Chapter 2

Simple Ontologies in RDF and RDF Schema

The *Resource Description Framework* RDF is a formal language for describing structured information. The goal of RDF is to enable applications to exchange data on the Web while still preserving their original meaning. As opposed to HTML and XML, the main intention now is not to display documents correctly, but rather to allow for further processing and re-combination of the information contained in them. RDF consequently is often viewed as the basic representation format for developing the Semantic Web.

The development of RDF began in the 1990s, and various predecessor languages have influenced the creation process of RDF. A first official specification was published in 1999 by the W3C, though the emphasis at this time still was clearly on the representation of *metadata* about Web resources. The term metadata generally refers to data providing information about given data sets or documents. In 1999, the latter were mainly expected to be Web pages, for which RDF could help to state information on authorship or copyright. Later the vision of the Semantic Web was extended to the representation of semantic information in general, reaching beyond simple RDF data as well as Web documents as primary subjects of such descriptions. This was the motivation for publishing a reworked and extended RDF specification in 2004.

As of today, numerous practical tools are available for dealing with RDF. Virtually every programming language offers libraries for reading and writing RDF documents. Various RDF stores – also called *triple stores* for reasons that shall become clear soon – are available for keeping and processing large amounts of RDF data, and even commercial database vendors are already providing suitable extensions for their products. RDF is also used to exchange (meta) data in specific application areas. The most prominent example of this kind of usage is likely to be RSS 1.0 for syndicating news on the Web.[1] But also metadata belonging to files of desktop applications are sometimes encoded using RDF, such as in the case of Adobe's RDF format XMP for embedding information in PDF files, or as annotations in the XML-based vector graphics format SVG. We will say more about such applications in Chapter 9.

---

[1] RSS 1.0 and 2.0 are different formats, which pursue the same goal but which, confusingly, are not based on each other. RSS 1.0 stands for *RDF Site Summary*, whereas RSS 2.0 is usually interpreted as *Really Simple Syndication*. See also Section 9.1.2.

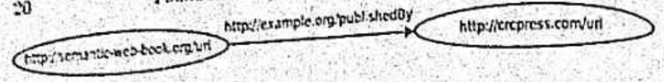

FIGURE 2.1: A simple RDF graph describing the relationship between this book and the publisher, CRC Press

This chapter introduces the basics of RDF. Initially, the representation of simple data is our main concern. In the subsequent sections, we have a closer look at the various syntactic formats available for exchanging RDF, and we address some further questions regarding the usage of RDF. Thereafter we consider some specific expressive features that go beyond the description of simple data. RDF is extended to the language *RDF Schema* (RDFS) for this purpose, allowing us to express also general information about a data set. The official formal semantics as used for properly interpreting RDF and RDFS in computer programs is explained in detail in Chapter 3.

2.1 Introduction to RDF

We begin by giving a very basic introduction to the RDF format that also highlights major differences to XML. As we shall see, RDF is based on a very simple graph-oriented data schema.

2.1.1 Graphs Instead of Trees

An RDF document describes a *directed graph*, i.e. a set of *nodes* that are linked by *directed edges* ("arrows"). Both nodes and edges are labeled with identifiers to distinguish them. Figure 2.1 shows a simple example of a graph of two nodes and one edge. In contrast, as recalled in Appendix A, information in XML is encoded in tree structures. Trees are perfectly suited for organizing information in electronic documents, where we are often confronted with strictly hierarchical structures. In addition, information in trees can often be fetched directly and be processed rather efficiently. Why then is RDF relying on graphs?

An important reason is that RDF was not conceived for the task of structuring documents, but rather for describing general relationships between objects of interest (in RDF one usually speaks of "resources"). The graph in Fig. 2.1, e.g., might be used to express that the book "Foundations of Semantic Web Technologies" was published by "CRC Press" if we interpret the given labels to refer to those objects. The relationship between book and publishing house in this case is information which does not in any obvious sense belong hierarchically below either of the resources. RDF therefore considers such relations as basic building blocks of information. Many such relationships together naturally form graphs, not hierarchical tree structures.

Another reason for choosing graphs is the fact that RDF was intended to serve as a description language for data on the WWW and other electronic networks. Information in these environments is typically stored and managed in decentralized ways, and indeed it is very easy to combine RDF data from multiple sources. For example, the RDF graphs from the website of this book could simply be joined with graphs from `http://semanticweb.org` – this would merely lead to a bigger graph that may or may not provide interesting new information. Note that we generally allow for graphs to consist of multiple unconnected components, i.e. of sub-graphs without any edges between them. Now it is easy to see why such a straightforward approach would not be feasible for combining multiple XML documents. An immediate problem is that the simple union of two tree structures is not a tree anymore, so that additional choices must be made to even obtain a well-formed XML document when combining multiple inputs. Moreover, related information items in trees might be separated by the strict structure: even if two XML files refer to the same resources, related information is likely to be found in very different locations in each tree. Graphs in RDF are therefore better suited for the composition of distributed information sources.

Note that these observations refer to the *semantic* way in which RDF structures information, not to the question of how to encode RDF data *syntactically*. We will see below that XML is still very useful for the latter purpose.

2.1.2 Names in RDF: URIs

We have claimed above that RDF graphs enable the simple composition of distributed data. This statement so far refers only to the graph structure in general, but not necessarily to the intended information in the composed graphs. An essential problem is that resources, just like in XML, may not have uniform identifiers within different RDF documents. Even when two documents contain information on related topics, the identifiers they use might be completely unrelated. On the one hand, it may happen that the same resource is labeled with different identifiers, for instance, since there is no globally agreed identifier for the book "Foundations of Semantic Web Technologies." On the other hand, it may occur that the same identifiers are used for different resources, e.g., "CRC" could refer to the publishing house as well as to the official currency of Puerto Rico. Such ambiguity would obviously be a major problem when trying to process and compose information automatically.

To solve the latter problem, RDF uses so-called *Uniform Resource Identifiers* (*URIs*) as names to clearly distinguish resources from each other. URIs are a generalization of URLs (Uniform Resource Locators), i.e. of Web addresses as they are used for accessing online documents. Every URL is also a valid URI, and URLs can indeed be used as identifiers in RDF documents that talk about Web resources. In numerous other applications, however, the goal is not to exchange information about Web pages but about many different kinds of objects. In general this might be any object that has a clear identity in the context of the given application: books, places, people, publishing houses, events, relationships among such things, all kinds of abstract concepts, and many more. Such resources can obviously not be retrieved online and hence their URIs are used exclusively for unique identification. URIs that are not URLs are sometimes also called *Uniform Resource Names* (URNs).

Even if URIs can refer to resources that are not located on the Web, they are still based on a similar construction scheme as common Web addresses. Figure 2.2 gives an overview of the construction of URIs, and explains their relevant parts. The main characteristic of any URI is its initial scheme part. While schemes like `http` are typically associated with a protocol for transmitting information, we also find such schemes in many URIs that do not refer to an actual Web location. The details of the protocol are obviously not relevant when using a URI only as a name. The book "Foundations of Semantic Web Technologies" could, e.g., use the URI `http://semantic-web-book.org/uri` and it would not matter whether or not a document can be retrieved at the corresponding location, and whether this document is relevant in the given context. As we shall see later on, RDF makes use of various mechanisms of XML to abbreviate URIs when convenient.

As shown in Fig. 2.1, nodes and edges in RDF graphs both are labeled with URIs to distinguish them from other resources. This rule has two possible exceptions: RDF allows for the encoding of data values which are not URIs, and it features so-called *blank nodes* which do not carry any name. We will take a closer look at both cases next. Later we will also return to the question of finding good URIs in practice, in a way that ensures maximal utility and reliability in semantic applications. For now we are satisfied with the insight that URIs, if they are well-chosen, provide us with a robust mechanism for distinguishing different entities, thus avoiding confusion when combining RDF data from distributed sources.

2.1.3 Data Values in RDF: Literals

URIs allow us to name abstract resources, even those that cannot be represented or processed directly by a computer. URIs in this case are merely references to the intended objects (people, books, publishers, ...). While URIs can always be treated as names, the actual "intended" interpretation of particular URIs is not given in any formal way, and specific tools may have their own way of interpreting certain URIs. A certain Web Service, e.g., may recognize URIs that refer to books and treat them in a special way by displaying purchasing options or current prices. This degree of freedom is useful and in fact unavoidable when dealing with arbitrary resources. The situation is different when dealing with concrete data values such as numbers, times,

The general construction scheme of URIs is summarized below, where parts in brackets are optional:

`scheme:[//authority]path[?query][#fragment]`

The meaning of the various URI parts is as follows:

`scheme` The name of a URI scheme that classifies the type of URI. Schemes may also provide additional information on how to handle URIs in applications. Examples: `http`, `ftp`, `mailto`, `file`, `irc`

`authority` URIs of some URI schemes refer to "authorities" for structuring the available identifiers further. On the Web, this is typically a domain name, possibly with additional user and port details. The authority part of a URI is optional and can be recognized by the preceding `//`. Examples: `semantic-web-book.org`, `john@example.com`, `example.org:8080`

`path` The path is the main part of many URIs, though it is possible to use empty paths, e.g., in email addresses. Paths can be organized hierarchically using `/` as separator. Examples: `/etc/passwd`, `this/path/with/-:_~/is/../okay` (paths without initial `/` are only allowed if no authority is given)

`query` The query is an optional part of the URI that provides additional non-hierarchical information. It can be recognized by its preceding `?`. In URLs, queries are typically used for providing parameters, e.g., to a Web Service. Example: `q=Semantic+Web+book`

`fragment` The optional fragment part provides a second level of identifying resources, and its presence is recognized by the preceding `#`. In URLs, fragments are often used to address a sub-part of a retrieved resource, such as a section in an HTML file. URIs with different fragments are still different names for the purpose of RDF, even if they may lead to the same document being retrieved in a browser. Example: `section1`

Not all characters are allowed in all positions of a URI, and illegal symbols are sometimes encoded by specific means. For the purpose of this book it suffices to know that basic Latin letters and numbers are allowed in almost any position. Moreover, the use of non-Latin characters that abound in many languages is widely allowed in all current Semantic Web formats as well. URIs that are extended in this way are known as *International Resource Identifiers* (IRIs), and they can be used in any place where URIs are considered in this book.

FIGURE 2.2: The basic construction scheme for URIs

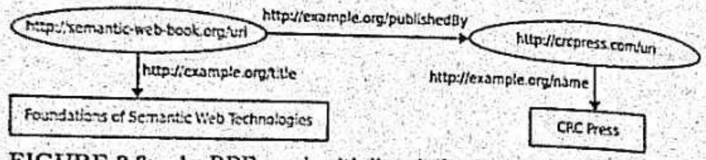

FIGURE 2.3: An RDF graph with literals for describing data values

or truth values: in these cases, we would expect every application to have a minimal understanding of the concrete meaning of such values. The number 42, e.g., has the same numeric interpretation in any context.

Data values in RDF are represented by so-called *literals*. These are reserved names for RDF resources of a certain datatype. The value of every literal is generally described by a sequence of characters, such as the string consisting of the symbols "4" and "2" in the above example. The interpretation of such sequences is then determined based on a given *datatype*. Knowing the datatype is crucial for understanding the intended meaning: the character sequences "42" and "042", e.g., refer to the same natural number but to different text strings.

For the time being, we will consider only literals for which no datatype has been given. Such *untyped* literals are always interpreted as text strings. The slightly more complex form of literals that contains an additional datatype identifier will be explained later on.

