 'SubClassOf' '(' **axiomAnnotations** **subClassExpression** **superClassExpression** ')'
subClassExpression := **ClassExpression**
superClassExpression := **ClassExpression**

Example:

Consider the ontology consisting of the following axioms.

SubClassOf(*a:Baby a:Child*)	Each baby is a child.
SubClassOf(*a:Child a:Person*)	Each child is a person.
ClassAssertion(*a:Baby a:Stewie*)	Stewie is a baby.

Since *a:Stewie* is an instance of *a:Baby*, by the first subclass axiom *a:Stewie* is classified as an instance of *a:Child* as well. Similarly, by the second subclass axiom *a:Stewie* is classified as an instance of *a:Person*. This style of reasoning can be applied to any instance of *a:Baby* and not just *a:Stewie*; therefore, one can conclude that *a:Baby* is a subclass of *a:Person*. In other words, this ontology entails the following axiom:

```
SubClassOf( a:Baby a:Person )
```

Example:

Consider the ontology consisting of the following axioms.

SubClassOf(*a:PersonWithChild* ObjectSomeValuesFrom(*a:hasChild* ObjectUnionOf(*a:Boy a:Girl*)))	A person that has a child has either at least one boy or a girl.
SubClassOf(*a:Boy a:Child*)	Each boy is a child.
SubClassOf(*a:Girl a:Child*)	Each girl is a child.
SubClassOf(ObjectSomeValuesFrom(*a:hasChild a:Child*) *a:Parent*)	If some object has a child, then this object is a parent.

The first axiom states that each instance of *a:PersonWithChild* is connected to an individual that is an instance of either *a:Boy* or *a:Girl*. (Because of the open-world semantics of OWL 2, this does not mean that there must be only one such individual or that all such individuals must be instances of either *a:Boy* or of *a:Girl*.) Furthermore, each instance of *a:Boy* or *a:Girl* is an instance of *a:Child*. Finally, the last axiom says that all individuals that are connected by *a:hasChild* to an instance of *a:Child* are instances of *a:Parent*. Since this reasoning holds for each instance of *a:PersonWithChild*, each such instance is also an instance of *a:Parent*. In other words, this ontology entails the following axiom:

```
SubClassOf( a:PersonWithChild a:Parent )
```

9.1.2 Equivalent Classes

An equivalent classes axiom EquivalentClasses(CE_1 ... CE_n) states that all of the class expressions CE_i, $1 \le i \le n$, are semantically equivalent to each other. This axiom allows one to use each CE_i as a synonym for each CE_j — that is, in any expression in the ontology containing such an axiom, CE_i can be replaced with CE_j without affecting the meaning of the ontology. An axiom EquivalentClasses(CE_1 CE_2) is equivalent to the following two axioms:

```
SubClassOf( CE1 CE2 )
SubClassOf( CE2 CE1 )
```

Axioms of the form EquivalentClasses(C CE), where C is a class and CE is a class expression, are often called *definitions*, because they define the class C in terms of the class expression CE.

EquivalentClasses := 'EquivalentClasses' '(' **axiomAnnotations** **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

EquivalentClasses(*a:Boy* ObjectIntersectionOf(*a:Child a:Man*))	A boy is a male child.
ClassAssertion(*a:Child a:Chris*)	Chris is a child.
ClassAssertion(*a:Man a:Chris*)	Chris is a man.
ClassAssertion(*a:Boy a:Stewie*)	Stewie is a boy.

The first axiom defines the class *a:Boy* as an intersection of the classes *a:Child* and *a:Man*; thus, the instances of *a:Boy* are exactly those instances that are both an instance of *a:Child* and an instance of *a:Man*. Such a definition consists of two directions. The first direction implies that each instance of *a:Child* and *a:Man* is an instance of *a:Boy*; since *a:Chris* satisfies these two conditions, it is classified as an instance of *a:Boy*. The second direction implies that each *a:Boy* is an instance of *a:Child* and of *a:Man*; thus, *a:Stewie* is classified as an instance of *a:Man* and of *a:Boy*.

Example:

Consider the ontology consisting of the following axioms.

EquivalentClasses(*a:MongrelOwner* ObjectSomeValuesFrom(*a:hasPet a:Mongrel*))	A mongrel owner has a pet that is a mongrel.
EquivalentClasses(*a:DogOwner* ObjectSomeValuesFrom(*a:hasPet a:Dog*))	A dog owner has a pet that is a dog.
SubClassOf(*a:Mongrel a:Dog*)	Each mongrel is a dog.
ClassAssertion(*a:MongrelOwner a:Peter*)	Peter is a mongrel owner.

By the first axiom, each instance x of *a:MongrelOwner* must be connected via *a:hasPet* to an instance of *a:Mongrel*; by the third axiom, this individual is an instance of *a:Dog*; thus, by the second axiom, x is an instance of *a:DogOwner*. In other words, this ontology entails the following axiom:

```
SubClassOf( a:MongrelOwner a:DogOwner )
```

By the fourth axiom, *a:Peter* is then classified as an instance of *a:DogOwner*.

9.1.3 Disjoint Classes

A disjoint classes axiom DisjointClasses(CE_1 ... CE_n) states that all of the class expressions CE_i, $1 \le i \le n$, are pairwise disjoint; that is, no individual can be at the same time an instance of both CE_i and CE_j for $i \ne j$. An axiom DisjointClasses(CE_1 CE_2) is equivalent to the following axiom:

```
SubClassOf( CE1 ObjectComplementOf( CE2 ) )
```

DisjointClasses := 'DisjointClasses' '(' **axiomAnnotations** **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

DisjointClasses(*a:Boy a:Girl*)	Nothing can be both a boy and a girl.
ClassAssertion(*a:Boy a:Stewie*)	Stewie is a boy.

The axioms in this ontology imply that *a:Stewie* can be classified as an instance of the following class expression:

```
ObjectComplementOf( a:Girl )
```

Furthermore, if the ontology were extended with the following assertion, the ontology would become inconsistent:

```
ClassAssertion( a:Girl a:Stewie )
```

9.1.4 Disjoint Union of Class Expressions

A disjoint union axiom DisjointUnion(C CE_1 ... CE_n) states that a class C is a disjoint union of the class expressions CE_i, $1 \le i \le n$, all of which are pairwise disjoint. Such axioms are sometimes referred to as *covering* axioms, as they state that the extensions of all CE_i exactly cover the extension of C. Thus, each instance of C is an instance of exactly one CE_i, and each instance of CE_i is an instance of C. Each such axiom can be seen as a syntactic shortcut for the following two axioms:

```
EquivalentClasses( C ObjectUnionOf( CE1 ... CEn ) )
DisjointClasses( CE1 ... CEn )
```

DisjointUnion := 'DisjointUnion' '(' **axiomAnnotations** **Class** **disjointClassExpressions** ')'
disjointClassExpressions := **ClassExpression** **ClassExpression** { **ClassExpression** }

Example:

Consider the ontology consisting of the following axioms.

DisjointUnion(*a:Child a:Boy a:Girl*)	Each child is either a boy or a girl, each boy is a child, each girl is a child, and nothing can be both a boy and a girl.
ClassAssertion(*a:Child a:Stewie*)	Stewie is a child.
ClassAssertion(ObjectComplementOf(*a:Girl*) *a:Stewie*)	Stewie is not a girl.

By the first two axioms, *a:Stewie* is either an instance of *a:Boy* or *a:Girl*. The last assertion eliminates the second possibility, so *a:Stewie* is classified as an instance of *a:Boy*.

9.2 Object Property Axioms

OWL 2 provides axioms that can be used to characterize and establish relationships between object property expressions. For clarity, the structure of these axioms is shown in two separate figures, Figure 14 and Figure 15. The **SubObjectPropertyOf** axiom allows one to state that the extension of one object property expression is included in the extension of another object property expression. The **EquivalentObjectProperties** axiom allows one to state that the extensions of several object property expressions are the same. The **DisjointObjectProperties** axiom allows one to state that the extensions of several object property expressions are pairwise disjoint — that is, that they do not share pairs of connected individuals. The **InverseObjectProperties** axiom can be used to state that two object property expressions are the inverse of each other. The **ObjectPropertyDomain** and **ObjectPropertyRange** axioms can be used to restrict the first and the second individual, respectively, connected by an object property expression to be instances of the specified class expression.

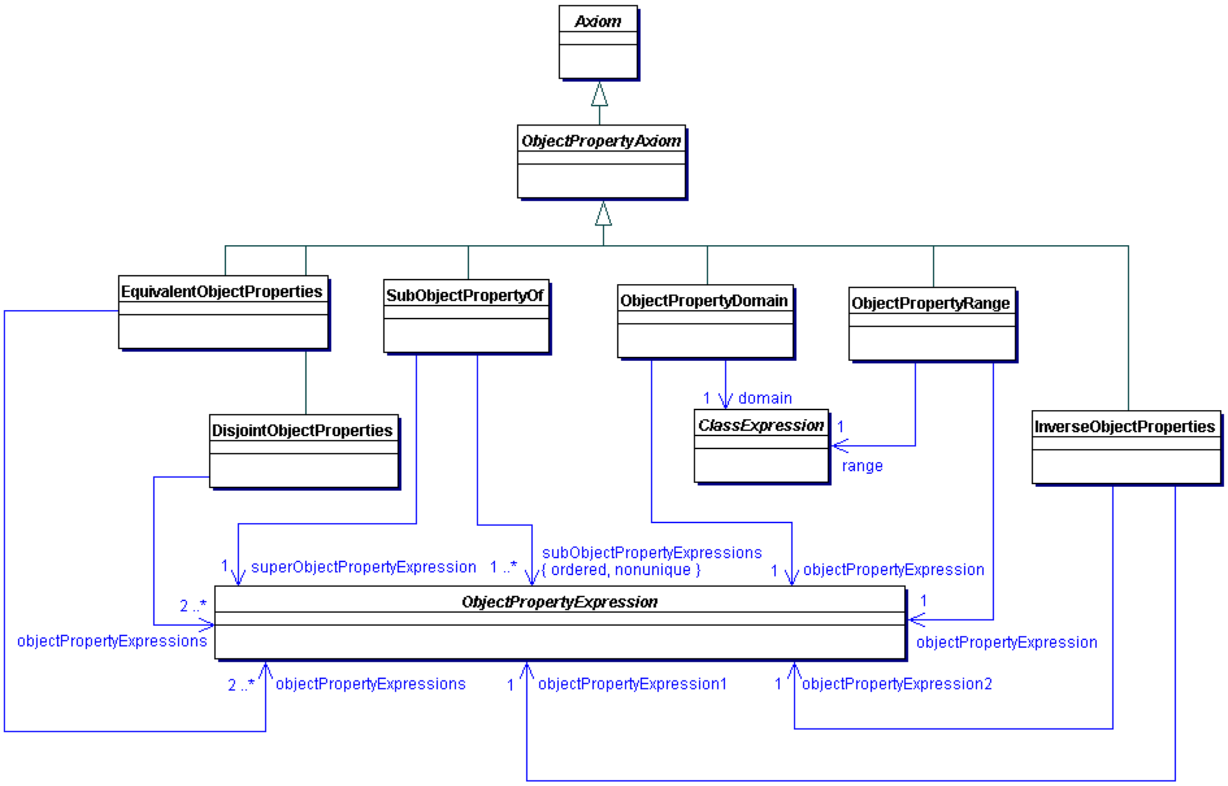

Figure 14. Object Property Axioms in OWL 2, Part I

The **FunctionalObjectProperty** axiom allows one to state that an object property expression is functional — that is, that each individual can have at most one outgoing connection of the specified object property expression. The **InverseFunctionalObjectProperty** axiom allows one to state that an object property expression is inverse-functional — that is, that each individual can have at most one incoming connection of the specified object property expression. Finally, the **ReflexiveObjectProperty**, **IrreflexiveObjectProperty**, **SymmetricObjectProperty**, **AsymmetricObjectProperty**, and **TransitiveObjectProperty** axioms allow one to state that an object property expression is reflexive, irreflexive, symmetric, asymmetric, or transitive, respectively.

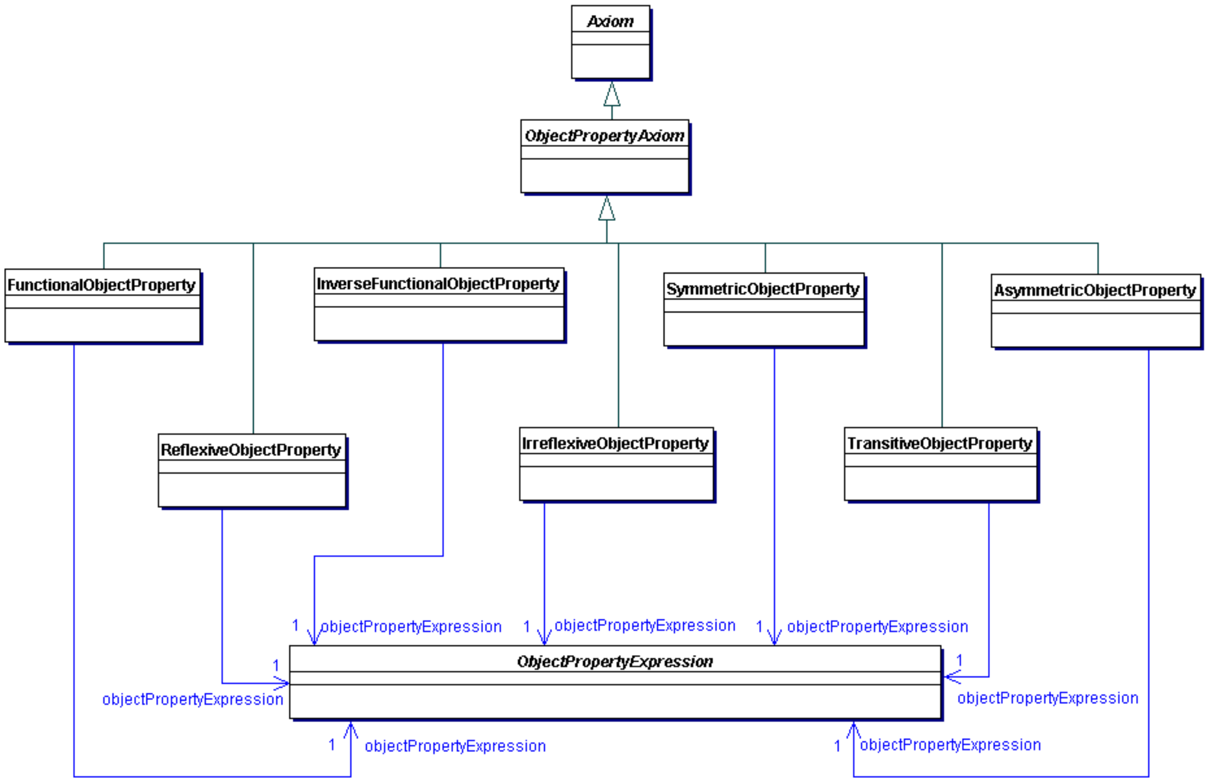

Figure 15. Axioms Defining Characteristics of Object Properties in OWL 2, Part II

ObjectPropertyAxiom :=
  SubObjectPropertyOf | **EquivalentObjectProperties** |
  DisjointObjectProperties | **InverseObjectProperties** |
  ObjectPropertyDomain | **ObjectPropertyRange** |
  FunctionalObjectProperty | **InverseFunctionalObjectProperty** |
  ReflexiveObjectProperty | **IrreflexiveObjectProperty** |
  SymmetricObjectProperty | **AsymmetricObjectProperty** |
  TransitiveObjectProperty

9.2.1 Object Subproperties

Object subproperty axioms are analogous to subclass axioms, and they come in two forms.

