Chapter 2

The Semantic Web and the RDF language

The present chapter introduces the Semantic Web and its philosophy. This involves two main ideas. The first is to associate meta-information with Internet-based resources. The second is to reason about this type of information. We show how these two ideas can help in solving the problems mentioned in the previous chapter.

Having introduced the main concepts we continue the chapter by describing technologies that can be used for representing meta-information in a uniform way. First we introduce the XML language, which forms the basis of the Semantic Web as a standard information exchange format. Then we describe the RDF language; this has an XML notation as well as other representations and can be used to associate meta-information to an arbitrary resource. By doing this we can extend web contents with computer-processable semantics.

Subsequently, we introduce the RDF schema language, which provides the background knowledge that is essential to do reasoning on meta-information. We discuss the similarities and differences between RDF schemas and traditional object-oriented modelling paradigms.

We conclude the chapter by presenting several applications that directly or indirectly use RDF descriptions during their operation.

2.1. Introduction

The Semantic Web approach was originated by Tim Berners-Lee, the father of the World Wide Web and related technologies (URI, HTTP, HTML etc.). The approach is based on two fundamental ideas.

The first idea is to associate *meta-information* with Internet-based resources. Metadata are pieces of information about other data.[1] Examples of metadata are the title of a book, the creator and creation date of a homepage, the type and size of a file and the fact that a lion, a chimpanzee and a banana can be seen on a particular picture. For these examples, the "real data" are the specific bit-streams representing the book, the HTML code of the homepage, the file and the picture. The distinction between data and metadata is often not very sharp; something which acts like data in one situation becomes metadata in another, and vice versa.

[1] The concept of metadata comes from library science.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

The Semantic Web approach interprets the concept of Internet-based resources rather broadly and asserts that it is possible to associate meta-information with practically anything which is uniquely identifiable. Such objects are, for example, a homepage or an arbitrary part of it, a picture, a video clip, a file or a hardware device. But, in a broader sense, a coffee mug or a pair of gloves are also such resources, granted that unique identifiers are assigned to them.

Besides the ability to associate meta-information with resources, the other fundamental idea of the Semantic Web is the ability to reason about the meta-information. For example, one must be able to figure out in some way that the picture with the chimpanzee on it shows *animals*, even though the meta-information mentions only a *chimpanzee* and a *lion*. No-one said that the picture displays an *animal*, let alone several *animals*.

2.1.1. New ideas

As a matter of fact, the idea of the Semantic Web is not entirely new. We can already find meta-information on the web, given both explicitly and implicitly, which helps us to exploit the semantic contents of the Internet. We mentioned in Section 1.4 that the information associated with individual web pages by search engines, such as their relevance and the text around the anchors, as well as the location, size and frequency of expressions gathered from the pages, can all be considered as indirectly given metadata.

We can also find examples of explicit metadata, supplied intentionally by the page creator. For example, we may provide certain meta-information about a web page using the `META` HTML-element in the head of the page. The two most important and most frequently used attributes of this element are `description` and `keywords`. The former is used to give a short summary of the page contents, which is displayed by the search engines in the search results page. This is definitely meta-information and, even though it is free text, its contents are meaningful for human beings and it can be fairly useful in text searches. Furthermore, with the `keywords` attribute we can specify (as the name suggests) keywords that are representative of the content. Below we reproduce some `META` elements from the homepage of W3C:

```
<meta name="keywords" content="W3C, World Wide Web, Web.../>
<meta name="description"
      content="The World Wide Web Consortium (W3C) is an
      international consortium where Member .../>
```

For more on the `META` elements in HTML documents and their support in various search engines, see the HTML specification [14].

Unfortunately, the use of the `META` element is very limited. First, the specification defines only a handful of attributes and does not allow the introduction of new attributes. Second, the metadata given in a `META` element is often not expressive enough. For example, we can specify – using the `keywords` attribute – that our homepage has something to do with Hungary, but there is no way to tell exactly what the relation is between the two. Does it contain pictures of the Hungarian landscape or analytical essays about the economical state of the country? Furthermore, we can assign meta-information only to an entire page not to arbitrary parts of it, e.g. we can specify the creator of the page but we cannot specify the photographer of a particular picture or the programmer who wrote the source code snippet embedded into the page.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

It is important to note that meta-information is often used in cases which have little to do with the Internet. Examples include the `id3v1` and `id3v2` tags stored in MP3 files, information attached to various image formats, such as the EXIF tags in JPEG files or the author, last modification date, notes etc. information in Word documents. As a further example, think of any file system, which uses (implicitly or explicitly) attached meta-information to store access rights and times of individual files. The important idea in the Semantic Web approach is that these metadata are associated with arbitrary resources in a *uniform* and *structured* manner. The goal is that we should be able to attach meta-information to a Word document just as we would do it in the case of an MP3 file or a web page. At the present time this is clearly not possible, since the exact way in which the meta-information is stored is dependent on the format of the data itself, i.e. there is some binary format in the case of Word documents, free text inside a `META` element for web pages etc.

Exercise 2.1: Add `META` elements to your homepage (or if you do not have a homepage, use the HTML page in Figure 1.1) describing the content and keywords of it. Try this page in a browser to see what happens.

2.1.2. The usefulness of metadata

Using meta-information provides a generic means for associating meaning with various resources. If we are able to describe the properties of a resource in a format processable by and comprehensible to machines then we are making a huge step towards intelligent search. By using metadata, the deep web suddenly becomes searchable: we gain access to information stored in databases and flash-based web pages, images are returned as search results whose content it would be impossible to determine with automated tools if there were no metadata, and so on.

For any of these, all we need is that the databases provide metadata about themselves ("this database contains information on the works of Hermann Hesse", "this database stores specifications of Sony digital cameras"). Similarly, we should be able to figure out that a web page with a flash animation in fact presents information about the movie *Terminator 3*. The detail level of the metadata can vary, but even such generic, high-level descriptions as the above can help the search engines in, for example, forwarding the user to the login page of the appropriate web database, when they entered "Hermann Hesse" as the search query.

In fact, metadata can help in all situations where the real contents of a resource – such as a homepage – are not obvious to the web crawler from the source of the page itself. Imagine that our page contains pictures of the Amazon river and its various bridges but that the names of the files do not imply their contents and there is practically no textual information on the page. In such a case, if the `keywords` element contains the words "Amazon", "river" and "bridge", a search engine will probably return the page for a search on the Amazon river. However, without this piece of meta-information it would be almost impossible to automatically index the page as one that has something to do with the Amazon river. It would probably present difficulties even for a human being, to figure out, just by looking at the pictures, exactly which river they depict.

One of the biggest problems when providing meta-information is how to achieve a uniform way to do this. In the following section we present the Extensible Markup Language

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

(XML) language, a standard data exchange format which amply answers the requirements for a uniform data storage mechanism. This is followed by the introduction of the Resource Description Framework (RDF) language, the core language of the Semantic Web idea, which can be used to associate metadata with arbitrary resources.

2.2. The XML language

In order to become familiar with the XML syntax of the RDF standard, it is necessary to know XML itself at a basic level. It is especially important to know about the concept of XML namespaces. Extensible Markup Language, like RDF, is a recommendation of the World Wide Web Consortium (W3C) [21]. The primary purpose of XML is to describe data and their structure in a computer-processable format and to provide a standard means for inter-computer data exchange. The goal is to make the Internet suitable for machine-to-machine communication.

To better grasp the role of XML, think of it as a multi-layer model. In the lowest layer we ensure that textual data of an arbitrary natural language can be *represented* on the web; this is usually achieved by using Unicode encoding. Above this there is XML, which gives a syntactic structure to data, thus making them machine *processable*. Higher layers, such as RDF, provide various semantics to the data, making information on the web *comprehensible* to computers.

2.2.1. XML as a metalanguage

Extensible Markup Language is – as its name suggests – a general purpose, markup language for creating other, special purpose, markup languages.

The definition of a new language using XML is called an *application* of XML. A particular XML document is an *instance* of such a language or, using the terminology of formal languages, a *sentence* of a language.

In fact, XML is merely a simple subset of a very complex and generic description language which had existed long before the Internet became widely used. This language is *Standard Generalized Markup Language* (SGML), ISO Standard 8879:1986. There are numerous applications of SGML; it is good to know that one of these is none other than HTML. This is why there are so many syntactic similarities between XML and HTML documents.

In the following, first we will see how XML documents are constructed and what syntactic elements can be used in them. Then we show how one can define a new XML language using the so-called XML schemas. Finally we will talk about how XML documents can be presented and displayed.

2.2.2. Basic syntax; XML elements

An XML document is a text file that is designed to store data in a *structured manner*. It is very important to understand that an XML document does not *do* anything. Like any other text document, all it does is store information, nothing more. It is the task of the processing application to decide what to do with the data stored in the XML document. We can store in XML format our existing Word documents, emails, notes etc. Knowing this, we may consider XML merely as a means of storage that is aimed at being a data exchange format

```
<?xml version="1.0" encoding="iso-8859-1"?>
<message>
   <from>Little Red Riding Hood</from>
   <to>Granny</to>
   <body>I'll visit you this afternoon!</body>
   <ps>I'll bring some cookies</ps>
   <ps>they say there's a wolf in the forest</ps>
</message>
```

Figure 2.1. A message from Little Red Riding Hood to Granny in XML format.

for systems wishing to cooperate, thereby providing the foundations of machine-to-machine communication.

The most important parts of an XML document are *elements* and *attributes*. An XML element consists of three parts: an opening tag, the data itself and a closing tag. For the sake of simplicity, if it will not lead to confusion we often refer to an element by its opening tag. HTML uses similar concepts to XML but, as we said, this is not a coincidence. An HTML source file also uses elements and attributes to structure our data, but the set of available elements is closed, their meaning having been defined in advance. XML, however, does not define elements like those in HTML, such as `<HEAD>`, `<BODY>` and `<IMG>`; this is the responsibility of the creator of the XML document, the choice of the appropriate set of elements being a design issue. It means that there are no "reserved words" in XML: that is to say, the name of an element can be an arbitrary character sequence (observing some simple rules, such as that a name cannot begin with a digit). Look at Figure 2.1. The first line specifies that this is an XML document and that it uses Latin-1 character coding.

Our example depicts a simple message in XML format. The *root element* of the document is `<message>`. An XML document must contain exactly one root element. The `<from>`, `<to>`, `<body>` and `<ps>` elements are all *children* of the root element and *siblings* of each other. On the basis of its contents, an XML element can be *complex, mixed, simple* or *empty*. Complex elements have children: such an element is `<message>` in the example. A simple element contains only literal or numerical data; such an element is `<from>Little Red Riding Hood</from>`. A mixed element contains both literals and child elements, while an empty element consists only of the opening and the closing tags.

