OWL 2 Web Ontology Language – with Gellish Expression Format Primer (Second Edition)

W3C Recommendation 11 December 2012 –

Amended and issued by:

dr. ir. Andries van Renssen, 9 August 2013
Gellish.net
info@gellish.net

This (not-amended) version:
http://www.w3.org/TR/2012/REC-owl2-primer-20121211/
Latest version (series 2):
http://www.w3.org/TR/owl2-primer/
Latest Recommendation:
http://www.w3.org/TR/owl-primer
Previous version:
http://www.w3.org/TR/2012/PER-owl2-primer-20121018/
Editors:

Pascal Hitzler, Wright State University
Markus Krötzsch, University of Oxford
Bijan Parsia, University of Manchester
Peter F. Patel-Schneider, Nuance Communications
Sebastian Rudolph, FZI Research Center for Information Technology

Please refer to the **errata** for this document, which may include some normative corrections.

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.

See also translations.

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.

Gellish Formal English is a formal ontological language that is a formalized subset of natural English with a large amount of standardized meaning (semantics), in the form of standardized concepts, especially kinds of relations. The full semantics of Formal English is defined in the Gellish Formal English Dictionary-Taxonomy, which actually is an Ontology (see the Gellish Upper Ontology in the TOPini section in particular at http://www.formalenglish.net/dictionary). Semantically Formal English is a superset of OWL 2. The formal definition of the OWL semantics is applicable also to the equivalent Gellish concepts. The 2008 version of Gellish Formal English is free of charge available from http://www.gellish.net/index.php/downloads.html. The latest OWL 2 compliant version is available via the webshop http://www.gellish.net/index.php/shop.html. Formal English is described in the book 'Semantic Modeling in Formal English' (http://www.lulu.com/shop/dr-ir-andries-van-renssen/semantic-modeling-in-formal-english/paperback/product-21538016.html).

The Gellish Expression Format is a syntax for expressions in Formal English or in other formalized natural languages in the Gellish family of formal languages. The Gellish Expression Format is basically a tabular format that can be directly used for database definitions. The format can also be used as an alternative format for OWL expressions, as is shown in this Primer. The Gellish Expression Format syntax is defined in the document 'Gellish Syntax, Definition of Universal Semantic Databases and Messages' (see http://www.gellish.net/index.php/downloads.html). That document also defines a collection of kinds of Contextual Facts, comparable with the Dublin Core.

This primer provides an approachable introduction to OWL 2, including orientation for those coming from other disciplines, a running example showing how OWL 2 can be used to represent first simple information and then more complex information, how OWL 2 manages ontologies, and finally the distinctions between the various sublanguages of OWL 2.

This document does not describe Gellish Formal English, but the OWL constructs are annotated with a description of the equivalent solutions that are available in Gellish Formal English. This document therefore describes only the subset of Formal English that covers the OWL 2 semantics.

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.

Please Send Comments

Please send any comments to public-owl-comments@w3.org (public archive). Although work on this document by the OWL Working Group is complete, comments may be addressed in the errata or in future revisions. Open discussion among developers is welcome at public-owl-dev@w3.org (public archive).

Endorsed By W3C

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

Patents

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. This document is informative only. 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.

Table of Contents

1 Introduction

The W3C OWL 2 Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit. OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies. OWL is part of the W3C's Semantic Web technology stack, which includes RDF [*RDF Concepts*] and SPARQL [*SPARQL*]. Gellish Formal English is a formal language that defines many more concepts and kinds of relations, which include also identical or similar concepts as defined in OWL.

The key goal of the primer is to help develop insight into OWL, its strengths, and its weaknesses. The core of the primer is an introduction to most of the language features of OWL by way of a running example. Most of the examples in the primer are taken from a sample ontology (which is presented entirely in an appendix). This sample ontology is designed to touch the key language features of OWL in an understandable way and not, in itself, to be an example of a good ontology. The annotations describe how Gellish Formal English can be implemented as an extension of OWL.

1.1 Guide to this Document

This document is intended to provide an initial understanding about OWL 2. In particular it is supposed to be accessible for people yet unfamiliar with the topic. Therefore, we start with giving some high-level introduction on the nature of OWL 2 in Section 2 before Section 3 provides some very basic notions in knowledge representation and explains how they relate to terms used in OWL 2. Readers familiar with knowledge representation and reasoning might only skim through this section to get acquainted with the OWL 2 terminology.

Sections 4–8 describe most of the language features that OWL provides, starting from very basic ones and proceeding to the more sophisticated. Section 4 presents and discusses the elementary modeling features of OWL 2 before in Section 5 complex classes are introduced. Section 6 addresses advanced modeling features for properties. Section 7 focuses on advanced modeling related to datatypes. Section 8 concludes with extra-logical features used mainly for ontology management purposes.

In Section 9 we address the differences between OWL 2 DL and OWL 2 Full, the two semantic views of OWL, while in Section 10 we describe the three tractable sublanguages of OWL 2, called profiles. Tool support for OWL 2 is addressed in Section 11 and in Section 12 we give pointers on where to continue reading after our informal introduction to OWL 2.

Finally, Section 13 lists the complete example ontology used in this document.

For a comprehensive listing of the OWL 2 language features, see the OWL 2 Quick Reference Guide [*OWL 2 Quick Guide*] which provides links into the corresponding sections of the appropriate documents concerning syntax and examples.

For readers already familiar with OWL 1, OWL 2 New Features and Rationale [*OWL 2 New Features and Rationale*] provides a comprehensive overview of what has changed in OWL 2.

1.2 OWL Syntaxes

OWL is a language to be used in the Semantic Web, so names in OWL are international resource identifiers (IRIs) [*RFC 3987*]. As IRIs are long, we will often make use of abbreviation mechanisms for writing them in OWL. The way in which such abbreviations work is specific to each syntactic format that can be used to encode OWL ontologies, but the examples in this document can generally be understood without knowing these details. Appropriate declarations for namespaces and related mechanisms are given further below in Section 8.2.

There are various syntaxes available for OWL which serve various purposes. The Functional-Style syntax [*OWL 2 Specification*] is designed to be easier for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners. The RDF/XML syntax for OWL is just RDF/XML, with a particular translation for the OWL constructs [*OWL 2 RDF Mapping*]. This is the only syntax that is mandatory to be supported by all OWL 2 tools. The Manchester syntax [*OWL 2 Manchester Syntax*] is an OWL syntax that is designed to be easier for non-logicians to read. The OWL XML syntax is an XML syntax for OWL defined by an XML schema [*OWL 2 XML*]. There are tools that can translate between the different syntaxes for OWL. In many syntactic forms, OWL language constructs are also represented by IRIs, and some declarations might be needed to use the abbreviated forms as in the examples. Again, necessary details are found in Section 8.2.

The examples and the sample ontology in the appendix can be viewed as any of the four different syntaxes, and we provide both RDF/XML [*RDF Syntax*] and Turtle [*Turtle*] serializations for the RDF-based syntax. You can control which syntaxes are shown throughout the document by using the buttons below.

The buttons below can be used to show or hide the available syntaxes.

2 What is OWL 2?

OWL 2 is a language for expressing *ontologies*. The term *ontology* has a complex history both in and out of computer science, but we use it to mean a certain kind of computational artifact – i.e., something akin to a program, an XML schema, or a web page – generally presented as a document. An ontology is a set of precise descriptive statements about some part of the world (usually referred to as the *domain of interest* or the *subject matter* of the ontology). Precise descriptions satisfy several purposes: most notably, they prevent misunderstandings in human communication and they ensure that software behaves in a uniform, predictable way and works well with other software.

In order to precisely describe a domain of interest, it is helpful to come up with a set of central terms – often called vocabulary – and fix their meaning. Besides a concise natural language definition, the meaning of a term can be characterized by stating how this term is interrelated to the other terms. A *terminology*, providing a vocabulary together with such interrelation information constitutes an essential part of a typical OWL 2 document. Besides this terminological knowledge, an ontology might also contain so called assertional knowledge that deals with concrete objects of the considered domain rather than general notions.

OWL 2 is not a programming language: OWL 2 is *declarative*, i.e. it describes a state of affairs in a logical way. Appropriate tools (so-called reasoners) can then be used to infer further information about that state of affairs. How these inferences are realized algorithmically is not part of the OWL document but depends on the specific implementations. Still, the correct answer to any such question is predetermined by the formal semantics (which comes in two versions: the *Direct Semantics* [*OWL 2 Direct Semantics*] and the *RDF-Based Semantics* [*OWL 2 RDF-Based Semantics*]). Only implementations that comply with these semantics will be regarded as OWL 2 conformant (see [*OWL 2 Conformance*]). Through its declarative nature, the activity of creating OWL 2 documents is conceptually different from programming. Still, as in both cases complex formal documents are created, certain notions from software engineering can be transferred to ontology engineering, such as methodological and collaborative aspects, modularization, patterns, etc.

OWL 2 is not a schema language for syntax conformance. Unlike XML, OWL 2 does not provide elaborate means to prescribe how a document should be structured syntactically. In particular, there is no way to enforce that a certain piece of information (like the social security number of a person) has to be syntactically present. This should be kept in mind as OWL has some features that a user might misinterpret this way.

OWL 2 is not a database framework. Admittedly, OWL 2 documents store information and so do databases. Moreover a certain analogy between assertional information and database content as well as terminological information and database schemata can be drawn. However, usually there are crucial differences in the underlying assumptions (technically: the used semantics). If some fact is not present in a database, it is usually considered false (the so-called *closed-world assumption*) whereas in the case of an OWL 2 document it may simply be missing (but possibly true), following the *open-world assumption*. Moreover, database schemata often come with the prescriptive constraint semantics mentioned above. Still, technically, databases provide a viable backbone in many ontology-oriented systems.

3 Modeling Knowledge: Basic Notions

OWL 2 is a knowledge representation language, designed to formulate, exchange and reason with knowledge about a domain of interest. Some fundamental notions should first be explained to understand how knowledge is represented in OWL 2. These basic notions are:

- **Axioms:** the basic statements that an OWL or Formal English ontology expresses

- **Entities:** elements used to refer to real-world objects
- **Expressions:** combinations of entities to form complex descriptions from basic ones

While OWL 2 aims to capture knowledge, the kind of "knowledge" that can be represented by OWL does of course not reflect all aspects of human knowledge. OWL can be considered as a powerful general-purpose modeling language for certain parts of human knowledge. The results of the modeling processes are called *ontologies* – a terminology that also helps to avoid confusion since the term "model" is often used in a rather different sense in knowledge representation.

Now, in order to formulate knowledge explicitly, it is useful to assume that it consists of elementary pieces that are often referred to as *statements* or *propositions*. Statements like "it is raining" or "every man is mortal" are typical examples for such basic propositions. Indeed, every OWL 2 ontology is essentially just a collection of such basic "pieces of knowledge." Statements that are made in an ontology are called *axioms* in OWL 2, and the ontology asserts that its axioms are true. In general, OWL statements might be either true or false given a certain state of affairs. This distinguishes them from *entities* and *expressions* as described further below. In Formal English statements may also have a probability, and not only statemenst are made, but also possibilities, requirements, promises, denials, etc.

When humans think, they draw consequences from their knowledge. An important feature of OWL is that it captures this aspect of human intelligence for the forms of knowledge that it can represent. But what does it mean, generally speaking, that a statement is a consequence of other statements? Essentially it means that this statement is true whenever the other statements are. In OWL terms: we say, a set of statements *A entails* a statement *a* if in any state of affairs wherein all statements from *A* are true, also *a* is true. Moreover, a set of statements may be *consistent* (that is, there is a possible state of affairs in which all the statements in the set are jointly true) or *inconsistent* (there is no such state of affairs). The formal semantics of OWL specifies, in essence, for which possible "states of affairs" a particular set of OWL statements is true.