As can be seen in Fig. 2.3, rectangular boxes are used to distinguish literals from URIs when drawing RDF graphs. Another special trait of literals is that they may never be the origin of edges in an RDF graph. In practice, this means that we cannot make direct statements about literals.[2] This constraint needs to be taken into account when modeling data in RDF. Moreover, it is not allowed to use literals as labels for edges in RDF graphs – a minor restriction since it is hard to see what could be intended with such a labeling. Note that it is still allowed to use the same URI for labeling both nodes and edges in a graph, so at least in RDF there is no principle separation between resources used for either purpose.

2.2 Syntax for RDF

Up to this point, we have described RDF graphs by means of drawing diagrams. This way of representing RDF is easy to read and still precise, yet it is clearly not suitable for processing RDF in computer systems. Even for humans, understanding visual graphs works without much effort only if the graphs are very small – practically relevant data sets with thousands or millions of nodes do obviously not lend themselves to being stored and communicated in pictures. This section thus introduces ways of representing RDF by means of character strings that can easily be kept in electronic documents. This requires us to split the original graph into smaller parts that can be stored one by one. Such a transformation of complex data structures into linear strings is called *serialization*.

2.2.1 From Graphs to Triples

Computer science has various common ways of representing graphs as character strings, e.g., by using an adjacency matrix. RDF graphs, however, are typically very sparse graphs within which the vast majority of possible relationships do not hold. In such a case it makes sense to represent the graph as the set of edges that are actually given, and to store each edge on its own. In the example of Fig. 2.1 this is exactly one edge, uniquely determined by its start `http://semantic-web-book.org/uri`, label `http://example.org/publishedBy`, and endpoint `http://crcpress.com/uri`. Those three distinguished parts are called *subject*, *predicate*, and *object*, respectively.

It is easy to see that every RDF graph can, in essence, be completely described by its edges. There are of course many ways for drawing such graphs, but the details of the visual layout clearly have no effect on the information the graph conveys. Now every such edge corresponds to an *RDF triple* "subject-predicate-object." As we have seen above, each part of a triple can be a URI, though the object might also be a literal. Another special case is *blank nodes* that we will consider later.

2.2.2 Simple Triple Syntax: N3, N-Triples and Turtle

Our earlier observations suggest that one denotes RDF graphs simply as a collection of all their triples, given in arbitrary order. This basic idea has indeed been taken up in various concrete proposals for serializing RDF. A realization that dates back to 1998 is Tim Berners-Lee's *Notation 3* (N3), which also includes some more complex expressions such as paths and rules. The RDF recommendation of 2004 therefore proposed a less complicated part of N3 under the name *N-Triples* as a possible syntax for RDF. N-triples in

---

[2] The reason for this restriction is in fact historic, and an official resolution of the RDF-Core working group notes that it could be waived in future Semantic Web languages; see `http://www.w3.org/2000/03/rdf-tracking/#rdfms-literalsubjects`.

turn was further extended to incorporate various convenient abbreviations, leading to the RDF syntax *Turtle* which is hitherto not described in an official standardization document. Both N-Triple and Turtle are essentially parts of N3, restricted to covering only valid RDF graphs. Here we consider the more modern Turtle syntax.

The graph of Fig. 2.3 is written in Turtle as follows:

```
<http://semantic-web-book.org/uri>
   <http://example.org/publishedBy>  <http://crcpress.com/uri> .
<http://semantic-web-book.org/uri>
   <http://example.org/title>
       "Foundations of Semantic Web Technologies" .
<http://crcpress.com/uri>
   <http://example.org/name>          "CRC Press" .
```

URIs are thus written in angular brackets, literals are written in quotation marks, and every statement is terminated by a full stop. Besides those specific characteristics, however, the syntax is a direct translation of the RDF graph into triples. Spaces and line breaks are only relevant if used within URIs or literals, and are ignored otherwise. Our lengthy names force us to spread single triples over multiple lines. Due to the hierarchical structure of URIs, the identifiers in RDF documents typically use similar prefixes. Turtle offers a mechanism for abbreviating such URIs using so-called *namespaces*. The previous example can be written as follows:

```
@prefix book: <http://semantic-[illegible]book.org/> .
@prefix ex: <http://example.org/> .
@prefix crc: <http://crcpress.com/> .

book:uri  ex:publishedBy  crc:uri .
book:uri  ex:title        "Foundations of Semantic Web Technologies" .
crc:uri   ex:name         "CRC Press" .
```

URIs are now abbreviated using prefixes of the form "prefix:" and are no longer enclosed in angular brackets. Without the latter modification it would be possible to confuse the abbreviated forms with full URIs, e.g., since it is allowable to use a prefix "http:" in namespace declarations. The prefix text that is used for abbreviating a particular URI part can be chosen freely, but it is recommended to select abbreviations that are easy to read and that refer the human reader to what they abbreviate. Identifiers of the form "prefix:name" are also known as *QNames* (for *qualified names*).

It frequently happens that RDF descriptions contain many triples with the same subject, or even with the same subject and predicate. For those common cases, Turtle provides further shortcuts as shown in the following example:

```
@prefix book: <http://semantic-web-book.org/> .
@prefix ex: <http://example.org/> .
@prefix crc: <http://crcpress.com/> .

book:uri  ex:publishedBy  crc:uri ;
          ex:title        "Foundations of Semantic Web Technologies" .
crc:uri   ex:name         "CRC Press", "CRC" .
```

The semicolon after the first line terminates the triple, and at the same time fixes the subject book:uri for the next triple. This allows us to write many triples for one subject without repeating the name of the subject. The comma in the last line similarly finishes the triple, but this time both subject and predicate are re-used for the next triple. Hence the final line in fact specifies two triples providing two different names. The overall RDF graph therefore consists of four edges and four nodes. It is possible to combine semicolon and comma, as shown in the next example with four triples:

```
@prefix book: <http://semantic-web-book.org/> .
@prefix ex: <http://example.org/> .

book:uri  ex:author book:Hitzler, book:Krötzsch, book:Rudolph ;
          ex:title  "Foundations of Semantic Web Technologies" .
```

The above abbreviations are not contained in the official (normative) W3C syntax N-Triples which allows neither namespaces, nor comma or semicolon. Yet, Turtle's syntactic shortcuts are frequently encountered in practice, and they have influenced the triple syntax of W3C's more recent SPARQL specification, introduced in Chapter 7.

2.2.3 The XML Serialization of RDF

The Turtle representation of RDF can easily be processed by machines but is still accessible for humans with relatively little effort. Yet, triple representations like Turtle are by far not the most commonly used RDF syntax in practice. One reason for this might be that many programming languages do not offer standard libraries for processing Turtle syntax, thus requiring developers to write their own tools for reading and writing to files. In contrast, essentially every programming language offers libraries for processing XML files, so that application developers can build on existing solutions for storage and pre-processing. As of today, the main syntax for RDF therefore is the XML-based serialization RDF/XML that is introduced in this section. This syntax also offers a number of additional features and abbreviations that can be convenient to represent advanced features which we will encounter later on, but at the same time it imposes some additional technical restrictions. Readers who are not familiar with the basics of XML may wish to consult Appendix A for a quick introduction.

The differences of the data models of XML (trees) and RDF (graphs) are no obstacle, since XML only provides the syntactic structure used for organizing an RDF document. Since XML requires hierarchic structures, the encoding of triples now must as well be hierarchical. The space efficient Turtle descriptions of the previous section have illustrated that it is often useful to assign multiple predicate-object pairs to a single subject. Accordingly, triples in RDF/XML are grouped by their subjects. The following example encodes the RDF graph from Fig. 2.1:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex ="http://example.org/">

  <rdf:Description rdf:about="http://semantic-web-book.org/uri">
    <ex:publishedBy>
      <rdf:Description rdf:about="http://crcpress.com/uri">
      </rdf:Description>
    </ex:publishedBy>
  </rdf:Description>

</rdf:RDF>
```

After an optional specification of XML version and encoding, the document starts with a first node of type `rdf:RDF`. This element is generally used as the root of any RDF/XML document. At this place, we also declare the global (XML-)namespaces for `ex:` and `rdf:`. Just as in Turtle, namespaces allow us to abbreviate URIs with QNames, this time building upon the existing XML namespace mechanism. While abbreviations for namespaces are still mostly arbitrary, it is a convention to use the prefix `rdf:` for the RDF namespace as given in the above example. In the following, elements that have a special meaning in the RDF serialization are recognized by that prefix.

Nested within the element `rdf:RDF`, we find the encoding of the sole triple of the above example. Subject and object are described by elements of the type `rdf:Description`, where the XML attribute `rdf:about` defines the identifier of the resource. The predicate of the encoded triple is represented directly as the element `ex:publishedBy`.

Multiple triples can be encoded by representing each of them by a separate element of type `rdf:Description`, which may lead to multiple such elements referring to the same subject. Likewise, the order of the triples is of course not important. However, it is also possible to nest elements of type `rdf:Description`, possibly leading to a more concise serialization. The following example encodes the graph from Fig. 2.3:[3]

```
<rdf:Description rdf:about="http://semantic-web-book.org/uri">
  <ex:title>Foundations of Semantic Web Technologies</ex:title>
  <ex:publishedBy>
    <rdf:Description rdf:about="http://crcpress.com/uri">
      <ex:name>CRC Press</ex:name>
    </rdf:Description>
  </ex:publishedBy>
</rdf:Description>
```

Here we can see how literals are represented simply as the contents of a predicate-element. The name "CRC Press" is given directly by nesting XML elements instead of creating a second top-level subject element for describing `http://crcpress.com/uri`. Some further abbreviations are allowed:

```
<rdf:Description rdf:about="http://semantic-web-book/uri"
             ex:title= "Foundations of Semantic Web Technologies">
  <ex:publishedBy rdf:resource="http://crcpress.com/uri" />
</rdf:Description>
<rdf:Description rdf:about="http://crcpress.com/uri"
                                         ex:Name="CRC Press" />
```

This syntax requires some explanation. First of all, all predicates with literal objects have been encoded as XML attributes. This abbreviation is admissible only for literals – objects referred to by URIs cannot be encoded in this way, since they would then be misinterpreted as literal strings.

Moreover, the element `ex:publishedBy` makes use of the special attribute `rdf:resource`. This directly specifies the object of the triple, such that no further nested element of type `rdf:Description` is necessary. This is the reason why `ex:publishedBy` has no content so that it can be written as an empty-element tag, as opposed to giving separate start and end tags. This shortcut notation is only allowed for URIs, i.e., every value of `rdf:resource` is considered as a URI.

Since we thus avoid a nested description of `http://crcpress.com/uri`, another description appears at the outer level. The predicate `ex:Name` is again encoded as an XML attribute and the otherwise empty description can be closed immediately.

---

[3] In many cases, we show only the interesting parts of an RDF/XML document in examples. The declaration of `rdf:RDF` can always be assumed to be the same as in the initial example on page 28.

We see that RDF/XML provides a multitude of different options for representing RDF. Some of those options stem from the underlying XML syntax. As an example, it is not relevant whether or not an element without contents is encoded by an empty-element tag instead of giving both start and end tags. A larger amount of freedom, however, is provided by RDF since the same triples can be encoded in many different ways. Our previous two examples certainly do not describe the same XML tree, yet they encode the same RDF graph.

W3C Validator The *W3C Validator* is a Web Service that can be employed to check the validity of RDF/XML documents with respect to the official specification. A simple online form is provided to upload XML-encoded RDF which is then validated. Valid documents are processed to extract individual triples and a visualization of the corresponding graph, whereas invalid documents lead to error messages that simplify the diagnosis of problems. This Web Service can also be used to investigate RDF/XML examples given within this book, though it should not be forgotten that many examples are only partial and must be augmented with a suitable `rdf:RDF` declaration to become valid.
The W3C Validator is found at `http://www.w3.org/RDF/Validator/`

2.2.4 RDF in XML: URIs and Other Problems

Namespaces in RDF/XML have been introduced above as a way of abbreviating URIs in the RDF/XML serialization. The truth, however, is that namespaces in RDF/XML are an indispensable part of the encoding rather than an optional convenience. The reason is that RDF/XML requires us to use resource identifiers as names of XML elements and attributes. But all URIs necessarily contain a colon – a symbol that is not allowed in XML names! Using namespaces, we can "hide" a URI's own colon within the declared prefix.

On the other hand, namespaces can only be used for abbreviating XML tags and attributes, but are not allowed within attribute values and plain text contents between XML tags. This is the reason why we used the complete URI `http://semantic-web-book/uri` in all previous examples, instead of employing a QName `book:uri` as in our earlier Turtle examples. An attribute assignment of the form `rdf:about="book:uri"` is not correct, and book in this case would be interpreted as the scheme part of a URI but not as an XML namespace prefix.

Thus we are in the unfortunate situation of having to write the same URI differently in different positions of an RDF/XML document. The next section introduces a method that still allows us to at least abbreviate URIs in cases where namespaces cannot be used. XML has a number of further syntactic restrictions that may complicate the encoding of arbitrary RDF graphs. It

is, e.g., not allowed to use a hyphen directly after a colon in XML tags, even though hyphens might occur within URIs. It may thus become necessary to declare auxiliary namespaces merely for the purpose of exporting single elements in a valid way.

Another practical problem is that the percent sign `%` occurs frequently within URLs since it is used to escape forbidden characters. The string `%20`, e.g., encodes the space character. Just like colon, the percent sign is not allowed in XML tags, and it can happen that existing URLs cannot be used as URIs in RDF. Fortunately, many problems of this kind are already addressed by existing RDF programming libraries, so that application developers do not need to focus on such serialization issues. Yet they should be aware that there are valid URIs that cannot be encoded at all in XML.

2.2.5 Shorter URIs: XML Entities and Relative URIs

In the above examples, we have always used absolute URIs as values of the attributes `rdf:about` and `rdf:resource`, as the use of namespaces would not be admissible in this context. This section discusses two methods for abbreviating such values as well. While these abbreviations are of course optional additions to the basic syntax, they are very widely used and thus indispensable for understanding most of today's RDF documents.

A simple method to abbreviate values in XML is the use of so-called *XML entities*. An entity in XML is a kind of shortcut that can be declared at the beginning of a document, and referred to later in the document instead of giving its complete value. The following is a concrete example of an XML document using this feature:

```
<?xml version="1.0" encoding="utf-8"?> <!DOCTYPE rdf:RDF[
      <!ENTITY book 'http://semantic-web-book.org/'>
]>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex ="http://example.org/">

  <rdf:Description rdf:about="&book;uri">
    <ex:title>Foundations of Semantic Web Technologies</ex:title>
  </rdf:Description>

</rdf:RDF>
```

An obvious novelty in this example is the initial entity declaration enclosed in `<!DOCTYPE rdf:RDF[` and `]>`. This part of the XML document constitutes its *document type declaration* which might provide a so-called *Document Type Definition* (DTD). A DTD can be used to declare entities as above, but also to define a number of further restrictions on the contents of the XML document. All document type declarations used in this book, however, are plain entity definitions, so that we can content ourselves with knowing that entities in RDF/XML are defined as above. In our example, the entity defined is called book and its value is `http://semantic-web-book.org/`. Further entities could easily be defined by providing additional lines of the form `<!ENTITY name 'value'>`.

We may now refer to our newly defined entity by writing `&book;` as in the value of `rdf:about` above, and the XML document is interpreted just as if we had written the declared value of the entity at this position. Such entity references are allowed in XML attribute values and within plain text data contained in an element, such as the texts used for serializing data literals in Section 2.2.3. Entities cannot be used within names of XML elements and attributes – there we have to stick to the use of namespaces. In our current example, defining a new entity does not actually shorten the document, but usually entities for common URI prefixes lead to much more concise serializations and may also increase readability. XML also provides a small number of pre-defined entities that are useful for encoding certain symbols that would otherwise be confused with parts of the XML syntax. These entities are `<` (<), `>` (>), `&` (&), `'` ('), and `"` (").

There is another common case in which URIs might be abbreviated: in many RDF documents, URIs primarily stem from a common *base namespace*. A website that exports data in RDF, e.g., is likely to use many URIs that begin with the site's domain name. XML has the concept of a *base URI* that can be set for elements in a document using the attribute `xml:base`. Other attributes in the XML document then may, instead of full URIs, use so-called *relative references*. Such entries refer to a full URIs which are obtained by preceding the entries with the given base URI, as illustrated by the following example:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex ="http://example.org/"
         xml:base ="http://semantic-web-book.org/" >

  <rdf:Description rdf:about="uri">
    <ex:publishedBy rdf:resource="http://crcpress.com/uri" />
  </rdf:Description>

</rdf:RDF>
```

The relative reference `rdf:about="uri"` is thus interpreted as the URI `http://semantic-web-book/uri`. Values of `rdf:resource` or `rdf:datatype` (explained later) can be abbreviated in the same fashion. Relative references are distinguished from full URIs by lacking a scheme part; see Fig. 2.2. It is possible to use relative references even without declaring the intended base URI beforehand: in this case, the base URI of the document – based on the URL it was retrieved from – is used. This mechanism is less robust since locations of documents may change; hence it is suggested to provide explicit base URIs whenever needed.

A second common use of `xml:base` for abbreviating URIs relates to the attribute `rdf:ID`. This attribute can be used just like `rdf:about`, but it always expects a single fragment identifier as its value, whereas complete URIs are not allowed. The full URI is then constructed by using the given value as a fragment for the base URI (which thus should not contain any fragment), i.e. we can obtain the URI by extending the base URI with the symbol # followed by the value of `rdf:ID`.

Thus we find that `rdf:ID="name"` has essentially the same meaning as `rdf:about="#name"`. The most relevant difference of both ways of writing URIs is that every value of `rdf:ID` must be used only once for a given base URI. An RDF/XML document may thus contain one element with a given ID, but it may still contain further elements that refer to the same URI by means of `rdf:about` and `rdf:resource`.

The Turtle syntax for RDF provides a similar support for relative references, which are resolved by using the base URI (URL) of the document. Setting the base URI explicitly is not encompassed by the current specification, even though the syntax `@base` was proposed for this purpose. Overall, relative references in Turtle are of only minor importance since namespace declarations can be used without the restrictions of the XML syntax.

Figure 2.4 provides an overview of the various forms of abbreviation mechanisms that we have introduced for RDF/XML. Note that a principal difference between XML entities and (base) namespaces is that the former can be declared only once for the whole document, whereas the latter may be declared in arbitrary XML start tags or empty-element tags. The namespaces then apply to the element within which they were declared, and to all subelements thereof. Moreover, entities can be used not only for abbreviating URIs but provide shortcuts for arbitrary text content, even within literal values.

2.2.6 Where Do URIs Come From? What Do They Mean?

Does the use of URIs, which is strictly required throughout RDF, allow for a semantically unambiguous interpretation of all RDF-encoded information? The answer is clearly no. It is still possible to use different URIs for the same resource, just as it is still possible to use the same URI for different things. A possible solution for this problem is the use of well-defined *vocabularies*. As in XML, the term vocabulary in RDF is most commonly used to refer to collections of identifiers with a clearly defined meaning. A typical example is provided by RDF itself: the URI

`http://www.w3.org/1999/02/22-rdf-syntax-ns#Description`,

Namespace declaration	Usage: `namespace:name` in XML element names Declaration: `xml:namespace="`*<uri>*`"` in XML start tags or empty-element tags; declarations affect XML subtree; multiple declarations possible
Entity declaration	Usage: `&entity;` in XML attribute values or character content (RDF literal values) of elements Declaration: `<!ENTITY entity '`*text*`'>` in initial `DOCTYPE` declaration; declaration affects whole document; only one declaration possible
Predefined entities	Usage: `<`, `>`, `&`, `'`, or `"` in XML attribute values or character content (RDF literal values) of elements Declaration: predefined in XML, no declaration
Base namespace	Usage: non-URI name as value for `rdf:about`, `rdf:resource`, `rdf:ID`, or `rdf:datatype` Declaration: `xml:base="`*<uri>*`"` in XML start tags or empty-element tags; declarations affect XML subtree; multiple declarations possible

FIGURE 2.4: Summary of abbreviation mechanisms in RDF/XML.

e.g., has a generally accepted well-defined meaning which applications may take into account.

But vocabularies are not just used to define the RDF/XML syntax as such; they are also commonly used to describe information. An example of a particularly popular vocabulary is *FOAF* (*Friend Of A Friend*), which defines URIs to describe people and their relationships (see Section 9.1.2 for details). Even though FOAF is not specified by an official standardization authority, its URIs are sufficiently well known to avoid confusion. Both FOAF and RDF itself illustrate that the intended use and meaning of vocabularies are typically not encoded in a machine-readable way.

One of the major misconceptions regarding the Semantic Web is the belief that semantic technologies enable computers to truly *understand* complex concepts such as "person." The unambiguous assignment of URIs indeed allows us to refer to such concepts, and to use them in a multitude of semantic relationships – actually comprehending the contents of the encoded statements, however, is still the task of the human user. This should be obvious based on our daily experiences of the capabilities and limitations of today's computers, but new technologies often are accompanied by a certain amount of inflated expectations. It is still possible to describe a certain amount of complex relationships that may refer to a certain vocabulary in a way that is readable by machines: this is the main aim of the *ontology languages* RDF Schema and OWL that we consider later on.

In many cases, a vocabulary for a certain topic area is not readily available, and it is clearly never possible to assign URIs to all conceivable resources. Therefore it is required to introduce new URIs on demand, and various proposals and guidelines have been developed for coining new URIs on the Semantic Web. It also makes sense to take the relationship between URLs and URIs into account in this context.

In some situations, very concrete guidelines are available for creating suitable URIs. There is, e.g., an official policy for turning phone numbers into URIs using the scheme `tel`. Similar proposals exist for deriving URIs for books and journals from the ISSN or ISBN numbers.

In numerous other cases, however, it is required to coin completely new URIs. A first objective in this case must be to ensure that the chosen URI is not used elsewhere, possibly with a different intended meaning. This is often surprisingly easy to do by taking advantage of the existing hierarchic mechanisms for managing URLs. By choosing URIs that - when viewed as URLs – refer to locations on the Web over which one has complete control, one can usually avoid clashes with existing URIs. Moreover, it is then possible to make a document available at the corresponding location, providing an authoritative explanation of the intended meaning. The information about the proper usage of a URI thus becomes accessible worldwide.

An important related aspect is the distinction between Web pages and other (abstract) resources. The URL `http://en.wikipedia.org/wiki/Othello` e.g., at first appears to be a suitable URI for Shakespeare's drama, since it contains an unambiguous description of this resource. If an RDF document assigns an author to this URI, however, it is not clear whether this refers to the existing HTML page or to the drama. One thus could be led to believe that Shakespeare has edited pages on Wikipedia, or that Othello was written collaboratively by authors such as "*User:The_Drama_Llama*". It is thus obvious why URLs of existing documents are not well-suited as URIs for abstract concepts.