The basic form is `SubObjectPropertyOf( OPE₁ OPE₂ )`. This axiom states that the object property expression OPE_1 is a subproperty of the object property expression OPE_2 — that is, if an individual x is connected by OPE_1 to an individual y, then x is also connected by OPE_2 to y.

The more complex form is `SubObjectPropertyOf( ObjectPropertyChain( OPE₁ ... OPEₙ ) OPE )`. This axiom states that, if an individual x is connected by a sequence of object property expressions OPE_1, ..., OPE_n with an individual y, then x is also connected with y by the object property expression OPE. Such axioms are also known as *complex role inclusions* [*SROIQ*].

SubObjectPropertyOf := 'SubObjectPropertyOf' '(' **axiomAnnotations** **subObjectPropertyExpression** **superObjectPropertyExpression** ')'
subObjectPropertyExpression := **ObjectPropertyExpression** | **propertyExpressionChain**
propertyExpressionChain := 'ObjectPropertyChain' '(' **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'
superObjectPropertyExpression := **ObjectPropertyExpression**

Example:

Consider the ontology consisting of the following axioms.

SubObjectPropertyOf(*a:hasDog a:hasPet*)	Having a dog implies having a pet.
ObjectPropertyAssertion(*a:hasDog a:Peter a:Brian*)	Brian is a dog of Peter.

Since *a:hasDog* is a subproperty of *a:hasPet*, each tuple of individuals connected by the former property expression is also connected by the latter property expression. Therefore, this ontology entails that *a:Peter* is connected to *a:Brian* by *a:hasPet*; that is, the ontology entails the following assertion:

```
ObjectPropertyAssertion( a:hasPet a:Peter a:Brian )
```

Example:

Consider the ontology consisting of the following axioms.

`SubObjectPropertyOf( ObjectPropertyChain( a:hasMother a:hasSister ) a:hasAunt )`	The sister of someone's mother is that person's aunt.
`ObjectPropertyAssertion( a:hasMother a:Stewie a:Lois )`	Lois is the mother of Stewie.
`ObjectPropertyAssertion( a:hasSister a:Lois a:Carol )`	Carol is a sister of Lois.

The axioms in this ontology imply that *a:Stewie* is connected by *a:hasAunt* with *a:Carol*; that is, the ontology entails the following assertion:

```
ObjectPropertyAssertion( a:hasAunt a:Stewie a:Carol )
```

9.2.2 Equivalent Object Properties

An equivalent object properties axiom EquivalentObjectProperties(OPE_1 ... OPE_n) states that all of the object property expressions OPE_i, $1 \le i \le n$, are semantically equivalent to each other. This axiom allows one to use each OPE_i as a synonym for each OPE_j — that is, in any expression in the ontology containing such an axiom, OPE_i can be replaced with OPE_j without affecting the meaning of the ontology. The axiom EquivalentObjectProperties(OPE_1 OPE_2) is equivalent to the following two axioms:

```
SubObjectPropertyOf( OPE1 OPE2 )
SubObjectPropertyOf( OPE2 OPE1 )
```

EquivalentObjectProperties := 'EquivalentObjectProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`EquivalentObjectProperties( a:hasBrother a:hasMaleSibling )`	Having a brother is the same as having a male sibling.
`ObjectPropertyAssertion( a:hasBrother a:Chris a:Stewie )`	Stewie is a brother of Chris.
`ObjectPropertyAssertion( a:hasMaleSibling a:Stewie a:Chris )`	Chris is a male sibling of Stewie.

Since *a:hasBrother* and *a:hasMaleSibling* are equivalent properties, this ontology entails that *a:Chris* is connected by *a:hasMaleSibling* with *a:Stewie* — that is, it entails the following assertion:

```
ObjectPropertyAssertion( a:hasMaleSibling a:Chris a:Stewie )
```

Furthermore, the ontology also entails that that *a:Stewie* is connected by *a:hasBrother* with *a:Chris* — that is, it entails the following assertion:

```
ObjectPropertyAssertion( a:hasBrother a:Stewie a:Chris )
```

9.2.3 Disjoint Object Properties

A disjoint object properties axiom DisjointObjectProperties(OPE_1 ... OPE_n) states that all of the object property expressions OPE_i, $1 \le i \le n$, are pairwise disjoint; that is, no individual x can be connected to an individual y by both OPE_i and OPE_j for $i \ne j$.

DisjointObjectProperties := 'DisjointObjectProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`DisjointObjectProperties( a:hasFather a:hasMother )`	Fatherhood is disjoint with motherhood.
`ObjectPropertyAssertion( a:hasFather a:Stewie a:Peter )`	Peter is Stewie's father.
`ObjectPropertyAssertion( a:hasMother a:Stewie a:Lois )`	Lois is the mother of Stewie.

In this ontology, the disjointness axiom is satisfied. If, however, one were to add the following assertion, the disjointness axiom would be invalidated and the ontology would become inconsistent:

```
ObjectPropertyAssertion( a:hasMother a:Stewie a:Peter )
```

9.2.4 Inverse Object Properties

An inverse object properties axiom InverseObjectProperties(OPE_1 OPE_2) states that the object property expression OPE_1 is an inverse of the object property expression OPE_2. Thus, if an individual x is connected by OPE_1 to an individual y, then y is also connected by OPE_2 to x, and vice versa. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
EquivalentObjectProperties( OPE1 ObjectInverseOf( OPE2 ) )
```

InverseObjectProperties := 'InverseObjectProperties' '(' **axiomAnnotations ObjectPropertyExpression ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`InverseObjectProperties( a:hasFather a:fatherOf )`	Having a father is the opposite of being a father of someone.
`ObjectPropertyAssertion( a:hasFather a:Stewie a:Peter )`	Peter is Stewie's father.
`ObjectPropertyAssertion( a:fatherOf a:Peter a:Chris )`	Peter is Chris's father.

This ontology entails that *a:Peter* is connected by *a:fatherOf* with *a:Stewie* — that is, it entails the following assertion:

```
ObjectPropertyAssertion( a:fatherOf a:Peter a:Stewie )
```

Furthermore, the ontology also entails that *a:Chris* is connected by *a:hasFather* with *a:Peter* — that is, it entails the following assertion:

```
ObjectPropertyAssertion( a:hasFather a:Chris a:Peter )
```

9.2.5 Object Property Domain

An object property domain axiom `ObjectPropertyDomain( OPE CE )` states that the domain of the object property expression OPE is the class expression CE — that is, if an individual x is connected by OPE with some other individual, then x is an instance of CE. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( ObjectSomeValuesFrom( OPE owl:Thing ) CE )
```

ObjectPropertyDomain := 'ObjectPropertyDomain' '(' **axiomAnnotations ObjectPropertyExpression ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyDomain( a:hasDog a:Person )`	Only people can own dogs.
`ObjectPropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.

By the first axiom, each individual that has an outgoing *a:hasDog* connection must be an instance of *a:Person*. Therefore, *a:Peter* can be classified as an instance of *a:Person*; that is, this ontology entails the following assertion:

```
ClassAssertion( a:Person a:Peter )
```

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The domain axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasDog* can point only from individuals that are known to be instances of *a:Person*; furthermore, since the example ontology does not explicitly state that *a:Peter* is an instance of *a:Person*, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the domain constraint.

9.2.6 Object Property Range

An object property range axiom `ObjectPropertyRange( OPE CE )` states that the range of the object property expression OPE is the class expression CE — that is, if some individual is connected by OPE with an individual x, then x is an instance of CE. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing ObjectAllValuesFrom( OPE CE ) )
```

ObjectPropertyRange := 'ObjectPropertyRange' '(' **axiomAnnotations ObjectPropertyExpression ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyRange( a:hasDog a:Dog )`	The range of the *a:hasDog* property is the class *a:Dog*.
`ObjectPropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.

By the first axiom, each individual that has an incoming *a:hasDog* connection must be an instance of *a:Dog*. Therefore, *a:Brian* can be classified as an instance of *a:Dog*; that is, this ontology entails the following assertion:

```
ClassAssertion( a:Dog a:Brian )
```

Range axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The range axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasDog* can point only to individuals that are known to be instances of *a:Dog*; furthermore, since the example ontology does not explicitly state that *a:Brian* is an instance of *a:Dog*, one might expect the range constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the range constraint.

9.2.7 Functional Object Properties

An object property functionality axiom `FunctionalObjectProperty( OPE )` states that the object property expression `OPE` is functional — that is, for each individual x, there can be at most one distinct individual y such that x is connected by `OPE` to y. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing ObjectMaxCardinality( 1 OPE ) )
```

FunctionalObjectProperty := 'FunctionalObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`FunctionalObjectProperty( a:hasFather )`	Each object can have at most one father.
`ObjectPropertyAssertion( a:hasFather a:Stewie a:Peter )`	Peter is Stewie's father.
`ObjectPropertyAssertion( a:hasFather a:Stewie a:Peter_Griffin )`	Peter Griffin is Stewie's father.

By the first axiom, *a:hasFather* can point from *a:Stewie* to at most one distinct individual, so *a:Peter* and *a:Peter_Griffin* must be equal; that is, this ontology entails the following assertion:

```
SameIndividual( a:Peter a:Peter_Griffin )
```

One might expect the previous ontology to be inconsistent, since the *a:hasFather* property points to two different values for *a:Stewie*. OWL 2, however, does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

```
DifferentIndividuals( a:Peter a:Peter_Griffin )
```

9.2.8 Inverse-Functional Object Properties

An object property inverse functionality axiom `InverseFunctionalObjectProperty( OPE )` states that the object property expression `OPE` is inverse-functional — that is, for each individual x, there can be at most one individual y such that y is connected by `OPE` with x. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing ObjectMaxCardinality( 1 ObjectInverseOf( OPE ) ) )
```

InverseFunctionalObjectProperty := 'InverseFunctionalObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`InverseFunctionalObjectProperty( a:fatherOf )`	Each object can have at most one father.
`ObjectPropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is Stewie's father.
`ObjectPropertyAssertion( a:fatherOf a:Peter_Griffin a:Stewie )`	Peter Griffin is Stewie's father.

By the first axiom, at most one distinct individual can point by *a:fatherOf* to *a:Stewie*, so *a:Peter* and *a:Peter_Griffin* must be equal; that is, this ontology entails the following assertion:

```
SameIndividual( a:Peter a:Peter_Griffin )
```

One might expect the previous ontology to be inconsistent, since there are two individuals that *a:Stewie* is connected to by *a:fatherOf*. OWL 2, however, does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

```
DifferentIndividuals( a:Peter a:Peter_Griffin )
```

9.2.9 Reflexive Object Properties

An object property reflexivity axiom `ReflexiveObjectProperty( OPE )` states that the object property expression `OPE` is reflexive — that is, each individual is connected by `OPE` to itself. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing ObjectHasSelf( OPE ) )
```

ReflexiveObjectProperty := 'ReflexiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ReflexiveObjectProperty( a:knows )`	Everybody knows themselves.
`ClassAssertion( a:Person a:Peter )`	Peter is a person.

By the first axiom, *a:Peter* must be connected by *a:knows* to itself; that is, this ontology entails the following assertion:

```
ObjectPropertyAssertion( a:knows a:Peter a:Peter )
```

9.2.10 Irreflexive Object Properties

An object property irreflexivity axiom `IrreflexiveObjectProperty( OPE )` states that the object property expression `OPE` is irreflexive — that is, no individual is connected by `OPE` to itself. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( ObjectHasSelf( OPE ) owl:Nothing )
```

IrreflexiveObjectProperty := 'IrreflexiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`IrreflexiveObjectProperty( a:marriedTo )`	Nobody can be married to themselves.

If this ontology were extended with the following assertion, the irreflexivity axiom would be contradicted and the ontology would become inconsistent:

```
ObjectPropertyAssertion( a:marriedTo a:Peter a:Peter )
```

9.2.11 Symmetric Object Properties

An object property symmetry axiom `SymmetricObjectProperty( OPE )` states that the object property expression `OPE` is symmetric — that is, if an individual x is connected by `OPE` to an individual y, then y is also connected by `OPE` to x. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubObjectPropertyOf( OPE ObjectInverseOf( OPE ) )
```

SymmetricObjectProperty := 'SymmetricObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`SymmetricObjectProperty( a:friend )`	If x is a friend of y, then y is a friend of x.
`ObjectPropertyAssertion( a:friend a:Peter a:Brian )`	Brian is a friend of Peter.

Since *a:friend* is symmetric, *a:Peter* must be connected by *a:friend* to *a:Brian*; that is, this ontology entails the following assertion:

```
ObjectPropertyAssertion( a:friend a:Brian a:Peter )
```

9.2.12 Asymmetric Object Properties

An object property asymmetry axiom `AsymmetricObjectProperty( OPE )` states that the object property expression `OPE` is asymmetric — that is, if an individual x is connected by `OPE` to an individual y, then y cannot be connected by `OPE` to x.