There are OWL tools – reasoners – that can automatically compute consequences. The way ontological axioms interact can be very subtle and difficult for people to understand. This is both a strength and a weakness of OWL 2. It is a strength because OWL 2 tools can discover information that a person would not have spotted. This allows knowledge engineers to model more directly and the system to provide useful feedback and critique of the modeling. It is a weakness because it is comparatively difficult for humans to immediately foresee the actual effect of various constructs in various combinations. Tool support ameliorates the situation but successful knowledge engineering often still requires some amount of training and experience.

Having a closer look at statements in OWL, we see that they are rarely "monolithic" but more often have some internal structure that can be explicitly represented. They normally refer to objects of the world and describe them e.g. by putting them into categories (like "Mary is female") or saying something about their relation ("John and Mary are married"). All atomic constituents of statements, be they objects (John,

Mary), categories (female) or relations (married) are called *entities*. In OWL 2, we denote objects as *individuals*, categories as *classes* and relations as *properties*. Properties in OWL 2 are further subdivided. *Object properties* relate objects to objects (like a person to their spouse), while *datatype properties* assign data values to objects (like an age to a person). *Annotation properties* are used to encode information about (parts of) the ontology itself (like the author and creation date of an axiom) instead of the domain of interest.

As a central feature of OWL, names of entities can be combined into *expressions* using so called *constructors*. As a basic example, the atomic classes “female” and “professor” could be combined conjunctively to describe the class of female professors. The latter would be described by an OWL class expression, that could be used in statements or in other expressions. In this sense, expressions can be seen as new entities which are defined by their structure. In OWL, the constructors for each sort of entity vary greatly. The expression language for classes is very rich and sophisticated, whereas the expression language for properties is much less so. These differences have historical as well as technical reasons.

4 Classes, Properties, and Individuals – And Basic Modeling With Them

After these general considerations, we now engage in the details of modeling with OWL 2. In the subsequent sections, we introduce the essential modeling features that OWL 2 offers, provide examples and give some general comments on how to use them. We proceed from basic features, which are essentially available in any modeling language, to more advanced constructs.

Thereby we will represent information about a particular family. Note that we do not intend this example to be representative of the sorts of domains OWL should be used for, or as a canonical example of good modeling with OWL, or a correct representation of the rather complex, shifting, and culturally dependent domain of families. Instead, we intend it to be a rather simple exhibition of various features of OWL.

4.1 Classes and Instances

We start by introducing the persons we are talking about. This can be done as follows:

Functional-Style Syntax

```
ClassAssertion( :Person :Mary )
```

Gellish Expression Format syntax

Mary is classified as a person

or using the extended version with Gellish UID’s that enable to distinguish homonyms:

Intention	UID of fact	UID of left hand object	Name of left hand object	UID of relation type	Name of relation type	UID of right hand object	Name of right hand object
statement	201	101	Mary	1225	is classified as a	990010	person

Notes on Gellish Formal English Formal English:

1. The intention 'statement' can also be another intention, such as 'question'.
2. The fact as well as other things have a unique identifier (UID) that is language independent. UID's below 1000 and above 1 billion are free.
3. The concept 'person' is not part of OWL 2, however it.is defined in the Gellish Formal English Dictionary-Taxonomy and thus it is part of the Gellish Formal English language.
4. Each Formal English expression is accompanied by a number of standard contextual facts that express context for interpretation, such as: who expressed it, when, and with which status, etc.

This statement talks about an individual named Mary and states that this individual is a person. More technically, *being a person* is expressed by stating that Mary belongs to (or "is a member of" or, even more technically, "is an instance of") the *class* of all persons or "is classified as a" person. In general classes are used to group individuals that have something in common in order to refer to them. Hence, classes essentially represent sets of individuals. In modeling, classes are often used to denote the set of objects comprised by a concept of human thinking, like the concept *person* or the concept *woman*. Consequently, we can use the same type of statement to indicate that Mary is a woman by expressing that she is an instance of the class of women or "is classified as a" woman:

Functional-Style Syntax

```
ClassAssertion( :Woman :Mary )
```

Gellish Expression Format syntax

Mary is classified as a woman

or using the extended version with Gellish UID's that enable to distinguish homonyms:

Intention	UID of fact	UID of left hand	Name of left hand	UID of relation type	Name of relation type	UID of right hand	Name of right hand

		object	object			object	object
statement	202	101	Mary	1225	is classified as a	990013	woman

Hereby it also becomes clear that class membership is not exclusive: as there may be diverse criteria to group individuals (like gender, age, shoe size, etc.), one individual may well belong to several classes simultaneously. In Formal English it is unlikely that a shoe-size would be accepted as a class of woman. But sizes of shoes would be.

4.2 Class Hierarchies

In the previous section, we were talking about two classes: the class of all persons and that of all women. To the human reader it is clear that these two classes are in a special relationship: Person is more general than Woman, meaning that whenever we know some individual to be a woman, that individual must be a person. However, this correspondence cannot be derived from the labels “Person” and “Woman” but is part of the human background knowledge about the world and our usage of those terms. Therefore, in order to enable a system to draw the desired conclusions, it has to be informed about this correspondence. In OWL 2, this is done by a so-called subclass axiom:

Functional-Style Syntax

```
SubClassOf( :Woman :Person )
```

Gellish Expression Format syntax

woman is a subclass of person

or synonymously:

woman is a kind of person

Notes on Gellish Formal English:

1. This statement and many others are already included in the Gellish Dictionary-Taxonomy.

The presence of this axiom in an ontology enables reasoners to infer for every individual which is specified as an instance of the class Woman, that it is an instance of the class Person as well. As a rule of thumb, a subclass relationship between two classes A and B can be specified, if the phrase “every A is a B” makes sense and is correct.

It is common in ontological modeling to use subclass statements not only for sporadically declaring such interdependencies, but to model whole *class hierarchies* by specifying the generalization relationships of all classes in the domain of interest. Suppose we also want to state that all mothers are women:

Functional-Style Syntax

```
SubClassOf( :Mother :Woman )
```

Gellish Expression Format syntax

mother is a subclass of woman

Then a reasoner could not only derive for every single individual that is classified as mother, that it is also a woman (and consequently a person), but also that Mother must be a subclass of Person – coinciding with our intuition. Technically, this means that the subclass relationship between classes is *transitive*. Besides this, it is also *reflexive*, meaning that every class is its own subclass – this is intuitive as well since clearly, every person is a person etc.

Classes in our vocabulary may effectively refer to the same sets, and OWL provides a mechanism by which they are considered to be semantically equivalent. For example, we use the term Person and Human interchangeably, meaning that every instance of the class Person is also an instance of class Human, and vice versa. Two classes are considered equivalent if they contain exactly the same individuals. The following example states that the class Person is equivalent to the class Human.

Functional-Style Syntax

```
EquivalentClasses( :Person :Human )
```

Gellish Expression Format syntax

person is a synonym of human

or using the extended version with Gellish UID's that enable to distinguish homonyms:

Intention	UID of fact	UID of left hand object	Name of left hand object	UID of relation type	Name of relation type	UID of right hand object	Name of right hand object
statement	203	990010	human	1981	is a synonym of	990010	person

Notes on Gellish Formal English:

1. The equivalence statement in OWL 2 appears to define that the two terms are synonym terms for the same concept.
2. Note that both terms, human and person, share the same UID in Gellish. Thus in Gellish the software doesn't need to search the for synonyms.
3. Gellish has an additional 'equivalence relation' that specifies that things of two kinds are equivalent, although not identical.

Stating that Person and Human are equivalent amounts exactly to the same as stating that both Person is a subclass of Human and Human is a subclass of Person.

4.3 Class Disjointness

In Section 4.1, we stated that an individual can be an instance of several classes. However, in some cases membership in one class specifically excludes membership in another. For example, if we consider the classes Man and Woman, we know that no individual can be an instance of both classes (for the sake of the example, we disregard biological borderline cases). This "incompatibility relationship" between classes is referred to as *(class) disjointness*. Again, the information that two classes are disjoint is part of our background knowledge and has to be explicitly stated for a reasoning system to make use of it. This is done as follows:

Functional-Style Syntax

```
DisjointClasses( :Woman :Man )
```

Gellish Expression Format syntax

woman is disjoint with man

In practice, disjointness statements are often forgotten or neglected. The arguable reason for this could be that intuitively, classes are considered disjoint unless there is other evidence. By omitting disjointness statements, many potentially useful consequences can get lost. Note that in our example, the disjointness axiom is needed to deduce that Mary is not a man. Moreover, given the above axioms, a reasoner can infer the disjointness of the classes Mother and Man.

4.4 Object Properties

In the preceding sections we were concerned with describing single individuals, their class memberships, and how classes can relate to each other based on their instances. But more often than not, an ontology is also meant to specify how the individuals relate to other individuals. These relationships are central when describing a family. We start by indicating that Mary is John's wife.

Functional-Style Syntax

```
ObjectPropertyAssertion( :hasWife :John :Mary )
```

Gellish Expression Format syntax

John is married with Mary

Note on Gellish:

1. In OWL a kind of relationship is called a property. In Gellish it is called a kind of relation or relation type. The term property is used in Gellish for physical properties, not for relations.

Hereby, the entities describing in which way the individuals are related – like hasWife in our case, are called *properties.*

Note that the order in which the individuals are written is important. While “Mary is John's wife” might be true, “John is Mary's wife” certainly isn't. Indeed, this is a common source of modeling errors that can be avoided by using property names which allow only one unique intuitive reading. In case of nouns (like “wife”), such unambiguous names might be constructions with “of” or with “has” (wifeOf or hasWife). For verbs (like “to love”) an inflected form (loves) or a passive version with “by” (lovedBy) would prevent unintended readings.

Note on Gellish:

1. Gellish synonym phrases, such as <is the husband of> and <has as wife> for the same kind of relation clearly state who has the role of husband and who the role of wife.
2. The role husband is a separate concept in Gellish, whereas a role of that kind can be played by a man. Such a role is an extrinsic aspect of the role player.

We can also state that two individuals are *not* connected by a property. The following, for example, states that Mary is not Bill's wife.

Functional-Style Syntax

```
NegativeObjectPropertyAssertion( :hasWife :Bill :Mary )
```

Gellish Expression Format syntax

denial Bill is married with Mary

Negative property assertions provide a unique opportunity to make statements where we know something that is not true. This kind of information is particularly important in OWL where the default stance is that anything is possible until you say otherwise.

4.5 Property Hierarchies

In Section 4.2 we argued that it is useful to specify that one class membership implies another one. Essentially the same situation can occur for properties: whenever B is known to be A's wife, it is also known to be A's spouse (note, that this is not true the other way round). OWL allows to specify this statement as follows:

Functional-Style Syntax

```
SubObjectPropertyOf( :hasWife :hasSpouse )
```

Gellish Expression Format syntax

has as wife	is a kind of	has as spouse

Note on Gellish:

1. This is a specialization of kinds of relations, called properties in OWL

There is also a syntactic shortcut for property equivalence, which is similar to class equivalence.

4.6 Domain and Range Restrictions

Frequently, the information that two individuals are interconnected by a certain property allows to draw further conclusions about the individuals themselves. In particular, one might infer class memberships. For instance, the statement that B is the wife of A obviously implies that B is a woman while A is a man. So in a way, the statement that two individuals are related via a certain property carries implicit additional information about these individuals. In our example, this additional information can be expressed via class memberships. OWL provides a way to state this correspondence:

Functional-Style Syntax

```
ObjectPropertyDomain( :hasWife :Man )
ObjectPropertyRange( :hasWife :Woman )
```

Gellish Expression Format syntax

has as wife	has by definition as first role a	husband
has as wife	has by definition as second role a	wife
husband	is by definition played by a	man
wife	is by definition played by a	woman

Notes on Gellish Formal English:

1. These and other definitions of kinds of roles and allowed kinds of role players are part of the definition of kinds of relations in Gellish. They are not in OWL; in OWL these are only examples of possible definitions.
2. The concepts husband and wife as kinds of roles are explicitly defined in Gellish, but are missing in OWL.