On the other hand, we would still like to construct URIs that point to existing Web documents. For users of RDF, it would certainly be useful if URIs could be used to learn more about their intended usage – like an inherent user documentation closely tied to any RDF document. But how can this be accomplished without using existing URLs? One option is the use of fragment identifiers. By writing, e.g., `http://en.wikipedia.org/wiki/Othello#uri` one does not use the URL of an existing Web document (since the fragment "uri" is not defined on the page retrieved at the base URL). Yet, when resolving this URI in a browser, one obtains the same explanatory document as before. This solution is also suggested by the possibility of using relative references together with the attribute `rdf:ID` explained earlier.

An alternative option is the use of redirects: even if no document is found at a given URL, a Web server may redirect users to an alternative page. This is a core functionality of the HTTP protocol. Since the user-side application notices any such HTTP redirect, the retrieved page can still be distinguished from the resource that the original URI referred to. The automatic redirect also has the advantage that a single URI may redirect either to a human-readable HTML description, or to a machine-readable RDF document – the server may select which of those is appropriate based on information that the client provides when requesting the data. This method is known as *content negotiation*. An example is the URI `http://semanticweb.org/id/Markus`: when viewed in a browser, it provides details on the encoded resource; when accessed by an RDF-processing tool such as Tabulator,[4] it returns RDF-based metadata.

The above technical tricks allow us to create unambiguous URIs that link to their own documentation, and this explains to some extent why many URIs still refer to common URL schemes such as `http`.

---

[4] An RDF browsing tool; see `http://www.w3.org/2005/ajar/tab`.

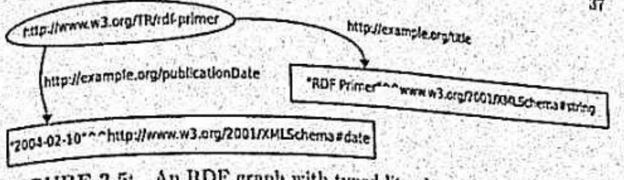

FIGURE 2.5: An RDF graph with typed literals

2.3 Advanced Features

We already have learned about all the basic features of RDF. There are, however, a number of additional and derived expressive means, which are highly important in applications. This section introduces a number of these advanced features in detail. In each case, we consider presentations using RDF graphs, Turtle syntax, and RDF/XML.

2.3.1 Datatypes in RDF

We have already seen in Section 2.1.3 that RDF allows us to describe data values by means of literals. So far, however, all literals we considered have been nothing more than mere character strings. Practical applications of course require many further datatypes, e.g., to denote numbers or points in time. Datatypes usually have a major effect on the interpretation of a given value. A typical example is the task of sorting data values: The natural order of the values "10", "02", "2" is completely different depending on whether we interpret them as numbers or as strings. The latter are usually sorted alphabetically, yielding "02" < "10" < "2", while the former would be sorted numerically to obtain "2" = "02" < "10".

RDF therefore allows literals to carry an explicit datatype. Staying true to our established principles, each datatype is uniquely identified by a URI, and might be chosen rather arbitrarily. In practice, however, it is certainly most useful to refer to datatype URIs that are widely known and supported by many software tools. For this reason, RDF suggests the use of XML Schema. Figure 2.5 illustrates how additional datatype information might be added to an RDF graph. The subject of this example is the RDF Primer document, identified by its actual URL, for which a title text and publication date are provided. These data values are specified by a literal string in quotation marks, followed by ^^ and the URI of some datatype. As datatypes, we have used "string" for simple character sequences, and "date" for calendar days.

It can be seen from the graphical representation that typed literals in RDF are considered as single elements. Any such literal therefore essentially behaves just like a single untyped literal. From this we can readily derive the Turtle syntax for the RDF document in Fig. 2.5:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
<http://www.w3.org/TR/rdf-primer>
      <http://example.org/title>  "RDF Primer"^^xsd:string ;
      <http://example.org/publicationDate> "2004-02-10"^^xsd:date .
```

As the example shows, datatype URIs in Turtle can be abbreviated using namespaces. If they were written as complete URIs, they would need to be enclosed in angular brackets just as any other URI. The representation in RDF/XML is slightly different, using an additional XML attribute `rdf:datatype`:

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-primer">
  <ex:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   RDF Primer
  </ex:title>
  <ex:publicationDate
           rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
   2004-02-10
  </ex:publicationDate>
</rdf:Description>
```

The general restrictions on the use of namespaces in XML also apply to the previous example. Since datatype URIs are specified as XML attribute values, they cannot be abbreviated by namespaces. We may, however, introduce XML entities for arriving at a more concise serialization.

To obtain a better understanding of RDF's datatype mechanism, it makes sense to have a closer look at the meaning of datatypes. Intuitively, we would expect any datatype to describe a certain *value space*, such as, e.g., the natural numbers. This fixes the set of possible values that literals of a datatype denote. A second important component is the set of all admissible literal strings. This so-called *lexical space* of a datatype enables implementations to recognize whether or not a given literal syntactically belongs to the specified datatype. The third and final component of each datatype then is a well-defined mapping from the lexical space to the value space, assigning a concrete value to every admissible literal string.

As an example, we consider the datatype *decimal* that is defined in XML Schema. The value space of this datatype is the set of all rational numbers that can be written as finite decimal numbers. We thus exclude irrational numbers like π, and rational numbers like 1/3 that would require infinitely many digits in decimal notation. Accordingly, the lexical space consists of all character strings that contain only numerals 0 to 9, at most one occurrence of `.`, and an optional initial symbol + or −. The mapping between lexical space and value space is the well-known interpretation of decimal numbers as rationals. The literal strings `3.14`, `+03.14`, and `3.14000`, e.g., are multiple possible ways to refer to the rational number 3.14. It is common in many datatypes that a single value can be denoted in multiple different ways. Applications that support a datatype thus recognize syntactically different RDF literals as being semantically equal.

Most of the common XML datatypes allow for a meaningful interpretation in RDF, yet the RDF specification leaves it to individual implementations to decide which datatypes are supported. In particular, a software tool can conform to the RDF specification without recognizing any additional XML datatypes.

The sole exception to this general principle is RDF's only built-in datatype `rdf:XMLLiteral`. This datatype allows the embedding of well-formed XML snippets as literal values in RDF. As such, the datatype specifically addresses a possible use case of RDF/XML where it might be convenient to use well-formed XML directly in the place of literal values.

The datatype `rdf:XMLLiteral` is most commonly used together with an additional function for pre-processing and normalizing XML data. This is achieved by means of the attribute `rdf:parseType`, which we shall also encounter in various other contexts later on:

```
<rdf:Description rdf:about="http://semantic-web-book/uri">
  <ex:title rdf:parseType="Literal">
    Foundations of
    <br />
    <b>Semantic Web Technologies</b>
  </ex:title>
</rdf:Description>
```

In this example, we have embedded text that uses HTML mark-up into an RDF document. Due to the setting `rdf:parseType="Literal"`, the given XML fragment is normalized internally, and transformed into a literal of type `rdf:XMLLiteral`. Even though XML snippets that are used in this way need not be complete XML documents, it is required that their opening and closing tags are balanced. If this cannot be guaranteed in an application, it is also common to embed XML fragments into RDF/XML as string literals, using pre-defined entities like `&` and `<` to replace XML syntax that would otherwise be confused with the remainder of the RDF document.

At this point we should also make sure that we have not allowed ourselves to be confused by some rather similar terms which have been introduced: RDF literals generally are syntactic identifiers for data values in RDF, whereas the value Literal of the attribute rdf:parseType merely leads to the creation of one particular kind of literals belonging to the datatype rdf:XMLLiteral.

2.3.2 Language Settings and Datatypes

Now that we are more familiar with the comprehensive datatype system of RDF and XML Schema, it is in order to take another look at data values in RDF. The first obvious question is which datatype literals without any type assignment actually have. In fact, such *untyped literals* simply have no type at all, even though they behave very similarly to typed literals of type xsd:string for most practical purposes.

An important difference between typed and untyped literals is revealed when introducing language information into RDF. XML in general supports the specification of language information that tells applications whether part of an XML document's content is written in a particular (natural) language. This is achieved by means of the attribute xml:lang. A typical example is the language setting in (X)HTML documents as found on the Web, which often contain attribute assignments such as xml:lang="en" or xml:lang="de-ch" in their initial html tag. Not surprisingly, such language information in XML is managed in a hierarchical way, i.e. all child elements of an element with a language setting inherit this setting, unless they supply another value for xml:lang.

Language information can also be provided in RDF/XML, but this is semantically relevant only for untyped literals. For instance, one could write the following:

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-primer">
   <ex:title xml:lang="fr">Initiation à RDF</ex:title>
   <ex:title xml:lang="en">RDF Primer</ex:title>
</rdf:Description>
```

In serializations of RDF other than RDF/XML, language information is supplied by means of the symbols @. In Turtle this might look as follows:

```
<http://www.w3.org/TR/rdf-primer> <http://example.org/title>
   "Initiation à RDF"@fr, "RDF Primer"@en .
```

This syntax again shows that language settings are really part of the data value in RDF. The above example thus describes a graph of two triples with the same subject and predicate. Likewise, in the graphical representation of RDF, the labels of literal nodes are simply extended to include language settings just as in Turtle.

Similar yet distinct

Both language settings and datatypes in RDF are considered to be part of literals, and their absence or presence thus leads to different literals. This might sometimes lead to confusion in the users of RDF, and it may also impose some challenges when merging datasets. Consider, e.g., the following RDF description in Turtle:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
<http://crcpress.com/uri> <http://example.org/Name> "CRC Press" ,
                                                 "CRC Press"@en ,
                                          "CRC Press"^^xsd:string .
```

This example does indeed encode three different triples. Literals with language settings always constitute a pair of a literal value and a language code, and thus can never be the same as any literal without language setting. The untyped literal "CRC Press", according to the RDF specification, represents "itself", i.e. there is no distinction between lexical space and value space. Whether or not the untyped value "CRC Press" is part of the value space of xsd:string of course is not addressed in the RDF specification, since XML Schema datatypes are not part of this standard.

In practice, however, many applications expect the two literals without language settings to be equal, which probably agrees with the intuition of many users.

As mentioned above, language settings are only allowed for untyped literals. The justification for this design of the current RDF standard was that the semantics of typed data values is not dependent on any language. The number 23, e.g., should have the same (mathematical) meaning in any language. This view was also extended to strings: values of type xsd:string therefore are indeed assumed to encode a sequence of characters, not a text of a concrete language. Another possibly unexpected consequence of this decision is that values of type rdf:XMLLiteral do not inherit language settings from their parent elements, even though this would be assumed if the RDF/XML document was considered as XML. RDF thus introduces an exception to the otherwise strictly hierarchical scope of xml:lang in XML.

2.3.3 Many-Valued Relationships

So far, we have represented only very simple binary relationships between resources, thus essentially describing a directed graph. But does such a simple graph structure also allow for the representation of more complex data structures? In this section, we will see how relationships between more than two resources can indeed be encoded in RDF.