The most striking property of an XML document is that it stores data in a hierarchy: the document defines a tree, with the root element at its root. In Figure 2.1 we define a two-level tree with one root and five child elements (see Figure 2.2). Thus an XML document defines not only the *name* and *contents* of individual elements but also their *hierarchical relations*. This makes XML also suitable for describing the structure of the data.

Figure 2.3 shows a second example of an XML document. Here we represent the employee database of a company in XML format. Employees have names, salaries and Social Security Numbers. Some also have phone numbers (perhaps more than one). It is clear that the database is easy to maintain: upon the arrival of a new employee a new `<employee>` element must be added and when someone leaves the appropriate entry must be deleted.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

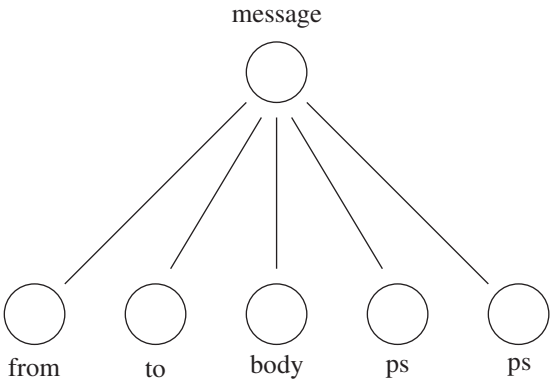

Figure 2.2. The XML tree of Little Red Riding Hood's message.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employees>
   <employee>
      <name>Jane Doe</name>
      <salary>4500</salary>
      <SSN>965342876</SSN>
   </employee>
   <employee>
      <name>John Stiles</name>
      <salary>6100</salary>
      <SSN>823457201</SSN>
      <phone>555-136-8531/2465</phone>
      <phone>917-555-7642</phone>
   </employee>
</employees>
```

Figure 2.3. Employee database of a company in XML format.

Since an XML document is text based and uses elements to delimit data (with mandatory closing tags), it is necessarily larger than a corresponding binary file would be. This is a deliberate design decision, as the dense storage of data was not a priority of the XML recommendation. One advantage of the text-based format is that the resulting file is readable by human beings as well, so there is no need to run an application to transform or display it. Another important aspect is error handling. Imagine just how many problems we could run into when trying to open a corrupted binary file. What are the user's options when the application returns a "sorry, but the file cannot be opened" kind of error message? If the given document is in XML, it is much easier to find and fix the error in question. The verbosity can still lead to difficulties since no-one likes to store files of several hundred megabytes

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

when the same in binary format would take up less than a tenth of that. The biggest problem, however, is not this (the present rate of increase in storage capacities can easily take care of it), but the much greater limitations of network *bandwidth*. XML files are often used for standard communication. As an example, consider the *SOAP protocol*, developed for web service interfaces, which uses XML to encode messages to be exchanged between applications. Fortunately most modern communication protocols, such as HTTP/1.1, are able to compress data on the fly, thereby significantly saving on bandwidth.

Exercise 2.2: Extend the XML source shown in Figure 2.3 to include the age of both employees. Examine the structure of the document by closing and opening parts of the XML tree (this can be done by clicking on the − or + sign in front of a composite element).

2.2.3. XML attributes

XML elements can have an arbitrary number of attributes (properties), but every attribute may occur at most once in each element. An attribute has a name and a value, separated by the equality sign. The well-known `<IMG>` element in HTML also uses attributes to identify a picture and its alternative text. We can see an example of this here:

```
<img src="dolphin.gif" alt="A cute dolphin">
```

An alternative XML representation of the message in Figure 2.1 is shown in Figure 2.4. There the sender and the recipient of the message are specified as attributes of the `<message>` element, rather than as its children elements. Note that an element with an attribute is always considered *complex*, but the values of attributes can only be *simple* (a literal, a numerical value etc.).

Using attributes instead of elements does not have any substantial benefits. This is more of a design and modelling decision. Attributes can be useful in cases when they carry only auxiliary information, not the core of the data. Their use can help to make the XML source more readable and structurally cleaner. An example of this is the following complex element:

```
<movie type="mpeg4">matrix.avi</movie>
```

In this example, the information that the format of the video is mpeg4 is less important than the file name from which it is available.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<message from="Little Red Riding Hood" to="Granny">
   <body>I'll visit you this afternoon!</body>
   <ps>I'll bring some cookies</ps>
   <ps>they say there's a wolf in the forest</ps>
</message>
```

Figure 2.4. Message from Little Red Riding Hood to Granny with XML attributes.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.2.4. XML namespaces

The XML element and attribute names that we have used so far are called *local names*. Local names are perfectly valid as names, although as soon as we try to merge multiple XML documents we must face the problem of conflicting names. It can happen, for example, that an XML document embedded into an HTML page also uses the `<body>` element name (like our message example). XML defines *XML namespaces* to avoid such conflicts.

The idea behind XML namespaces is very simple. It is advantageous to specify element and attribute names by adding a *prefix* such as this:

```
<n:body>...</n:body>
```

A local name extended with a prefix is called a *universal* or *qualified name*. We assume that the qualified name is unique, thereby avoiding any further conflicts. If the prefix is too simple, we are not much better off since the same prefix (`n` in this example) could accidentally be used in another XML document as well. The specification therefore requires that all prefixes must be associated with uniform resource identifiers (URIs), identifying XML namespaces. So, a prefix indirectly tells us to which namespace the given element actually belongs.

A URI is a literal with a well-defined structure. You can read more on URIs in Section 2.3. It is worth knowing in advance that a URI is generally assigned to an organisation or a person. An XML namespace can be considered unique because who is allowed to use it, namely its owner, can be determined unambiguously. When merging two such documents there will be conflicts, which must be resolved manually. Fortunately these conflicts happen very rarely.

To declare a new namespace prefix we must use special XML attributes, which can be added to arbitrary elements. There are two such attributes, and they are the following:

`xmlns:`*`prefix`* `xmlns.`

The former is used to associate a namespace to the namespace prefix called *`prefix`*. The latter specifies the *default namespace*. All XML element names without a prefix automatically belong to this namespace. In the absence of default namespace, an unprefixed XML element is considered to be a simple local name. Note that default namespace declarations do not apply to attribute names.

All namespace prefixes have a scope. The specification prescribes that an XML namespace prefix is visible only within the element which declares it (including the opening and closing tags of the element). Therefore it matters where a namespace prefix is declared. If we want the namespace prefix to be visible throughout the entire document, it should be defined as an attribute of the root element.

Figure 2.5 shows an XML document declaring and using two namespaces. The element called `<priority>` is in the default namespace; all other elements are in the namespace with the prefix `n`. Note that attribute names are also prefixed.

It is important to note that the URIs have no other role than to act as globally unique identifiers avoiding name conflicts. This means that the URIs used do not have to be valid (e.g. they do not need to work in a browser), so we are allowed to write anything we like. Specific URIs, however, play an important role since the implementations of concrete web technologies may rely on them. In particular, by looking for elements belonging to a given namespace, applications can identify those parts of an XML document in which they are interested.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<n:message
     xmlns:n="http://swexpld.org/"
     xmlns="uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882"
     n:from="Little Red Riding Hood" n:to="Granny">

   <n:body>I'll visit you this afternoon!</n:body>
   <n:ps>I'll bring some cookies</n:ps>
   <n:ps>they say there's a wolf in the forest</n:ps>
   <priority>important</priority>
</n:message>
```

Figure 2.5. A message from Little Red Riding Hood to Granny with XML namespaces.

For example, RDF applications process only those elements within an XML document that belong to the RDF namespace. More about this can be found in Section 2.3.

Finally, we mention that so-called *entities* can serve a role very similar to those of namespaces. An entity declaration can be used to instruct the XML parser to replace each reference to that entity with the value of the entity. The value can be an arbitrary character sequence, with the exception of a few special characters. For example, we declare entity `s` in the following snippet:

```
<!ENTITY s "http://swexpld.org/">
```

Following this declaration, anywhere within the XML document the reference `&s;wolf` becomes exactly equivalent to writing `http://swexpld.org/wolf` in that place.

Exercise 2.3: View the XML source in Figure 2.5 in a browser. Notice how the XML attributes and namespaces are displayed.

2.2.5. XML schemas; validity of documents

A piece of XML text is well formed if it adheres to the format requirements of XML documents; that is, all opening tags have a closing pair, the attributes are key-value pairs, and so on. However, well-formedness does not imply that the document satisfies any further, application-specific, requirements. In the example shown in Figure 2.3 we implicitly assumed that the names of people are string literals and not, say, numerical values. We also assumed that their salaries are non-negative and that they are guaranteed to have an SSN but not necessarily a phone number.

The real strength of XML stems from the fact that it is possible to check not only whether an XML document is well formed but also whether it is *an instance of a given language*. Without further information, it is impossible to figure out whether the language containing the example in Figure 2.3 allows further elements in the description of an employee (such as hobbies, marital status etc.). Likewise, there is no way to know whether the salary must be a numerical value or whether it could be something different; and so on. The reason is that an

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

XML document, as it is, can be an instance of multiple languages. When we are preparing an XML document, we are in fact creating a file according to an undetermined grammar. If we want to allow an application that can parse XML documents to be able also to verify whether these documents are instances of the language understood by the application, the language itself must be specified in a standard manner. This is what so-called *XML schemas* are for.

XML schemas describe which elements and attributes are allowed in the XML documents of the given language, what is allowed in the contents and the attributes of these elements and how they are related. For the document in Figure 2.1, for example, we can prescribe that there must be exactly one sender, one recipient and one body and there can be zero or more postscripts. We can also declare that the recipient must come first, followed by the sender, then the body and finally the optional postscripts. The schema corresponding to this verbal description can be seen in Figure 2.6. Provided that both the XML schema and an XML document are given, it is possible to tell whether the document is an instance of the language described by the schema.

In the schema we declared that all documents in our XML application begin with the `<message>` complex element. The `xs:sequence` element in the schema specifies that the given elements must follow each other in the given order. The `type` attribute, as expected, defines the type of the elements. In the case of the `<ps>` element, the `minOccurs` and `maxOccurs` attributes are used to prescribe that there can be an arbitrary number of these in a valid XML document. All the other elements (`from`, `to` and `body`) must occur exactly once, as in the absence of the occurrence indicators the default value, 1, is used.

By providing this schema, it becomes possible to reject messages that do not adhere to it (e.g. the sender is undetermined). One benefit of using XML as a data format is that the

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

<xs:element name="message">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="from" type="xs:string"/>
         <xs:element name="to" type="xs:string"/>
         <xs:element name="body" type="xs:string"/>
         <xs:element name="ps" type="xs:string"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

</xs:schema>
```

Figure 2.6. XML schema for messages.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

processing applications can be simpler than otherwise. Imagine that we are applying for a job, and we provide our personal data in an XML document adhering to a schema given by the company. The submitted XML document will not surprise the processing application at the company because, once it passes the schema verification, it is guaranteed not to contain "invalid" entries, e.g. a numerical value where a boolean is expected, it is guaranteed to have a SSN field (the schema can also specify the required number of digits[2]), and so on.