Having these two axioms in place and given e.g. the information that Sasha is related to Hillary via the property hasWife, a reasoner would be able to infer that Sasha is a man and Hillary a woman.

4.7 Equality and Inequality of Individuals

Note that from the information given so far, it can be deduced that John and Mary are not the same individual as they are known to be instances of the disjoint classes Man and Woman, respectively. However, if we add information about another family member, say Bill, and indicate that he is a man, then there is nothing said so far that implies that John and Bill are not the same. OWL does not make the assumption that different names are names for different individuals. (This lack of a required "unique names assumption" is particularly well-suited to Semantic Web applications where names may be coined by different organizations at different times unknowingly referring to the same individual.) Hence, if we want to exclude the option of John and Bill being the same individual, this has to be explicitly specified as follows:

Functional-Style Syntax

```
DifferentIndividuals( :John :Bill )
```

Gellish Expression Format syntax

102	John
103	Bill

Notes on Gellish Formal English:

1. The different UID's that represent things in Gellish imply that they are different things.
2. OWL does not allow for homonyms, such as two persons with the same name. The different UID's in Gellish allow for homonyms provided they are distinguished by their different language communities.

It is also possible to state that two names refer to (denote) the same individual. For example, we can say that James and Jim are the same individual.

Functional-Style Syntax

```
SameIndividual( :James :Jim )
```

Gellish Expression Format syntax

104 James
104 Jim

Notes on Gellish Formal English:

1. The identical UID's imply that the two names are different names for the same thing.
2. Gellish also enables to specify in which language communities these names are the preferred terms.

This would enable a reasoner to infer that any information given about the individual James also holds for the individual Jim.

4.8 Datatypes

So far, we have seen how we can describe individuals via class memberships and via their relationships to other individuals. In many cases, however, individuals are to be described by data values. Think of a person's birth date, his age, his email address etc. For this purpose, OWL provides another kind of properties, so-called *Datatype properties*. These properties relate individuals to data values (instead of to other individuals), and many of the XML Schema datatypes [*XML Schema Datatypes*] can be used. The following is an example using a datatype property. It states that John's age is 51.

Functional-Style Syntax

```
DataPropertyAssertion( :hasAge :John "51"^^xsd:integer )
```

Gellish Expression Format syntax

John	has aspect	age of John	
age of John	is classified as a	age	
age of John	has on scale a value equal to	51	year

Notes on Gellish Formal English:

1. In OWL the concept age is not defined, although hasAge can be. Gellish has defined kinds of properties such as age in its dictionary-taxonomy.
2. Gellish enables specification of the date of validity of this statement.
3. Numbers, such as 51, are defined in Gellish as being a number, possibly a whole number or natural number.

Likewise, we can state that Jack's age is *not* 53.

Functional-Style Syntax

```
NegativeDataPropertyAssertion( :hasAge :Jack "53"^^xsd:integer )
```

Gellish Expression Format syntax

denial age of John has on scale a value equal to 53 year

Notes on Gellish Formal English:

1. OWL ingnores the unit of measure here. Ages of baby's are usually given in days, weeks or months.
2. Gellish has defined other kinds of relations between properties and numbers, such as greater than, unequal to, etc.

Domain and range can also be stated for datatype properties as it is done for object properties. In that case, however, the range will be a datatype instead of a class. The following states that the hasAge property is only used to relate persons with non-negative integers.

Functional-Style Syntax

```
DataPropertyDomain( :hasAge :Person )
DataPropertyRange( :hasAge xsd:nonNegativeInteger )
```

Gellish Expression Format syntax

person can have as aspect a age
age can be quantified on scale year

Notes on Gellish Formal English:

1. The OWL statement that ranges of properties, such as has Age, relate to datatypes is not a semantic statement, but a syntactic. A range of a property can be expressed in many ways.
2. The quantification relation on scale relates an aspect (property), such as age, to a number. There the domain is a age and the range is a number.
3. Most of this kind of statements are superfluous in Gellish as Gellish defined already that age is a duration and it defined that a duration can be expressed on a time scale. Year is one of the units of measures that are a time scale.

We would like to point out at this stage a common mistake which easily occurs when using property domains and ranges. In the example just given, which states that the hasAge property is only used to relate persons with non-negative integers, assume that we also specify the information that Felix is in the class Cat and that Felix hasAge 9. From the combined information, it would then be possible to deduce that Felix is also in the class Person, which is probably not intended. This is a commonly modeling error: note that a domain (or range) statement is not a constraint on the knowledge, but allows a reasoner to infer further knowledge. If we state – as in our example – that an age is only given for persons, then everything we give an age for automatically becomes a person.

Note on Gellish: This common mistake in OWL modelling is less common in Gellish modelling, because Gellish makes a distinction whether something can be the case

or is by definition the case. In OWL the above statement specifies that it is by definition the case, which is incorrect.

5 Advanced Class Relationships

In the previous sections we have dealt with classes as something "opaque" carrying a name. We used them to characterize individuals, and related them to other classes via subclass or disjointness statements.

We will now demonstrate how named classes, properties, and individuals can be used as building blocks to define new classes.

5.1 Complex Classes

By means of the language elements described so far, simple ontologies can be modeled. In order to express more complex knowledge, OWL provides logical class constructors. In particular, OWL provides language elements for logical and, or, and not. The corresponding OWL terms are borrowed from set theory: *(class) intersection*, *union* and *complement*. These constructors combine atomic classes – i.e. classes with names – to complex classes.

The *intersection* of two classes consists of exactly those individuals which are instances of both classes. The following example states that the class Mother consists of exactly those objects which are instances of both Woman and Parent:

Functional-Style Syntax

```
EquivalentClasses(
  :Mother
  ObjectIntersectionOf( :Woman :Parent )
)
```

Gellish Expression Format syntax

mother	is by definition a role of a	woman
mother	is a kind of	parent

Notes on Gellish Formal English:

1. Semantically the concept mother is a kind of role, as is parent, because the concept is dependent on a relation between persons and is not determined by intrinsic aspects of a woman. Therefore, it is semantically better to use the Gellish kind of relation (which is not available in OWL) <is by definition a role of a>:
2. The intersection relation in OWL is equivalent to multiple supertypes. The intersection relation in Gellish is a relation between collections and not between kinds of things (classes).

An example for an inference which can be drawn from this is that all instances of the class Mother are also in the class Parent.

The *union* of two classes contains every individual which is contained in at least one of these classes. Therefore we could characterize the class of all parents as the union of the classes Mother and Father:

Functional-Style Syntax

```
EquivalentClasses(
  :Parent
  ObjectUnionOf( :Mother :Father )
)
```

Gellish Expression Format syntax

mother	is a collected kind in	all subtypes of parent
father	is a collected kind in	all subtypes of parent
all subtypes of parent	is a complete collection of subtypes of	parent

Notes on Gellish Formal English:

1. In Gellish kinds (classes) are distinct from collections and therefore the union relation is not applicable to kinds. OWL defines classes as (practically infinite) collections and thus uses union of set relations to define the equivalence, although in practice only subsets of collections of all members can be united.
2. In Gellish it is possible to specify that a collections of subtypes is an incomplete collection of subtypes.

The *complement* of a class corresponds to logical negation: it consists of exactly those objects which are not members of the class itself. The following definition of childless persons uses the class complement and also demonstrates that class constructors can be nested:

Functional-Style Syntax

```
EquivalentClasses(
  :ChildlessPerson
  ObjectIntersectionOf(
    :Person
    ObjectComplementOf( :Parent )
  )
)
```

Gellish Expression Format syntax

childless person | is by definition a role of a | person
denial | childless person | is a kind of | parent

Notes on Gellish Formal English:

1. The concept childless person is defined as a role because the concept is dependent on a relation between persons and not by an intrinsic aspect of a person..

All the above examples demonstrate the usage of class constructors in order to *define* new classes as combination of others. But, of course, it is also possible to use class constructors together with a subclass statement in order to indicate necessary, but not sufficient, conditions for a class. The following statement indicates that every Grandfather is both a man and a parent (whereas the converse is not necessarily true):

Functional-Style Syntax

```
SubClassOf(
  :Grandfather
  ObjectIntersectionOf( :Man :Parent )
)
```

Gellish Expression Format syntax

grandfather | is by definition a role of a | man
grandfather | is a kind of | parent

Notes on Gellish Formal English:

1. The Manchester syntax illustrates that “equivalent to” implies a completeness of the definition model, whereas “subclass of” is a necessary but possibly not sufficiently complete definition model.
   In Gellish this requires the statement that the collection of expressions that form the definition model is incomplete.
   The practicality of this is questionable, as completeness of definition models is a difficult topic that is only solved in simple academic examples..

In general, complex classes can be used in every place where named classes can occur, hence also in class assertions. This is demonstrated by the following example which asserts that Jack is a person but not a parent.

Functional-Style Syntax

```
ClassAssertion(
  ObjectIntersectionOf(
    :Person
    ObjectComplementOf( :Parent )
  )
  :Jack
```

```
)
```

Gellish Expression Format syntax

Jack	is classified as a		person
denial	Jack	is classified by role as a	parent

Notes on Gellish Formal English:

1. The kind of relation <is classified by role as a> has as synonym <has a role as a>
2. In Gellish it is allowed to replace the classification by role relation by a classification relation. Then, if the classifier (e.g. parent) is correctly defined as a subtype of role, then it can be deduced by logic that the classification is in fact a classification by role.

5.2 Property Restrictions

Property restrictions provide another type of logic-based constructors for complex classes. As the name suggests, property restrictions use constructors involving properties.

One property restriction called *existential quantification* defines a class as the set of all individuals that are connected via a particular property (AvR: in Gellish a particular kind of relation) to another individual which is an instance of a certain class. This is best explained by an example, like the following which defines the class of parents as the class of individuals that are linked to a person by the hasChild property.

Functional-Style Syntax

```
EquivalentClasses(
  :Parent
  ObjectSomeValuesFrom( :hasChild :Person )
)
```

Gellish Expression Format syntax

parent has by definition as child a[1,n] person

Notes on Gellish Formal English:

1. The right hand cardinality constraint specifies that a parent has by definition one or more persons as child. This explicit cardinality specification is superfluous, because the defaulit for a definition relation is [1,n].
2. The concept 'parent' is a role of a person. According to the rules of Gellish, a kind of relation, such as <has by definition as child a>, therefore applies to the person that is a player of such a role.

This means that there is an expectation that for every instance of Parent, there exists at least one child, and that child is a member of the class Person. This is useful to

capture *incomplete* knowledge*. For example, Sally tells us that Bob is a parent,* and therefore we can infer that he has at least one child even if we don't know their name. Natural language indicators for the usage of existential quantification are words like “some,” or “one.”

Another property restriction, called *universal quantification* is used to describe a class of individuals for which all related individuals must be instances of a given class. We can use the following statement to indicate that somebody is a happy person exactly if all their children are happy persons.

Functional-Style Syntax

```
EquivalentClasses(
  :HappyPerson
  ObjectAllValuesFrom( :hasChild :HappyPerson )
)
```

Gellish Expression Format syntax

happy person has by definition as child only a happy person

Notes on Gellish Formal English:

1. The kind of relation <has by definition as child only a> will be a user defined extension of Gellish that is a subtype of the universal quantification relation and a subtype of the <has by definition as child a> relation.
2. This example is semantically not well defined as its self referential nature causes that it cannot not hold for all children, because there will always be a child that does not have children and thus does not have happy children.