AsymmetricObjectProperty := 'AsymmetricObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`AsymmetricObjectProperty( a:parentOf )`	If x is a parent of y, then y is not a parent of x.
`ObjectPropertyAssertion( a:parentOf a:Peter a:Stewie )`	Peter is a parent of Stewie.

If this ontology were extended with the following assertion, the asymmetry axiom would be invalidated and the ontology would become inconsistent:

```
ObjectPropertyAssertion( a:parentOf a:Stewie a:Peter )
```

9.2.13 Transitive Object Properties

An object property transitivity axiom `TransitiveObjectProperty( OPE )` states that the object property expression `OPE` is transitive — that is, if an individual x is connected by `OPE` to an individual y that is connected by `OPE` to an individual z, then x is also connected by `OPE` to z. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubObjectPropertyOf( ObjectPropertyChain( OPE OPE ) OPE )
```

TransitiveObjectProperty := 'TransitiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`TransitiveObjectProperty( a:ancestorOf )`	If x is an ancestor of y and y is an ancestor of z, then x is an ancestor of z.
`ObjectPropertyAssertion( a:ancestorOf a:Carter a:Lois )`	Carter is an ancestor of Lois.
`ObjectPropertyAssertion( a:ancestorOf a:Lois a:Meg )`	Lois is an ancestor of Meg.

Since *a:ancestorOf* is transitive, *a:Carter* must be connected by *a:ancestorOf* to *a:Meg* — that is, this ontology entails the following assertion:

```
ObjectPropertyAssertion( a:ancestorOf a:Carter a:Meg )
```

9.3 Data Property Axioms

OWL 2 also provides for data property axioms. Their structure is similar to object property axioms, as shown in Figure 16. The **SubDataPropertyOf** axiom allows one to state that the extension of one data property expression is included in the extension of another data property expression. The **EquivalentDataProperties** allows one to state that several data property expressions have the same extension. The **DisjointDataProperties** axiom allows one to state that the extensions of several data property expressions are disjoint with each other — that is, they do not share individual–literal pairs. The **DataPropertyDomain** axiom can be used to restrict individuals connected by a property expression to be instances of the specified class; similarly, the **DataPropertyRange** axiom can be used to restrict the literals pointed to by a property expression to be in the specified unary data range. Finally, the **FunctionalDataProperty** axiom allows one to state that a data property expression is functional — that is, that each individual can have at most one outgoing connection of the specified data property expression.

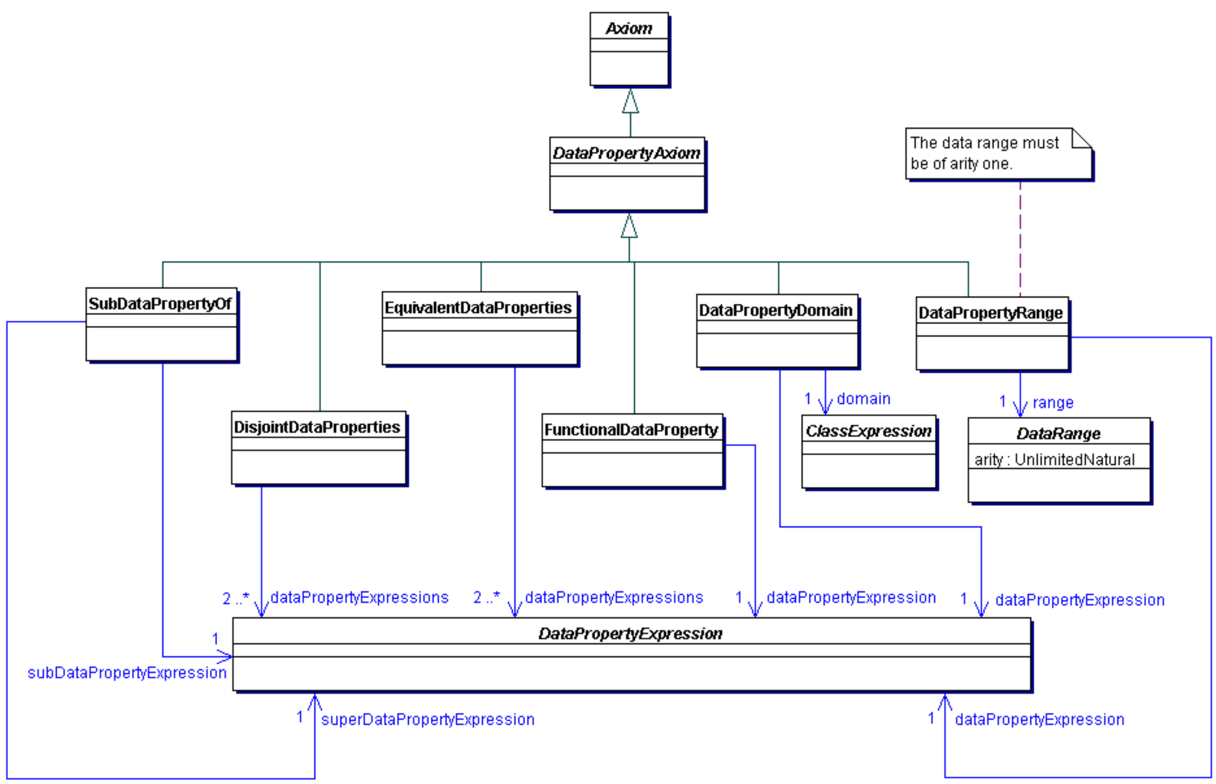

Figure 16. Data Property Axioms of OWL 2

DataPropertyAxiom :=
SubDataPropertyOf | **EquivalentDataProperties** | **DisjointDataProperties** |
DataPropertyDomain | **DataPropertyRange** | **FunctionalDataProperty**

9.3.1 Data Subproperties

A data subproperty axiom SubDataPropertyOf(DPE_1 DPE_2) states that the data property expression DPE_1 is a subproperty of the data property expression DPE_2 — that is, if an individual x is connected by DPE_1 to a literal y, then x is connected by DPE_2 to y as well.

SubDataPropertyOf := 'SubDataPropertyOf' '(' **axiomAnnotations subDataPropertyExpression superDataPropertyExpression** ')'
subDataPropertyExpression := **DataPropertyExpression**
superDataPropertyExpression := **DataPropertyExpression**

Example:

Consider the ontology consisting of the following axioms.

SubDataPropertyOf(*a:hasLastName a:hasName*)	A last name of someone is his/her name as well.
DataPropertyAssertion(*a:hasLastName a:Peter* "Griffin")	Peter's last name is "Griffin".

Since *a:hasLastName* is a subproperty of *a:hasName*, each individual connected by the former property to a literal is also connected by the latter property to the same literal. Therefore, this ontology entails that *a:Peter* is connected to "Griffin" through *a:hasName*; that is, the ontology entails the following assertion:

DataPropertyAssertion(*a:hasName a:Peter* "Griffin")

9.3.2 Equivalent Data Properties

An equivalent data properties axiom EquivalentDataProperties(DPE_1 ... DPE_n) states that all the data property expressions DPE_i, $1 \le i \le n$, are semantically equivalent to each other. This axiom allows one to use each DPE_i as a synonym for each DPE_j — that is, in any expression in the ontology containing such an axiom, DPE_i can be replaced with DPE_j without affecting the meaning of the ontology. The axiom EquivalentDataProperties(DPE_1 DPE_2) can be seen as a syntactic shortcut for the following axiom:

SubDataPropertyOf(DPE_1 DPE_2)
SubDataPropertyOf(DPE_2 DPE_1)

EquivalentDataProperties := 'EquivalentDataProperties' '(' **axiomAnnotations DataPropertyExpression DataPropertyExpression** { **DataPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

EquivalentDataProperties(*a:hasName a:seLlama*)	*a:hasName* and *a:seLlama* (in Spanish) are synonyms.
DataPropertyAssertion(*a:hasName a:Meg* "Meg Griffin")	Meg's name is "Meg Griffin".
DataPropertyAssertion(*a:seLlama a:Meg* "Megan Griffin")	Meg's name is "Megan Griffin".

Since *a:hasName* and *a:seLlama* are equivalent properties, this ontology entails that *a:Meg* is connected by *a:seLlama* with "Meg Griffin" — that is, it entails the following assertion:

DataPropertyAssertion(*a:seLlama a:Meg* "Meg Griffin")

Furthermore, the ontology also entails that *a:Meg* is also connected by *a:hasName* with "Megan Griffin" — that is, it entails the following assertion:

DataPropertyAssertion(*a:hasName a:Meg* "Megan Griffin")

9.3.3 Disjoint Data Properties

A disjoint data properties axiom DisjointDataProperties(DPE_1 ... DPE_n) states that all of the data property expressions DPE_i, $1 \le i \le n$, are pairwise disjoint; that is, no individual x can be connected to a literal y by both DPE_i and DPE_j for $i \ne j$.

DisjointDataProperties := 'DisjointDataProperties' '(' **axiomAnnotations DataPropertyExpression DataPropertyExpression** { **DataPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

DisjointDataProperties(*a:hasName a:hasAddress*)	Someone's name must be different from his address.
DataPropertyAssertion(*a:hasName a:Peter* "Peter Griffin")	Peter's name is "Peter Griffin".
DataPropertyAssertion(*a:hasAddress a:Peter* "Quahog, Rhode Island")	Peter's address is "Quahog, Rhode Island".

In this ontology, the disjointness axiom is satisfied. If, however, one were to add the following assertion, the disjointness axiom would be invalidated and the ontology would become inconsistent:

```
DataPropertyAssertion( a:hasAddress a:Peter "Peter Griffin" )
```

9.3.4 Data Property Domain

A data property domain axiom DataPropertyDomain(DPE CE) states that the domain of the data property expression DPE is the class expression CE — that is, if an individual x is connected by DPE with some literal, then x is an instance of CE. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( DataSomeValuesFrom( DPE rdfs:Literal) CE )
```

DataPropertyDomain := 'DataPropertyDomain' '(' **axiomAnnotations DataPropertyExpression ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`DataPropertyDomain( a:hasName a:Person )`	Only people can have names.
`DataPropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is "Peter Griffin".

By the first axiom, each individual that has an outgoing *a:hasName* connection must be an instance of *a:Person*. Therefore, *a:Peter* can be classified as an instance of *a:Person* — that is, this ontology entails the following assertion:

```
ClassAssertion( a:Person a:Peter )
```

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. Thus, the domain axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasName* can point only from individuals that are known to be instances of *a:Person*; furthermore, since the example ontology does not explicitly state that *a:Peter* is an instance of *a:Person*, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the domain constraint.

9.3.5 Data Property Range

A data property range axiom DataPropertyRange(DPE DR) states that the range of the data property expression DPE is the data range DR — that is, if some individual is connected by DPE with a literal x, then x is in DR. The arity of DR *must* be one. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing DataAllValuesFrom( DPE DR ) )
```

DataPropertyRange := 'DataPropertyRange' '(' **axiomAnnotations DataPropertyExpression DataRange** ')'

Example:

Consider the ontology consisting of the following axioms.

`DataPropertyRange( a:hasName xsd:string )`	The range of the *a:hasName* property is *xsd:string*.
`DataPropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is "Peter Griffin".

By the first axiom, each literal that has an incoming *a:hasName* link must be in *xsd:string*. In the example ontology, this axiom is satisfied. If, however, the ontology were extended with the following assertion, then the range axiom would imply that the literal "42"^^*xsd:integer* is in *xsd:string*, which is a contradiction and the ontology would become inconsistent:

```
DataPropertyAssertion( a:hasName a:Peter "42"^^xsd:integer )
```

9.3.6 Functional Data Properties

A data property functionality axiom FunctionalDataProperty(DPE) states that the data property expression DPE is functional — that is, for each individual x, there can be at most one distinct literal y such that x is connected by DPE with y. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing DataMaxCardinality( 1 DPE ) )
```

FunctionalDataProperty := 'FunctionalDataProperty' '(' **axiomAnnotations DataPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`FunctionalDataProperty( a:hasAge )`	Each object can have at most one age.
`DataPropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )`	Meg is seventeen years old.

By the first axiom, *a:hasAge* can point from *a:Meg* to at most one distinct literal. In this example ontology, this axiom is satisfied. If, however, the ontology were extended with the following assertion, the semantics of functionality axioms would imply that `"15"^^xsd:integer` is equal to `"17"^^xsd:integer`, which is a contradiction and the ontology would become inconsistent:

```
DataPropertyAssertion( a:hasAge a:Meg "15"^^xsd:integer )
```

Example:

Note that some datatypes from the OWL 2 datatype map distinguish between equal and identical data values, and that the semantics of cardinality restrictions and functional data properties in OWL 2 is defined with respect to the latter. Consider the following example:

`FunctionalDataProperty( a:hasAge )`	Each object can have at most one age.
`DataPropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )`	Meg is seventeen years old.
`DataPropertyAssertion( a:hasAge a:Meg "17.0"^^xsd:decimal )`	Meg is seventeen years old.
`DataPropertyAssertion( a:hasAge a:Meg "+17"^^xsd:int )`	Meg is seventeen years old.

Literals `"17"^^xsd:integer`, `"17.0"^^xsd:decimal`, and `"+17"^^xsd:int` are all mapped to the identical data value — the integer 17. Therefore, the individual *a:Meg* is connected by the *a:hasAge* property to one distinct data value, so this ontology is satisfiable.

In contrast, consider the following ontology:

`FunctionalDataProperty( a:numberOfChildren )`	An individual can have at most one value for *a:numberOfChildren*.
`DataPropertyAssertion( a:numberOfChildren a:Meg "+0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *+0*.
`DataPropertyAssertion( a:numberOfChildren a:Meg "-0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *-0*.

Literals `"+0"^^xsd:float` and `"-0"^^xsd:float` are mapped to distinct data values *+0* and *-0* in the value space of *xsf:float*; these data values are equal, but not identical. Therefore, the individual *a:Meg* is connected by the *a:numberOfChildren* property to two distinct data values, which violates the functionality restriction on *a:numberOfChildren* and makes the ontology unsatisfiable.