XML schemas offer much more than we have described here. For example, one can specify the domain of certain elements (the value must fall into an interval or must be one of a number of predefined values, and so on). This is just one instance of a constraint – there can be many others. For further details, see the specification of the XML schema [107].

Exercise 2.4: Try a web-based XML schema validator, such as [31], to validate the XML document in Figure 2.1 against the XML schema presented in Figure 2.6. Modify the XML document, e.g. change the order of the elements `from` and `to`, and run the validator again.

2.2.6. Transformation and visualisation of XML documents

The *eXtensive Stylesheet Language Transformation* (XSLT) [28] is a language that can be used to transform one XML document into another. The transformation itself works by building a new XML tree from nodes matching certain patterns in the source document. The patterns are given using the so-called XPath [26] language, described in Section 3.3, where the XML query languages are also elaborated. The new XML document resulting from the transformation can differ arbitrarily from the source: elements can be omitted, new elements can be inserted and so on. For example, given an XML document storing employees, using an appropriate XSLT transformation we can easily create an HTML page which lists the workers and their data in a tabular form, setting the names in boldface.

The transformation defined by XSLT is called a *stylesheet* (hence the expansion of the XSLT acronym). This name was chosen because an original purpose of XSLT was to transform XML documents into XSL-FO [13] files. XSL-FO is a generic-purpose formatting language, which describes how to present various documents to people. In other words, an XSL-FO document represents a stylesheet. An XSL-FO document can in turn be translated into a PDF, PS or plain text using a *Formatting Object Processor* (FOP). Such an FOP can be seen in [2].

The XSLT, XPath and XSL-FO languages all belong to the same family, called *eXtensive Stylesheet Languages* (XSLs). They all share the basic idea of separating XML data storage and visual display. Consequently, the application processing the XML files does not have to be concerned with how, where or on what device the results will be presented. At the presentation stage, the "human readable" form can be generated using an XSLT transformation (or something similar).

The idea of separating the data and its appearance appeared in a precursor of XSL, the *Cascading Style Sheet* (CSS) language. The goal of CSS is to associate formatting rules

[2] Although it is still possible that we provided an invalid, non-existent, number.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<html>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=iso-8859-1" >
    <title>Notice</title>
    <style type="text/css">
     h1      { color: red;
               font-weight: bold; }
     p       { color: black; }
    .thing   { color: green;
               font-style: italic; }
    </style>
  </head>

  <body>
     <h1>Attention</h1>
     <p>Bring the following necessities to the excursion:
     </p>
     <ul>
        <li class="thing">Raincoat</li>
        <li class="thing">Electric torch</li>
        <li class="thing">Warm clothes</li>
        <li class="thing">Food</li>
     </ul>
  </body>
</html>
```

Figure 2.7. Using stylesheets: the separation of contents and appearance.

with HTML documents, for example, to specify that the contents of `<H3>` elements must be rendered in red font. Such a description is also called a stylesheet, although CSS is significantly different from XSL stylesheets. The CSS specifications have many fewer formatting options than those given in XSL-FO. However, the simplicity of CSS pages is often an advantage, because they are much easier and faster to write.

Figure 2.7 shows an HTML page using a CSS stylesheet. We have avoided using any formatting-oriented HTML elements. The example notifies the participants on an imaginary excursion what to bring with them. Here, the CSS specification itself is built into the page, for better readability, but in most cases it is stored in a separate file and there is only a link from the HTML code. The stylesheet specifies that the list items must be printed in green with italics. Figure 2.8 shows how a browser displays this page.

Exercise 2.5: Following the example shown in Figure 2.7, add a stylesheet to some simple HTML page of yours that uses no CSS at the moment. Check the resulting web page in a browser.

---

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

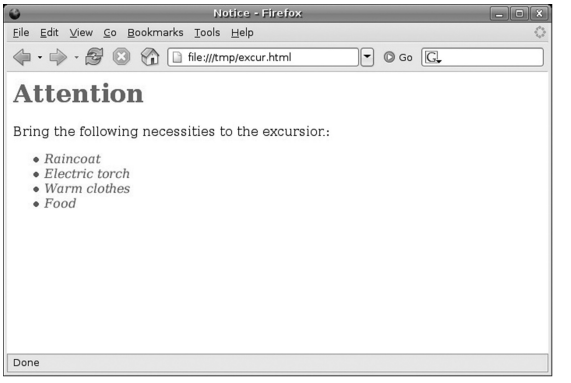

Figure 2.8. Web page that uses a stylesheet.

2.2.7. An XML application: the XHTML language

We have already mentioned that HTML is a specific SGML *application*. That is, HTML is a language defined using the SGML metalanguage and, by using it, we can create an arbitrary number of documents (i.e. web pages). Naturally, we have to obey the rules of the language during the creation of web pages; in other words, we always have to construct a sentence of the HTML language. It is worth knowing that today's browsers are pretty lax when it comes to grammar strictness, and they often accept normally invalid HTML documents (missing closing tags etc.), eventually bringing more harm than comfort.

We have also seen that XML, similarly to SGML, is a metalanguage, which raises the question: can we write HTML in XML? The answer is yes, it has already been done by W3C. The resulting XML application is the XHTML language, which combines the benefits of HTML and XML. An XHTML document is also an XML document; therefore it is suitable for encapsulating snippets of XML-based languages (such as RDF, SVG, XSQL etc.), and XML-based technologies (XSLT, XPath etc.) can also be used on it. Furthermore it is important that the early XHTML standards (XHTML 1.0 and 1.1) differed only very little from regular HTML, so that browsers need only minor modifications in order to be able to parse XHTML documents.

The first draft of XHTML version 2 was published by W3C in August 2002; the latest at the time of writing this book is the eighth draft. Version 2 broke away from legacies and distanced itself from HTML. Many traditional HTML elements were removed, such as the `<b>`, `<i>`, `<br>` and `<hr>` elements. The last, for example, is a request for the browser to display a horizontal rule in the page where the markup appears. The elements that were removed determined how the page should look, but they did not say anything about the contents, and the importance and machine-processability of the contents is a key point of XML.[3] The idea is that an XHTML document merely specifies in a structured manner the data composing the bulk of the page. The presentation is taken care of by stylesheets given

[3] It is interesting also that HTML was originally been designed to define the structure of web pages, and new presentation-oriented elements as well as the formatting semantics of some existing elements were only added later.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

in CSS or by XSLT transformations. These are responsible for deciding how certain parts of the document should be displayed.

This approach is very useful in practical applications, since it makes it possible to prepare a web page regardless of what kind of a device it will be displayed on (PDAs, mobile phones, customised browsers etc.), allowing the creator to concentrate on the content and the structure.

2.2.8. Evaluation of XML

Nowadays, XML is used practically everywhere: in deployment descriptors of web applications, in data files of word processors (Microsoft Office uses XML, for example), in database interfaces etc. The success of XML is all the more apparent knowing that it reached this level of popularity in only a couple of years.

However, XML is still basically a data-exchange format and in fact forms a very important milestone on the road to the creation of the Semantic Web. The first step was how the Internet changed the way in which applications communicate with each other. Before the dawn of the Internet various applications communicated on dedicated channels, using ad hoc protocols and techniques. This also included the case when someone wrote the data on a floppy disk and manually transferred it to another machine, to be read by the other application there. The Internet with its de facto standards and protocol families regularised the channels of communication.

Still, the problem of various applications using different data formats remained. Therefore a common syntax had to be assented to before each data interchange, since without this agreement no-one would have understood anyone else. No matter how badly a customer wants to buy a product, if the seller does not understand the request because the two sides represent the same data differently then a sale cannot be made. That XML is a standard data-exchange format is a huge step forward in this respect. Using XML there is no need to design and implement specific parsers for each communication channel. Using XML schemas, it can also be verified whether a received XML document adheres to our formal requirements.

2.3. The basics of the RDF language

With XML, a huge step forward was made towards the machine processability of web content by the standardisation of the syntax of data exchange.

However, the use of XML still requires that the meaning of the data to be transferred is agreed by the parties. XML is no help if one participant misinterprets a message and sends \$100 instead of \$100 000, just because the other counts everything in thousands! This is a general consideration: what if one party understands the same element differently from the other?

The Semantic Web, the next generation of the World Wide Web, is aimed at making not just the syntax but also the semantics of transmitted and stored data unambiguous to the parties involved. If this can be achieved then naturally a web crawler will also "understand" the contents of certain Internet resources. The general requirement is to be able to store data in such a way that it is processable and comprehensible for machines. The former is taken care of by XML and the latter is answered by RDF, which is to be presented shortly. Using RDF it

is possible to associate meta-information – meaning, if you like – with arbitrary web contents in a standard way. However, providing such concrete information is not enough; one also has to provide some background knowledge to enable, for example, intelligent search based on automatic inference.[4] The computer-processable form of such background knowledge, for a specific field of interest, is called a *terminological system* or *ontology*.

The RDF language is only the first step towards the Semantic Web. However, it provides the foundations for further steps: the RDF schema extension uses RDF as its base notation to support the building of lightweight ontologies, suitable for describing simple terminological systems. Similarly, the web ontology language OWL, capable of describing heavyweight ontologies, uses both RDF and RDF schemas as its building blocks.

In the following we first define the concept of URIs and their role in RDF. We will then present a simple example of an RDF description. This is followed by an explanation of the RDF data model, which gives a syntax-independent means of making RDF statements. We then introduce the modelling paradigm of the language, where we show how best to describe real-world concepts in RDF. In the final part of this section we present the XML syntax of RDF and then explain the various language constructs which can be used during modelling.

2.3.1. URIs and their role

The *Resource Description Framework* is a language suitable for associating metadata with arbitrary *resources* [11, 69]. In the world of RDF, everything that has a *URI* (Uniform Resource Identifier) is a resource; URIs are string literals, which identify objects and resources usually found on the web. Examples of resources are web pages and parts thereof: an image, an arbitrary file, an audio sample, a group of resources etc. It is important that as far as URIs – and thus RDF – are concerned, resources do not have to be attached to the web: even a coffee mug standing on a desk can be a resource, granted that it has an associated URI (which can be achieved). Therefore, sometimes we say that everything which has a URI is *on the web*.

Uniform Resource Identifiers have a fundamental role in the Semantic Web. Their most important property is their uniqueness, which enables the construction of unambiguous statements. No matter where two metadata descriptions appear, if they use the same URI (e.g. that referring to a person) then they both state something about the very same resource. For example, that the given person has blue eyes on the one hand and that his or her height is $5'11''$ on the other. This implies that it is fairly easy to combine various pieces of meta-information originating from different sources, since they can be joined simply by using the same URIs. It can also be said that the usage of URIs embodies the "anyone can say anything" idea, since all one needs is the appropriate URI to make arbitrary statements about a particular resource.

It is important to distinguish URIs from the perhaps better known *Uniform Resource Locators* (URLs) and *Uniform Resource Names* (URNs); these are special URIs. The former identify resources by their location, while URNs are persistent, location-independent resource identifiers. For more information about URNs, we refer to the standard specification [82]. Here are a few examples of URIs:

[4] For example, it may be known that a specific picture on the web shows a fox terrier. To return this picture in a search for dogs, one has to know that fox terriers are dogs.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
http://www.cs.uwyo.edu/index.html

mailto:lukacsy@math.bme.hu

file:///c:/examples/cat.rdf

urn:issn:1564-3417

uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882
```

The first example URI identifies a web page, the second refers to a person via his email address and the third is a file in a local file system. All three are URLs. The fourth example, a URN, identifies a publication location independently via its ISSN number. Finally, the fifth URI is a unique 128-bit identifier (written in hexadecimal notation), called a *Universally Unique Identifier* (UUID); these are generated in such a way that it is reasonable to assume that they will be different from other, similarly generated, identifiers.