This example also shows that OWL statements are allowed to be in a certain way self-referential; the class HappyPerson is used on both sides of the equivalence statement.

The usage of property restrictions may cause some conceptual confusion to “modeling beginners.” As a rule of thumb, when translating a natural language statement into a logical axiom, existential quantification occurs far more frequently. Natural language indicators for the usage of universal quantification are words like “only,” “exclusively,” or “nothing but.”

There is one particular misconception concerning the *universal role restriction*. As an example, consider the above happiness axiom. The intuitive reading suggests that in order to be happy, a person must have at least one happy child. Yet, this is not the case: any individual that is not a “starting point” of the property hasChild is a class member of any class defined by universal quantification over hasChild. Hence, by our above statement, every childless person would be qualified as happy. In order to formalize the aforementioned intended reading, the statement would have to read as follows:

Functional-Style Syntax

```
EquivalentClasses(
```

```
  :HappyPerson
  ObjectIntersectionOf(
    ObjectAllValuesFrom( :hasChild :HappyPerson )
    ObjectSomeValuesFrom( :hasChild :HappyPerson )
  )
)
```

Gellish Expression Format syntax

happy person	has by definition as child only a	[1,n] happy person

This example also illustrates how property restrictions can be nested with complex classes.

Property restrictions can also be used to describe classes of individuals that are related to one particular individual. For instance we could define the class of John's children:

Functional-Style Syntax

```
EquivalentClasses(
  :JohnsChildren
  ObjectHasValue( :hasParent :John )
)
```

Gellish Expression Format syntax

Child of John	has by definition as parent	John

Notes on Gellish Formal English:

1. Johns children is a collection with a changing number of elements in the collection when children are born. Child of John is a category that has no number of elements and does not change. OWL mixes these two concepts.
2. This kind of relation should be defined as a subtype of a relation between a class and an individual thing.

As a special case of individuals being interlinked by properties, an individual might be linked to itself. The following example shows how to represent the idea that all narcissists love themselves.

Functional-Style Syntax

```
EquivalentClasses(
  :NarcisticPerson
  ObjectHasSelf( :loves )
)
```

Gellish Expression Format syntax

narcistic person	is by definition involved in a	self loving

Notes on Gellish Formal English:

1. The semantics of this OWL 'property' is unclear about the question whether 'loves' is an unary relation or a binary relation. In general such a relation has two roles: lover and being loved. The term 'Self', 'HasSelf', or 'ObjectHasSelf' seem to be a trick to model it as a binary relation, with an implied constraint that the role player of the first role (lover) is by definition the same as the role player of the second role (loved). This can be modelled in Gellish as such as follows:

narcistic person	is by definition a lover of a narcistic person
is by definition a lover of a	has by definition as first role a lover
is by definition a lover of a	has by definition as second role a loved
lover	is by definition played by the player of loved

2. I prefer modelling the semantics by defining a self loving activity as a separate concept with only one person involved.

5.3 Property Cardinality Restrictions

Using universal quantification, we can say something about all of somebody's children, whereas existential quantification allows us to refer to (at least) one of them. However, we might want to specify the number of individuals involved in the restriction. Indeed, we can construct classes depending on the number of children. The following example states that John has at most four children who are themselves parents:

Functional-Style Syntax

```
ClassAssertion(
  ObjectMaxCardinality( 4 :hasChild :Parent )
  :John
)
```

Gellish Expression Format syntax

John	has as child a	[0,4] parent

Notes on Gellish Formal English:

1. As parent is a role of a person, the implied semantics is that John has max 4 persons as child, whereas those persons are parents.

Note that this statement allows John to have arbitrarily many further children who are not parents.

Likewise, it is also possible to declare a minimum number by saying that John is an instance of the class of individuals having at least two children who are parents:

Functional-Style Syntax

```
ClassAssertion(
  ObjectMinCardinality( 2 :hasChild :Parent )
  :John
)
```

Gellish Expression Format syntax

```
John        has as child a        [2,n] parent
```

If we happen to know the exact number of John's children who are parents, this can be specified as follows:

Functional-Style Syntax

```
ClassAssertion(
  ObjectExactCardinality( 3 :hasChild :Parent )
  :John
)
```

Gellish Expression Format syntax

```
John        has as child a        [3,3] parent
```

In a cardinality restriction, providing the class is optional; if we just want to talk about the number of all of John's children we can write the following:

Functional-Style Syntax

```
ClassAssertion(
  ObjectExactCardinality( 5 :hasChild )
  :John
)
```

Gellish Expression Format syntax

```
John        has as child a        [5,5] person
```

5.4 Enumeration of Individuals

A very straightforward way to describe a class is just to enumerate all its instances. OWL provides this possibility, e.g. we can create a class of birthday guests:

Functional-Style Syntax

```
EquivalentClasses(
  :MyBirthdayGuests
  ObjectOneOf( :Bill :John :Mary)
)
```

Gellish Expression Format syntax

My birthday guests	has as element	Bill
My birthday guests	has as element	John
My birthday guests	has as element	Mary

Notes on Gellish Formal English:

1. It is semantically questionably to state that MyBirthdayGuests is a class. It is clearly an individual collection of individuals. Furthermore, the group can grow, which is questionable for a class. In Gellish it is defined as an individual collection.
2. Enumeration of individuals is thus a collection of individual things and not a kind of thing (class).
3. The class 'my birthday guest' would be a proper class, but that is not defined by the members, but by a person being a guest on my birthday. Class membership is the specified by a classification relation as in par. 4.1.

Note that this axiom provides more information than simply asserting class membership of Bill, John, and Mary as described in Section 4.1. In addition to that, it also stipulates that Bill, John, and Mary are the *only* members of MyBirthdayGuests. Therefore, classes defined this way are sometimes referred to as *closed classes* or enumerated sets. If we now assert Jeff as an instance of MyBirthdayGuests, the consequence is that Jeff must be equal to one of the above three persons.

Note on Gellish:

1. Specification that the collection contains only three elements can be done by specifying the aspect 'number of elements' as follows:

My birthday guests	has number of elements	3

2. If Jeff is specified as element of the collection as well, then either his UID is the same as one of the other UID's, or that statement is in conflict with the statement that the number of elements is three.

6 Advanced Use of Properties

Until now we focused on classes and properties that were merely used as building blocks for class expressions. In the following, we will see what other modeling capabilities with properties OWL 2 offers.

6.1 Property Characteristics

Sometimes one property can be obtained by taking another property and changing its direction, i.e. inverting it. For example, the property hasParent can be defined as the inverse property of hasChild:

Functional-Style Syntax

```
InverseObjectProperties( :hasParent :hasChild )
```

Gellish Expression Format syntax

has as parent is an inverse of has as child

Notes on Gellish Formal English:

1. In Gellish this is the same relation, but a different expression. In OWL these are two different 'properties' (relations), whereas one can be deduced from the other.

This would for example allow to deduce for arbitrary individuals A and B, where A is linked to B by the hasChild property, that B and A are also interlinked by the hasParent property. However, we do not need to explicitly assign a name to the inverse of a property if we just want to use it, say, inside a class expression. Instead of using the new hasParent property for the definition of the class Orphan, we can directly refer to it as the hasChild-inverse:

Functional-Style Syntax

```
EquivalentClasses(
  :Orphan
  ObjectAllValuesFrom(
    ObjectInverseOf( :hasChild )
    :Dead
  )
)
```

Gellish Expression Format syntax

orphan has by definition as parent a [2,2] dead parent

or

dead parent [2,2] has by definition as child a [1,n] orphan

Notes on Gellish Formal English:

1. Instead of using the combination of the term 'inverse' and a 'phrase', the left and right hand terms are inverted.

In some cases, a property and its inverse coincide, or in other words, the direction of a property doesn't matter. For instance the property hasSpouse relates A with B exactly if it relates B with A. For obvious reasons, a property with this characteristic is called *symmetric*, and it can be specified as follows

Functional-Style Syntax

```
SymmetricObjectProperty( :hasSpouse )
```

Gellish Expression Format syntax

has as spouse is a kind of symmetric relation

Notes on Gellish Formal English:

1. Whether true symmetric relations really exist is questionable, because there is always a difference in perspective. Therefore, in Gellish a phrase for a kind of relation always differs from its inverse phrase.

On the other hand, a property can also be *asymmetric* meaning that if it connects A with B it never connects B with A. Clearly (excluding paradoxical scenarios resulting from time travels), this is the case for the property hasChild and is expressed like this:

Functional-Style Syntax

```
AsymmetricObjectProperty( :hasChild )
```

Gellish Expression Format syntax

has as child is a kind of antisymmetric relation

Note that being asymmetric is a much stronger notion than being non-symmetric. Likewise, being symmetric is a much stronger notion than being non-asymmetric.

Previously, we considered subproperties in analogy to subclasses. It turns out that it also make sense to transfer the notion of class disjointness to properties: two properties are disjoint if there are no two individuals that are interlinked by both properties. Following common law, we can thus state that parent-child marriages cannot occur:

Functional-Style Syntax

```
DisjointObjectProperties( :hasParent :hasSpouse )
```

Gellish Expression Format syntax

parent shall not simultaneously towards the same relator be played by a player in the role of a spouse

Notes on Gellish Formal English:

1. Gellish makes a distinction between what shall not be the case and what cannot be the case. The example is apparently something that is forbidden “by common law”, but not impossible.
2. Disjointness also exists in Gellish. It is trivial that an individual relation (‘property’) cannot be classified as a ‘hasParent’ relation and also as a

'hasSpouse' relation, because those are two distinct relations. Therefore the example is not a disjointness relation.

Properties can also be *reflexive*: such a property relates everything to itself. For the following example, note that everybody has himself as a relative.

Functional-Style Syntax

```
ReflexiveObjectProperty( :hasRelative )
```

Gellish Expression Format syntax

has as relative	is a kind of	reflexive relation

Note that this does not necessarily mean that every two individuals which are related by a reflexive property are identical.

Properties can furthermore be *irreflexive*, meaning that no individual can be related to itself by such a role. A typical example is the following which simply states that nobody can be his own parent.

Functional-Style Syntax

```
IrreflexiveObjectProperty( :parentOf )
```

Gellish Expression Format syntax

is a parent of	is a kind of	irreflexive relation

Next, consider the hasHusband property. As every person can have only one husband (which we take for granted for the sake of the example), every individual can be linked by the hasHusband property to at most one other individual. This kind of properties are called *functional* and are described as follows:

Functional-Style Syntax

```
FunctionalObjectProperty( :hasHusband )
```

Gellish Expression Format syntax

has as husband	is a kind of	single-valued relation

Notes on Gellish Formal English:

1. Single-valued relation is a synonym of FunctionalProperty.
2. A single-valued relation is a relation with 1:1 or n:1 simultaneous cardinalities. In the Gellish Expression Format syntax the cardinalities can be explicitly specified.
3. A single-valued relation in Gellish specifies only the right hand simultaneous cardinality.

4. Strictly speaking this is an incorrect example statement, because a proper subtype should be validity context independent.

Note that this statement does not require every individual to have a husband, it just states that there can be no more than one. Moreover, if we additionally had a statement that Mary's husband is James and another that Mary's husband is Jim, it could be inferred that Jim and James must refer to the same individual.

Note on Gellish: The above conclusion is only true if both statements are true. In Gellish the intention 'statement' still allows that the expression is an erroneous opinion. Thus the conclusion can also be the there are two Mary's (homonyms) or that one or both statements are incorrect.

It is also possible to indicate that the inverse of a given property is functional:

Functional-Style Syntax

```
InverseFunctionalObjectProperty( :hasHusband )
```

Gellish Expression Format syntax

has as husband is a kind of inverse single-valued relation

Notes on Gellish Formal English:

1. Inverse single-valued relation is a synonym of InverseFunctionalProperty.

This indicates that an individual can be husband of at most one other individual. The example also indicates the difference between functionality and inverse functionality, as in a polygynous situation the former axiom is valid whereas the latter isn't.