9.4 Datatype Definitions

A datatype definition `DatatypeDefinition( DT DR )` defines a new datatype `DT` as being semantically equivalent to the data range `DR`; the latter *must* be a unary data range. This axiom allows one to use the *defined* datatype `DT` as a synonym for `DR` — that is, in any expression in the ontology containing such an axiom, `DT` can be replaced with `DR` without affecting the meaning of the ontology. The structure of such axiom is shown in Figure 17.

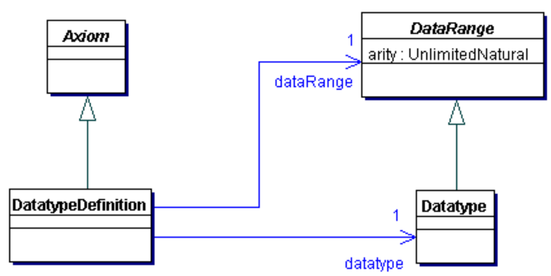

Figure 17. Datatype Definitions in OWL 2

DatatypeDefinition := 'DatatypeDefinition' '(' **axiomAnnotations** **Datatype** **DataRange** ')'

The datatypes defined by datatype definition axioms support no facets so they *must not* occur in datatype restrictions. Furthermore, such datatypes have empty lexical spaces and therefore they *must not* occur in literals. Finally, datatype definitions are not substitutes for declarations: if an OWL 2 ontology is to satisfy the typing constraints of OWL 2 DL from Section 5.8.1, it *must* explicitly declare all datatypes that occur in datatype definitions.

Example:

Consider the ontology consisting of the following axioms.

`Declaration( Datatype( a:SSN ) )`	*a:SSN* is a datatype.
`DatatypeDefinition( a:SSN DatatypeRestriction( xsd:string xsd:pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ) )`	A social security number is a string that matches the given regular expression.
`DataPropertyRange( a:hasSSN a:SSN )`	The range of the *a:hasSSN* property is *a:SSN*.

The second axiom defines *a:SSN* as an abbreviation for a datatype restriction on *xsd:string*. In order to satisfy the typing restrictions from Section 5.8.1, the first axiom explicitly declares *a:SSN* to be a datatype. The datatype *a:SSN* can be used just like any other datatype; for example, it is used in the third axiom to define the range of the *a:hasSSN* property. The only restriction is that *a:SSN* supports no facets and therefore cannot be used in datatype restrictions, and that there can be no literals of datatype *a:SSN*.

9.5 Keys

A key axiom HasKey(CE (OPE_1 ... OPE_m) (DPE_1 ... DPE_n)) states that each (named) instance of the class expression CE is uniquely identified by the object property expressions OPE_i and/or the data property experssions DPE_j — that is, no two distinct (named) instances of CE can coincide on the values of all object property expressions OPE_i and all data property expressions DPE_j. In each such axiom in an OWL ontology, m or n (or both) *must* be larger than zero. A key axiom of the form HasKey(*owl:Thing* (OPE) ()) is similar to the axiom InverseFunctionalObjectProperty(OPE), the main differences being that the former axiom is applicable only to individuals that are explicitly named in an ontology, while the latter axiom is also applicable to anonymous individuals and individuals whose existence is implied by existential quantification. The structure of such axiom is shown in Figure 18.

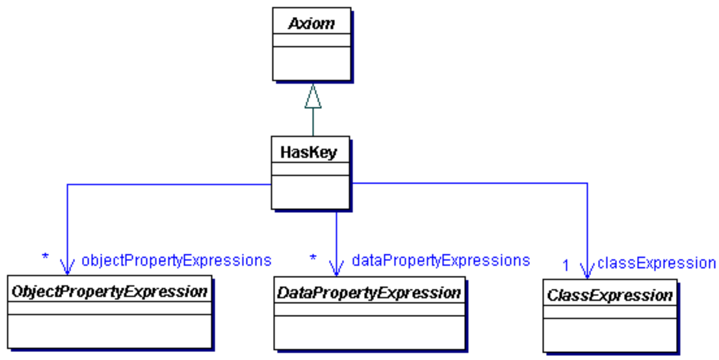

Figure 18. Key Axioms in OWL 2

HasKey := 'HasKey' '(' **axiomAnnotations** **ClassExpression** '(' { **ObjectPropertyExpression** } ')' '(' { **DataPropertyExpression** } ')' ')'

Example:

Consider the ontology consisting of the following axioms.

HasKey(*owl:Thing* () (*a:hasSSN*))	Each object is uniquely identified by its social security number.
DataPropertyAssertion(*a:hasSSN* *a:Peter* "123-45-6789")	Peter's social security number is "123-45-6789".
DataPropertyAssertion(*a:hasSSN* *a:Peter_Griffin* "123-45-6789")	Peter Griffin's social security number is "123-45-6789".

The first axiom makes *a:hasSSN* the key for instances of the *owl:Thing* class; thus, only one individual can have a particular value for *a:hasSSN*. Since the values of *a:hasSSN* are the same for the individuals *a:Peter* and *a:Peter_Griffin*, these two individuals are equal — that is, this ontology entails the following assertion:

SameIndividual(*a:Peter* *a:Peter_Griffin*)

One might expect the previous ontology to be inconsistent, since the *a:hasSSN* has the same value for two individuals *a:Peter* and *a:Peter_Griffin*. However, OWL 2 does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

DifferentIndividuals(*a:Peter* *a:Peter_Griffin*)

Example:

The effect of a key axiom can be "localized" to instances of a particular class expression. Consider the following example:

HasKey(*a:GriffinFamilyMember* () (*a:hasName*))	Each member of the Griffin family is uniquely identified by its name.
DataPropertyAssertion(*a:hasName* *a:Peter* "Peter")	Peter's name is "Peter".
ClassAssertion(*a:GriffinFamilyMember* *a:Peter*)	Peter is a member of the Griffin family.
DataPropertyAssertion(*a:hasName* *a:Peter_Griffin* "Peter")	Peter Griffin's name is "Peter".
ClassAssertion(*a:GriffinFamilyMember* *a:Peter_Griffin*)	Peter Griffin is a member of the Griffin family.
DataPropertyAssertion(*a:hasName* *a:StPeter* "Peter")	St. Peter's name is "Peter".

The effects of the first key axiom are "localized" to the class *a:GriffinFamilyMember* — that is, the data property *a:hasName* uniquely identifies only instances of that class. The individuals *a:Peter* and *a:Peter_Griffin* are instances of *a:GriffinFamilyMember*, so the key axiom implies that *a:Peter* and *a:Peter_Griffin* are the same individuals — that is, the ontology implies the following assertion:

SameIndividual(*a:Peter* *a:Peter_Griffin*)

The individual *a:StPeter*, however, is not an instance of *a:GriffinFamilyMember*, so the key axiom is not applicable to it. Therefore, the ontology implies neither that *a:Peter* and *a:StPeter* are the same individuals, nor does it imply that *a:Peter_Griffin* and *a:StPeter* are the same. Keys can be made global by "localizing" them to the *owl:Thing* class, as shown in the previous example.

Example:

A key axiom does not make all the properties used in it functional. Consider the following example:

`HasKey( a:GriffinFamilyMember () ( a:hasName ) )`	Each member of the Griffin family is uniquely identified by its name.
`DataPropertyAssertion( a:hasName a:Peter "Peter" )`	Peter's name is "Peter".
`DataPropertyAssertion( a:hasName a:Peter "Kichwa-Tembo" )`	Peter's name is "Kichwa-Tembo".
`ClassAssertion( a:GriffinFamilyMember a:Peter )`	Peter is a member of the Griffin family.

This ontology is consistent — that is, the fact that the individual *a:Peter* has two distinct values for *a:hasName* does not cause an inconsistency since the *a:hasName* data property is not necessarily functional.

If desired, the properties used in a key axiom can always be made functional explicitly. Thus, if the example ontology were extended with the following axiom, it would become inconsistent.

```
FunctionalDataProperty( a:hasName )
```

The semantics of key axioms is specific in that these axioms apply only to individuals explicitly introduced in the ontology by name, and not to unnamed individuals (i.e., the individuals whose existence is implied by existential quantification). This makes key axioms equivalent to a variant of DL-safe rules [*DL-Safe*]. Thus, key axioms will typically not affect class-based inferences such as the computation of the subsumption hierarchy, but they will play a role in answering queries about individuals.

Example:

Consider the ontology consisting of the following axioms.

`HasKey( a:Person () ( a:hasSSN ) )`	Each person is uniquely identified by their social security number.
`DataPropertyAssertion( a:hasSSN a:Peter "123-45-6789" )`	Peter's social security number is "123-45-6789".
`ClassAssertion( a:Person a:Peter )`	Peter is a person.
`ClassAssertion( ObjectSomeValuesFrom( a:marriedTo ObjectIntersectionOf( a:Man DataHasValue( a:hasSSN "123-45-6789" ) ) ) a:Lois )`	Lois is married to some man whose social security number is "123-45-6789".
`SubClassOf( a:Man a:Person )`	Each man is a person.

The fourth axiom implies existence of some individual x that is an instance of *a:Man* and whose value for the *a:hasSSN* data property is "123-45-6789"; by the fifth axiom, x is an instance of *a:Person* as well. Furthermore, the second and the third axiom say that *a:Peter* is an instance of *a:Person* and that the value of *a:hasSSN* for *a:Peter* is "123-45-6789". Finally, the first axiom says that *a:hasSSN* is a key property for instances of *a:Person*. Thus, one might expect x to be equal to *a:Peter*, and for the ontology to entail the following assertion:

```
ClassAssertion( a:Man a:Peter )
```

The inferences in the previous paragraph, however, cannot be drawn because of the DL-safe semantics of key axioms: x is an individual that has not been explicitly named in the ontology; therefore, the semantics of key axioms does not apply to x. Therefore, this OWL 2 ontology does not entail the mentioned assertion.

9.6 Assertions

OWL 2 supports a rich set of axioms for stating *assertions* — axioms about individuals that are often also called *facts*. For clarity, different types of assertions are shown in three separate figures, Figure 19, 20, and 21. The **SameIndividual** assertion allows one to state that several individuals are all equal to each other, while the **DifferentIndividuals** assertion allows for the opposite — that is, to state that several individuals are all different from each other. (More precisely, that the several different individuals in the syntax are also semantically different.) The **ClassAssertion** axiom allows one to state that an individual is an instance of a particular class.

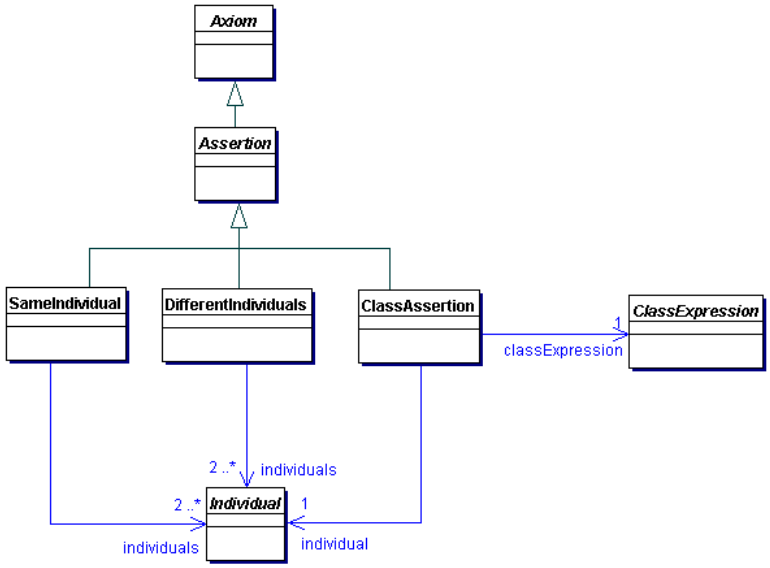

Figure 19. Class and Individual (In)Equality Assertions in OWL 2

The **ObjectPropertyAssertion** axiom allows one to state that an individual is connected by an object property expression to an individual, while **NegativeObjectPropertyAssertion** allows for the opposite — that is, to state that an individual is not connected by an object property expression to an individual.

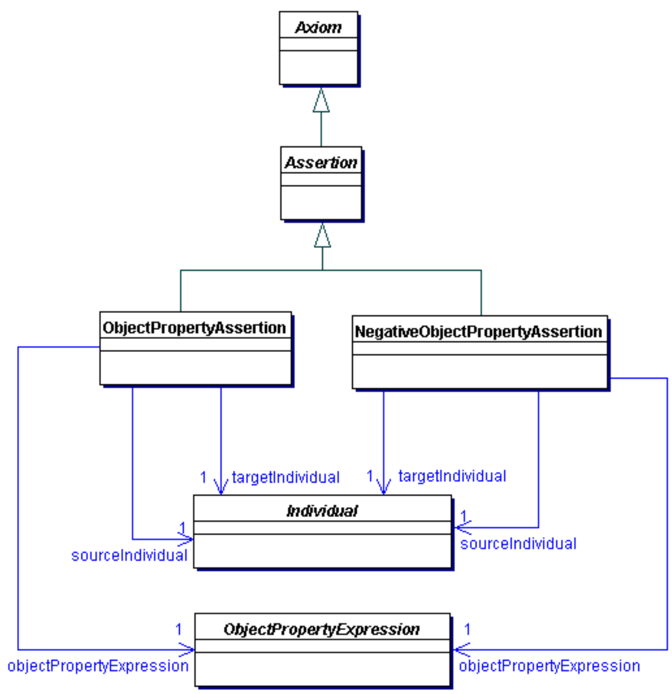

Figure 20. Object Property Assertions in OWL 2

The **DataPropertyAssertion** axiom allows one to state that an individual is connected by a data property expression to a literal, while **NegativeDataPropertyAssertion** allows for the opposite — that is, to state that an individual is not connected by a data property expression to a literal.