As far as RDF is concerned, it is important to make a distinction between two kinds of URI. *Absolute* URIs uniquely identify resources, no matter where and when they are used. *Relative* URIs are meaningful only in a given context, namely when a so-called *base* URI can be determined. The base URI is determined by several factors, as defined by the standard, along with their priorities. For example, if a particular document format is suitable for specifying a base URI explicitly, and a document in this format does so, then that will be the base URI. Otherwise, the base URI of the encapsulating resource (if there is such an object) is inherited. If the base is still unknown, the URI identifying the document itself will be used as a base.

Both absolute and relative URIs may have a *fragment identifier*, which is separated from the body of the URI by a # sign. According to the specification [17], "the fragment identifier component of a URI allows indirect identification of a secondary resource by reference to a primary resource and additional identifying information". The semantics of a fragment identifier depends on the media type [41] of the document actually retrieved. In a case where retrieval is not possible, the semantics is undefined.[5] As a typical usage, a fragment identifies part of a document, for example, in the case of an HTML page it refers to the element with the attribute `name` matching the fragment identifier.

It is also possible that a URI consists only of a fragment identifier, such as `#Person`. This refers to the specific part of the document identified by the actual base URI.

Exercise 2.6: Enter the Semantic Web Activity page of W3C [112] and within the navigation panel called "Further links", on the right, check the section entitled "On this page". Notice that navigation within the site is achieved using fragment identifiers.

In the following subsection we present a simple introductory example, which is intended to show the reader the basics of the RDF language.

[5] Recall that a URI in general is "only" an identifier, and is not necessarily associated with an accessible and downloadable document (even a desk lamp can have a URI).

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.3.2. An introductory RDF example

The fundamental idea of RDF is to associate – through given properties – resources identified by URIs with other resources or with plain literals. Such an association is called a *triple*. Before going into further details about RDF, let us consider the RDF description of the following informal *statement*: "The e-mail address of John Doe (who is a person) is johndoe@freemail.org". This can be seen in Figure 2.9 in RDF graph representation using three triples. The graph expresses the following facts: there is a resource identified by the `http://freemail.org/~doe/#about` URI. The *type* of this resource is the URI `http://www.w3.org/2000/10/swap/pim/contact#Person`, its *name* is John Doe and its e-mail *address* is `mailto:johndoe@freemail.org` URI. In fact, the properties used in the graph (name, address, type) are also resources identified by URIs, but for brevity we have used simple names here.

The RDF standard also defines an XML-based syntax. Figure 2.10 shows the XML equivalent of the graph seen in Figure 2.9. The process of transforming an RDF graph into an equivalent XML description, or set of triples, is called *serialisation* or *linearisation*. Thus Figure 2.10 is the serialisation of the graph in Figure 2.9.

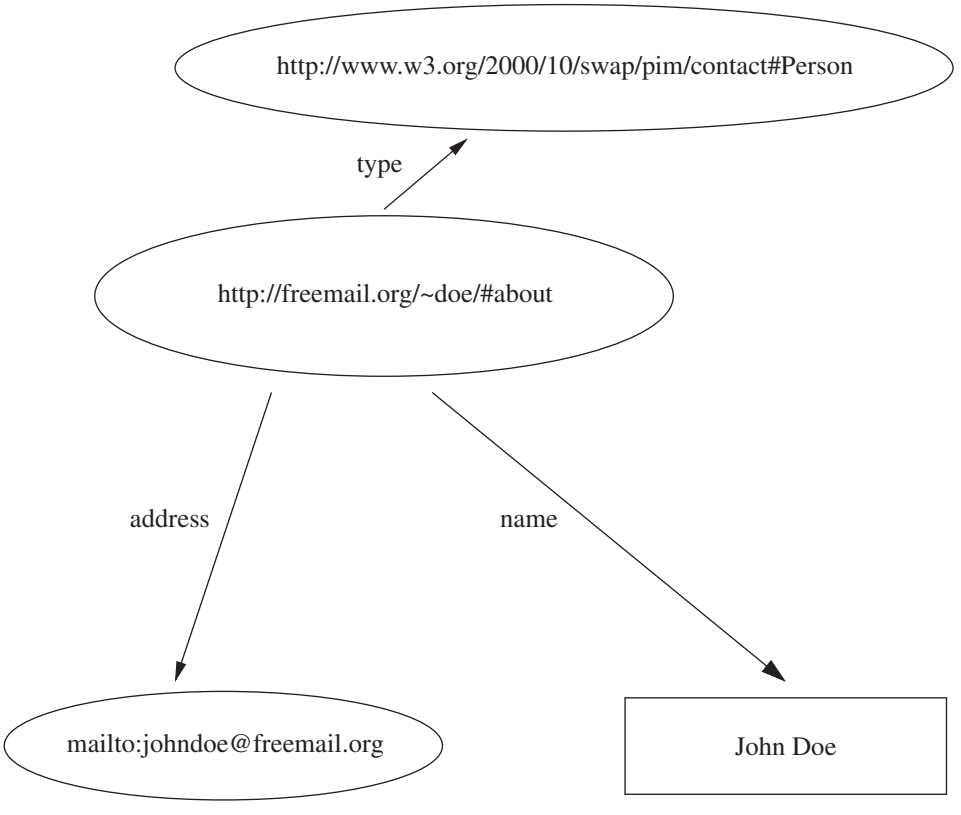

Figure 2.9. The first RDF example represented as an RDF graph.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://swexpld.org/utils#">

 <rdf:Description rdf:about=
  "http://freemail.org/~doe/#about">
   <s:name>John Doe</s:name>
   <s:address rdf:resource="mailto:johndoe@freemail.org"/>
   <rdf:type rdf:resource=
    "http://www.w3.org/2000/10/swap/pim/contact#Person"/>
 </rdf:Description>

</rdf:RDF>
```

Figure 2.10. The first RDF example in XML form.

The XML elements used in this example will be described in detail in Section 2.4.

In the next section we introduce the data model of the RDF and describe the precise syntax and semantics of RDF statements.

2.3.3. The RDF data model

The principal use of RDF is to associate metadata with resources. In other words, RDF is a framework that, in order to attach semantics to the pieces of information found on the web, allows one to assign metadata to them. The strength of RDF lies in the fact that by using URIs and RDF descriptions it becomes possible to associate meta-information to almost anything.

The RDF specification defines the *RDF data model*, a data model based on set theory which is used to describe metadata. The model is composed of the following parts:

- *The set of resources*. This is the set of all things about which one can make RDF statements. The elements of this set are called *resources*. RDF resources are identified by URIs, with optional fragment identifiers. Examples of RDF resources are a web page, an image found on a page and in general everything that has a URI. Considering that practically everything (a book, a refrigerator etc.) can have a URI, this set includes all imaginable entities.
- *The set of properties*. The elements of this set are called *properties*. Properties are features with values that can be attached to resources. The properties themselves are also RDF resources (in other words, Properties is a subset of Resources), and as such they also are referred to by URIs. In the case of a property, one can determine what resources they can be attached to, what values they can take and their relation to other properties.
- *The set of literals*. The elements of this set are *literals*, i.e. character sequences.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

- *The set of statements*. The elements of this set are *triples* or *statements*[6] that consist of *subject*, *predicate* and *object*. The subject is an arbitrary RDF resource, the predicate is any RDF property and the object (that is, the value of the property) is either an RDF resource or a literal. As a consequence, an RDF statement is nothing other than a cohesive group of three resources or of two resources and a literal.

The RDF data model does not define the syntax of RDF statements. An *RDF description* is a set of statements (that is, the set *Statements* represented in some syntax). Accordingly, the order of the triples does not matter. The *meaning* (semantics) associated by the data model to such a description is that the triples in the set are *true*.

Notice that in fact RDF describes *binary* (i.e. two-argument) relations between identifiable objects or, to put it another way, makes statements about them. It is a matter of taste whether we look on an RDF triple as a statement or as an association binding values to a resource.

In the next subsection we show how one can model the real world with this data model.

2.3.4. Basics of RDF-based modelling

We have just discussed how the RDF data model can be described in mathematical terms. This does not require a strict computer-processable syntax. However, we do need the latter since without it we would not be able to write down, store and transmit the statements, i.e. the metadata. Therefore, the RDF specification defines three standard data-model representations, one of which also has a concrete syntax. RDF data can be represented as a *set of triples*, as a *labelled directed graph* or as *XML data*. The graph model is of primary importance from the theoretical point of view but, as far as machine processability and data transfer are concerned, the XML representation is the principal format.

As an example, take the following statement: a painting called "The Night Watch" was created by Rembrandt van Rijn. For this we will need a URI which identifies the painting, such as

```
http://www.rembrandtpainting.net/rembrandt_night_watch.htm
```

Then the decomposition of the statement into parts as required by the structure of RDF triples can be seen in Figure 2.11. (Note that the RDF terminology is somewhat inconsistent with the terms used in English grammar.)

subject (`resource`)	http://www.rembrandtpainting.net/rembrandt_night_watch.htm
predicate (`property`)	painted by
object (`literal`)	Rembrandt van Rijn

Figure 2.11. Decomposition of an RDF statement.

[6] Sometimes the word "axiom" will also be used as a synonym of the word "statement".

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
{[http://...night_watch.htm], paintedBy, "Rembrandt" }
```

Figure 2.12. An RDF statement represented as a triple.