Note on Gellish:

1. In Gellish there is a distinction between what is the case (about individual things), and for kinds of things what can be the case, what shall be the case and what is by definition the case. The semantics of this expression seems to be that e (validity context dependent) requirement (shall be the case) is specified. In Gellish this would be that in a particular validity context:

   shall have as husband a is a kind of inverse single-valued relation

   or

   man shall be a husband of a [1,1] wife

2. In Gellish the FunctionalProperty and InverseFunctionalProperty can be combined in one statement as follows:

   [1,1] man shall be a husband of a [1,1] wife

Now have a look at a property hasAncestor which is meant to link individuals A and B whenever A is a direct descendant of B. Clearly, the property hasParent is a "special case" of hasAncestor and can be defined as a subproperty thereof. Still, it would be

nice to "automatically" include parents of parents (and parents of parents of parents). This can be done by defining hasAncestor as *transitive* property. A transitive property interlinks two individuals A and C whenever it interlinks A with B and B with C for some individual B.

Functional-Style Syntax

```
TransitiveObjectProperty( :hasAncestor )
```

Gellish Expression Format syntax

has as ancestor is a kind of transitive relation

6.2 Property Chains

While the last example from the previous section implied the presence of an hasAncestor property whenever there is a chain of hasParent properties, we might want to be a bit more specific and define, say, a hasGrandparent property instead. Technically, this means that we want hasGrandparent to connect all individuals that are linked by a chain of exactly two hasParent properties. In contrast to the previous hasAncestor example, we do not want hasParent to be a special case of hasGrandparent nor do we want hasGrandparent to refer to great-grandparents etc. We can express that every such chain has to be spanned by a hasGrandparent property as follows:

Functional-Style Syntax

```
SubObjectPropertyOf(
  ObjectPropertyChain( :hasParent :hasParent )
  :hasGrandparent
)
```

Gellish Expression Format syntax

has as grandparent is by definition a relation chain of [2,2] has as parent

6.3 Keys

In OWL 2 a collection of (data or object) properties can be assigned as a key to a class expression. This means that each named instance of the class expression is uniquely identified by the set of values which these properties attain in relation to the instance.

A simple example of this would be the identification of a person by her social security number.

Functional-Style Syntax

```
HasKey( :Person () ( :hasSSN ) )
```

Gellish Expression Format syntax

person	can have as unique key a	SSN of person

or

person	can have as unique key a	SSN collection
SSN collection	is a kind of	collection of kinds of relations
SSN of person	is an element in collection of concepts	SSN collection

whereas

SSN of person is the name of the kind of relation:

person	can have as identifier a	SSN

Notes on Gellish Formal English:

1. The two ways of modeling illustrate that in Gellish the relation < can have as unique key a> can relate either to a kind of relation or to a collection of kinds of relations.
2. The above second way of modelling defines an SSN collection as a kind of key (collection of kinds of relations), called SSN collection. That collection contains as element the relation 'SSN of person', which is the name of the kind of relation 'person <can have as identifier a> SSN'. This means that a person can uniquely be identified (in a particular language community context) by one relation, being a relation with a social security number. The language community context is explicit in a Gellish Expression Format.

7 Advanced Use of Datatypes

In Section 4.8, we learned that individuals can be endowed with numerical information, essentially by connecting them to a data value via a datatype property – just like object properties link to other domain individuals. In fact, these parallels extend to the more advanced features of datatype usage.

Note on Gellish: An OWL 'datatype property' is comparable with a 'constraining relation'or its subtype 'qualification relation' in Gellish, which constrains or qualifies an aspect.

First of all, data values are (Gellish: can be) grouped into datatypes and we have seen in Section 4.8 how a range restriction on a datatype property can be used to indicate the kind of values this property can link to. Moreover, it is possible to express and define new datatypes by constraining or combining existing ones. Datatypes can be restricted via so-called *facets*, borrowed from XML Schema Datatypes [*XML Schema Datatypes*]. In the following example, we define a new datatype for a person's age by constraining the datatype integer to values between (inclusively) 0 and 150.

Functional-Style Syntax

```
DatatypeDefinition(
  :personAge
  DatatypeRestriction( xsd:integer
```

```
    xsd:minInclusive "0"^^xsd:integer
    xsd:maxInclusive "150"^^xsd:integer
  )
)
```

Gellish Expression Format syntax

age of a person	shall have on scale a value greater than or equal to	0	year
age of a person	shall have on scale a value less than or equal to		
150	year		

Notes on Gellish Formal English:

1. An age is not a proper Datatype, because age requires a unit of measure in order enable to relate it to a number.
2. The Gellish expression also includes a unit of measure, which is missing in the OWL expressions.
3. Apparently this is a constraint or requirement for data entry, whereas in OWL it seems to be a statement.
4. The OWL concept 'EquivalentClass' is semantically questionable, as the range is not equivalent to an age. Furthermore in onother context the range will be chose differently.

Likewise, datatypes can be combined just like classes by complement, intersection and union. Thereby, assuming we have already defined a datatype minorAge, we can define the datatype majorAge by excluding all data values of minorAge from personAge:

Functional-Style Syntax

```
DatatypeDefinition(
  :majorAge
  DataIntersectionOf(
    :personAge
    DataComplementOf( :minorAge )
  )
)
```

Gellish Expression Format syntax

major age	is by definition within range	person age range
major age	is by definition outside range	minor age

Notes on Gellish Formal English:

1. The range for the age of a person is a concept that differs from the age of a person. It is assumed that the first concept is meant in this example.
2. In Gellish a characteristic such as age is not a mathematical range, but a property range. It can be related to a mathematical range by using a scale for mapping.
3. Time and age are continuous aspects and not discrete collections of times or ages.

4. A mathematical range is considered to be continuous and not a collection of discrete values.

Moreover, a new datatype can be generated by just enumerating the data values it contains.

Functional-Style Syntax

```
DatatypeDefinition(
  :toddlerAge
  DataOneOf( "1"^^xsd:integer "2"^^xsd:integer )
)
```

Gellish Expression Format syntax

1	is an element in collection of concepts	one or two
2	is an element in collection of concepts	one or two

Notes on Gellish Formal English:

1. This example is apparently about a collection of whole numbers and not about a collection of ages. A collection of ages should include the conceps ‘one year’ and ‘two years’.
2. The concept ‘Datatype’ is in fact a role of a collection or range of allowed values for values of aspects.
3. The specification that the numbers are integer (which is a way of binary encoding of whole numbers) is semantically not relevant. The numbers 1 and 2 should be defined as whole numbers, but that is not relevant for the specification of their inclusion in a collection.
4. The specification that ages are in whole numbers (integers) is apparently a requirement that holds in a particular context and not a generally valid definition or true statement. Baby’s age is often measured in parts of years, such as 1.5 year.

In Section 6.1, we saw ways of characterizing object properties. Some of those are also available for datatype properties. For example, we can express that every person has only one age by characterizing the hasAge datatype property as functional:

Functional-Style Syntax

```
FunctionalDataProperty( :hasAge )
```

Gellish Expression Format syntax

age	shall have as scale value a	[1,1] whole number

Notes on Gellish Formal English:

1. A person has one age, which can be quantified by various numbers in the course of time and on different scales. Thus it is not a good example of a single-valued property (or a ‘FunctionalProperty’).

New classes can be defined by restrictions on datatype properties. The following example defines the class teenager as all individuals whose age is between 13 and 19 years.

Functional-Style Syntax

```
SubClassOf(
  :Teenager
  DataSomeValuesFrom( :hasAge
    DatatypeRestriction( xsd:integer
      xsd:minExclusive "12"^^xsd:integer
      xsd:maxInclusive "19"^^xsd:integer
    )
  )
)
```

Gellish Expression Format syntax

teenager	is a kind of	person	
teenager	has by definition as qualitative intrinsic aspect a	teenager age	
teenager age	has by definition on scale a value less than or equal to	19	year
teenager age	has by definition on scale a value greater than	12	year

Notes on Gellish Formal English:

1. The example in OWL defines a teenager as a SubClass of age and not as a SubClass of person, which is incorrect.
2. The kind of relation on the second line in the Gellish model specifies < has by definition as qualitative intrinsic aspect a> implies that the teenager age is by definition qualified (or kwantified). It also indicates that teenager age is an intrinsic aspect, which means it is defined as an aspect of a teenager.
3. The example in OWL does not recognize teenager age as a separate concept. The explicit definitionof teenager age in Gellish would be:

teenager age	is by definition an intrinsic aspect of a	teenager
teenager age	is by definition an intrinsic	age
teenager age	is a kind of	intrinsic aspect

8 Document Information and Annotations

In the following, we describe features of OWL 2 which do not actually contribute to the “logical” knowledge specified in the ontology. Rather these are used to provide additional information about the ontology itself, axioms, or even single entities.

8.1 Annotating Axioms and Entities

In many cases, we want to furnish parts of our OWL ontology with information that actually does not describe the domain itself but talks about the description of the domain. OWL provides annotations for this purpose. An OWL annotation simply associates property-value pairs with parts of an ontology, or the entire ontology itself.

Even annotations themselves can be annotated. Annotation information is not really part of the logical meaning of an ontology.

So, for example, we could add information to one of the classes of our ontology, giving a natural language description of its meaning.

Functional-Style Syntax

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

Gellish Expression Format syntax

person is a kind of party a party that is a human being.

Or, when the description or definition is a separate entity in the model:

person is described by description-1
description-1 is a qualification of description a party that is a human being.

Notes on Gellish Formal English:

1. Gellish makes a distinction between comment on an expression and a description of something and a textual definition of something. In this example the description is assumed.
2. There is a third option in Gellish, which is used in case the description is provided in a separate physical document or electronic file. This is described in the Gellish documentation.

The following is an example of an axiom with an annotation.

Functional-Style Syntax

```
SubClassOf(
  Annotation( rdfs:comment "States that every man is a person." )
  :Man
  :Person
)
```

Gellish Expression Format syntax

man is a kind of person <some definition> <some comment>

Notes on Gellish Formal English:

1. As the Gellish Expression Format syntax is tabular the table contains separate columns for definitions of the left hand objects in the expressions and for comments on the whole (binary) expression and for a term or phrase that names the expression, for example for text to be displayed in user interfaces.

Often such annotations are used in tools to provide access to natural language text to be displayed in help interfaces.

8.2 Ontology Management

In OWL, general information about a topic is almost always gathered into an ontology that is then used by various applications. We can also provide a name for OWL ontologies, which is generally the place where the ontology document is located in the web. Particular information about a topic can also be placed in an ontology, if it is used by different applications.

Functional-Style Syntax

```
Ontology(<http://example.com/owl/families>
  ...
)
```

Gellish Expression Format syntax

Gellish English version: 8.3 9-aug-2013
http://example.com/owl/families

Notes on Gellish Formal English:

1. The header of a Gellish Expression Format contains a collection of fields (A1..An), of which field A6 optionally contains the name of the collection of expressions in the table, possibly preceded by a path to (or address of) the location of the source on an electronic network.

We place OWL ontologies into OWL documents, which are then placed into local filesystems or on the World Wide Web. Aside from containing an OWL ontology, OWL documents also contain information about transforming the short names normally used in OWL ontologies (e.g., Person) into IRIs, by providing the expansion for prefixes. The IRI is then the concatenation of the prefix expansion and the reference.