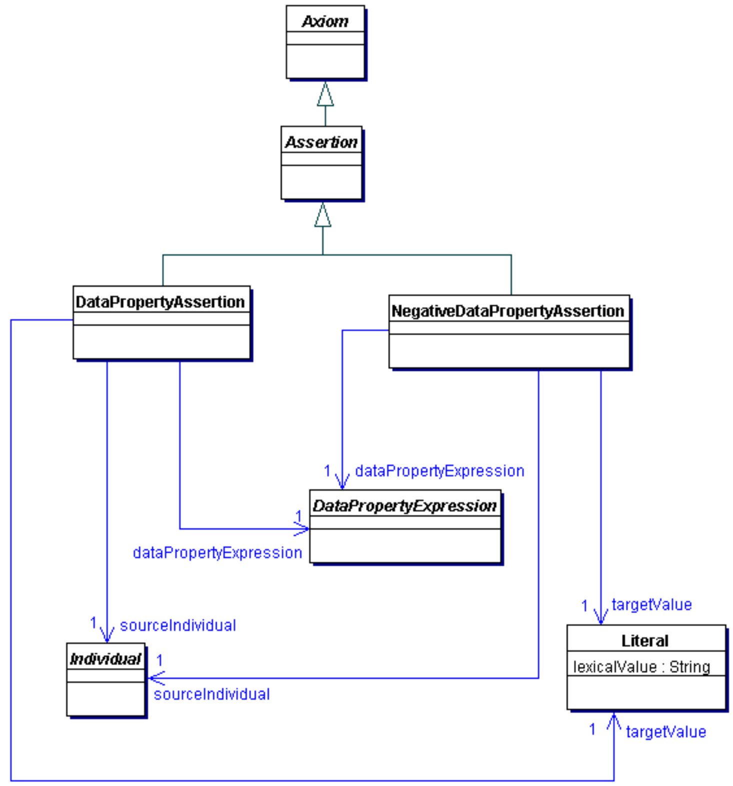

Figure 21. Data Property Assertions in OWL 2

Assertion :=
    SameIndividual | **DifferentIndividuals** | **ClassAssertion** |
    ObjectPropertyAssertion | **NegativeObjectPropertyAssertion** |
    DataPropertyAssertion | **NegativeDataPropertyAssertion**

sourceIndividual := **Individual**
targetIndividual := **Individual**
targetValue := **Literal**

9.6.1 Individual Equality

An individual equality axiom `SameIndividual(` a_1 `...` a_n `)` states that all of the individuals a_i, $1 \le i \le n$, are equal to each other. This axiom allows one to use each a_i as a synonym for each a_j — that is, in any expression in the ontology containing such an axiom, a_i can be replaced with a_j without affecting the meaning of the ontology.

SameIndividual := 'SameIndividual' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

Example:

Consider the ontology consisting of the following axioms.

`SameIndividual(` *a:Meg a:Megan* `)`	Meg and Megan are the same objects.
`ObjectPropertyAssertion(` *a:hasBrother a:Meg a:Stewie* `)`	Meg has a brother Stewie.

Since *a:Meg* and *a:Megan* are equal, one individual can always be replaced with the other one. Therefore, this ontology entails that *a:Megan* is connected by *a:hasBrother* with *a:Stewie* — that is, the ontology entails the following assertion:

`ObjectPropertyAssertion(` *a:hasBrother a:Megan a:Stewie* `)`

9.6.2 Individual Inequality

An individual inequality axiom `DifferentIndividuals(` a_1 `...` a_n `)` states that all of the individuals a_i, $1 \le i \le n$, are different from each other; that is, no individuals a_i and a_j with $i \ne j$ can be derived to be equal. This axiom can be used to axiomatize the *unique name assumption* — the assumption that all different individual names denote different individuals.

DifferentIndividuals := 'DifferentIndividuals' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyAssertion( a:fatherOf a:Peter a:Meg )`	Peter is Meg's father.
`ObjectPropertyAssertion( a:fatherOf a:Peter a:Chris )`	Peter is Chris's father.
`ObjectPropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is Stewie's father.
`DifferentIndividuals( a:Peter a:Meg a:Chris a:Stewie )`	Peter, Meg, Chris, and Stewie are all different from each other.

The last axiom in this example ontology axiomatizes the unique name assumption (but only for the four names in the axiom). If the ontology were extended with the following axiom stating that *a:fatherOf* is functional, then this axiom would imply that *a:Meg*, *a:Chris*, and *a:Stewie* are all equal, thus invalidating the unique name assumption and making the ontology inconsistent.

```
FunctionalObjectProperty( a:fatherOf )
```

9.6.3 Class Assertions

A class assertion `ClassAssertion( CE a )` states that the individual a is an instance of the class expression CE.

ClassAssertion := 'ClassAssertion' '(' **axiomAnnotations** **ClassExpression** **Individual** ')'

Example:

Consider the ontology consisting of the following axioms.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`SubClassOf( a:Dog a:Mammal )`	Each dog is a mammal.

The first axiom states that *a:Brian* is an instance of the class *a:Dog*. By the second axiom, each instance of *a:Dog* is an instance of *a:Mammal*. Therefore, this ontology entails that *a:Brian* is an instance of *a:Mammal* — that is, the ontology entails the following assertion:

```
ClassAssertion( a:Mammal a:Brian )
```

9.6.4 Positive Object Property Assertions

A positive object property assertion `ObjectPropertyAssertion( OPE a1 a2 )` states that the individual a_1 is connected by the object property expression OPE to the individual a_2.

ObjectPropertyAssertion := 'ObjectPropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

Example:

Consider the ontology consisting of the following axioms.

`ObjectPropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.
`SubClassOf( ObjectSomeValuesFrom( a:hasDog owl:Thing ) a:DogOwner )`	Objects that have a dog are dog owners.

The first axiom states that *a:Peter* is connected by *a:hasDog* to *a:Brian*. By the second axiom, each individual connected by *a:hasDog* to an individual is an instance of *a:DogOwner*. Therefore, this ontology entails that *a:Peter* is an instance of *a:DogOwner* — that is, the ontology entails the following assertion:

```
ClassAssertion( a:DogOwner a:Peter )
```

9.6.5 Negative Object Property Assertions

A negative object property assertion `NegativeObjectPropertyAssertion( OPE a1 a2 )` states that the individual a_1 is not connected by the object property expression OPE to the individual a_2.

NegativeObjectPropertyAssertion := 'NegativeObjectPropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

Example:

Consider the ontology consisting of the following axiom.

`NegativeObjectPropertyAssertion( a:hasSon a:Peter a:Meg )`	Meg is not a son of Peter.

The ontology would become inconsistent if it were extended with the following assertion:

```
ObjectPropertyAssertion( a:hasSon a:Peter a:Meg )
```

9.6.6 Positive Data Property Assertions

A positive data property assertion `DataPropertyAssertion( DPE a lt )` states that the individual a is connected by the data property expression DPE to the literal `lt`.

DataPropertyAssertion := 'DataPropertyAssertion' '(' **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** ')'

Example:

Consider the ontology consisting of the following axioms.

```
DataPropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )
SubClassOf(
    DataSomeValuesFrom( a:hasAge
       DatatypeRestriction( xsd:integer
          xsd:minInclusive "13"^^xsd:integer
          xsd:maxInclusive "19"^^xsd:integer
       )
    )
    a:Teenager
)
```

Meg is seventeen years old.

Objects that are older than 13 and younger than 19 (both inclusive) are teenagers.

The first axiom states that *a:Meg* is connected by *a:hasAge* to the literal `"17"^^xsd:integer`. By the second axiom, each individual connected by *a:hasAge* to an integer between 13 and 19 is an instance of *a:Teenager*. Therefore, this ontology entails that *a:Meg* is an instance of *a:Teenager* — that is, the ontology entails the following assertion:

```
ClassAssertion( a:Teenager a:Meg )
```

9.6.7 Negative Data Property Assertions

A negative data property assertion `NegativeDataPropertyAssertion( DPE a lt )` states that the individual a is not connected by the data property expression DPE to the literal `lt`.

NegativeDataPropertyAssertion := 'NegativeDataPropertyAssertion' '(' **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** ')'

Example:

Consider the ontology consisting of the following axiom.

```
NegativeDataPropertyAssertion( a:hasAge a:Meg "5"^^xsd:integer )
```

Meg is not five years old.

The ontology would become inconsistent if it were extended with the following assertion:

```
DataPropertyAssertion( a:hasAge a:Meg "5"^^xsd:integer )
```

10 Annotations

OWL 2 applications often need ways to associate additional information with ontologies, entities, and axioms. To this end, OWL 2 provides for *annotations* on ontologies, axioms, and entities.

Example:

One might want to associate human-readable labels with IRIs and use them when visualizing an ontology. To this end, one might use the *rdfs:label* annotation property to associate such labels with ontology IRIs.

Various OWL 2 syntaxes, such as the functional-style syntax, provide a mechanism for embedding comments into ontology documents. The structure of such comments is, however, dependent on the syntax, so these are simply discarded during parsing. In contrast, annotations are "first-class citizens" in the structural specification of OWL 2, and their structure is independent of the underlying syntax.

Example:

Since it is based on XML, the OWL 2 XML Syntax [*OWL 2 XML Serialization*] allows the embedding of the standard XML comments into ontology documents. Such comments are not represented in the structural specification of OWL 2 and, consequently, they should be ignored during document parsing.

10.1 Annotations of Ontologies, Axioms, and other Annotations

Ontologies, axioms, and annotations themselves can be annotated using annotations shown in Figure 22. As shown in the figure, such annotations consist of an annotation property and an annotation value, where the latter can be anonymous individuals, IRIs, and literals.

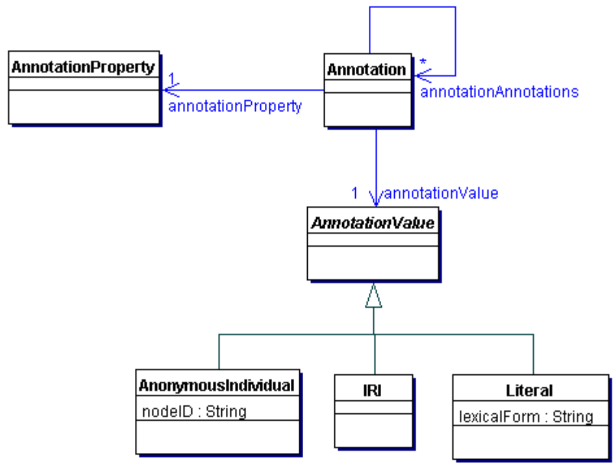

Figure 22. Annotations of Ontologies and Axioms in OWL 2

Annotation := 'Annotation' '(' **annotationAnnotations** **AnnotationProperty** **AnnotationValue** ')'
annotationAnnotations := { **Annotation** }
AnnotationValue := **AnonymousIndividual** | **IRI** | **Literal**

10.2 Annotation Axioms

OWL 2 provides means to state several types of axioms about annotation properties, as shown in Figure 23. These statements are treated as axioms only in order to simplify the structural specification of OWL 2.

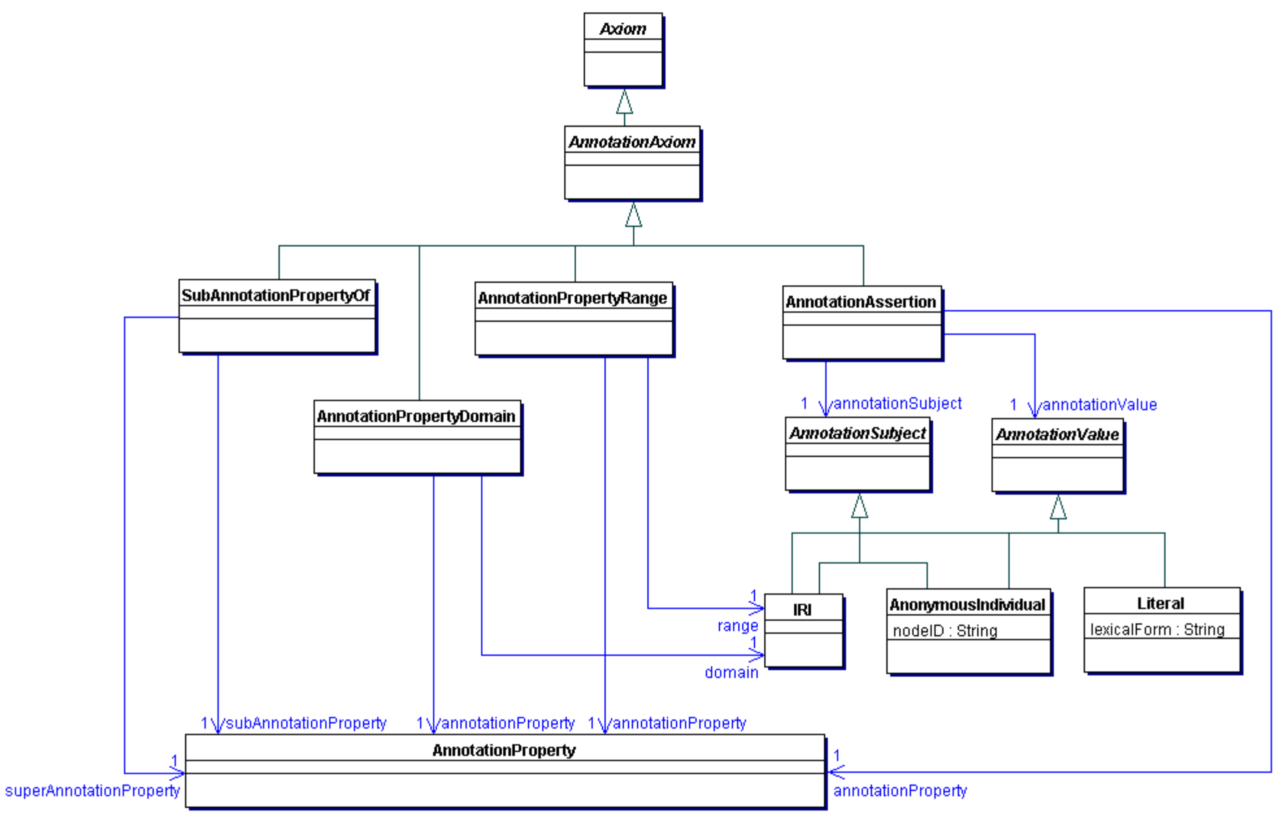

Figure 23. Annotations of IRIs and Anonymous Individuals in OWL 2

AnnotationAxiom := **AnnotationAssertion** | **SubAnnotationPropertyOf** | **AnnotationPropertyDomain** | **AnnotationPropertyRange**

10.2.1 Annotation Assertion

An annotation assertion `AnnotationAssertion( AP as av )` states that the annotation subject as — an IRI or an anonymous individual — is annotated with the annotation property AP and the annotation value av.