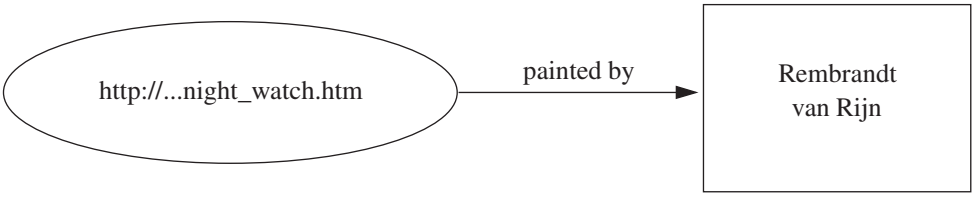

Figure 2.13. An RDF statement represented as a graph.

For the sake of simplicity we have listed the predicate ("painted by") in the example as a plain literal (in reality this must be a resource identified by a URI). The statement can also be depicted as a triple, as seen in Figure 2.12,[7] where the URI denoting the subject of the statement is delimited by square brackets `[]` and literals are surrounded by double quotes " ". We will use this notation for triples in the rest of the book.

The XML form of the statement will be given in Section 2.4, where the XML syntax of RDF is described.

Finally, the statement represented as a graph is shown in Figure 2.13. An RDF graph (not surprisingly) has nodes and arcs (directed edges). The nodes contain URIs optionally extended by fragment identifiers, or literals, thus the nodes represent RDF resources or literals, respectively. The arcs are labelled by (optionally extended) URIs: these identify RDF properties. It is important to note that an RDF graph *may only contain absolute URIs*. RDF is also suitable for making statements about properties: as properties are resources, they can appear as the subject or object of a statement.

From a modelling point of view it is rather unfortunate that the object of our example statement, the painter's name, is given as a literal, because a name does not identify the person unambiguously and, anyway, we might simply find it strange to have as a literal the painter of a painting. Let us not forget that RDF statements express *facts*, which are unconditionally true. Therefore it is better if we use as a URI an object which unambiguously identifies Rembrandt van Rijn (e.g. points to a web page describing his life and work). If we do not have such an unambiguous URI, we may still want to strengthen our statement by providing additional details about the painter. This can be achieved by introducing a so-called *intermediate* resource (also called a *blank* or *anonymous* resource), which is special in the sense that it cannot be identified at all: among other things, it does not even have a URI. Such a resource appears as a *blank* node in the graph, as can be seen in Figure 2.14. The graph expresses the fact that the painting was painted by someone whose name is Rembrandt van Rijn and who was born in 1606. The blank node in the graph provides a fairly exact identification of the artist without referring to the resource representing him.

[7] In the rest of this chapter we will shorten the URI of the painting and the name of the painter for typographical reasons.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

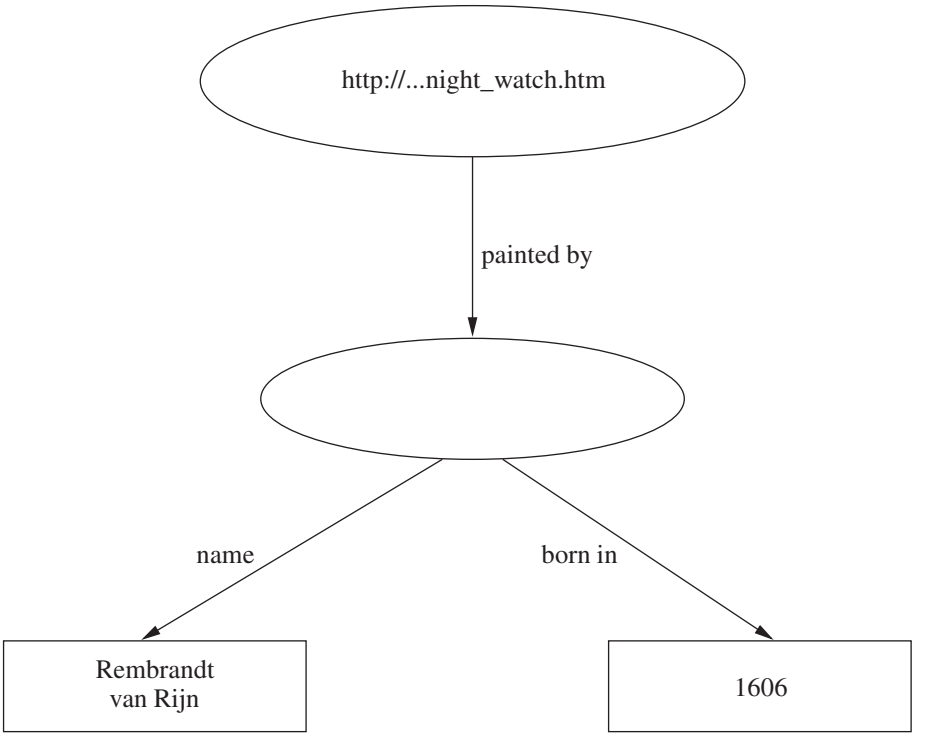

Figure 2.14. An RDF graph with a blank node.

Blank nodes are often used to improve the structuring of the information. When, for example, we decide to represent an address as a structured entity rather than a single literal ("Shaftesbury Rd, Cambridge, CB2 8RU"), it helps to introduce an intermediate resource for the address itself. In this case, the street address, the city and the post code can be attached to this node as literals. Even though four new nodes have been created instead of one, it is well worth the added cost. The structured nature of the information can become a great asset in a later phase, when it is processed by machines. Note that a similar technique is used when higher-arity relations (i.e. relations with more than two arguments) are described in RDF (see Subsection 2.5.1).

When an RDF graph which contains blank nodes has to be serialised (written as triples or in XML form), naming these nodes is often unavoidable after all. These names should be easily distinguishable from URIs and literals appearing in the description, but they do not need to be globally unique as long as there is no name conflict within a single RDF description (it might be necessary to rename or renumber these when merging multiple descriptions) [69]. An example of such a serialisation can be seen in Figure 2.15, where the intermediate resource was simply named `im_resource1`.

Earlier we said that an RDF statement is nothing other than an ordered sequence of three resources, identified by URIs, or two resources and a literal. Now we have seen that an

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
{[http://.....night_watch.htm], paintedBy, [im_resource1]}
{[im_resource1],name,"Rembrandt van Rijn"}
{[im_resource1],bornIn,"1606"}
```

Figure 2.15. An RDF statement with an intermediate resource.

intermediate resource can also appear in an RDF statement even though no URI is assigned to it. Extending thus our earlier definition, we state that the subject of a statement is either a resource identified by a URI or a blank resource. Similarly, the object of a statement is a resource identified by a URI; a literal or a blank resource. However, a predicate can still only be a resource identified by a URI; in other words, there can be no unlabelled arcs in an RDF graph.

Exercise 2.7: Consider the sentence "File `c:/project/house.jpg` is a picture made by Jim and it depicts a house painted white." Model this sentence in RDF. Describe your solution using both RDF graph and RDF triple representations.

Now we will proceed to the introduction of the XML-based syntax of RDF.

2.4. The XML syntax for RDF

This section introduces the use of the main XML constructs and namespaces in RDF descriptions. We show how intermediate resources can be described in XML and how relative URIs can be used.

2.4.1. Basic XML syntax

The XML syntax of RDF – the final version of which was published as late as 2004 – is a linear form of the RDF graph. Figure 2.16 displays the XML equivalent of the statement seen earlier (the given painting was painted by Rembrandt). Note that the XML description is rather verbose, given that the XML code seen in the figure represents exactly the same RDF expression as that depicted by the graph in Figure 2.13 and the triple in Figure 2.12.

The XML form of an RDF graph is a valid XML document: therefore such an XML description must start with the XML declaration introduced earlier:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

This tells the processing application that there follows an XML data stream conforming to the specified version and using the given character encoding. The subsequent `<rdf:RDF>` XML element denotes that the enclosed content (which is delimited by the closing `</rdf:RDF>` tag) should be considered as RDF data. The significance of this is that the RDF contents can be inserted into other, larger, XML sources, and in that case one should be able to unambiguously isolate them.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://swexpld.org/utils/">

 <rdf:Description rdf:about="http://...night_watch.htm">
   <s:paintedBy>Rembrandt van Rijn</s:paintedBy>
 </rdf:Description>

</rdf:RDF>
```

Figure 2.16. The XML form of an RDF statement.

Within the `<rdf:RDF>` element we find two attributes providing XML namespace declarations. The first assigns the prefix `rdf` to the namespace identified by the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#`:

```
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

This namespace is part of the RDF specification. Using the `rdf:` prefix in qualified names will make the corresponding local name part of the above namespace. We have already seen an example of such a name in the element name `rdf:RDF` itself. Note that while the namespace is required to be the above URI, the `rdf` prefix name is just a recommendation and in theory we can use any other prefix for denoting the RDF namespace.

The second declaration is in the form

```
xmlns:s="http://swexpld.org/utils/"
```

which assigns the name `s` to the namespace identified by the given URI. Since these two namespaces have been declared in the root element of the document, they are valid throughout the entire document according to the XML specification. Note that it is not compulsory to place the namespace declarations in the root element, but with a few rare exceptions this is the recommended procedure for RDF documents.

The next three lines describe the RDF statement represented by the graph in Figure 2.13:

```
<rdf:Description rdf:about="http://...night_watch.htm">
  <s:paintedBy>Rembrandt van Rijn</s:paintedBy>
</rdf:Description>
```

The subject of the statement can be described using the element `<rdf:Description>` from the `rdf` namespace. The URI identifying the subject is given as the `rdf:about` attribute of the `<rdf:Description>` element. Informally this first row means "here follows the *description* of the resource identified by the given URI".

The second row contains the simple XML element `<s:paintedBy>`. Such an element is called a *property element* and, together with its contents, it specifies the *predicate* and the *object* of the statement, the subject of which was given by the enclosing `<rdf:Description>` element. In our case the object of the statement is given by the contents of the element `<s:paintedBy>`, which is a plain literal. The predicate of the

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

statement is provided by the name of the element itself, which is a qualified XML name. However,we know that the predicate of an RDF statement must be identified by a URI. In order to understand how the `s:paintedBy` qualified name becomes an appropriate URI, let us make a small diversion.

The XML syntax of RDF uses qualified XML names to represent URIs written on the *arcs* of RDF graphs. *Properties* identified by absolute URIs (and optional fragment identifiers) in an RDF graph must appear in the XML form as `prefix:local name`. In the case of RDF/XML syntax, a qualified name is actually a specific notation of *appending* names (this is an extension of the XML specification).

According to this, the name `s:paintedBy` can be considered the equivalent of the URI `http://swexpld.org/utils/paintedBy`: thus the triple represented by the XML code in Figure 2.16 is