In our example we have so far used a number of prefixes, including `xsd` and the empty prefix. The former prefix has been used in compact names for XML Schema datatypes, whose IRIs are fixed by the XML Schema recommendation. We thus must use the standard expansion for `xsd`, which is `http://www.w3.org/2001/XMLSchema#`. The expansion we pick for the other prefix will affect the names of the classes, properties, and individuals in our ontology, as well as the name of the ontology itself. If we are going to put the ontology on the web, we should pick an expansion that is in some part of the web that we control, so that we are not using someone else's names by accident. (Here we use a made-up place that no one controls.) The two XML-based syntaxes need namespaces for built-in names and also can use XML entity references for namespaces. In general, it should be noted that the available abbreviation mechanisms and their specific syntax is different in each of the serializations of OWL, even in cases where similar keywords are used.

Functional-Style Syntax

```
Prefix(:=<http://example.com/owl/families/>)
Prefix(otherOnt:=<http://example.org/otherOntologies/families/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://example.com/owl/families>
    ...
)
```

Gellish Expression Format syntax

… A9: http://example.com/Gellish/families

Notes on Gellish Formal English:

1. Field A9 and following in the first collection of fields in the header of a Gellish Expression Format can contain references (IRI's) to other Gellish Expression Formats that are required to be used in combination with the table for its proper interpretation.
2. Gellish Expression Formats may contain specifications of synonyms, abbreviations, codes, translations, etc. that are defined in public or private language communities. References tables can thus be used to introduce such alias terms.

It is also common in OWL to reuse general information that is stored in one ontology in other ontologies. Instead of requiring the copying of this information, OWL allows the import of the contents of entire ontologies in other ontologies, using import statements, as follows:

Functional-Style Syntax

```
Import( <http://example.org/otherOntologies/families.owl> )
```

Gellish Expression Format syntax

… A9: http://example.com/Gellish/families

Notes on Gellish Formal English:

1. In Gellish combined use or import is the same, because collections of proper Gellish expressions can interoperate by definition and can thus be combined as and when required.

As the Semantic Web and ontology construction is distributed, it is common for ontologies to use different names for the same concept, property, or individual. As we have seen, several constructs in OWL can be used to state that different names refer to the same class, property, or individual, so, for example, we could – instead of tediously renaming entities – tie the names used in our ontology to the names used in an imported ontology as follows:

Functional-Style Syntax

```
SameIndividual( :John otherOnt:JohnBrown )
SameIndividual( :Mary otherOnt:MaryBrown )
EquivalentClasses( :Adult otherOnt:Grownup )
EquivalentObjectProperties( :hasChild otherOnt:child )
EquivalentDataProperties( :hasAge otherOnt:age )
```

Gellish Expression Format syntax

otherOnt	John	is a synonym of	John Brown
otherOnt	Mary	is a synonym of	Mary Brown
otherOnt	adult	is a synonym of	Grown up
otherOnt	hasChild	is a synonym of	child
otherOnt	has Age	is a synonym of	age

Notes on Gellish Formal English:

1. The 'otherOnt' is in Gellish an indication of a language community (context).
2. In Gellish there can be different language communities within the same ontology.
3. Gellish uses UID to uniquely represent a concept. Thus the synonyms will have the same UID.
4. The use of UID not only supports the use of synonyms, it also enables the use of homonyms.

8.3 Entity Declarations

To help with managing ontologies, OWL has the notion of declarations. The basic idea is that each class, property, or individual is supposed to be declared in an ontology, and then it can be used in that ontology and ontologies that import that ontology.

In the Manchester syntax, declarations are implicit. Constructs that provide information about a class, property, or individual implicitly declare that class, property, or individual if needed. The other syntaxes have explicit declarations.

Functional-Style Syntax

```
Declaration( NamedIndividual( :John ) )
Declaration( Class( :Person ) )
Declaration( ObjectProperty( :hasWife ) )
Declaration( DataProperty( :hasAge ) )
```

Gellish Expression Format syntax

John	is classified as a	person
person	is a kind of	party
has as wife	is a kind of	binary relation between individual things
has as aspect	is a kind of	binary relation between individual things

Notes on Gellish Formal English:

1. Individual things are declared in Gellish by a classification relation. Kinds of things are declared by a specialization relation.

However, an IRI may denote different entity-types (e.g. both an individual and a class) at the same time. This possibility, called “punning,” has been introduced to allow for a certain amount of metamodeling; we give an example of this in Section 9. Still, OWL 2 does require some discipline in using and reusing names. To allow a more readable syntax, and for other technical reasons, OWL 2 DL requires that a name is not used for more than one property type (object, datatype or annotation property) nor can an IRI denote both a class and a datatype. Moreover, “built-in” names (such as those used by RDF and RDFS and various syntaxes of OWL) cannot be freely used in OWL.

Notes on Gellish Formal English:

1. The above described constraints for OWL do not apply for Gellish, due to the use of unique identifiers (UID’s) and language community contexts in Gellish.

9 OWL 2 DL and OWL 2 Full

There are two alternative ways of assigning meaning to ontologies in OWL 2 called the Direct Semantics [*OWL 2 Direct Semantics*] and the RDF-Based Semantics [*OWL 2 RDF-Based Semantics*]. The Direct Semantics can be applied to ontologies that are in the OWL 2 DL subset of OWL 2, which is defined in OWL 2 functional-style syntax document [*OWL 2 Specification*]. Ontologies that are not in OWL 2 DL are often said to belong to *OWL 2 Full*, and can only be interpreted under RDF-Based Semantics. Informally, the term *OWL 2 DL* is often used to refer to OWL 2 ontologies interpreted using the Direct Semantics, but it is also possible to interpret OWL 2 DL ontologies under RDF-Based Semantics.

The Direct Semantics [*OWL 2 Direct Semantics*] provides a meaning for OWL 2 in a Description Logic [*Description Logics*] style. The RDF-Based Semantics [*OWL 2 RDF-Based Semantics*] is an extension of the semantics for RDFS [*RDF Semantics*] and is based on viewing OWL 2 ontologies as RDF graphs.

When thinking about ontologies the differences between these two semantics are generally quite slight. Indeed, given an OWL 2 DL ontology, many inferences drawn using the Direct Semantics remain valid inferences under the RDF-Based Semantics – see the correspondence theorem in Section 7.2 of the RDF-Based Semantics document [*OWL 2 RDF-Based Semantics*]. The two main differences are that under the Direct Semantics annotations have no formal meaning and under the RDF-Based Semantics there are some extra inferences that arise from the RDF view of the universe.

Notes on Gellish Formal English:

1. In Gellish description text and comment text does not have formal (computer interpretable) meaning, because their internal structure and terminology are free and are typically expressed in natural language. Thus they do not necessarily comply with a formal language definition and are intended for human interpretation.

Conceptually, we can think of the difference between OWL 2 DL (under Direct Semantics) and OWL 2 Full (under RDF-Based Semantics) in two ways:

- One can see OWL 2 DL as a syntactically restricted version of OWL 2 Full where the restrictions are designed to make life easier for implementors. In fact, since OWL 2 Full (under the RDF-Based Semantics) is undecidable, OWL 2 DL (under the Direct Semantics) makes writing a reasoner that, in principle, can return all "yes or no" answers (subject to resource constraints) possible. As a consequence of its design, there are several production quality reasoners that cover the entire OWL 2 DL language under the Direct Semantics. There are no such reasoners for OWL 2 Full under the RDF-Based Semantics.

- One can see OWL 2 Full as the most straightforward extension of RDFS. As such, the RDF-Based Semantics for OWL 2 Full follows the RDFS semantics and general syntactic philosophy (i.e., everything is a triple and the language is fully reflective).

Of course, the two semantics have been designed together and thus have influenced each other. For example, one design goal of OWL 2 was to bring OWL 2 DL syntactically closer to OWL 2 Full (that is, to allow more RDF Graphs/OWL 2 Full ontologies to be legal OWL 2 DL ontologies). This led to the incorporation of so-called *punning* into OWL 2, e.g., using the same IRI as a name for both a class and an individual. An example of such usage would be the following, which states that John is a father, and that father is a social role.

Functional-Style Syntax

```
ClassAssertion( :Father :John )
ClassAssertion( :SocialRole :Father )
```

Gellish Expression Format syntax

John	is classified (by role) as a	father
father	is a particular	social role

Notes on Gellish Formal English:

1. The OWL example seems to intent that the two terms ‘father’ denote the same concept (with the same UID in Gellish), but *used* in different ways. Assuming such an intention, the second line expresses a classification of a class. This does not imply that ‘father’ is used as an individual. The problem in OWL occurs, because the same expression is used to classify classes as is used to classify individuals.
2. If different concepts would be meant, having the same name but different UID’s in Gellish, then it would be a homonym. This allowed in Gellish and solved by specifying the names for the two UID’s in two different language communities.

Note that in the first statement, Father is used as a class, while in the second it is used as an individual. In this sense, SocialRole acts as a metaclass for the class Father.

Notes on Gellish Formal English:

1. In Gellish a ‘classification of a class’ relation differs from a ‘classification of an individual thing’ relation. The classification of a class does imply that the classified thing is a class. In Gellish an <is a particular> relation relates a class to a metaclass.

In OWL 1, a document containing these two statements would be an OWL 1 Full document, but not an OWL 1 DL document. In OWL 2 DL, however, this is allowed. It must be noted, though, that the Direct Semantics of OWL 2 DL accommodates this by understanding the class Father and the individual Father as two different views on the same IRI, i.e. they are interpreted semantically as if they were distinct. In particular, if two individuals are equal, then the classes that they denote are equivalent under RDF-Based Semantics, whereas there is no such relationship between the classes under Direct Semantics. This is possibly the one difference between the two semantics that is most relevant in practice.

10 OWL 2 Profiles

In addition to OWL 2 DL and OWL 2 Full, OWL 2 specifies three profiles. OWL 2, in general, is a very expressive language (both computationally and for users) and thus can be difficult to implement well and to work with. These additional profiles are designed to be approachable subsets of OWL 2 sufficient for a variety of applications. As with OWL 2 DL, computational considerations are a major requirement of these profiles (and they are all much easier to implement with robust scalability given existing technology), but there are many subsets of OWL 2 that have good computational properties. The selected OWL 2 profiles were identified as having substantial user communities already, although there were several others not included and one should expect more to emerge. The [*OWL 2 Profiles*] document provides a clear template for specifying additional profiles.

In order to guarantee scalable reasoning, the existing profiles share some limitations regarding their expressiveness. In general, they disallow negation and disjunction, as these constructs complicate reasoning and have proved to be only rarely needed for modeling. For example, in none of the profiles it is possible to specify that every person is either male or female. Further specific modeling restrictions of the profiles will be dealt with in the sections on the individual profiles.

We discuss each profile and its design rationale, and provide some guidance for users in selecting which profile to work with. Please be aware that this discussion is not comprehensive, nor can it be. Part of any decision has to be based on available tooling and how that fits in with the rest of your system or workflow. A more extended discussion and comparison of the profiles can be found in the literature [*OWL 2 Profiles Introduction*].

By and large, different profiles can be distinguished syntactically, and some of the profiles contain others. For example, OWL 2 DL can be seen as a syntactic fragment of OWL 2 Full and OWL 2 QL is a syntactic fragment of OWL 2 DL (and thus of OWL 2 Full). None of these profiles below is a subset of another. Ideally, one can use a reasoner (or other tool) that is conforming for a superprofile on the subprofile with no change in the results derived. In particular, every conforming OWL 2 DL reasoner is

also a conforming reasoner for OWL 2 EL, OWL 2 RL, and OWL 2 QL (but may differ in performance since the OWL 2 DL reasoner is tuned for a more general set of cases). Each of the two semantics of OWL 2 (see Section 9) can be used for any of the profiles, but it is most common to use Direct Semantics for OWL 2 EL and OWL 2 QL, and RDF-Based Semantics for OWL 2 RL.