AnnotationAssertion := 'AnnotationAssertion' '(' **axiomAnnotations** **AnnotationProperty** **AnnotationSubject** **AnnotationValue** ')'
AnnotationSubject := **IRI** | **AnonymousIndividual**

Example:

The following axiom assigns a human-readable comment to the IRI *a:Person*.

```
AnnotationAssertion( rdfs:label a:Person "Represents the set of all people." )
```

Since the annotation is assigned to an IRI, it applies to all entities with the given IRI. Thus, if an ontology contains both a class and an individual *a:Person*, the above comment applies to both entities.

10.2.2 Annotation Subproperties

An annotation subproperty axiom `SubAnnotationPropertyOf(` AP_1 AP_2 `)` states that the annotation property AP_1 is a subproperty of the annotation property AP_2.

SubAnnotationPropertyOf := 'SubAnnotationPropertyOf' '(' **axiomAnnotations** **subAnnotationProperty** **superAnnotationProperty** ')'
subAnnotationProperty := **AnnotationProperty**
superAnnotationProperty := **AnnotationProperty**

10.2.3 Annotation Property Domain

An annotation property domain axiom `AnnotationPropertyDomain( AP U )` states that the domain of the annotation property AP is the IRI U.

AnnotationPropertyDomain := 'AnnotationPropertyDomain' '(' **axiomAnnotations** **AnnotationProperty** **IRI** ')'

10.2.4 Annotation Property Range

An annotation property range axiom `AnnotationPropertyRange( AP U )` states that the range of the annotation property AP is the IRI U.

AnnotationPropertyRange := 'AnnotationPropertyRange' '(' **axiomAnnotations** **AnnotationProperty** **IRI** ')'

11 Global Restrictions on Axioms in OWL 2 DL

The axiom closure *Ax* (with anonymous individuals standardized apart as explained in Section 5.6.2) of each OWL 2 DL ontology *O must* satisfy the *global restrictions* defined in this section. As explained in the literature [*SROIQ*], this restriction is necessary in order to obtain a decidable language. The formal definition of these conditions is rather technical, so it is split into two parts. Section 11.1 first introduces the notions of a property hierarchy and of *simple* object property expressions. These notions are then used in Section 11.2 to define the actual conditions on *Ax*.

11.1 Property Hierarchy and Simple Object Property Expressions

For an object property expression `OPE`, the *inverse property expression* `INV(OPE)` is defined as follows:

- If `OPE` is an object property `OP`, then `INV(OPE)` = `ObjectInverseOf( OP )`.
- if `OPE` is of the form `ObjectInverseOf( OP )` for `OP` an object property, then `INV(OPE)` = `OP`.

The set *AllOPE(Ax)* of all object property expressions w.r.t. *Ax* is the smallest set containing `OP` and `INV(OP)` for each object property `OP` occurring in *Ax*.

An object property expression `OPE` is *composite* in the set of axioms *Ax* if

- `OPE` is equal to *owl:topObjectProperty* or *owl:bottomObjectProperty*, or
- *Ax* contains an axiom of the form
  - `SubObjectPropertyOf( ObjectPropertyChain(` OPE_1 ... OPE_n `) OPE )` with n > 1, or
  - `SubObjectPropertyOf( ObjectPropertyChain(` OPE_1 ... OPE_n `) INV(OPE) )` with n > 1, or
  - `TransitiveObjectProperty( OPE )`, or
  - `TransitiveObjectProperty( INV(OPE) )`.

The relation → is the smallest relation on *AllOPE(Ax)* for which the following conditions hold (A → B means that → holds for A and B):

- if *Ax* contains an axiom SubObjectPropertyOf(OPE_1 OPE_2), then OPE_1 → OPE_2 holds; and
- if *Ax* contains an axiom EquivalentObjectProperties(OPE_1 OPE_2), then OPE_1 → OPE_2 and OPE_2 → OPE_1 hold; and
- if *Ax* contains an axiom InverseObjectProperties(OPE_1 OPE_2), then OPE_1 → INV(OPE_2) and INV(OPE_2) → OPE_1 hold; and
- if *Ax* contains an axiom SymmetricObjectProperty(OPE), then OPE → INV(OPE) holds; and
- if OPE_1 → OPE_2 holds, then INV(OPE_1) → INV(OPE_2) holds as well.

The *property hierarchy* relation $\rightarrow^*$ is the reflexive-transitive closure of →.

An object property expression OPE is *simple* in *Ax* if, for each object property expression OPE' such that OPE' $\rightarrow^*$ OPE holds, OPE' is not composite.

Example:

Roughly speaking, a simple object property expression has no direct or indirect subproperties that are either transitive or are defined by means of property chains, where the notion of indirect subproperties is captured by the property hierarchy. Consider the following axioms:

SubObjectPropertyOf(ObjectPropertyChain(*a:hasFather a:hasBrother*) *a:hasUncle*)	The brother of someone's father is that person's uncle.
SubObjectPropertyOf(*a:hasUncle a:hasRelative*)	Having an uncle implies having a relative.
SubObjectPropertyOf(*a:hasBiologicalFather a:hasFather*)	Having a biological father implies having a father.

The object property *a:hasUncle* occurs in an object subproperty axiom involving a property chain, so it is not simple. Consequently, the object property *a:hasRelative* is not simple either, because *a:hasUncle* is a subproperty of *a:hasRelative* and *a:hasUncle* is not simple. In contrast, the object property *a:hasBiologicalFather* is simple, and so is *a:hasFather*.

11.2 The Restrictions on the Axiom Closure

The set of axioms *Ax* satisfies the *global restrictions* of OWL 2 DL if all of the following conditions hold.

Restriction on *owl:topDataProperty*. The *owl:topDataProperty* property occurs in *Ax* only in the **superDataPropertyExpression** part of **SubDataPropertyOf** axioms.

Without this restriction, *owl:topDataProperty* could be used to write axioms about datatypes, which would invalidate Theorem DS1 from the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*]. That is, the consequences of an ontology would then not necessarily depend only on the datatypes used in the ontology, but would also depend on the datatypes selected in the OWL 2 datatype map. Thus, if an implementation or a future revision of OWL decided to extend the set of supported datatypes, it would run the risk of possibly changing the consequences of certain ontologies.

Restrictions on Datatypes.

- Each datatype occurring in *Ax* satisfies exactly one of the following conditions: it is *rdfs:Literal*, or it is contained in the OWL 2 datatype map, or it is defined by a single datatype definition axiom in *Ax*.
- A strict partial order (i.e., an irreflexive and transitive relation) < on the set of all datatypes in *Ax* exists such that, for each axiom of the form DatatypeDefinition(DT DR) and each datatype DT_1 occurring in DR, we have DT_1 < DT.

Example:

The first condition ensures that all datatypes in *Ax* are given a well-defined interpretation and that datatype definitions do not redefine the datatypes from the OWL 2 datatype map. The second condition ensures that datatype definitions are acyclic — that is, if a datatype DT_1 is used in a definition of DT, then DT is not allowed to be used in the definition of DT_1 — and it is illustrated by the following example:

Declaration(Datatype(*a:SSN*))	*a:SSN* is a datatype.
Declaration(Datatype(*a:TIN*))	*a:TIN* is a datatype.
Declaration(Datatype(*a:TaxNumber*))	*a:TaxNumber* is a datatype.
DatatypeDefinition(*a:SSN* DatatypeRestriction(*xsd:string xsd:pattern* "[0-9]{3}-[0-9]{2}-[0-9]{4}"))	A social security number is a string that matches the given regular expression.
DatatypeDefinition(*a:TIN* DatatypeRestriction(*xsd:string xsd:pattern* "[0-9]{11}"))	A TIN — a tax identification number used in Germany — is a string consisting of 11 digits.
DatatypeDefinition(*a:TaxNumber* DataUnionOf(*a:SSN a:TIN*))	A tax number is either a social security number of a TIN.

These datatype definitions are acyclic: *a:SSN* and *a:TIN* are defined in terms of *xsd:string*, and *a:TaxNumber* is defined in terms of *a:SSN* and *a:TIN*. To verify this condition formally, it suffices to find one strict partial order < on these datatypes such that each datatype is defined only in terms of the datatypes that are smaller w.r.t. <. For example, it can be readily verified that the partial order < given below fulfills the above conditions.

xsd:string < *a:SSN* < *a:TaxNumber*
xsd:string < *a:TIN* < *a:TaxNumber*

Note that order < is allowed to be partial — that is, some datatypes can be incomparable under <. In the above example, datatypes *a:SSN* and *a:TIN* are incomparable under <. Since neither of these two datatypes is defined in terms of the other datatype, the order between the two datatypes is irrelevant.

The restriction on datatypes is necessary to ensure validity of Theorem DS1 from the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*]. Furthermore, the restriction is natural given that data ranges describe the set of values exactly. For example, if an axiom defining *a:SSN* in terms of *a:TIN* and *a:TaxNumber* were added to the above axioms, then datatypes *a:SSN*, *a:TIN*, and *a:TaxNumber* could not be simply "unfolded", which is contrary to the intended meaning of these datatypes. This situation, however, is disallowed since no ordering < satisfying the mentioned restrictions exists for the extended axiom set.

Restriction on Simple Roles. Each class expression and each axiom in *Ax* of type from the following two lists contains only simple object properties.

- **ObjectMinCardinality**, **ObjectMaxCardinality**, **ObjectExactCardinality**, and **ObjectHasSelf** .
- **FunctionalObjectProperty**, **InverseFunctionalObjectProperty**, **IrreflexiveObjectProperty**, **AsymmetricObjectProperty**, and **DisjointObjectProperties**.

This restriction is necessary in order to guarantee decidability of the basic reasoning problems for OWL 2 DL [*Description Logics*].

Restriction on the Property Hierarchy. A strict partial order (i.e., an irreflexive and transitive relation) < on *AllOPE(Ax)* exists that fulfills the following conditions:

- $OP_1 < OP_2$ if and only if $INV(OP_1) < OP_2$ for all object properties OP_1 and OP_2 occurring in *AllOPE(Ax)*.
- If $OPE_1 < OPE_2$ holds, then $OPE_2 \rightarrow^* OPE_1$ does not hold;
- Each axiom in *Ax* of the form `SubObjectPropertyOf( ObjectPropertyChain( OPE1 ... OPEn ) OPE )` with $n \geq 2$ fulfills the following conditions:
    - OPE is equal to *owl:topObjectProperty*, or
    - $n = 2$ and $OPE_1 = OPE_2 = OPE$, or
    - $OPE_i < OPE$ for each $1 \leq i \leq n$, or
    - $OPE_1 = OPE$ and $OPE_i < OPE$ for each $2 \leq i \leq n$, or
    - $OPE_n = OPE$ and $OPE_i < OPE$ for each $1 \leq i \leq n\text{-}1$.

This restriction is necessary in order to guarantee decidability of the basic reasoning problems for OWL 2 DL [*Description Logics*].

Example:

The main goal of this restriction is to prevent cyclic definitions involving object subproperty axioms with property chains. Consider the following ontology:

`SubObjectPropertyOf( ObjectPropertyChain( a:hasFather a:hasBrother ) a:hasUncle )`	The brother of someone's father is that person's uncle.
`SubObjectPropertyOf( ObjectPropertyChain( a:hasUncle a:hasWife ) a:hasAuntInLaw )`	The wife of someone's uncle is that person's aunt-in-law.

The first axiom defines *a:hasUncle* in terms of *a:hasFather* and *a:hasBrother*, and the second axiom defines *a:hasAuntInLaw* in terms of *a:hasUncle* and *a:hasWife*. The second axiom depends on the first one, but not vice versa; hence, these axioms are not cyclic and can occur together in the axiom closure of an OWL 2 DL ontology. To verify this condition formally, it suffices to find one strict partial order < on object properties such that each property is defined only in terms of the properties that are smaller w.r.t. <. For example, it can be readily verified that the partial order < given below fulfills the above conditions.

a:hasFather < *a:hasUncle*
a:hasBrother < *a:hasUncle*
a:hasUncle < *a:hasAuntInLaw*
a:hasWife < *a:hasAuntInLaw*

The first two conditions on < are needed to satisfy the first axiom, while the remaining two conditions on < are needed to satisfy the second axiom from the example OWL 2 DL ontology.

Example:

In contrast to the previous example, the following axioms are cyclic and do not satisfy the restriction on the property hierarchy.

`SubObjectPropertyOf( ObjectPropertyChain( a:hasFather a:hasBrother ) a:hasUncle )`	The brother of someone's father is that person's uncle.
`SubObjectPropertyOf( ObjectPropertyChain( a:hasChild a:hasUncle ) a:hasBrother )`	The uncle of someone's child is that person's brother.

The first axiom defines *a:hasUncle* in terms of *a:hasBrother*, while the second axiom defines *a:hasBrother* in terms of *a:hasUncle*; these two definitions are thus cyclic and cannot occur together in the axiom closure of an OWL 2 DL ontology. To verify this condition formally, note that, for < to satisfy the third subcondition of the third condition, we need *a:hasBrother* < *a:hasUncle* (due to the first axiom) and *a:hasUncle* < *a:hasBrother* (due to the second axiom); by transitivity of < we then have *a:hasUncle* < *a:hasUncle* and *a:hasBrother* < *a:hasBrother*; however, this contradicts the requirement that < is irreflexive. Thus, an order < satisfying all the required conditions does not exist.

Example:

A particular kind of cyclic definitions is known not to lead to decidability problems. Consider the following ontology:

`SubObjectPropertyOf( ObjectPropertyChain( a:hasChild a:hasSibling ) a:hasChild )`	The sibling of someone's child is that person's child.

The above definition is cyclic, since the object property *a:hasChild* occurs in both the subproperty chain and as a superproperty. As per the fourth and the fifth subcondition of the third condition, however, axioms of this form do not violate the restriction on the property hierarchy.

Restrictions on the Usage of Anonymous Individuals.