```
{[...],[http://swexpld.org/utils/paintedBy],"Rembrandt..."}
```

This is almost the same as what we saw in Figure 2.12. The only difference is that in Figure 2.12 the predicate of the statement is written as a plain literal for brevity.

The XML code of our example is terminated with the closing tag of the `<rdf:RDF>` element.

Let us note that whenever the content of a property element is a literal (such as `Rembrandt van Rijn` in our example), it is possible to replace the property element by an attribute. The name of the attribute refers to the property while its value is the content of the property element. For example, the abbreviated alternative form of our example is the following:

```
<rdf:Description rdf:about="http://...night_watch.htm"
                 s:paintedBy="Rembrandt van Rijn" />
```

Finally, we mention that since qualified names in XML form are merely syntactic sugar, there are multiple ways of transforming a URI into a qualified XML name (depending on where the URI is split in two). We have to consider, however, that XML does not allow arbitrary characters in qualified names, which means that, theoretically, not every RDF graph can be serialised into XML format. The RDF specification [11] actually suggests that the URI should be split immediately after the last character that is not considered a letter by the XML standard.

In the next section we first demonstrate how to make more compact RDF descriptions if several statements share the same subject. We then show how to write down a statement whose object is not a literal but a resource.

2.4.1.1. Shared subjects

The RDF source in Figure 2.10 is the serialisation of the RDF graph in Figure 2.9. It is worth noting various properties of the XML form. First, the `<rdf:Description>` element in the example has multiple property elements: that is, this element has multiple children. This is a compact version of the full, more verbose, form shown in Figure 2.17. In the compact form, each property element defines the predicate and object of an RDF statement but the subject of all statements is common. Since it often happens that we would like to state several things about a resource, the compact form can come in very handy.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://swexpld.org/utils#">

 <rdf:Description rdf:about=
  "http://freemail.org/~doe/#about">
    <s:name>John Doe</s:name>
 </rdf:Description>

 <rdf:Description rdf:about=
  "http://freemail.org/~doe/#about">
    <s:address rdf:resource="mailto:johndoe@freemail.org"/>
 </rdf:Description>

 <rdf:Description rdf:about=
  "http://freemail.org/~doe/#about">
    <rdf:type rdf:resource=
     "http://www.w3.org/2000/10/swap/pim/contact#Person"/>
 </rdf:Description>
</rdf:RDF>
```

Figure 2.17. Independent `<rdf:Description>` elements.

It is also worth noting that the declarations at the beginning of the XML document (XML version, character encoding, namespaces) only have to be written once, so larger RDF documents do not appear to be overly wordy. In fact, RDF triples can be translated into `<rdf:Description>` elements by computer in an automated manner. The XML syntax is based on such simple `<rdf:Description>` elements. Practically everything else is syntactic sugar, but for the sake of brevity we do not always stress this in the book.

2.4.2. Resources in object position

A further interesting point in the first example is the use of the `rdf:resource` attribute for referring to other resources. Let us recall that resource-to-resource relations are also present in the example in Figure 2.9. For example, the `address` of the resource identified by URI `http://freemail.org/~doe/#about` is also a resource, not a literal. Had we represented this part of the graph as

```
<rdf:Description rdf:about="http://freemail.org/~doe/#about">
    <s:address>mailto:johndoe@freemail.org</s:address>
</rdf:Description>
```

then we would have stated that the value of the `s:address` element is the `mailto:johndoe@freemail.org` *character sequence*, not the resource identified by the given URI.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

As can be seen, the `rdf:resource` attribute is used for describing resource-to-resource relations. The value of this attribute is always interpreted as a URI. The attribute always belongs to an element which must be the child of an `<rdf:Description>` element. The subject of the resulting RDF statement is determined by the `<rdf:Description>` element, its predicate comes from the name of the element containing the attribute and its object is the resource identified by the URI in the value of the attribute. The latter can be given in full form (unfortunately we cannot use qualified names in attribute values) or we can use XML entities; see Subsection 2.2.4.

Another case where resources appear in the object position will be described in Subsection 2.5.4. For the time being, we continue our discussion by showing how RDF intermediate resources (blank nodes in the graph) can be represented in XML form.

2.4.3. XML format of blank nodes

Intermediate resources can be represented in the XML syntax of RDF in numerous ways. Here we introduce two of the most often used solutions. A third will be described in connection with RDF containers in Subsection 2.5.4. What is important is that, whichever approach we choose, the graph corresponding to the XML form will be the *same*.

The most obvious solution for describing intermediate resources is an adaptation for XML of the approach seen earlier. Namely, the blank resource is assigned an identifier which is locally unique within the document. This can be done using the `rdf:nodeID` attribute, which may appear instead of the `rdf:about` and `rdf:resource` attributes, as can be seen in the following example:

```
<rdf:Description rdf:about="http://...night_watch.htm">
   <s:paintedBy rdf:nodeID="local_identifier1"/>
</rdf:Description>

<rdf:Description rdf:nodeID="local_identifier1">
   <s:name>Rembrandt van Rijn</s:name>
   <s:bornIn>1606</s:bornIn>
</rdf:Description>
```

Another possibility for serialising blank nodes is to use the `rdf:parseType` attribute, with the value `Resource`. The `rdf:parseType` attribute is a generic means of changing the interpretation of the property element to which it belongs. If its value is, for example, `Literal` then the content of the given property element should be always interpreted as a literal value, no matter what it is. In the following example we use some HTML tags inside the text describing the object of a statement, which, without the `rdf:parseType` attribute, would cause trouble:

```
...
<rdf:Description rdf:about="http://128.30.52.45">
   <dc:Title rdf:parseType="Literal">
   This is <I>my</I> computer!
   </dc:Title>
   <dc:Creator>Compaq</dc:Creator>
</rdf:Description>
...
```

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

Let us return to the question of blank nodes. By setting the value of the `rdf:parseType` attribute to `Resource`, we instruct the XML processor to encapsulate the property element into an additional `<rdf:Description>` element. This latter, imaginary, element describes a blank resource providing the object of the RDF statement, the predicate of which is determined by the name of the property element. The following snippet is the XML equivalent of the graph shown in Figure 2.14:

```
<rdf:Description rdf:about="http://...night_watch.htm">
   <s:paintedBy rdf:parseType="Resource">
      <s:name>Rembrandt van Rijn</s:name>
      <s:bornIn>1606</s:bornIn>
   </s:paintedBy>
</rdf:Description>
```

Note that, unlike the solution based on `rdf:nodeID` shown above, here we have only one `rdf:Description` element. The disadvantage is that the blank node described in this way can no longer be referenced from a different place (if, for example, we would like to add later a triple about the birthplace of the painter).

Exercise 2.8: Consider the RDF triples that you formulated as the solution of Exercise 2.7. Write down these in RDF/XML form. Use the RDF validator of W3C [111] to check the syntax of your document and to make sure that it really captures the set of triples with which you started.

Next, we show that relative URIs are also allowed in the XML form of RDF. We demonstrate how these relative URIs are resolved into absolute URIs.

2.4.4. Relative URIs in the XML syntax

The URIs in an RDF graph are present in the XML equivalent in various forms. We have seen that the URIs on graph arcs appear as qualified (element) names in the XML source. However, URIs in the nodes (those identifying the subjects and objects of statements) are listed as the values of XML attributes. Such attributes, among others, are `rdf:about`, used to describe subjects, and `rdf:resource`, used to specify objects of statements.

In our examples so far, the URIs defined as values of attributes were all absolute URIs, and we also mentioned that qualified names are in fact syntactic sugar for absolute URIs. The use of absolute URIs is hardly surprising, since we know that RDF graphs may only contain absolute URIs for identifying resources and that the XML form is always the equivalent of a graph.

However, the XML syntax of RDF allows the usage of *relative URIs* as well, in *all places* where absolute URIs are allowed. This is acceptable because in the case of the XML form of RDF the base URI is well defined, and the construction of an absolute URI from a relative URI can be guaranteed to succeed. The base URI is the URI of the document by default, but we may specify a different URI using the `xml:base` attribute.

Let us look at an example. In Figure 2.18 we see an XML document which uses a relative URI as the object of the RDF statement (`people/John`). Let us assume that the document

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/">

   <rdf:Description rdf:about="http://128.30.52.45">
      <dc:contributor rdf:resource="people/John"/>
   </rdf:Description>
</rdf:RDF>
```

Figure 2.18. The use of relative URIs in RDF.

is identified by the URI `http://swexpld.org/tricky/` (for example, because it is located there). Then the triple described by the document is the following:

```
{
 [http://128.30.52.45],
 [http://www.purl.org/dc/elements/1.1/contributor],
 [http://swexpld.org/tricky/people/John]
}
```

In some cases it might be necessary to specify the base URI of a document explicitly. In this way we can ensure that the resolution of the relative URIs always happens using the same base URI, independently of the location of the document. The base URI of an XML element can be specified using the `xml:base` attribute, which it is best to place in the root element. In that way we can determine the base URI for the whole document. Accordingly, to set the base URI in our example to `http://swexpld.org/base/`, the root element must be modified as follows:

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xml:base="http://swexpld.org/base/">
```

As a consequence of this modification of the base URI, the resolution of the relative URI now results in a different absolute URI and the XML document represents a different triple:

```
{
 [http://128.30.52.45],
 [http://purl.org/dc/elements/1.1/contributor],
 [http://swexpld.org/base/people/John]
}
```

Below we introduce an alternative to the `rdf:about` attribute that relies heavily on relative URIs.