10.1 OWL 2 EL

Working with OWL 2 EL is fairly similar to working with OWL 2 DL: one can use class expressions on both sides of a subClassOf statement and even infer such relations. For many large, class-expression oriented ontologies, by only a little simplification one can get an OWL 2 EL ontology and preserve the bulk of the meaning of the original ontology.

OWL 2 EL is designed with large biohealth ontologies in mind, such as SNOMED-CT, the NCI thesaurus, and Galen. Common characteristics of such ontologies include complex structural descriptions (e.g., defining certain body parts in terms of what parts they contain and are contained in or propagating diseases along part-subpart relations), huge numbers of classes, the heavy use of classification to manage the terminology, and the application of the resulting terminology to vast amounts of data. Thus, OWL 2 EL has a comparatively expressive class expression language and it has no restrictions on how they may be used in axioms. It also has fairly expressive property expressions, including property chains but excluding inverse.

Sensible use of OWL 2 EL is obviously not restricted to the biohealth domain: as with the other profiles, OWL 2 EL is domain independent. However, OWL 2 EL shines when your domain and your application require recognition of structurally complex objects. Such domains include system configurations, product inventories, and many scientific domains.

Besides negation and disjunction, OWL 2 EL also disallows universal quantification on properties. Therefore propositions like “all children of a rich person are rich” cannot be stated. Moreover, as all kinds of role inverses are not available, there is no way of specifying that, say, parentOf and childOf are inverses of each other.

The EL acronym reflects the profile's basis in the so-called EL family of description logics [*EL++*]; they are Languages providing mainly Existential quantification of variables.

The following is an example which uses some of the typical modeling features available in OWL 2 EL.

Functional-Style Syntax

```
SubClassOf(
  :Father
  ObjectIntersectionOf( :Man :Parent )
)

EquivalentClasses(
  :Parent
```

```
  ObjectSomeValuesFrom(
    :hasChild
    :Person
  )
)

EquivalentClasses(
  :NarcisticPerson
  ObjectHasSelf( :loves )
)

DisjointClasses(
  :Mother
  :Father
  :YoungChild
)

SubObjectPropertyOf(
  ObjectPropertyChain( :hasFather :hasBrother )
  :hasUncle
)

NegativeObjectPropertyAssertion(
  :hasDaughter
  :Bill
  :Susan
)
```

10.2 OWL 2 QL

OWL 2 QL can be realized using standard relational database technology (e.g., SQL) simply by expanding queries in the light of class axioms. This means it can be tightly integrated with RDBMSs and benefit from their robust implementations and multi-user features. Furthermore, it can be implemented without having to “touch the data,” so really as a translational/preprocessing layer. Expressively, it can represent key features of Entity-relationship and UML diagrams (at least those with functional restrictions). Thus, it is suitable both for representing database schemas and for integrating them via query rewriting. As a result, it can also be used directly as a high level database schema language, though users may prefer a diagram based syntax.

OWL 2 QL also captures many commonly used features in RDFS and small extensions thereof, such as inverse properties and subproperty hierarchies. OWL 2 QL restricts class axioms asymmetrically, that is, you can use constructs as the subclass that you cannot use as the superclass.

Among other constructs, OWL 2 QL disallows existential quantification of roles to a class expression, e.g. it can be stated that every person has a parent but not that every person has a female parent. Moreover property chain axioms and equality are not supported.

The QL acronym reflects the fact that query answering in this profile can be implemented by rewriting queries into a standard relational Query Language [*DL-Lite*].

The following is an example which uses some of the typical modeling features available in OWL 2 QL. The first axiom states that every childless person is a person for which there does not exist anybody who has the first person as parent.

Functional-Style Syntax

```
SubClassOf(
  :ChildlessPerson
  ObjectIntersectionOf(
    :Person
    ObjectComplementOf(
      ObjectSomeValuesFrom(
        ObjectInverseOf( :hasParent )
        owl:Thing
      )
    )
  )
)

DisjointClasses(
  :Mother
  :Father
  :YoungChild
)

DisjointObjectProperties(
  :hasSon
  :hasDaughter
)

SubObjectPropertyOf(
  :hasFather
  :hasParent
)
```

10.3 OWL 2 RL

The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2. This is achieved by defining a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies, and presenting a partial axiomatization of the OWL 2 semantics in the form of first-order implications that can be used as the basis for such an implementation.

Suitable rule-based implementations of OWL 2 RL under RDF-Based Semantics can be used with arbitrary RDF graphs. As a consequence, OWL 2 RL is ideal for enriching RDF data, especially when the data must be massaged by additional rules. From a modeling perspective, however, this pushes us farther away from working with class expressions: OWL 2 RL ensures we cannot (easily) talk about unknown individuals in our superclass expressions (this restriction follows from the nature of rules). Compared with OWL 2 QL, OWL 2 RL works better when you have already massaged your data into RDF and are working with it as RDF.

Among other constructs, OWL 2 RL disallows statements where the existence of an individual enforces the existence of another individual: for instance, the statement "every person has a parent" is not expressible in OWL RL.

OWL 2 RL restricts class axioms asymmetrically, that is, you can use constructs as the subclass that you cannot use as the superclass.

The RL acronym reflects the fact that reasoning in this profile can be implemented using a standard Rule Language [*DLP*].

The following is an example which uses some of the typical modeling features available in OWL 2 RL. The first – somewhat contrived – axiom states that for each of Mary, Bill, and Meg who is female, the following holds: she is a parent with at most one child, and all her children (if she has any) are female.

Functional-Style Syntax

```
SubClassOf(
  ObjectIntersectionOf(
    ObjectOneOf( :Mary :Bill :Meg )
    :Female
  )
  ObjectIntersectionOf(
    :Parent
    ObjectMaxCardinality( 1 :hasChild )
    ObjectAllValuesFrom( :hasChild :Female )
  )
)

DisjointClasses(
  :Mother
  :Father
  :YoungChild
)

SubObjectPropertyOf(
  ObjectPropertyChain( :hasFather :hasBrother )
  :hasUncle
)
```

11 OWL Tools

In order to work with OWL ontologies, tool support is essential. Basically, there are two types of tools addressing the two main stages of the ontology lifecycle: *ontology editors* are used to create and edit ontologies, whereas *reasoners* are used to query ontologies for implicit knowledge, i.e., they determine whether a statement in question is a logical consequence of an ontology.

The currently most widely used OWL editor is Protégé, a free open-source editing framework developed at Stanford University. By virtue of its open plugin structure, it allows for the easy integration of special-purpose ontology editing components. Other editors include TopQuadrant's commercial TopBraid Composer and the open-source systems SWOOP and NeOn-Toolkit.

There are several reasoners for OWL DL which differ somewhat in terms of coverage of the supported reasoning features. For some of these, OWL 2 conformance is currently planned and the corresponding implementations are in progress. The Test Suite Status document lists to which extent some of the reasoners mentioned below comply with the test cases.

For reasoning within OWL DL, the most prominent systems are Fact++ by the University of Manchester, Hermit by Oxford University Computing Laboratory, Pellet by Clark & Parsia, LLC, and RacerPro by Racer Systems.

In addition to those general-purpose reasoners aiming at supporting all of OWL DL, there are reasoning systems tailored to the tractable profiles of OWL. CEL by Dresden University of Technology supports OWL EL. QuOnto by Sapienza Università di Roma supports OWL QL. ORACLE 11g supports OWL RL.

The open-source OWL API plays a rather prominent role as the currently most important development tool around OWL.

It must be mentioned that by the time this document was created, several OWL tools were under development, hence the current overview should be seen as a snapshot of this development rather than an up-to-date overview. Extensive listings of OWL tools can be found at semanticweb.org and in the ESW-Wiki.

12 What To Read Next

This short primer can only scratch the surface of OWL. There are many longer and more involved tutorials on OWL and how to use OWL tools that can be found by searching on the Web. Reading one of these documents and using a tool to build an OWL ontology is probably the best way to obtain a working knowledge about OWL. For learning more about the foundations of OWL, we recommend to consult first a textbook [*FOST*] and then the original articles cited therein. An extended introduction to the OWL 2 Profiles can be found in [*OWL 2 Profiles Introduction*], which is also available on the Web.

This short primer is also not a normative definition of OWL. The normative definition of the OWL syntax as well as informative descriptions of the meaning of each OWL

construct can be found in the OWL 2 Structural Specification and Functional Syntax document [*OWL 2 Specification*].

The OWL 2 Quick Reference Guide [*OWL 2 Quick Guide*] comes handy as a reference when looking for information about a specific language feature.

For those interested in more formal documents, the formal meaning of OWL 2 can be found in the OWL 2 Semantics documents [*OWL 2 Direct Semantics*] [*OWL 2 RDF-Based Semantics*].

The mapping between OWL syntax and RDF triples can be found in the OWL 2 Mapping to RDF Graphs document [*OWL 2 RDF Mapping*].

13 Appendix: The Complete Sample Ontology

Here we include the complete sample OWL ontology. Ontological axioms are ordered by top-level expressive features they use. Moreover, we follow a commonly-used ordering, with ontology and declaration information first, followed by information about properties, then classes and datatypes, then individuals.