- No anonymous individual occurs in *Ax* in an axiom of type from the following list:

    - **SameIndividual**, **DifferentIndividuals**, **NegativeObjectPropertyAssertion**, or **NegativeDataPropertyAssertion**.
- No anonymous individual occurs in *Ax* in a class expression of type from the following list:
    - **ObjectOneOf** or **ObjectHasValue**.
- The *anonymous individual graph* for *Ax* is the undirected graph *F* whose vertices are anonymous individuals occurring in *Ax*, and that contains an (undirected) edge between each pair of anonymous individuals `_:x` and `_:y` for each assertion in *Ax* of the form `ObjectPropertyAssertion( OPE _:x _:y )`. Such *F* is required to satisfy all of the following conditions:
    - *F* is a forest — that is, it should be possible to partition *F* into zero or more disjoint undirected trees;
    - for each pair of anonymous individuals `_:x` and `_:y` connected by an edge in *F*, the set *Ax* contains at most one assertion of the form `ObjectPropertyAssertion( OPE _:x _:y )` or `ObjectPropertyAssertion( OPE _:y _:x )`; and
    - each tree in *F* contains at least one anonymous individual `_:x` such that the set *Ax* contains at most one assertion of the form `ObjectPropertyAssertion( OPE _:x a )` or `ObjectPropertyAssertion( OPE a _:x )` with a a named individual.

Example:

These restrictions ensure that each OWL 2 DL ontology with anonymous individuals can be transformed to an equivalent ontology without anonymous individuals. Roughly speaking, this is possible if property assertions connect anonymous individuals in a tree-like way. Consider the following ontology:

`ObjectPropertyAssertion( a:hasChild a:Francis _:a1 )`	Francis has some (unknown) child.
`ObjectPropertyAssertion( a:hasChild _:a1 a:Meg )`	This unknown child has Meg...
`ObjectPropertyAssertion( a:hasChild _:a1 a:Chris )`	...Chris...
`ObjectPropertyAssertion( a:hasChild _:a1 a:Stewie )`	...and Stewie as children.

The connections between individuals *a:Francis*, *a:Meg*, *a:Chris*, and *a:Stewie* can be understood as a tree that contains `_:a1` as its root. Because of that, the anonymous individuals can be "rolled up"; that is, these four assertions can be replaced by the following equivalent assertion:

```
ClassAssertion(
    ObjectSomeValuesFrom( a:hasChild
       ObjectIntersectionOf(
          ObjectHasValue( a:hasChild a:Meg )
          ObjectHasValue( a:hasChild a:Chris )
          ObjectHasValue( a:hasChild a:Stewie )
       )
    )
    a:Francis
)
```

Example:

Unlike in the previous example, the following ontology does not satisfy the restrictions on the usage of anonymous individuals:

```
ObjectPropertyAssertion( a:hasSibling _:b1 _:b2 )
ObjectPropertyAssertion( a:hasSibling _:b2 _:b3 )
ObjectPropertyAssertion( a:hasSibling _:b3 _:b1 )
```

The following ontology does not satisfy these restrictions either:

```
ObjectPropertyAssertion( a:hasChild _:b1 _:b2 )
ObjectPropertyAssertion( a:hasDaughter _:b1 _:b2 )
```

In both of these examples, the anonymous individuals are connected by property assertions in a non-tree-like way. These assertions can therefore not be replaced with class expressions, which can lead to the undecidability of the basic reasoning problems.

12 Appendix: Internet Media Type, File Extension, and Macintosh File Type

Contact
: Ivan Herman / Sandro Hawke

See also
: How to Register a Media Type for a W3C Specification [*Register MIME*] and Internet Media Type registration, consistency of use [*MIME Consistency*].

The Internet Media Type / MIME Type for the OWL functional-style Syntax is `text/owl-functional`.

It is recommended that OWL functional-style Syntax files have the extension `.ofn` (all lowercase) on all platforms.

It is recommended that OWL functional-style Syntax files stored on Macintosh HFS file systems be given a file type of TEXT.

The information that follows will be submitted to the IESG for review, approval, and registration with IANA.

Type name
: text

Subtype name
: owl-functional

Required parameters
: None

Optional parameters
: charset This parameter may be required when transfering non-ASCII data across some protocols. If present, the value of charset should be UTF-8.

Encoding considerations
: The syntax of the OWL functional-style Syntax is expressed over code points in Unicode [*UNICODE*]. The encoding should be UTF-8 [*RFC 3629*], but other encodings are allowed.

Security considerations
: The OWL functional-style Syntax uses IRIs as term identifiers. Applications interpreting data expressed in the OWL functional-style Syntax should address the security issues of Internationalized Resource Identifiers (IRIs) [*RFC3987*] Section 8, as well as Uniform Resource Identifiers

(URI): Generic Syntax [*RFC 3986*] Section 7. Multiple IRIs may have the same appearance. Characters in different scripts may look similar (a Cyrillic "o" may appear similar to a Latin "o"). A character followed by combining characters may have the same visual representation as another character (LATIN SMALL LETTER E followed by COMBINING ACUTE ACCENT has the same visual representation as LATIN SMALL LETTER E WITH ACUTE). Any person or application that is writing or interpreting data in the OWL functional-style Syntax must take care to use the IRI that matches the intended semantics, and avoid IRIs that may look similar. Further information about matching of similar characters can be found in Unicode Security Considerations [*UNISEC*] and Internationalized Resource Identifiers (IRIs) [*RFC3987*] Section 8.

Interoperability considerations
: There are no known interoperability issues.

Published specification
: This specification.

Applications which use this media type
: No widely deployed applications are known to currently use this media type. It is expected that OWL tools will use this media type in the future.

Additional information
: None.

Magic number(s)
: OWL functional-style Syntax documents may have the strings "Prefix" or "Ontology" (case dependent) near the beginning of the document.

File extension(s)
: ".ofn"

Base IRI
: There are no constructs in the OWL functional-style Syntax to change the Base IRI.

Macintosh file type code(s)
: "TEXT"

Person & email address to contact for further information
: Ivan Herman, ivan@w3.org / Sandro Hawke, sandro@w3.org. Please send technical comments and questions about OWL to public-owl-comments@w3.org, a mailing list with a public archive at http://lists.w3.org/Archives/Public/public-owl-comments/

Intended usage
: COMMON

Restrictions on usage
: None

Author/Change controller
: The OWL functional-style Syntax is the product of the W3C OWL Working Group; W3C reserves change control over this specification.

13 Appendix: Complete Grammar (Normative)

This section contains the complete grammar of the functional-style syntax defined in this specification document. For easier reference, the grammar has been split into two parts.

13.1 General Definitions

```
nonNegativeInteger := a nonempty finite sequence of digits between 0 and 9
quotedString := a finite sequence of characters in which " (U+22) and \ (U+5C) occur only in pairs of the form \" (U+5C,
U+22) and \\ (U+5C, U+5C), enclosed in a pair of " (U+22) characters
languageTag := @ (U+40) followed a nonempty sequence of characters matching the langtag production from [BCP 47]
nodeID := a finite sequence of characters matching the BLANK_NODE_LABEL production of [SPARQL]


fullIRI := an IRI as defined in [RFC3987], enclosed in a pair of < (U+3C) and > (U+3E) characters
prefixName := a finite sequence of characters matching the as PNAME_NS production of [SPARQL]
abbreviatedIRI := a finite sequence of characters matching the PNAME_LN production of [SPARQL]
IRI := fullIRI | abbreviatedIRI


ontologyDocument := { prefixDeclaration } Ontology
prefixDeclaration := 'Prefix' '(' prefixName '=' fullIRI ')'
Ontology :=
    'Ontology' '(' [ ontologyIRI [ versionIRI ] ]
       directlyImportsDocuments
       ontologyAnnotations
       axioms
    ')'
ontologyIRI := IRI
versionIRI := IRI
directlyImportsDocuments := { 'Import' '(' IRI ')' }
ontologyAnnotations := { Annotation }
axioms := { Axiom }


Declaration := 'Declaration' '(' axiomAnnotations Entity ')'
Entity :=
    'Class' '(' Class ')' |
    'Datatype' '(' Datatype ')' |
    'ObjectProperty' '(' ObjectProperty ')' |
    'DataProperty' '(' DataProperty ')' |
    'AnnotationProperty' '(' AnnotationProperty ')' |
    'NamedIndividual' '(' NamedIndividual ')'


AnnotationSubject := IRI | AnonymousIndividual
```

```
AnnotationValue := AnonymousIndividual | IRI | Literal
axiomAnnotations := { Annotation }

Annotation := 'Annotation' '(' annotationAnnotations AnnotationProperty AnnotationValue ')'
annotationAnnotations := { Annotation }

AnnotationAxiom := AnnotationAssertion | SubAnnotationPropertyOf | AnnotationPropertyDomain | AnnotationPropertyRange

AnnotationAssertion := 'AnnotationAssertion' '(' axiomAnnotations AnnotationProperty AnnotationSubject AnnotationValue ')'

SubAnnotationPropertyOf := 'SubAnnotationPropertyOf' '(' axiomAnnotations subAnnotationProperty superAnnotationProperty
')'
subAnnotationProperty := AnnotationProperty
superAnnotationProperty := AnnotationProperty

AnnotationPropertyDomain := 'AnnotationPropertyDomain' '(' axiomAnnotations AnnotationProperty IRI ')'

AnnotationPropertyRange := 'AnnotationPropertyRange' '(' axiomAnnotations AnnotationProperty IRI ')'
```

13.2 Definitions of OWL 2 Constructs

```
Class := IRI

Datatype := IRI

ObjectProperty := IRI

DataProperty := IRI

AnnotationProperty := IRI

Individual := NamedIndividual | AnonymousIndividual

NamedIndividual := IRI

AnonymousIndividual := nodeID

Literal := typedLiteral | stringLiteralNoLanguage | stringLiteralWithLanguage
typedLiteral := lexicalForm '^^' Datatype
lexicalForm := quotedString
stringLiteralNoLanguage := quotedString
stringLiteralWithLanguage := quotedString languageTag


ObjectPropertyExpression := ObjectProperty | InverseObjectProperty

InverseObjectProperty := 'ObjectInverseOf' '(' ObjectProperty ')'

DataPropertyExpression := DataProperty


DataRange :=
    Datatype |
    DataIntersectionOf |
    DataUnionOf |
    DataComplementOf |
    DataOneOf |
    DatatypeRestriction

DataIntersectionOf := 'DataIntersectionOf' '(' DataRange DataRange { DataRange } ')'

DataUnionOf := 'DataUnionOf' '(' DataRange DataRange { DataRange } ')'

DataComplementOf := 'DataComplementOf' '(' DataRange ')'

DataOneOf := 'DataOneOf' '(' Literal { Literal } ')'

DatatypeRestriction := 'DatatypeRestriction' '(' Datatype constrainingFacet restrictionValue { constrainingFacet
restrictionValue } ')'
constrainingFacet := IRI
restrictionValue := Literal


ClassExpression :=
    Class |
    ObjectIntersectionOf | ObjectUnionOf | ObjectComplementOf | ObjectOneOf |
    ObjectSomeValuesFrom | ObjectAllValuesFrom | ObjectHasValue | ObjectHasSelf |
```

W3C Proposed Edited Recommendation

ObjectMinCardinality | **ObjectMaxCardinality** | **ObjectExactCardinality** |
DataSomeValuesFrom | **DataAllValuesFrom** | **DataHasValue** |
DataMinCardinality | **DataMaxCardinality** | **DataExactCardinality**

ObjectIntersectionOf := 'ObjectIntersectionOf' '(' **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

ObjectUnionOf := 'ObjectUnionOf' '(' **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

ObjectComplementOf := 'ObjectComplementOf' '(' **ClassExpression** ')'

ObjectOneOf := 'ObjectOneOf' '(' **Individual** { **Individual** }')'

ObjectSomeValuesFrom := 'ObjectSomeValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

ObjectAllValuesFrom := 'ObjectAllValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

ObjectHasValue := 'ObjectHasValue' '(' **ObjectPropertyExpression** **Individual** ')'

ObjectHasSelf := 'ObjectHasSelf' '(' **ObjectPropertyExpression** ')'

ObjectMinCardinality := 'ObjectMinCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

ObjectMaxCardinality := 'ObjectMaxCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

ObjectExactCardinality := 'ObjectExactCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

DataSomeValuesFrom := 'DataSomeValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

DataAllValuesFrom := 'DataAllValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

DataHasValue := 'DataHasValue' '(' **DataPropertyExpression** **Literal** ')'

DataMinCardinality := 'DataMinCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

DataMaxCardinality := 'DataMaxCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

DataExactCardinality := 'DataExactCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Axiom := **Declaration** | **ClassAxiom** | **ObjectPropertyAxiom** | **DataPropertyAxiom** | **DatatypeDefinition** | **HasKey** | **Assertion** | **AnnotationAxiom**

ClassAxiom := **SubClassOf** | **EquivalentClasses** | **DisjointClasses** | **DisjointUnion**

SubClassOf := 'SubClassOf' '(' **axiomAnnotations** **subClassExpression** **superClassExpression** ')'
subClassExpression := **ClassExpression**
superClassExpression := **ClassExpression**

EquivalentClasses := 'EquivalentClasses' '(' **axiomAnnotations** **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

DisjointClasses := 'DisjointClasses' '(' **axiomAnnotations** **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

DisjointUnion := 'DisjointUnion' '(' **axiomAnnotations** **Class** **disjointClassExpressions** ')'
disjointClassExpressions := **ClassExpression** **ClassExpression** { **ClassExpression** }

ObjectPropertyAxiom :=
SubObjectPropertyOf | **EquivalentObjectProperties** |
DisjointObjectProperties | **InverseObjectProperties** |
ObjectPropertyDomain | **ObjectPropertyRange** |
FunctionalObjectProperty | **InverseFunctionalObjectProperty** |
ReflexiveObjectProperty | **IrreflexiveObjectProperty** |
SymmetricObjectProperty | **AsymmetricObjectProperty** |
TransitiveObjectProperty

SubObjectPropertyOf := 'SubObjectPropertyOf' '(' **axiomAnnotations** **subObjectPropertyExpression** **superObjectPropertyExpression** ')'
subObjectPropertyExpression := **ObjectPropertyExpression** | **propertyExpressionChain**
propertyExpressionChain := 'ObjectPropertyChain' '(' **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'
superObjectPropertyExpression := **ObjectPropertyExpression**