2.4.5. The `rdf:ID` attribute

So far we have described resources in RDF that we assumed to have been identified by at least one URI. This URI was used as the value of the `rdf:about` attribute to construct RDF

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:s="http://swexpld.org/general/"
   xmlns:p="http://swexpld.org/utils/">

   <rdf:Description rdf:ID="employee1">
      <s:name>Jane Doe</s:name>
      <s:salary>4500</s:salary>
      <s:SSN>965342876</s:SSN>
   </rdf:Description>

   <rdf:Description rdf:ID="employee2">
      <s:name>John Stiles</s:name>
      <s:salary>6100</s:salary>
      <s:SSN>823457201</s:SSN>
      <p:phone rdf:parseType="Resource">
         <p:base>555-136-8531</p:base>
         <p:extension>2465</p:extension>
      </p:phone>
      <p:phone>917-555-7642</p:phone>
   </rdf:Description>

</rdf:RDF>
```

Figure 2.19. The use of the `rdf:ID` attribute (without any `xml:base`).

statements. In such a statement we claimed that the Night Watch was painted by Rembrandt van Rijn.

In many cases, however, we would like to make statements about things which do not yet have an associated URI or, if they do, we do not know what it is. For example, we might want to store the list of the employees or products of a company in RDF. Let us recall the employee database shown in Figure 2.3 in XML format. Figure 2.19 shows a representation of this data source in RDF.[8]

The main point of the example is that we used the `rdf:ID` attribute instead of the `rdf:about` to identify the subjects of the RDF statements. The meaning of the `rdf:ID` is similar to the `ID` attribute in the case of XML and HTML. Namely, `rdf:ID` defines a name which must be unique relative to the current base URI. In our example this allows us to "simulate" the process of assigning a unique identifier to a particular person (`rdf:ID` is also used for similar purposes in case of RDF reification, see Subsection 2.5.3 for the details).

[8] Unlike the original example, here the phone numbers with extensions are represented as structured entities; in this way the extension number and the base number are distinguished.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

The `rdf:ID` attribute can be very useful when we would like to define many different, but in some way coherent things in a single RDF document (e.g. various employees of a company). The distinctness of these things is guaranteed by the a special property of the `rdf:ID` attribute mentioned above: it must be true for a well-formed RDF XML source so that, within the scope of a given base URI (which usually means the entire document), an identifier listed in an `rdf:ID` attribute must appear *exactly once*. RDF parsers should verify this property.

Coherence[9] is ensured by the fact that in the case of `rdf:ID` we actually specify a fragment identifier. In practice this means that all resources named with `rdf:ID` use the same base URI. More precisely, the URIs identifying the subjects are constructed using the following rule: take the current base URI, append the character # and finally append the value of `rdf:ID`.

According to this rule, the two absolute URIs in our example are the following (assuming that the location of the document is `http://swexpld.org/stg.rdf`):

```
http://swexpld.org/stg.rdf#employee1 (*1)
http://swexpld.org/stg.rdf#employee2 (*2)
```

If the example is depicted as an RDF graph, the above absolute URIs appear as the labels of the corresponding root elements. Such a graph is shown in Figure 2.20, which, for the sake of simplicity, shows only the root element of `employee2`, omitting the SSN and the second phone number too. As can be seen, nothing in the graph implies that `rdf:ID` was used instead of `rdf:about`.

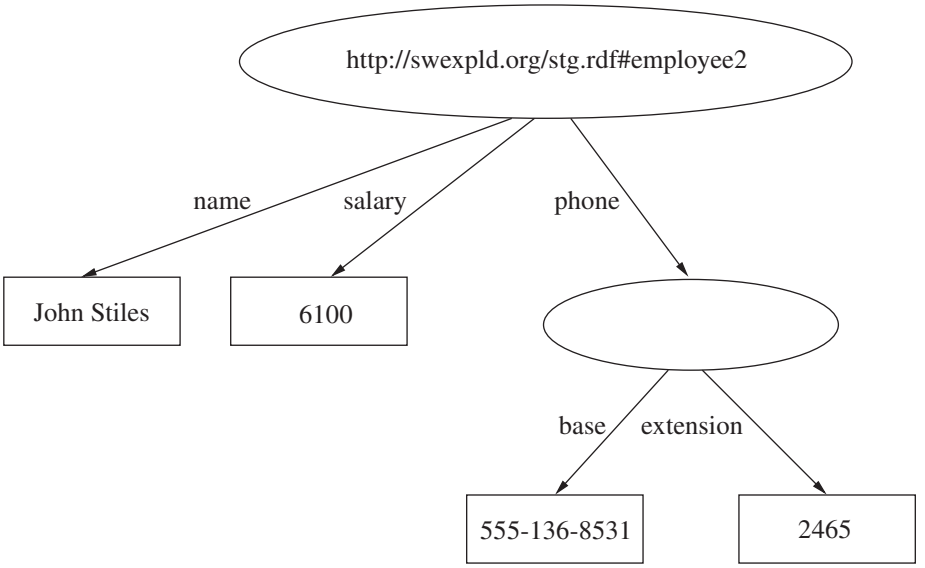

Figure 2.20. An RDF graph and the `rdf:ID` attribute.

[9] Note that by coherence we mean only syntactic similarity; RDF itself does not assume any relation between two resources simply because their URIs differ only slightly.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

The base URI has a particularly important role when the `rdf:ID` attribute (or relative URIs in general) are being used, because, as we have seen, relative URIs are resolved using the base URI. Imagine what would happen if we published the document in Figure 2.19 on several sites on the web. Depending on the location we would get different subject identifiers (e.g. `http://swexpld.org#employee1` etc.), resulting in different RDF statements. To avoid that, it is vitally important to specify explicitly the base URI using the `xml:base` attribute, introduced in Subsection 2.4.4 (which is not the case for the RDF source shown in Figure 2.19).

Whenever an `rdf:ID` appears in the scope of an `xml:base` specification, the resulting absolute URI depends only on the value of these two attributes. For example, no matter where the enclosing document is located, the label of the root element of the RDF graph represented by the following XML example will always be the URI `http://swexpld.org/base/#something`.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="http://swexpld.org/base/">
  <rdf:Description rdf:ID="something">
    ...
  </rdf:Description>
</rdf:RDF>
```

A resource identified by an `rdf:ID` attribute can be referred to with the `rdf:about` attribute within the same document (strictly speaking, within the scope of the same base URI declaration). Note, however, that the `rdf:about` attribute has to be preceded by a `#` character. The last statement in the following RDF description adds to the employee database the information that Jane Doe has bought herself a mobile phone.

```
...
<rdf:Description rdf:ID="employee1">
   <s:name>Jane Doe</s:name>
   <s:salary>4500</s:salary>
   <s:SSN>965342876</s:SSN>
</rdf:Description>

<rdf:Description rdf:ID="employee2">
   <s:name>John Stiles</s:name>
   ...
</rdf:Description>

<rdf:Description rdf:about="#employee1">
   <p:phone>615-555-5588</p:phone>
</rdf:Description>
...
```

If we refer to a resource identified by an `rdf:ID` either from an external document or from outside the scope of the `xml:base` declaration then we have to use the full

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:n="http://swexpld.org/styles/"

   <rdf:Description
     rdf:about="http://swexpld.org/stg.rdf#employee2">
      <n:hasEyeColour
         rdf:resource="http://swexpld.org/colours#blue"/>
   </rdf:Description>

</rdf:RDF>
```

Figure 2.21. External reference to an RDF resource.

URI, as expected. Figure 2.21 shows an example document which is different from that in Figure 2.19; it tells us that John Stiles has blue eyes.

The last example emphasises the point that information about a particular resource can be distributed on the Internet. Various places can store data about an employee of a given company. For example, it is possible that the author of the RDF document in Figure 2.21 could be the same as the author of that in Figure 2.19. If so, the author could have modified the original RDF source, instead of creating a new file, by adding a sixth child to the appropriate `<rdf:Description>` element. However, it is quite possible that two different people (or computers, for that matter) created these two files. This makes it clear that using RDF anyone can make arbitrary statements about any resource identifiable by a URI, without any practical limitations: a neat manifestation of the freedom of speech.

Exercise 2.9: What are the triples corresponding to the following RDF/XML document? Verify your solution using the W3C RDF validator [111].

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:s="http://swexpld.org/util/"
   xml:base="http://swexpld.org/base/">

<rdf:Description rdf:ID="id87">
   <s:address rdf:parseType="Resource">
      <s:street>Kossuth Lajos</s:street>
      <s:district>V</s:district>
      <s:city>Budapest</s:city>
   </s:address>
</rdf:Description></rdf:RDF>
```

---

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.4.6. Summary of and alternatives to the XML syntax

The XML syntax of RDF can be rather awkward to read for a human, but it has the huge benefit of being easily processable by machines because its base is XML. Furthermore, RDF sources written in XML are usually not generated by hand, or read directly by people, but in most cases we employ an RDF-based ontology editor, such as Protégé [104, 90], OilEd [6] or LORE [74]. Then the complexity of the format does not pose a problem. However, to make teaching RDF easier, and not to scare away novice users with an obscure and potentially unfamiliar XML syntax, Tim Berners-Lee created the *Notation 3* syntax [15], N3 for short. This notation is a practical alternative to the XML syntax given in the RDF specification; among other uses it is employed in the SPARQL query language, to be introduced in Subsection 3.3.6. The N3 syntax also serves as a basis for other RDF serialisations such as N-triples [9] or Turtle [8]. An example of a very simple N3 description can be seen below:

```
@prefix s:  <http://swexpld.org/utils/> .
<http://...night_watch.htm> s:paintedBy "Rembrandt" .
```

Thus, triples in N3 end with a period. A semicolon introduces another property of the same subject, while a comma introduces another object with the same predicate and subject. An example of such composite descriptions is shown below (we state that Bob, who is 34, has children Phil and Ryan):

```
...
p:Bob s:hasChild p:Phil, p:Ryan ;
      s:age 34 .
```

There are also other representations of RDF such as the JSON serialisation [39].

This concludes the introduction of the RDF XML syntax, and now we move on to the description of special RDF constructs.

2.5. Special RDF constructs

In this section first we observe how one can describe non-binary relations in RDF. Then we show how to state that a resource is an instance of a particular class. This is followed by the introduction of so-called reified statements, with which we can make RDF statements about RDF statements. Finally, we describe the containers and collections which can be used in RDF definitions and the concept of types as used by the RDF framework.

2.5.1. Non-binary relations in RDF

We have seen that one can describe binary relations between resources identified by URIs using RDF. In real life, however, we often encounter higher-arity, non-binary, relations as well.

For example, we call three integers a *Pythagorean triple* if a right-angled triangle can be constructed from segments of such lengths. For example, the numbers 3, 4 and 5 form a Pythagorean triple. The three numbers are connected by a single relation; only exactly three numbers *together* form a Pythagorean triple, so it is meaningless to ask what the relation is between 3 and 4 in the above triple.

As a further example, we could mention a `kick` relation, which relates a football player, a match and a number. Its meaning is that the given player scored the given amount of

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

goals during the specified match. Similarly to the previous example, it is meaningless to try to explain the relation between the match and the number, or the player and the number, without the third value.

It appears that non-binary statements play a significant role in real life. However, it is important to note that everything can be described using only binary relations. For example, we can create an intermediate resource to represent a Pythagorean triple, and bind the three numbers to it with three separate statements (first number, second number, third number). This technique can be used to translate any non-binary relation into a group of binary relations. Note that this transformation often also makes the data representation more structured. With regard to higher-arity relations, the `address` relation might connect a person, a postal code, a street name and a house number. But perhaps it would be more informative to transform this into a set of binary relations and create an intermediate resource representing the address itself, associating `postal code`, `street name`, and `house number` with it and connecting it with a further binary relation to the resource representing the person.

Thus the limitation that RDF can only define binary relations does not pose a theoretical barrier. Nonetheless, in some special cases RDF provides a means of helping represent non-binary relations. We can use the `rdf:value` property for this. This property allows us to assign a "main value" to a relation. The interpretation of what this main value means is at the discretion of the processing application. A good example of the use of the `rdf:value` element occurs when there is something simple to express (e.g. the price of an object) but it is worth modelling this as a complex entity (for example, understandably we would want to provide the currency of the price as well).