Functional-Style Syntax

```
 Prefix(:=<http://example.com/owl/families/>)
 Prefix(otherOnt:=<http://example.org/otherOntologies/families/>)
 Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
 Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
 Ontology(<http://example.com/owl/families>
   Import( <http://example.org/otherOntologies/families.owl> )

   Declaration( NamedIndividual( :John ) )
   Declaration( NamedIndividual( :Mary ) )
   Declaration( NamedIndividual( :Jim ) )
   Declaration( NamedIndividual( :James ) )
   Declaration( NamedIndividual( :Jack ) )
   Declaration( NamedIndividual( :Bill ) )
   Declaration( NamedIndividual( :Susan ) )
   Declaration( Class( :Person ) )
   AnnotationAssertion( rdfs:comment :Person "Represents the set of all
people." )
   Declaration( Class( :Woman ) )
   Declaration( Class( :Parent ) )
   Declaration( Class( :Father ) )
   Declaration( Class( :Mother ) )
   Declaration( Class( :SocialRole ) )
   Declaration( Class( :Man ) )
   Declaration( Class( :Teenager ) )
   Declaration( Class( :ChildlessPerson ) )
   Declaration( Class( :Human ) )
   Declaration( Class( :Female ) )
```

```
Declaration( Class( :HappyPerson ) )
Declaration( Class( :JohnsChildren ) )
Declaration( Class( :NarcisticPerson ) )
Declaration( Class( :MyBirthdayGuests ) )
Declaration( Class( :Dead ) )
Declaration( Class( :Orphan ) )
Declaration( Class( :Adult ) )
Declaration( Class( :YoungChild ) )
Declaration( ObjectProperty( :hasWife ) )
Declaration( ObjectProperty( :hasChild ) )
Declaration( ObjectProperty( :hasDaughter ) )
Declaration( ObjectProperty( :loves ) )
Declaration( ObjectProperty( :hasSpouse ) )
Declaration( ObjectProperty( :hasGrandparent ) )
Declaration( ObjectProperty( :hasParent ) )
Declaration( ObjectProperty( :hasBrother ) )
Declaration( ObjectProperty( :hasUncle ) )
Declaration( ObjectProperty( :hasSon ) )
Declaration( ObjectProperty( :hasAncestor ) )
Declaration( ObjectProperty( :hasHusband ) )
Declaration( DataProperty( :hasAge ) )
Declaration( DataProperty( :hasSSN ) )
Declaration( Datatype( :personAge ) )
Declaration( Datatype( :minorAge ) )
Declaration( Datatype( :majorAge ) )
Declaration( Datatype( :toddlerAge ) )

SubObjectPropertyOf( :hasWife :hasSpouse )
SubObjectPropertyOf(
  ObjectPropertyChain( :hasParent :hasParent )
  :hasGrandparent
)
SubObjectPropertyOf(
  ObjectPropertyChain( :hasFather :hasBrother )
  :hasUncle
)
SubObjectPropertyOf(
  :hasFather
  :hasParent
)

EquivalentObjectProperties( :hasChild otherOnt:child )
InverseObjectProperties( :hasParent :hasChild )
EquivalentDataProperties( :hasAge otherOnt:age )
DisjointObjectProperties( :hasSon :hasDaughter )
ObjectPropertyDomain( :hasWife :Man )
ObjectPropertyRange( :hasWife :Woman )
DataPropertyDomain( :hasAge :Person )
DataPropertyRange( :hasAge xsd:nonNegativeInteger )

SymmetricObjectProperty( :hasSpouse )
AsymmetricObjectProperty( :hasChild )
```

```
DisjointObjectProperties( :hasParent :hasSpouse )
ReflexiveObjectProperty( :hasRelative )
IrreflexiveObjectProperty( :parentOf )
FunctionalObjectProperty( :hasHusband )
InverseFunctionalObjectProperty( :hasHusband )
TransitiveObjectProperty( :hasAncestor )
FunctionalDataProperty( :hasAge )

SubClassOf( :Woman :Person )
SubClassOf( :Mother :Woman )
SubClassOf(
  :Grandfather
  ObjectIntersectionOf( :Man :Parent )
)
SubClassOf(
  :Teenager
  DataSomeValuesFrom( :hasAge
    DatatypeRestriction( xsd:integer
      xsd:minExclusive "12"^^xsd:integer
      xsd:maxInclusive "19"^^xsd:integer
    )
  )
)
SubClassOf(
  Annotation( rdfs:comment "States that every man is a person." )
  :Man
  :Person
)
SubClassOf(
  :Father
  ObjectIntersectionOf( :Man :Parent )
)
SubClassOf(
  :ChildlessPerson
  ObjectIntersectionOf(
    :Person
    ObjectComplementOf(
      ObjectSomeValuesFrom(
        ObjectInverseOf( :hasParent )
        owl:Thing
      )
    )
  )
)
SubClassOf(
  ObjectIntersectionOf(
    ObjectOneOf( :Mary :Bill :Meg )
    :Female
  )
  ObjectIntersectionOf(
    :Parent
    ObjectMaxCardinality( 1 :hasChild )
```

```
      ObjectAllValuesFrom( :hasChild :Female )
    )
  )

  EquivalentClasses( :Person :Human )
  EquivalentClasses(
    :Mother
    ObjectIntersectionOf( :Woman :Parent )
  )
  EquivalentClasses(
    :Parent
    ObjectUnionOf( :Mother :Father )
  )
  EquivalentClasses(
    :ChildlessPerson
    ObjectIntersectionOf(
      :Person
      ObjectComplementOf( :Parent )
    )
  )
  EquivalentClasses(
    :Parent
    ObjectSomeValuesFrom( :hasChild :Person )
  )
  EquivalentClasses(
    :HappyPerson
    ObjectIntersectionOf(
       ObjectAllValuesFrom( :hasChild :HappyPerson )
       ObjectSomeValuesFrom( :hasChild :HappyPerson )
    )
  )
  EquivalentClasses(
    :JohnsChildren
    ObjectHasValue( :hasParent :John )
  )
  EquivalentClasses(
    :NarcisticPerson
    ObjectHasSelf( :loves )
  )
  EquivalentClasses(
    :MyBirthdayGuests
    ObjectOneOf( :Bill :John :Mary)
  )
  EquivalentClasses(
    :Orphan
    ObjectAllValuesFrom(
      ObjectInverseOf( :hasChild )
      :Dead
    )
  )
  EquivalentClasses( :Adult otherOnt:Grownup )
  EquivalentClasses(
```

```
    :Parent
    ObjectSomeValuesFrom(
      :hasChild
      :Person
    )
  )

  DisjointClasses( :Woman :Man )
  DisjointClasses(
    :Mother
    :Father
    :YoungChild
  )
  HasKey( :Person () ( :hasSSN ) )

  DatatypeDefinition(
    :personAge
    DatatypeRestriction( xsd:integer
      xsd:minInclusive "0"^^xsd:integer
      xsd:maxInclusive "150"^^xsd:integer
    )
  )
  DatatypeDefinition(
    :minorAge
    DatatypeRestriction( xsd:integer
      xsd:minInclusive "0"^^xsd:integer
      xsd:maxInclusive "18"^^xsd:integer
    )
  )
  DatatypeDefinition(
    :majorAge
    DataIntersectionOf(
      :personAge
      DataComplementOf( :minorAge )
    )
  )
  DatatypeDefinition(
    :toddlerAge
    DataOneOf( "1"^^xsd:integer "2"^^xsd:integer )
  )

  ClassAssertion( :Person :Mary )
  ClassAssertion( :Woman :Mary )
  ClassAssertion(
    ObjectIntersectionOf(
      :Person
      ObjectComplementOf( :Parent )
    )
    :Jack
  )
  ClassAssertion(
    ObjectMaxCardinality( 4 :hasChild :Parent )
```

```
    :John
  )
  ClassAssertion(
    ObjectMinCardinality( 2 :hasChild :Parent )
    :John
  )
  ClassAssertion(
    ObjectExactCardinality( 3 :hasChild :Parent )
    :John
  )
  ClassAssertion(
    ObjectExactCardinality( 5 :hasChild )
    :John
  )
  ClassAssertion( :Father :John )
  ClassAssertion( :SocialRole :Father )

  ObjectPropertyAssertion( :hasWife :John :Mary )
  NegativeObjectPropertyAssertion( :hasWife :Bill :Mary )
  NegativeObjectPropertyAssertion(
    :hasDaughter
    :Bill
    :Susan
  )
  DataPropertyAssertion( :hasAge :John "51"^^xsd:integer )
  NegativeDataPropertyAssertion( :hasAge :Jack "53"^^xsd:integer )

  SameIndividual( :James :Jim )
  SameIndividual( :John otherOnt:JohnBrown )
  SameIndividual( :Mary otherOnt:MaryBrown )
  DifferentIndividuals( :John :Bill )
)
```

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.
- 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.

- Several editorial clarifications and improvements were made.

14.3 Changes Since Last Call

This section summarizes the changes to this document since the Last Call Working Draft of 11 June, 2009.

- Errors in some of the examples were fixed.
- The example ontology was fixed so as to be consistent and syntactically correct.
- A section on OWL 2 tools was added.
- Various links to other OWL 2 documents were added.
- A note was added pointing out that a property being asymmetric is a much stronger notion than its being non-symmetric, and that being symmetric is a much stronger notion than being non-asymmetric.
- A note on the origin of the profile names was added, and it was pointed out that none of the profiles is a subset of another.
- Post Last Call changes to the OWL 2 syntax were incorporated.
- Several editorial clarifications and improvements, minor corrections and fixes, and cosmetic 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 Jie Bao (RPI), Michel Dumontier (Carleton University), Christine Goldbreich (Université de Versailles St-Quentin and LIRMM), Henson Graves (Lockheed Martin), Ivan Herman (W3C/ERCIM), Rinke Hoekstra (University of Amsterdam), Doug Lenat (Cycorp), Deborah L. McGuinness (RPI), Alan Rector (University of Manchester), Alan Ruttenberg (Science Commons) Uli Sattler (University of Manchester), Michael Schneider (FZI), and Mike Smith (Clark & Parsia) for their thorough reviews and helpful comments.

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

[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*.

[DLP]
Description Logic Programs: Combining Logic Programs with Description Logic. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. in Proc. of the 12th Int. World Wide Web Conference (WWW 2003), Budapest, Hungary, 2003. pp.: 48–57

[DL-Lite]
Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. Diego Calvanese, Giuseppe de Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati. J. of Automated Reasoning 39(3):385–429, 2007

[EL++]
Pushing the EL Envelope. Franz Baader, Sebastian Brandt, and Carsten Lutz. In Proc. of the 19th Joint Int. Conf. on Artificial Intelligence (IJCAI 2005), 2005

[FOST]
Foundations of Semantic Web Technologies. Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. Chapman & Hall/CRC, 2009, ISBN: 9781420090505.

[OWL 2 Conformance]
OWL 2 Web Ontology Language: Conformance (Second Edition) Michael Smith, Ian Horrocks, Markus Krötzsch, Birte Glimm, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-conformance-20121211/. Latest version available at http://www.w3.org/TR/owl2-conformance/.

[OWL 2 Manchester Syntax]
OWL 2 Web Ontology Language: Manchester Syntax (Second Edition) Matthew Horridge, Peter F. Patel-Schneider. W3C Working Group Note, 11 December 2012, http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/. Latest version available at http://www.w3.org/TR/owl2-manchester-syntax/.

[OWL 2 New Features and Rationale]
OWL 2 Web Ontology Language: New Features and Rationale (Second Edition) Christine Golbreich, Evan K. Wallace, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-new-features-20121211/. Latest version available at http://www.w3.org/TR/owl2-new-features/.

[OWL 2 Profiles]
OWL 2 Web Ontology Language: Profiles (Second Edition) Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/. Latest version available at http://www.w3.org/TR/owl2-profiles/.

[OWL 2 Profiles Introduction]
OWL 2 Profiles: An Introduction to Lightweight Ontology Languages. Markus Krötzsch. Reasoning Web 2012. Lecture Notes in Computer Science, vol. 7487, pp. 112–183, Springer 2012. To appear. Preprint available at http://korrekt.org/page/OWL_2_Profiles

[OWL 2 Quick Reference Guide]
OWL 2 Web Ontology Language: Quick Reference Guide (Second Edition) Jie Bao, Elisa F. Kendall, Deborah L. McGuinness, Peter F. Patel-Schneider, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-quick-reference-20121211/. Latest version available at http://www.w3.org/TR/owl2-quick-reference/.

[OWL 2 RDF-Based Semantics]
OWL 2 Web Ontology Language: RDF-Based Semantics (Second Edition) Michael Schneider, editor. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/. Latest version available at http://www.w3.org/TR/owl2-rdf-based-semantics/.

[OWL 2 RDF Mapping]
OWL 2 Web Ontology Language: Mapping to RDF Graphs (Second Edition) Peter F. Patel-Schneider, Boris Motik, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/. Latest version available at http://www.w3.org/TR/owl2-mapping-to-rdf/.

[OWL 2 Direct Semantics]
OWL 2 Web Ontology Language: Direct Semantics (Second Edition) Boris Motik, Peter F. Patel-Schneider, Bernardo Cuenca Grau, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/. Latest version available at http://www.w3.org/TR/owl2-direct-semantics/.

[OWL 2 Specification]
OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition) Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/. Latest version available at http://www.w3.org/TR/owl2-syntax/.

[OWL 2 XML Serialization]
OWL 2 Web Ontology Language: XML Serialization (Second Edition) Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, eds. W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211/. Latest version available at http://www.w3.org/TR/owl2-xml-serialization/.

[pD*]
Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Herman J. ter Horst. J. of Web Semantics 3(2–3):79–115, 2005

[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 Semantics]
RDF Semantics. Patrick Hayes, ed., W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/. Latest version available as http://www.w3.org/TR/rdf-mt/.

[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/.

[RFC 3987]
RFC 3987: Internationalized Resource Identifiers (IRIs). M. Duerst and M. Suignard. IETF, January 2005, http://www.ietf.org/rfc/rfc3987.txt

[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/.

[Turtle]
Turtle: Terse RDF Triple Language. Eric Prud'hommeaux and Gavin Carothers. W3C Last Call Working Draft, 10 July 2012, http://www.w3.org/TR/2012/WD-turtle-20120710/. Latest version available at http://www.w3.org/TR/turtle/.

[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/.