Let us first consider an example. The following excerpt from an RDF

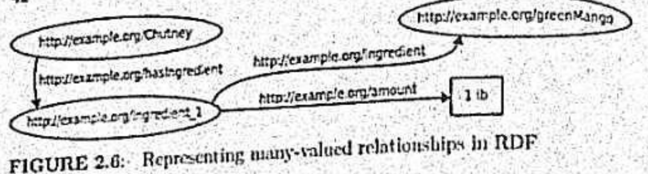

FIGURE 2.6: Representing many-valued relationships in RDF

description formalizes ingredients of a cooking recipe:

```
@prefix ex: <http://example.org/> .
ex:Chutney  ex:hasIngredient  "1lb green mango",
                              "1tsp. Cayenne pepper" .
```

This encoding, however, is not really satisfying, since ingredients and their amounts are modeled as plain strings of text. Thus it is, e.g., not possible to query for all recipes that contain green mango, unless the whole text including the specified amount is queried. It would therefore be more useful to describe ingredients and their amounts in separate resources. Let us attempt the following modeling:

```
@prefix ex: <http://example.org/> .
ex:Chutney  ex:ingredient  ex:greenMango;     ex:amount  "1lb" ;
            ex:ingredient  ex:CayennePepper;  ex:amount  "1tsp." .
```

It is not hard to see that this encoding is even less suitable than our initial approach. While ingredients and amounts are described separately, there is no relationship at all between the individual triples. We could therefore as well be dealing with 1 tsp. of green mango and 1 lb of Cayenne pepper – a rather dangerous ambiguity! An alternative approach could be to model amounts via triples that use ingredients as their subjects. This would obviously clarify the association of amounts and ingredients, e.g., when writing ex:greenMango ex:amount "1lb". Yet this attempt would again yield undesired results (can you see why?[5]).

We are thus dealing with a true three-valued relationship between a recipe, an ingredient, and an amount – one also speaks of ternary (and generally *n-ary*) relations. RDF obviously cannot represent relationships with three or more values directly, but they can be described by introducing so-called *auxiliary nodes* into the graph. Consider the graph in Fig. 2.6. The node ex:Ingredient1 in this example plays the role of an explicit connection between recipe, ingredient, and amount. Further nodes could be introduced for all additional ingredients to link the respective components to each other.

As can be readily seen, this method can generally be used to connect an arbitrary number of objects to a subject. This, however, requires the introduction of several additional URIs. On the one hand, the auxiliary node itself needs an identifier; on the other hand, additional triples with new predicate names are created. Consequently, our example now contains two predicates ex:hasIngredient and ex:ingredient. Our choice of names in this case reflects the fact that the object of ex:ingredient plays a particularly prominent role among the many values in our example relationship. RDF offers a reserved predicate rdf:value that may be used to highlight a particular object of a many-valued relation as a "main" value. Instead of the graph of Fig. 2.6, we may thus choose to write:

```
@prefix ex:  <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
ex:Chutney      ex:hasIngredient  ex:ingredient1 .
ex:ingredient1  rdf:value    ex:greenMango;
                ex:amount    "1lb" .
```

The predicate rdf:value does not have a particular formal semantics. It is merely a hint to applications that a particular value of a many-valued relationship could be considered as its primary value. Most of today's applications, however, do not heed this additional information. Since, moreover, rdf:value does not play well with the ontology language OWL DL that we introduce in Chapter 4, it is often the best choice to use application-specific predicate names instead.

2.3.4 Blank Nodes

As shown in the previous section, modeling many-valued relationships may require the introduction of auxiliary nodes. Such nodes typically do not refer to resources that were meant to be described explicitly in the first place, and rather introduce helper resources with a merely structural function. It is therefore rarely useful to refer to such resources globally by means of a specific URI. In such cases, RDF allows us to introduce nodes without any URI, called *blank nodes* or simply *bnodes*.

An example for an RDF graph with a blank node is given in Fig. 2.7. This graph essentially describes the same structure as the graph in Fig. 2.6. The second RDF document, however, merely states that there is some resource taking the place of the blank node, but without providing a URI for referring to this resource. As the name "blank node" suggests, this feature is only

---

[5] Hint: Try to encode multiple recipes that require the same ingredient but in different amounts.

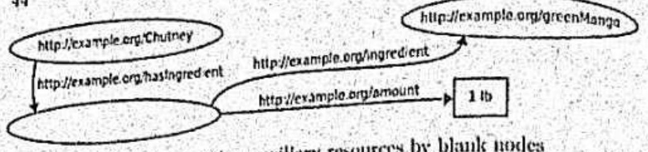

FIGURE 2.7: Representing auxiliary resources by blank nodes

available for subject and objects of RDF triples. Predicates (i.e. edges) must alway be specified by URIs.

Blank nodes cannot be addressed globally by means of URIs, and they do not carry any additional information within RDF graphs. Yet, the syntactic serialization of RDF necessitates referring to particular blank nodes at least in the context of the given document. The reason is that a single blank node may appear as a subject or object in arbitrarily many triples. Therefore, there must be a way for multiple triples to refer to the same blank node. To this end, blank nodes in a document may be denoted by means of (node) IDs. In RDF/XML this is done by using the attribute `rdf:nodeID` instead of `rdf:about`, `rdf:ID` or `rdf:resource`. The RDF/XML serialization for the graph in Fig. 2.7 could thus be as follows:

```
<rdf:Description rdf:about="http://example.org/Chutney">
   <ex:hasIngredient rdf:nodeID="id1" />
</rdf:Description>
<rdf:Description rdf:nodeID="id1">
   <ex:ingredient rdf:resource="http://example.org/greenMango" />
   <ex:amount>1lb</ex:amount>
</rdf:Description>
```

The label `id1` in this example is only relevant for the given document. Within other documents, in contrast, the same id might well refer to different resources. In particular, the semantics of an RDF document is not changed if all occurrences of a given node id are replaced by another id, as long as the latter was not used yet within this document. This reflects the fact that node IDs are only a syntactic tool to serialize blank nodes. If the given usage of a blank node does not actually require the use of this node in multiple positions, it is also allowed to omit the attribute `rdf:nodeID` entirely. This can be particularly useful when nesting descriptions in RDF/XML. The following example shows yet another possibility of introducing blank nodes without providing an id:

```
<rdf:Description rdf:about="http://example.org/Chutney">
  <ex:hasIngredient rdf:parseType="Resource">
    <ex:ingredient rdf:resource="http://example.org/greenMango" />
    <ex:amount>1lb</ex:amount>
  </ex:hasIngredient>
</rdf:Description>
```

The value `Resource` of the attribute `rdf:parseType` in this case leads to the automatic creation of a new blank node which does not have a node id within the given document. We already encountered `rdf:parseType` earlier with the value `Literal`, where a literal node of type `XMLLiteral` was newly created. In general, `rdf:parseType` modifies the way in which parts of the XML document are interpreted, usually leading to the generation of additional triples that have not been specified directly. All uses of `rdf:parseType` – including those discussed further below – can be avoided by serializing the encoded triples directly. Yet, such "syntactic sugar" is often rather useful for enhancing a document's readability.

In Turtle and similar triple-based serializations, blank nodes are encoded by using an underscore instead of a namespace prefix:

```
@prefix ex:  <http://example.org/> .
ex:Chutney    ex:hasIngredient  _:id1 .
_:id1         ex:ingredient  ex:greenMango;   ex:amount  "1lb" .
```

The given node id again is relevant only for the current document. Turtle allows us to abbreviate nested blank nodes in a way that is structurally similar to RDF/XML:

```
@prefix ex:  <http://example.org/> .
ex:Chutney    ex:hasIngredient
            [ ex:ingredient  ex:greenMango;   ex:amount  "1lb" ] .
```

The predicates and objects within square brackets refer to an implicit blank node without an id. The previous Turtle document thus corresponds to the same RDF graph structure as in the earlier examples. As a special case, it is also possible to write [] for a blank node that does not have an explicit id.

2.4 Simple Ontologies in RDF Schema

In the previous sections, we explained how propositions about single resources can be made in RDF. Essentially, three basic kinds of descriptive elements were used for this: we specified *individuals* (e.g., the authors of this textbook, a publisher or a cooking recipe), that in one way or the other were put into *relation* to each other. More casually, we learned that it is possible to assign *types* to literals and resources, thereby stating that they belong to a class of entities sharing certain characteristics (like natural numbers or ordered lists).

When describing new domains of interest, one would usually introduce new terms not only for individuals (like "Sebastian Rudolph" or "Karlsruhe Institute of Technology") and their relations (such as "employed by") but also for types or classes (e.g., "person", "university", "institution"). As pointed out in Section 2.2.6, a repertoire of such terms referring to individuals, relations and classes is usually called a vocabulary.

When introducing and employing such a vocabulary, the user will naturally have a concrete idea about the used terms' meanings. For example, it is intuitively clear that every university has to be an institution or that only persons can be employed by an institution.

From the "perspective" of a computer system, however, all the terms introduced by the user are merely character strings without any prior fixed meaning. Thus, the aforementioned semantic interrelations have to be explicitly communicated to the system in some format in order to enable it to draw conclusions that rely on this kind of human background knowledge.

By virtue of RDF Schema (short RDFS), a further part of the W3C RDF recommendation which we will deal with in the following sections, this kind of background information – so-called *terminological knowledge* or alternatively *schema knowledge* – about the terms used in the vocabulary can be specified.

In the first place, RDFS is nothing but another particular RDF vocabulary. Consequently, every RDFS document is a well-formed RDF document. This ensures that it can be read and processed by all tools that support just RDF, whereby, however, a part of the meaning specifically defined for RDFS (the RDFS semantics) is lost.

RDFS – whose name space `http://www.w3.org/2000/01/rdf-schema#` is usually abbreviated by `rdfs:` – does not introduce a topic-specific vocabulary for particular application domains like, e.g., FOAF does. Rather, the intention of RDFS is to provide generic language constructs by means of which a user-defined vocabulary can be semantically characterized. Moreover, this characterization is done *inside* the document, allowing an RDFS document to – roughly speaking – carry its own semantics. This allows for defining a new vocabulary and (at least partially) specifying its "meaning" in the document without necessitating a modification of the processing software's program logic. Indeed, any software with RDFS support automatically treats any RDFS-defined vocabulary in a semantically correct way.

The capability of specifying this kind of schema knowledge renders RDFS a knowledge representation language or *ontology language* as it provides means for describing a considerable part of the semantic interdependencies which hold in a domain of interest.

Let us dwell for a moment on the term *ontology language*. In Section 1.1 we have already discussed the notion *ontology* and its philosophical origin. On page 2 we said that in computer science, an ontology is a description of knowledge about a domain of interest, the core of which is a machine-processable specification with a formally defined meaning. It is in exactly this sense that RDFS is an ontology language: An RDFS document is a machine-processable specification which describes knowledge about some domain of interest. Furthermore, RDFS documents have a formally defined meaning, given by the formal semantics of RDFS. This formal semantics will be explained in Chapter 3, although the most important aspects of it will become intuitively clear in the following, when we introduce RDFS.

Let us remark that RDFS, despite its usefulness as an ontology language, also has its limitations, and we will explicate this in Section 3.4. Hence, RDFS is sometimes categorized as a representation language for so-called *lightweight* ontologies. Therefore, more sophisticated applications require more expressive representation languages such as OWL which will be discussed in Chapters 4 and 5, yet usually the higher expressivity comes at the expense of speed: the runtime of algorithms for automated inference tends to increase drastically when more expressive formalisms are used.

Hence the question which formalism to use should always be considered depending on the requirements of the addressed task; in many cases an RDFS representation might be sufficient for the intended purposes.[6]

2.4.1 Classes and Instances

Certainly, one basic functionality that any reasonable knowledge specification formalism should provide is the possibility to "type" resources, i.e. to mark them as elements of a certain aggregation. In RDF, this can be done via the predicate `rdf:type`. Generally, the predefined URI `rdf:type` is used to mark resources as *instances* of a *class* (i.e. belonging to that class). In order to clearly separate semantics and syntax, we always use the term "class" to denote a set of resources (being entities of the real world), whereas URIs which represent or refer to a class are called *class names*.

---

[6] This fact has been put into the common phrase "A little semantics goes a long way" coined by James Hendler.

As an example, it would be straightforward to describe this book as a textbook (which means: a member of the class of all textbooks):

```
book:uri   rdf:type   ex:Textbook .
```

This example also illustrates that it is possible (and, depending on the application domain, very reasonable) to introduce new, user-defined class names.