EquivalentObjectProperties := 'EquivalentObjectProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

DisjointObjectProperties := 'DisjointObjectProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

ObjectPropertyDomain := 'ObjectPropertyDomain' '(' **axiomAnnotations** **ObjectPropertyExpression** **ClassExpression** ')'

ObjectPropertyRange := 'ObjectPropertyRange' '(' **axiomAnnotations** **ObjectPropertyExpression** **ClassExpression** ')'

InverseObjectProperties := 'InverseObjectProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** ')'

FunctionalObjectProperty := 'FunctionalObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

InverseFunctionalObjectProperty := 'InverseFunctionalObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

ReflexiveObjectProperty := 'ReflexiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

IrreflexiveObjectProperty := 'IrreflexiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

SymmetricObjectProperty := 'SymmetricObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

AsymmetricObjectProperty := 'AsymmetricObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

TransitiveObjectProperty := 'TransitiveObjectProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

DataPropertyAxiom :=
    SubDataPropertyOf | **EquivalentDataProperties** | **DisjointDataProperties** |
    DataPropertyDomain | **DataPropertyRange** | **FunctionalDataProperty**

SubDataPropertyOf := 'SubDataPropertyOf' '(' **axiomAnnotations** **subDataPropertyExpression** **superDataPropertyExpression** ')'
subDataPropertyExpression := **DataPropertyExpression**
superDataPropertyExpression := **DataPropertyExpression**

EquivalentDataProperties := 'EquivalentDataProperties' '(' **axiomAnnotations** **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

DisjointDataProperties := 'DisjointDataProperties' '(' **axiomAnnotations** **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

DataPropertyDomain := 'DataPropertyDomain' '(' **axiomAnnotations** **DataPropertyExpression** **ClassExpression** ')'

DataPropertyRange := 'DataPropertyRange' '(' **axiomAnnotations** **DataPropertyExpression** **DataRange** ')'

FunctionalDataProperty := 'FunctionalDataProperty' '(' **axiomAnnotations** **DataPropertyExpression** ')'

DatatypeDefinition := 'DatatypeDefinition' '(' **axiomAnnotations** **Datatype** **DataRange** ')'

HasKey := 'HasKey' '(' **axiomAnnotations** **ClassExpression** '(' { **ObjectPropertyExpression** } ')' '(' { **DataPropertyExpression** } ')' ')'

Assertion :=
    SameIndividual | **DifferentIndividuals** | **ClassAssertion** |
    ObjectPropertyAssertion | **NegativeObjectPropertyAssertion** |
    DataPropertyAssertion | **NegativeDataPropertyAssertion**

sourceIndividual := **Individual**
targetIndividual := **Individual**
targetValue := **Literal**

SameIndividual := 'SameIndividual' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

DifferentIndividuals := 'DifferentIndividuals' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

ClassAssertion := 'ClassAssertion' '(' **axiomAnnotations** **ClassExpression** **Individual** ')'

ObjectPropertyAssertion := 'ObjectPropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

NegativeObjectPropertyAssertion := 'NegativeObjectPropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

DataPropertyAssertion := 'DataPropertyAssertion' '(' **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** ')'

NegativeDataPropertyAssertion := 'NegativeDataPropertyAssertion' '(' **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** ')'

14 Appendix: Change Log (Informative)

14.1 Changes Since Recommendation

This section summarizes the changes to this document since the Recommendation of 27 October 2009.

- With the publication of the XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes Recommendation of 5 April 2012, the elements of OWL 2 which are based on XSD 1.1 are now considered required, and the note detailing the optional dependency on the XSD 1.1 Candidate Recommendation of 30 April, 2009 has been removed from the "Status of this Document" section.
- References to and dependencies on the XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes Candidate Recommendation of 30 April, 2009 were amended to reflect the XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes Recommendation of 5 April 2012.
- The document parsing specification in Section 2.2 was made more precise by the addition of a canonical process (showing how a sequence of characters should be converted into a sequence of terminal symbols) and several examples.
- The restrictions on the axiom closure specified in Section 11.2 were explained in more detail.
- Minor typographical errors were corrected as detailed on the OWL 2 Errata page.

14.2 Changes Since Proposed Recommendation

This section summarizes the changes to this document since the Proposed Recommendation of 22 September, 2009.

- Some minor editorial changes were made.

14.3 Changes Since Candidate Recommendation

This section summarizes the changes to this document since the Candidate Recommendation of 11 June, 2009.

- The "Feature At Risk" warnings w.r.t. the owl:rational and rdf:XMLLiteral datatypes were removed: implementation support has been adequately demonstrated, and the features are no longer considered at risk (see Resolution 5 and Resolution 6, 05 August 2009).
- The definition of the OWL 2 datatype map was strengthened so as to make it clear that OWL 2 DL ontologies can include only the specified datatypes, facets and values.
- The definition of **HasKey** axioms was fixed to make it clear that each such axiom must involve at least one property.
- The restrictions in Section 5.2 on the usage of datatypes in an OWL 2 DL ontology were clarified.
- The restrictions in Section 5.7 on the allowed lexical forms of literals were weakened to apply to OWL 2 DL ontologies only.
- The restrictions in Section 7.5 on the allowed facets in facet restrictions were weakened to apply to OWL 2 DL ontologies only.
- The restrictions in Section 11.2 on the usage of datatypes were rephrased for clarity.
- The restrictions in Section 11.2 on the usage of anonymous individuals were rephrased for clarity.
- Sundry small editorial changes were made.

14.4 Changes Since Last Call

This section summarizes the changes to this document since the Last Call Working Draft of 21 April, 2009.

- Per the warning in an "at-risk" comment, the name of *owl:dateTime* was changed to *xsd:dateTime* to conform to the name that will be part of XML Schema.
- The name of rdf:text was changed to rdf:PlainLiteral.
- Two of the examples were fixed.
- Some minor editorial changes were made.

15 Acknowledgments

The starting point for the development of OWL 2 was the OWL1.1 member submission, itself a result of user and developer feedback, and in particular of information gathered during the OWL Experiences and Directions (OWLED) Workshop series. The working group also considered postponed issues from the WebOnt Working Group.

This document has been produced by the OWL Working Group (see below), and its contents reflect extensive discussions within the Working Group as a whole. The editors extend special thanks to Bernardo Cuenca Grau (Oxford University Computing Laboratory), Ivan Herman (W3C/ERCIM), Mike Smith (Clark & Parsia) and Vojtech Svatek (K-Space) for their thorough reviews.

The regular attendees at meetings of the OWL Working Group at the time of publication of this document were: Jie Bao (RPI), Diego Calvanese (Free University of Bozen-Bolzano), Bernardo Cuenca Grau (Oxford University Computing Laboratory), Martin Dzbor (Open University), Achille Fokoue (IBM Corporation), Christine Golbreich (Université de Versailles St-Quentin and LIRMM), Sandro Hawke (W3C/MIT), Ivan Herman (W3C/ERCIM), Rinke Hoekstra (University of Amsterdam), Ian Horrocks (Oxford University Computing Laboratory), Elisa Kendall (Sandpiper Software), Markus Krötzsch (FZI), Carsten Lutz (Universität Bremen), Deborah L. McGuinness (RPI), Boris Motik (Oxford University Computing Laboratory), Jeff Pan (University of Aberdeen), Bijan Parsia (University of Manchester), Peter F. Patel-Schneider (Bell Labs Research, Alcatel-Lucent), Sebastian Rudolph (FZI), Alan Ruttenberg (Science Commons), Uli Sattler (University of Manchester), Michael Schneider (FZI), Mike Smith (Clark & Parsia), Evan Wallace (NIST), Zhe Wu (Oracle Corporation), and Antoine Zimmermann (DERI Galway). We would also like to thank past members of the working group: Jeremy Carroll, Jim Hendler, and Vipul Kashyap.

16 References

16.1 Normative References

[BCP 47]
BCP 47 - Tags for Identifying Languages. A. Phillips and M. Davis, eds. IETF, September 2006. http://www.rfc-editor.org/rfc/bcp/bcp47.txt

[ISO 8601:2004]
ISO 8601:2004. Representations of dates and times. ISO (International Organization for Standardization).

[ISO/IEC 10646]
ISO/IEC 10646-1:2000. Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane and ISO/IEC 10646-2:2001. Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization. ISO (International Organization for Standardization).

[RFC 2119]
RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. Network Working Group, S. Bradner. IETF, March 1997, http://www.ietf.org/rfc/rfc2119.txt
[RFC 3629]
RFC 3629: UTF-8, a transformation format of ISO 10646. F. Yergeau. IETF, November 2003, http://www.ietf.org/rfc/rfc3629.txt
[RFC 3987]
RFC 3987: Internationalized Resource Identifiers (IRIs). M. Duerst and M. Suignard. IETF, January 2005, http://www.ietf.org/rfc/rfc3987.txt
[RDF Concepts]
Resource Description Framework (RDF): Concepts and Abstract Syntax. Graham Klyne and Jeremy J. Carroll, eds. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. Latest version available as http://www.w3.org/TR/rdf-concepts/.
[RDF Test Cases]
RDF Test Cases. Jan Grant and Dave Beckett, eds. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/. Latest version available as http://www.w3.org/TR/rdf-testcases/.
[RDF:PLAINLITERAL]
rdf:PlainLiteral: A Datatype for RDF Plain Literals Jie Bao, Sandro Hawke, Boris Motik, Peter F. Patel-Schneider, Axel Polleres, eds. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-rdf-plain-literal-20121018/. Latest version available at http://www.w3.org/TR/rdf-plain-literal-1/.
[SPARQL]
SPARQL Query Language for RDF. Eric Prud'hommeaux and Andy Seaborne, eds. W3C Recommendation, 15 January 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. Latest version available as http://www.w3.org/TR/rdf-sparql-query/.
[UML]
OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Object Management Group, OMG Available Specification, November 2007, http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.
[UNICODE]
The Unicode Standard. The Unicode Consortium, Version 5.1.0, ISBN 0-321-48091-0, as updated from time to time by the publication of new versions. (See http://www.unicode.org/unicode/standard/versions/ for the latest version and additional information on versions of the standard and of the Unicode Character Database).
[XML]
Extensible Markup Language (XML) 1.0 (Fifth Edition). Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, eds. W3C Recommendation, 26 November 2008, http://www.w3.org/TR/2008/REC-xml-20081126/. Latest version available as http://www.w3.org/TR/xml/.
[XML Schema Datatypes]
W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. David Peterson, Shudi (Sandy) Gao, Ashok Malhotra, C. M. Sperberg-McQueen, and Henry S. Thompson, eds. (Version 1.1) and Paul V. Biron, and Ashok Malhotra, eds. (Version 1.0). W3C Recommendation, 5 April 2012, http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. Latest version available as http://www.w3.org/TR/xmlschema11-2/.

16.2 Nonnormative References

[Description Logics]
The Description Logic Handbook: Theory, Implementation, and Applications, second edition. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, eds. Cambridge University Press, 2007. Also see the *Description Logics Home Page*.
[DL-Safe]
Query Answering for OWL-DL with Rules. Boris Motik, Ulrike Sattler and Rudi Studer. Journal of Web Semantics: Science, Services and Agents on the World Wide Web, 3(1):41–60, 2005.
[MIME Consistency]
Internet Media Type registration, consistency of use. Tim Bray, ed. W3C TAG Finding, 30 April 2004.
[MOF]
Meta Object Facility (MOF) Core Specification, version 2.0. Object Management Group, OMG Available Specification January 2006, http://www.omg.org/spec/MOF/2.0/PDF/.
[OWL 2 RDF Mapping]
OWL 2 Web Ontology Language: Mapping to RDF Graphs Peter F. Patel-Schneider, Boris Motik, eds. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-owl2-mapping-to-rdf-20121018/. Latest version available at http://www.w3.org/TR/owl2-mapping-to-rdf/.
[OWL 2 Direct Semantics]
OWL 2 Web Ontology Language: Direct Semantics Boris Motik, Peter F. Patel-Schneider, Bernardo Cuenca Grau, eds. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-owl2-direct-semantics-20121018/. Latest version available at http://www.w3.org/TR/owl2-direct-semantics/.
[OWL 2 RDF-Based Semantics]
OWL 2 Web Ontology Language: RDF-Based Semantics Michael Schneider, editor. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-owl2-rdf-based-semantics-20121018/. Latest version available at http://www.w3.org/TR/owl2-rdf-based-semantics/.
[OWL 2 Conformance]
OWL 2 Web Ontology Language: Conformance Michael Smith, Ian Horrocks, Markus Krötzsch, Birte Glimm, eds. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-owl2-conformance-20121018/. Latest version available at http://www.w3.org/TR/owl2-conformance/.
[OWL 2 XML Serialization]
OWL 2 Web Ontology Language: XML Serialization Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, eds. W3C Proposed Edited Recommendation, 18 October 2012, http://www.w3.org/TR/2012/PER-owl2-xml-serialization-20121018/. Latest version available at http://www.w3.org/TR/owl2-xml-serialization/.
[RDF Syntax]
RDF/XML Syntax Specification (Revised). Dave Beckett, ed. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. Latest version available as http://www.w3.org/TR/rdf-syntax-grammar/.
[Register MIME]
Register an Internet Media Type for a W3C Spec. Philippe Le Hégaret, ed. W3C Guidebook.
[RFC 3986]
RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee, R. Fielding, and L. Masinter. IETF, January 2005, http://www.ietf.org/rfc/rfc3986.txt
[SROIQ]
The Even More Irresistible SROIQ. Ian Horrocks, Oliver Kutz, and Uli Sattler. In Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). AAAI Press, 2006.
[UNISEC]
Unicode Security Considerations. Mark Davis and Michel Suignard. Unicode technical report 36, 23 July 2008, http://www.unicode.org/reports/tr36/tr36-7.html. Latest version available as http://www.unicode.org/reports/tr36/.
[XML Namespaces]
Namespaces in XML 1.0 (Second Edition). Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin, eds. W3C Recommendation, 16 August 2006, http://www.w3.org/TR/2006/REC-xml-names-20060816/. Latest version available as http://www.w3.org/TR/REC-xml-names/.