Obviously, there is no syntactic way of distinguishing URIs representing individuals (like book:uri) and class names (such as ex:Textbook). Hence, a single URI does not provide direct information whether it refers to a single object or a class. In fact, such a clear distinction is not always possible, even for some real world terms. Even with human background knowledge, it might be hard to decide whether the URI `http://www.un.org/#URI` denotes an individual single organization or the class of all its member states.

Nevertheless, it might be desirable to enforce some clarification by making a definite modeling decision in the context of an RDFS document. Therefore, RDFS provides the possibility to indicate class names by explicitly "typing" them as classes. In other words: it can be specified that, e.g., the class ex:Textbook belongs to the class of all classes. This "meta-class" is predefined in the RDFS vocabulary and denoted by the URI **rdfs:Class**. As we already know, class membership is expressed via **rdf:type**, hence the following triple characterizes the URI **ex:Textbook** as class name:

```
ex:Textbook   rdf:type   rdfs:Class .
```

On the other hand, the fact that **ex:Textbook** denotes a class is also an implicit but straightforward consequence of using it as object of a typing statement, hence, the preceding triple also follows from the triple

```
book:uri   rdf:type   ex:Textbook .
```

As an additional remark of little practical relevance, note that the class of all classes is obviously itself a class and hence contained in itself as an element. Therefore the proposition encoded by the following triple is always valid:

```
rdfs:Class   rdf:type   rdfs:Class .
```

Besides **rdfs:Class**, there are a few further class names predefined in the RDF and RDFS vocabularies and carrying a fixed meaning:

- **rdfs:Resource** denotes the class of all resources (i.e. for all elements of the considered domain of interest).
- **rdf:Property** refers to the class of all properties, and therefore to all resources that stand for relations.
- **rdf:XMLLiteral** has already been introduced as the only predefined datatype in RDF(S). At the same time, this name denotes the class of all values of this datatype.
- **rdfs:Literal** represents the class of all literal values, which implies that it comprises all datatypes as subclasses.
- The class denoted by **rdfs:Datatype** contains all datatypes as elements, for example, the class of XML literals. Note that this is another example of a class of classes (and hence a subclass of the **rdfs:Class** class).
- The class names **rdf:Bag**, **rdf:Alt**, **rdf:Seq** and **rdfs:Container** are used to declare lists and will be treated in more detail in Section 2.5.1.
- **rdfs:ContainerMembershipProperty**, denoting the class of containedness properties, will be dealt with in Section 2.5.1 as well.
- The class name **rdf:List** is used to indicate the class of all collections. In particular, the empty collection denoted by **rdf:nil** is an element of this class.
- **rdf:Statement** refers to the class of reified triples and will be dealt with in Section 2.5.2.

All those class names also exhibit a common notational convention: URIs representing classes are usually capitalized, whereas names for instances and properties are written in lower case. Note also that the choice for class names is not limited to nouns; it might be reasonable to introduce classes for qualities (expressed by adjectives) as well, e.g., **ex:Organic** for all organic compounds or **ex:Red** for all red things.

Finally, it is important to be aware that class membership is not exclusive: naturally, a resource can belong to several different classes, as illustrated by the following two triples:

```
book:uri   rdf:type   ex:Textbook .
book:uri   rdf:type   ex:WorthReading .
```

2.4.2 Subclasses and Class Hierarchies

Suppose an RDFS document contains one single triple referring to this textbook:

```
book:uri   rdf:type   ex:Textbook .
```

If we now searched for instances of the class of books denoted by ex:Book, the URI book:uri denoting "Foundations of Semantic Web Technologies" would not be among the results. Of course, human background knowledge entails that every textbook is a book and consequently every instance of the ex:Textbook class is also an instance of the ex:Book class. Yet, an automatic system not equipped with this kind of linguistic background knowledge is not able to come up with this conclusion. So what to do?

There would be the option to simply add the following triple to the document, explicitly stating an additional class membership:

```
book:uri   rdf:type   ex:Book .
```

In this case, however, the same problem would occur again and again for any further resource typed as textbook which might be added to the RDFS document. Consequently, for *any* triple occurring in the document and having the form

```
u   rdf:type   ex:Textbook .
```

the according triple

```
u   rdf:type   ex:Book .
```

would have to be explicitly added. Moreover, those steps would have to be repeated for any new information entered into the document. Besides the workload caused by this, it would also lead to an undesirable and unnecessary verbosity of the specification.

Clearly, a much more reasonable and less laborious way would be to just specify (one may think of it as a kind of "macro") that *every* textbook is also a book. This obviously means that the class of all textbooks is comprised by the class of all books, which is alternatively expressed by calling textbook a *subclass* of book or equivalently, calling book a *superclass* of textbook. Indeed, the RDFS vocabulary provides a predefined way to explicitly declare this subclass relationship between two classes, namely, via the predicate rdfs:subClassOf. The fact that any textbook is also a book can hence be succinctly stated by the following triple:

```
ex:Textbook   rdfs:subClassOf   ex:Book .
```

This enables any software that supports the RDFS semantics (see Chapter 3) to identify the individual denoted by book:uri as a book even without it being explicitly typed as such.

It is common and expedient to use subclass statements not only for sporadically declaring such interdependencies, but to model whole *class hierarchies* by exhaustively specifying the generalization-specification order of all classes in the domain of interest. For instance, the classification started in the example above could be extended by stating that book is a subclass of print media and the latter a superclass of journal:

```
ex:Book      rdfs:subClassOf   ex:PrintMedia .
ex:Journal   rdfs:subClassOf   ex:PrintMedia .
```

In accordance with the intuition, the RDFS semantics also implements transitivity of the subclass relationships, i.e. roughly speaking: subclasses of subclasses are subclasses. Therefore, from the triples written down in this section, the following triple – though not explicitly stated – can be deduced:

```
ex:Textbook   rdfs:subClassOf   ex:PrintMedia .
```

Moreover, the subclass relationship is defined to be reflexive, meaning that every class is its own subclass (clearly, the class of all books comprises the class of all books). Thus, once it is known that ex:Book refers to a class, the following triple can be concluded:

```
ex:Book   rdfs:subClassOf   ex:Book .
```

This fact also enables us to model the proposition that two classes contain the same individuals (in other words: they are extensionally equivalent) by establishing a mutual subclass relationship:

```
ex:MorningStar   rdfs:subClassOf   ex:EveningStar .
ex:EveningStar   rdfs:subClassOf   ex:MorningStar .
```

The most popular and elaborated class hierarchies can certainly be found in the area of biology, where – following the classical systematics – living beings are grouped into kingdoms, phyla, classes (as a biological term), orders, families, genus and species. From the RDFS document from Fig. 2.8, we can deduce that the individual Sebastian Rudolph is not only a human but also a mammal. Likewise, by the deducible membership in the class of primates, he logically makes a monkey of himself.

Documents containing only class hierarchies are usually referred to as *taxonomies* and subclass-superclass dependencies are often called taxonomic relations. Certainly, one reason why this kind of knowledge modelling is so intuitive is its closeness to human conceptual thinking. In most cases, a class hierarchy with sub- and superclasses can be conceived as a conceptual hierarchy with subordinate and superordinate concepts or, using common linguistic terminology: hyponyms and hypernyms.

2.4.3 Properties

A special role is played by those URIs used in triples in the place of the predicate. Examples from previous sections include `ex:hasIngredient`, `ex:publishedBy` and `rdf:type`. Although those terms are represented by URIs and hence denote resources, it remains a bit unclear how to concretely interpret them. A (or *the*) "publishedBy" can hardly be physically encountered in everyday life; therefore it seems inappropriate to consider it as class or individual. In the end, these "predicate URIs" describe relations between "proper" resources or individuals (referenced by subject and object in an RDF triple). As the technical term for such relations, we will use *property*.

In mathematics, a relation is commonly represented as the set of the pairs interlinked by that relation. According to that, the meaning of the URI `ex:isMarriedTo` would be just the set of all married couples. In this respect, properties resemble classes more than single individuals.

For expressing that a URI refers to a property (or relation), the RDF vocabulary provides the class name `rdf:Property` which by definition denotes the class of all properties. The fact that `ex:publishedBy` refers to a property can now again be stated by assigning the corresponding type:

```
ex:publishedBy   rdf:type   rdf:Property .
```

Note that `rdf:Property` itself denotes a class and not a property. It just contains properties as instances. Finally, in addition to explicitly being typed as such, a URI can also be identified as property name by its occurrence as predicate of a triple. Therefore, the RDFS semantics ensures that the above triple is also a consequence of any triple like

```
<?xml version="1.0" encoding="utf-8"?> <!DOCTYPE rdf:RDF[
      <!ENTITY ex 'http://example.org/'>
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:ex="http://www.semanticweb-grundlagen.de/Beispiele#">

  <rdfs:Class rdf:about="&ex;Animalia">
      <rdfs:label xml:lang="en">animals</rdfs:label>
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;Chordata">
      <rdfs:label xml:lang="en">chordates</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Animalia" />
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;Mammalia">
      <rdfs:label xml:lang="en">mammals</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Chordata" />
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;Primates">
      <rdfs:label xml:lang="en">primates</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Mammalia" />
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;Hominidae">
      <rdfs:label xml:lang="en">great apes</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Primates" />
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;Homo">
      <rdfs:label xml:lang="en">humans</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Hominidae" />
  </rdfs:Class>

  <rdfs:Class rdf:about="&ex;HomoSapiens">
      <rdfs:label xml:lang="en">modern humans</rdfs:label>
      <rdfs:subClassOf rdfs:resource="&ex;Homo" />
  </rdfs:Class>

  <ex:HomoSapiens rdf:about="&ex;SebastianRudolph" />
</rdf:RDF>
```

FIGURE 2.8: Example for class hierarchies in RDFS

```
book:uri  ex:publishedBy  crc:uri .
```

2.4.4 Subproperties and Property Hierarchies

In the previous section, we argued that properties can be conceived as sets of individual pairs and hence exhibit some similarity to classes. Thus, one might wonder whether modeling constructs in analogy to subclass relationships would also make sense for properties. This is indeed the case: RDFS allows for the specification of *subproperties*. For example, the property denoted by the URI `ex:isHappilyMarriedTo` is certainly a subproperty of the one `ex:isMarriedTo` refers to, as the happily married couples form a (most probably even proper) subset of all married couples. This connection can be declared as follows:

```
ex:isHappilyMarriedTo  rdf:subPropertyOf  ex:isMarriedTo.
```

Again, situations where this kind of information is of advantage are easy to imagine. For example by virtue of the above mentioned triple, the RDFS semantics allows us to deduce from the triple

```
ex:markus  ex:isHappilyMarriedTo  ex:anja .
```

that also the following triple must be valid:

```
ex:markus  ex:isMarriedTo  ex:anja .
```

Consequently, one single subproperty statement suffices to enable an RDFS-compliant information system to automatically recognize all pairs recorded as "happily married" additionally as "married". Note that this way, also properties can be arranged in complex hierarchies, although this is not as commonly done as for classes.

2.4.5 Property Restrictions

Frequently, the information that two entities are interconnected by a certain property allows us to draw further conclusions about the entities themselves. In particular, one might infer class memberships. For instance, the statement that one entity is married to another implies that both involved entities are persons.

Now it is not hard to see that the predicate's implicit additional information on subject and object can be expressed via class memberships: Whenever a triple of the form

```
a  ex:isMarriedTo  b .
```

occurs, one wants to assert, for example, that both following triples are valid as well:

```
a  rdf:type  ex:Person .
b  rdf:type  ex:Person .
```

As in the previously discussed cases, explicitly adding all those class membership statements to the RDF document would be rather cumbersome and would require us to repeat the process whenever new information is added to the document. Again, it seems desirable to have a "macro" or "template"-like mechanism which is entered just once and ensures the class memberships imposed by the predicates.

Fortunately the RDFS vocabulary provides means to do exactly this: one may provide information about a property's domain via `rdfs:domain` and its range via `rdfs:range`. The first kind of expression allows us to classify subjects, the second one to type objects that co-occur with a certain predicate in an RDF triple. The above mentioned class memberships imposed by the predicate `ex:isMarriedTo` can now be encoded by the following triples:

```
ex:isMarriedTo  rdfs:domain  ex:Person .
ex:isMarriedTo  rdfs:range   ex:Person .
```

In the same vein, literal values in the place of the object can be characterized by stipulating datatypes (e.g., in order to specify that a person's age should be a nonnegative number):

```
ex:hasAge  rdfs:range  xsd:nonNegativeInteger .
```

Obviously, domain and range restrictions constitute the "semantic link" between classes and properties because they provide the only way of describing the desired terminological interdependencies between those distinct kinds of ontology elements.

We would also like to address a frequent potential misconception. Suppose an RDFS document contains the triples:

```
ex:authorOf   rdfs:range   ex:Textbook .
ex:authorOf   rdfs:range   ex:Storybook .
```

According to the RDFS semantics, this expresses that every resource in the range of an authorship relation is *both* a textbook *and* a storybook; it does not mean that somebody may be author of a textbook *or* a storybook. The same holds for **rdfs:range** statements. So, every declared property restriction globally affects all occurrences of this property; hence one should be careful when restricting properties and make sure that always a sufficiently general class (i.e. one containing all possible resources that might occur in the subject resp. object position) is used.

Some further consideration is needed to prevent a confusion arising rather frequently. Consider the following RDF knowledge base:

```
ex:isMarriedTo    rdfs:domain   ex:Person .
ex:isMarriedTo    rdfs:range    ex:Person .
ex:instituteAIFB  rdf:type      ex:Institution .
```

Now assume the following triple was to be added to it:

```
ex:pascal         ex:isMarriedTo   ex:instituteAIFB .
```

Omitting deeper contemplations on a possible metaphorical truth of this statement, this example reveals a potential modeling flaw. A certain similarity to the "type mismatch" problem in programming shows up here. One might now expect that this kind of statement is automatically rejected by a system containing the above range statement (as a database system might reject changes if certain conditions are violated). However, RDF range and domain statements do not carry this kind of constraint semantics. The only indicator that something might be wrong is that by adding that triple to the knowledge base, counterintuitively, **ex:instituteAIFB** is additionally typed as a person, i.e. the triple

```
ex:instituteAIFB  rdf:type      ex:Person .
```

is a consequence of the above triples.

2.4.6 Additional Information in RDFS

In many cases it is desirable to endow an RDFS document with additional information which has no semantical impact but increases the understandability for human users. One might argue that – at least when using XML syntax – XML-style comments could just be used for this purpose. However, this would mean relinquishing the basic RDF(S) rationale to represent all knowledge as a graph, including all additional, comment-like information. Therefore, RDFS provides the possibility to embed additional information into the graph, thereby making it "semantically accessible." To this end, RDFS provides a predefined set of property names by means of which additional information can be encoded without relinquishing the basic idea to represent all knowledge as a graph. Thereby, all supplementary descriptions can be read and represented by any RDF-compliant software. We tacitly used **rdfs:label** in Section 2.4.2. Generally, this property-URI serves the purpose of accompanying a resource (which might be an individual but also a class or a property) with a name which is more handy than its URI. This sort of information can be used by tools visualizing the RDF document as a graph, where verbose URIs might impede readability. The object in a triple containing **rdfs:label** as predicate has to be a literal, a syntactic restriction which can be expressed within RDFS by the following triple:

```
rdfs:label   rdfs:range   rdfs:Literal .
```

rdfs:comment is used for assigning comprehensive human-readable comments to resources. Especially if new class or property terms are introduced, it is reasonable to write down their intended meaning in natural language. This facilitates the correct and consistent usage of the new terms by other users who might have a look at the documentation if in doubt. **rdfs:comment** also requires a literal as object.

By means of the expressions **rdfs:seeAlso** and **rdfs:isDefinedBy**, it is possible to link to resources that provide further information about the subject resource. This might be URLs of websites or URIs referring to print media. In particular, **rdfs:isDefinedBy** is used to state that the subject resource is (in some not further specified way) defined by the object resource. According to the RDFS semantics **rdfs:isDefinedBy** is stipulated to be a subproperty of **rdfs:seeAlso**.

As an example of the constructs introduced in this section consider the extended passage, given in Fig. 2.9, of the RDFS document from Fig. 2.8.

2.5 Encoding of Special Datastructures

You now know all the central ideas and notions of RDF(S) and the modeling features used most frequently. From here you might proceed directly to Chapter 3 to see how the semantics of RDF(S) – that we tried to intuitively convey

```
 :
 xmlns:wikipedia="http://en.wikipedia.org/wiki/"
 :
<rdfs:Class rdf:about="&ex;Primates">
  <rdfs:label xml:lang="en">primates</rdfs:label>
  <rdfs:comment>
   Order of mammals. Primates are characterized by an
   advanced brain. They mostly populate the tropical
   earth regions. The term 'Primates' was coined by
   Carl von Linné.
  </rdfs:comment>
  <rdfs:seeAlso rdf:resource="&wikipedia;Primates" />
  <rdfs:subClassOf rdfs:resource="&ex;Mammalia" />
</rdfs:Class>
```

FIGURE 2.9: Additional information in RDFS documents

in this chapter – is defined formally and how automated RDF(S) inferencing can be realized. Or you might go on reading if interested in what additional possibilities RDF(S) has in stock for modeling more complex datastructures: lists and nested propositions.

2.5.1 Lists in RDF

Many-valued relationships, as introduced in Section 2.3.3, are used for obtaining a more structured representation of a single object, separating its single components (ingredient and amount) in distinct triples. The structure of the triples referring to the auxiliary node thus was specific to the example at hand. In many other cases, in contrast, one simply wants to relate a subject to a set of objects that play similar roles, e.g., when describing the relationship of a book to the set of its authors. To address such use cases, RDF offers a number of specific constructs that can be used to describe structures that resemble lists. This can be achieved in two fundamentally different ways: with *containers* (open lists) or with *collections* (closed lists).

It is important to note that all of the following additional expressive features are only abbreviations for RDF graphs that could as well be encoded by specifying individual triples. As in the case of `rdf:value` above, lists in RDF do not have a specific formal semantics that distinguishes such structures from other RDF graphs.

2.5.1.1 RDF Container

RDF containers allow us to encode RDF graphs that resemble the one in Fig. 2.7 in a unified way. We have already seen that the introduction of

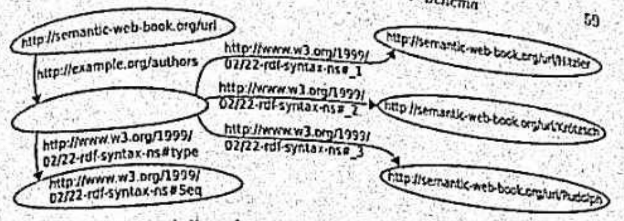

FIGURE 2.10: A list of type `rdf:Seq` in RDF

blank nodes can be abbreviated in various serializations of RDF. Containers introduce two additional changes:

- The triples of a list are denoted by standard identifiers instead of using specific URIs such as `ex:amount` in Fig. 2.7.
- It is possible to assign a *class* to a list, hinting at the desired (informal) interpretation.

An example of a corresponding list is the following specification of this book's authors:

```
<rdf:Description rdf:about="http://semantic-web-book/uri">
 <ex:authors>
  <rdf:Seq>
   <rdf:li rdf:resource="http://semantic-web-book.org/uri/Hitzler" />
   <rdf:li rdf:resource="http://semantic-web-book.org/uri/Krötzsch" />
   <rdf:li rdf:resource="http://semantic-web-book.org/uri/Rudolph" />
  </rdf:Seq>
 </ex:authors>
</rdf:Description>
```

We would normally expect an element of type `rdf:Description` instead of `rdf:Seq` in this XML serialization. As explained in Section 2.3.4, this syntax would then also conveniently introduce a blank node. Something quite similar happens in the above case, as can be seen when considering the resulting RDF graph in Fig. 2.10. This graph, however, displays some additional features that require further explanation.

First, we notice that Fig. 2.10 contains one more node than we may have expected. While book, authors, and the required blank auxiliary node are present, we see that, additionally, the blank node is typed as an instance of the class `rdf:Seq`. This typing mechanism is used to specify the intended usage of a given list. RDF provides the following types of lists:

- **rdf:Seq**: The container is intended to represent an ordered list, i.e. the order of objects is relevant.
- **rdf:Bag**: The container is intended to represent an unordered set, i.e. the order of objects is not relevant, even though it is inherently available in the RDF encoding.
- **rdf:Alt**: The container is intended to represent a set of alternatives. Even though the RDF document specifies multiple objects, only one of them is usually required in a particular application.

These class names can be used instead of `rdf:Seq` in the above example, but they do not otherwise affect the encoded RDF graph structure. Only when displaying or interpreting a list in a given application, this additional informal information may be taken into account.

A second aspect that can be noted in Fig. 2.10 is that the predicates of the individual list triples are not labeled like the corresponding elements in XML. Indeed, the RDF/XML serialization uses elements of type `rdf:li` for all elements. The graphical representation, in contrast, uses numbered predicates `rdf:_1` to `rdf:_3`. The XML syntax in this case is merely a syntactic simplification for encoding an RDF graph that uses predicates of the form `rdf:_n`. This encoding also applies if the list is of type `rdf:Bag` or `rdf:Alt`, even though the exact order may not be practically important in these cases. This method can be used to encode arbitrarily long lists – the RDF specification defines predicates of the form `rdf:_n` for any natural number *n*.

As explained above, the RDF/XML syntax for containers is merely a syntactic abbreviation that can also be avoided by serializing the corresponding RDF graph directly. Since blank nodes can be abbreviated in a simple way, Turtle does not provide a specific syntax for RDF containers. One simply specifies the according triples individually when denoting containers in Turtle.

2.5.1.2 Containers in RDFS

By introducing new predefined names, RDFS further extends the options for modeling lists described in the previous section. The URI `rdfs:Container` denotes the superclass of the three RDF container classes `rdf:Bag`, `rdf:Seq` and `rdf:Alt`, allowing us to mark a resource as list without specifying the precise type.

The URI `rdfs:ContainerMembershipProperty` is used in order to characterize properties, i.e. it refers to a class the instances of which are not individuals in the strict sense (as a person or a website), but themselves properties. The only class having properties as members that we have dealt with so far is the class of *all* properties denoted by `rdf:Property`. Consequently the following triple holds:

```
rdfs:ContainerMembershipProperty   rdfs:subClassOf   rdf:Property .
```

Now, what is the characteristic commonality of the properties contained in the class denoted by `rdfs:ContainerMembershipProperty`? All these properties encode the containedness of one resource in the other. Examples of such properties are those expressing containedness in a list: `rdf:_1`, `rdf:_2`, etc. Although this new class might seem somewhat abstract and of disputable use, there are possible application scenarios. For instance, a user might want to define a new type of list or container (as for cooking recipes) together with a specific containedness property (say, `ex:hasIngredient`). By typing this property with `rdfs:ContainerMembershipProperty`, the user makes this intended meaning explicit:

```
ex:hasIngredient   rdf:type   rdfs:ContainerMembershipProperty .
```

When using just RDF, finding out whether the resource `ex:shakespeare` is contained in the list `book:uri/Authors` would require asking for the validity of infinitely many triples:

```
book:uri/Authors   rdf:_1   ex:shakespeare .
book:uri/Authors   rdf:_2   ex:shakespeare .
book:uri/Authors   rdf:_3   ex:shakespeare .
...
```

RDFS provides the property name `rdfs:member` that denotes a property which is a superproperty of all the distinct containedness properties. As a practical consequence, the above mentioned problem can now be solved by querying for the validity of just one triple:

```
book:uri/Authors   rdfs:member   ex:shakespeare .
```

Yet, there is even more to it: according to the RDFS semantics, every instance of the `rdfs:ContainerMembershipProperty` class is a subproperty of the `rdfs:member` property. Now, let's come back to the aforementioned case of the self-defined container type. Even if the user now states

```
ex:cookie   ex:hasIngredient   ex:peanut .
```

using his proprietary property, the characterization of `ex:hasIngredient` as a containedness property enables us to deduce the validity of the following triple:

```
ex:cookie   rdfs:member   ex:peanut .
```

2.5.1.3 RDF Collections

The representation of lists as RDF containers is based on auxiliary predicates `rdf:_1`, `rdf:_2`, `rdf:_3`, etc. The resulting sequences of objects thus can always be extended by adding further triples, but it is not possible to express that a given list is complete and closed. RDF thus introduces so-called *collections* as a means for representing closed lists.

Like containers, collections are not introducing expressivity beyond what can already be stated by RDF triples, but they can allow for more concise RDF serializations. Let us first consider the following RDF/XML fragment:

```
<rdf:Description rdf:about="http://semantic-web-book/uri">
  <ex:authors rdf:parseType="Collection">
     <rdf:Description
          rdf:about="http://semantic-web-book.org/uri/Hitzler" />
     <rdf:Description
          rdf:about="http://semantic-web-book.org/uri/Krötzsch" />
     <rdf:Description
          rdf:about="http://semantic-web-book.org/uri/Rudolph" />
  </ex:authors>
</rdf:Description>
```

This syntactic description of a collection strongly resembles the explicit representation of a single triple, but it contains three objects instead of the usually unique `rdf:Description`. Furthermore, we encounter once more the attribute `rdf:parseType`, this time with the value `Collection`.

The corresponding RDF graph is the *linked list* that is shown in Fig. 2.11. We immediately see that the graph structure differs significantly from the containers introduced in the previous section. The underlying principle of this representation is that any non-empty list can be split into two components: a list head (the first element of the list) and a list rest. The rest of the list can again be decomposed in this way, or it is the empty list without further elements.

We can thus uniquely describe the closed list by specifying its head and rest. As can be seen in Fig. 2.11, the RDF predicates used for this purpose are `rdf:first` and `rdf:rest`. Since any list can be completely described in this way, the URIs of the individual list nodes are not significant. Consequently, all non-empty (partial) lists in Fig. 2.11 are represented by blank nodes. The only exception is the empty list, which cannot be decomposed further. It is represented in RDF by the URI `rdf:nil`.

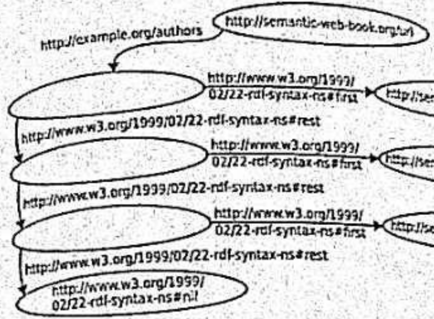

FIGURE 2.11: A collection in RDF

The empty list concludes the linked list, and indicates that no further elements follow. We may thus indeed speak of a closed list: By adding additional triples, one could merely produce RDF graphs that are not proper encodings of a collection, but one cannot add additional elements to the list.

Although collections could again be encoded in individual triples, Turtle also provides a more convenient notation for such lists using parentheses. A corresponding Turtle serialization of the above example could be as follows:

```
@prefix book:  <http://semantic-web-book.org/> .
book:uri   <http://example.org/authors>
     ( book:uri/Hitzler  book:uri/Krötzsch  book:uri/Rudolph ) .
```

2.5.2 Propositions About Propositions: Reification

Much more frequently than we are aware of, we make propositions referring to other propositions. As an example, consider the sentence: "The detective supposes that the butler killed the gardener." One naive attempt to model this situation might yield:

```
ex:detective   ex:supposes   "The butler killed the gardener." .
```

One of the problems arising from this way of modeling would be that the proposition in question – expressed by a literal – cannot be arbitrarily referenced in other triples (due to the fact that literals are only allowed to occur as triple objects). Hence it seems more sensible to use a URI for the proposition leading to something like:

```
ex:detective   ex:supposes   ex:theButlerKilledTheGardener .
```

Yet, this approach leaves us with the problem that the subordinate clause of our sentence is compressed into one URI and hence lacks structural transparency. Of course it is easy to model just the second part of the sentence as a separate triple:

```
ex:butler   ex:killed   ex:gardener .
```

Actually, a kind of "nested" representation, where the object of a triple is a triple on its own, would arguably best fit our needs. However, this would require a substantial extension of the RDF syntax.

One alternative option, called *reification*, draws its basic idea from the representation of many-valued relations as discussed in Section 2.3.3: an auxiliary node is introduced for the triple about which a proposition is to be made. This node is used as a "handle" to refer to the whole statement. Access to the inner structure of the represented triple is enabled by connecting the auxiliary node via the property-URIs `rdf:subject`, `rdf:predicate` and `rdf:object` with the respective triple constituents. The corresponding triple is then called *reified* ("thing-made", from lat. *res* thing and *facere* to make). Using this method, our above sentence could be described by the following four triples:

```
ex:detective  ex:supposes    ex:theory .
ex:theory     rdf:subject    ex:butler .
ex:theory     rdf:predicate  ex:hasKilled .
ex:theory     rdf:object     ex:gardener .
```

It is important to be aware that writing down a reified triple does not mean asserting its actual validity. In particular the previous specification does not allow us to conclude the triple:

```
ex:butler   ex:hasKilled   ex:gardener .
```

Note that this makes sense, as the detective's theory might turn out to be false.

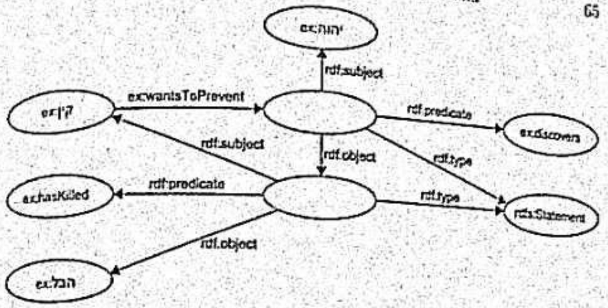

FIGURE 2.12: Example of multiple reification in graph representation

In addition to the reification properties already provided by the RDF vocabulary, RDFS contains the class name `rdf:Statement` that can be used to mark the "central" node of a reified triple:

```
ex:theory   rdf:type   rdf:Statement .
```

If the reified proposition is to be referenced only locally, it can be represented by a blank node. Note also that this way of modeling also allows for multiply nested propositions. An example of both modeling aspects is depicted in Fig. 2.12. With the knowledge acquired in this section you are certainly capable of decoding its content.[7]

2.6 An Example

In order to illustrate the essential modeling capabilities of RDFS, we give a small ontology as an example. For the sake of simplicity, we omit literals and datatypes. Suppose that an RDF document contains the following triples:

---

[7] Hint: It's also some sort of detective story; see Genesis 4:1-16 or Qur'an at 5:26-32 or Moses 5.16-41.

```
ex:vegetableThaiCurry  ex:thaiDishBasedOn   ex:coconutMilk .
ex:sebastian           rdf:type             ex:AllergicToNuts .
ex:sebastian           ex:eats              ex:vegetableThaiCurry .

ex:AllergicToNuts      rdfs:subClassOf      ex:Pitiable .
ex:thaiDishBasedOn     rdfs:domain          ex:Thai .
ex:thaiDishBasedOn     rdfs:range           ex:Nutty .
ex:thaiDishBasedOn     rdfs:subPropertyOf   ex:hasIngredient .
ex:hasIngredient       rdf:type    rdfs:ContainerMembershipProperty.
```

This RDFS specification models the existence of "vegetable thai curry", a Thai dish based on coconut milk.[8] Moreover we learn about a resource "Sebastian" belonging to the class of individuals allergic to nuts. The third triple states that Sebastian eats the vegetable Thai curry. These statements constitute the so-called *assertional knowledge* making propositions about the concrete entities of our domain of interest.

As *terminological knowledge*, our tiny ontology expresses that the class of nut-allergic individuals is a subclass of the class of pitiable things, that any Thai dish (based on something) belongs to the class of Thai things, and (reflecting the personal experience of the afflicted author) that any Thai dish is based only on ingredients belonging to the class of nutty things. Finally, we learn that whenever a (Thai) dish is based on something it also contains that "something" and that "having something as ingredient" constitutes a containedness property. Figure 2.13 shows the same ontology depicted as a graph and once more illustrates the distinction between terminological (or schema) knowledge and assertional (also: factual) knowledge. From knowledge specified in this way, it is now possible to derive implicit knowledge. In the next chapter, we provide the theoretical foundations for this and give an example showing how to automatically draw conclusions from the ontology introduced here.

---

[8] The usage of the property ex:thaiDishBasedOn is rather questionable from the perspective of a good modeling practice, as arguably too much information is squeezed into one property which thereby becomes overly specific. However, we used it for the sake of a small yet informative example. Moreover, we employed it to circumvent a modelling weakness of RDF: if the example were paraphrased using a ex:ThaiDish class and a ex:basedOn property, we would no longer be able to express the proposition "Everything a Thai dish is based on is nutty".

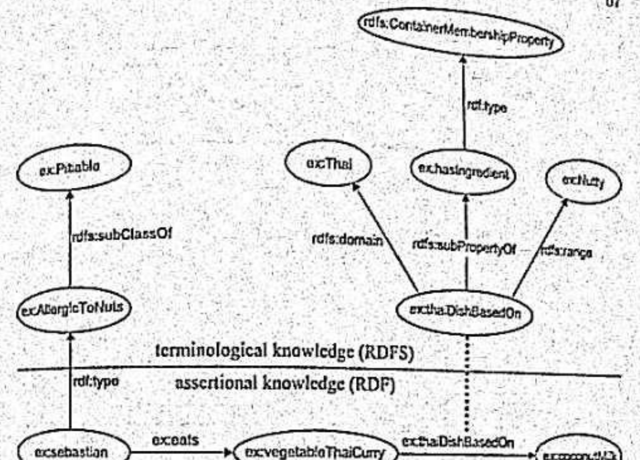

FIGURE 2.13: Graph representation of a simple RDFS ontology

2.7 Summary

In this chapter, we have introduced the description language RDF and its extension RDFS. Both rely on a data model of graph structures consisting of basic elements called triples, which are also used for encoding more complex data structures like lists. URIs admit the unique identification of nodes and edges in those triples. While RDF essentially serves the purpose of making propositions about the relationships of singular objects (individuals), RDFS provides means for specifying terminological knowledge in the form of class and property hierarchies and their semantic interdependencies.

2.7.1 Overview of RDF(S) Language Constructs

RDF(S) classes

```
rdfs:Class      rdf:Property
rdfs:Resource   rdfs:Literal
rdfs:Datatype   rdf:XMLLiteral
```