The following example represents a graph which still uses binary relations only. A product is connected to an intermediate resource with an `e:hasPrice` property, and the intermediate resource has two further properties, `rdf:value` and `e:hasCurrency` specifying the price currency. The description might be considered as a non-binary relation between the product, the price as a number and its currency:

```
<rdf:Description rdf:about="http://.../products#item10245">
   <e:hasPrice rdf:parseType="Resource">
     <rdf:value>6</rdf:value>
     <e:hasCurrency rdf:resource="http://.../currency/euro"/>
   </e:hasPrice>
</rdf:Description>
```

In our example, instead of `rdf:value` we could have taken a user-defined property, such as `e:amount`. The only difference would be that in the case of the standardised `rdf:value` an application might be able to answer 6 when asked the price of a given product (even though the object of the `e:hasPrice` property is the intermediate resource, not the number itself).

Exercise 2.10: How would you decompose into binary relations the ternary relation "player–season–goal" (that is, how many goals a football player scored in a given season, e.g. Ryan Giggs scored seven goals in the season 2005–6)?

---

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.5.2. Instances in RDF

In this section we introduce the way in which an RDF resource can be declared to be an instance of a class. We also show the simplified (abbreviated) XML syntax for describing class instantiation.

The RDF language makes it possible to specify that a resource is an *instance* of a given *class*. This functionality is provided by the `rdf:type` property, the object of which identifies the particular class. Earlier we used the following statement to state that John Doe is a *person*:

```
<rdf:type rdf:resource=
  "http://www.w3.org/2000/10/swap/pim/contact#Person"/>
```

This is in fact a completely ordinary RDF statement, analogous to that in which we specified that the name of a resource was Rembrandt van Rijn.

It is fair to say that RDF instantiation closely resembles the "instance of" relation of the object-oriented world. That is, if an RDF resource is the instance of the class of animals then we actually say that this particular resource is *an* animal. In our example the class of people is identified by the following URI:

```
http://www.w3.org/2000/10/swap/pim/contact#Person
```

Here we are actually assuming that this URI means the same (i.e. the class of people) to everyone. This is very important, as without this the RDF statement above could have different meaning for different processing peers (humans, computers). Another example is that claiming that the `s:name` of a resource is John Doe makes sense only if the resource identified by the URI `s:name` is mapped to the same semantics by everyone.

To achieve this, we need some way to express that a resource is a class or property and to specify certain characteristics of it. The RDF language itself does not give us the means to define our own classes and properties, but RDF schemas, to be discussed in Section 2.6, are well suited for this. However, RDF does provide a handful of *built-in classes*, which can be used to declare instances and which facilitate the use of higher-order statements and containers. These are introduced in detail in the next couple of subsections.

Declaring instances is very common in RDF, therefore the XML syntax allows a simplified form. Here, the `rdf:type` property element is omitted and the enclosing `<rdf:Description>` element is replaced by a new element, the name of which is the value of the removed `rdf:type` property, but in a different format (the value of the `rdf:type` property is a long full URI. e.g. `http://www.w3.org/2000/10/swap/pim/contact#Person`, but what we use instead is `<n:Person>`). For example, the statement shown in Figure 2.10 can also be written as seen in Figure 2.22. The two forms are interchangeable, their RDF graph being the same in both cases.

An arbitrary number of `rdf:type` properties can be associated with a resource; in this case the resource is the instance of multiple classes at the same time. However, the simplified form can only be used for one of these; the rest must be declared in the more verbose form.

A great advantage of the simplified syntax is that it allows us to store relatively complex pieces of information in a very XML-like style (the resource is a piece of clothing, a person, a computer etc.). In some cases this may facilitate the transition from XML-based to RDF-based data storage.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://swexpld.org/utils#"
 xmlns:n="http://www.w3.org/2000/10/swap/pim/contact#">

  <n:Person rdf:about="http://freemail.org/~doe/#about">
    <s:name>John Doe</s:name>
    <s:address rdf:resource="mailto:johndoe@freemail.org"/>
  </n:Person>

</rdf:RDF>
```

Figure 2.22. Simplified form of `rdf:type`.

Exercise 2.11: Let us assume that the employees in Figure 2.19 are actually instances of the class `Employee`. Modify the RDF source shown in the figure to represent this knowledge. Use the simplified syntax described above and compare your solution with the plain XML source presented in Figure 2.3.

In the following we show some examples of built-in RDF classes. First we introduce the `Statement` class, which includes so-called reified statements; this is followed by the definition of container classes.

2.5.3. Higher-order statements; reification

The RDF language lets us make statements about statements. Such statements are said to be *higher order*. Higher-order statements are necessary, since it is often important to know to whom a particular statement is attributed, when the statement was made and so on. A good example of a higher-order statement occurs when we state that *someone claims* that Rembrandt van Rijn was born in 1556.

The RDF equivalent of this natural language statement could be a statement the subject of which is the above mysterious someone (who is, by the way, not too well informed about dates), the predicate of which is that he or she claims and the object of which is the statement that Rembrandt was born in 1556. The problem is that the subject of an RDF statement must be a resource; therefore we need some way of modelling the statement as a special resource. This process is called *reification* and the resulting special resource is called a *reified statement*.

A reified statement is an instance of the `rdf:Statement` class. Therefore the meaning of the RDF statement

```
{[Uri1],[rdf:type],[rdf:Statement]}
```

is that the resource identified by `[Uri1]` is a reified statement, i.e. the model of another triple. The subject, predicate and object of the triple can be associated with the resource by

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

using the rdf:subject, rdf:predicate and rdf:object properties, respectively. Assuming that the painter is identified by the URI http://swexpld.org/painter#Rembrandt, the triples are

```
{[Uri1],[rdf:subject],[http://swexpld.org/painter#Rembrandt]}
{[Uri1],[rdf:predicate],[s:bornIn]}
{[Uri1],[rdf:object],"1556"}
```

Finally, the reified statement can be used to construct the original statement (assume that the URI http://swexpld.org/secret#he identifies the mysterious stranger):

```
{[http://swexpld.org/secret#he],[n:claims],[Uri1]}
```

The reified statement is usually depicted as a blank resource. The final XML form of the original statement can be seen in Figure 2.23 and the corresponding graph is shown in Figure 2.24. According to the simplified syntax, the rdf:type property element could have been omitted had we replaced the line

```
<rdf:Description rdf:nodeID="identifier1">
```

by

```
<rdf:Statement rdf:nodeID="identifier1">
```

It is important to note that the fact that a reified statement is present in an RDF graph does not necessarily mean that the modelled statement is also part of it. In other words, the RDF

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:n="http://swexpld.org/utils/terms">

 <rdf:Description rdf:nodeID="identifier1">
  <rdf:type rdf:resource=
   "http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  <rdf:subject rdf:resource=
   "http://swexpld.org/painters#Rembrandt"/>
  <rdf:predicate rdf:resource=
   "http://swexpld.org/utils#bornIn"/>
  <rdf:object>1556</rdf:object>
 </rdf:Description>

 <rdf:Description rdf:about="http://swexpld.org/secret#he">
   <n:claims rdf:nodeID="identifier1"/>
 </rdf:Description>

</rdf:RDF>
```

Figure 2.23. Reification in RDF/XML.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

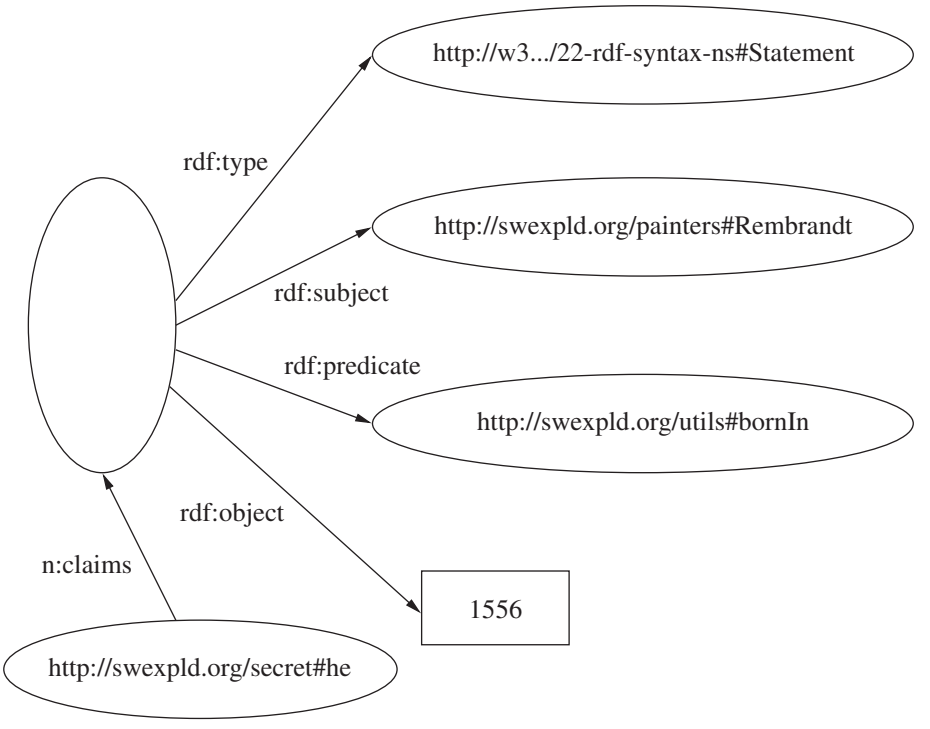

Figure 2.24. Reification in the RDF graph.

description in Figure 2.24 does *not* state that Rembrandt was born in 1556. All it says is that someone *claims* he was born in that year. This is a very important distinction because RDF statements express facts, which are considered true, and so which statements are represented by a graph is critical. Our example contains a single statement about an `[n:claims]` property, which has nothing to do directly with the birth date of the painter.

Note that a higher-order statement can also be created *without* reification if the triple in question already has a URI assigned to it (which means that someone else has already done the reification). For example, if `U` is a URI identifying a triple, the following statement is a higher-order statement (it is given in both XML and triple form):

```
<rdf:Description rdf:about="U">
   <n:author>John Doe</n:author>
</rdf:Description>

{[U],[n:author],"John Doe"}
```

We would also like to call the reader's attention to a peculiar feature and deficiency of RDF reification. A reified statement, according to the RDF specification, is about a *specific* triple in a *specific* RDF graph. This is obvious, since we use reification to declare who made a certain claim etc. However, statements containing the very same subject, property and object may appear in more than one graph. Therefore it can happen that several people made the

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

mistake of giving the birth of Rembrandt van Rijn as 1556, and we would like to make a statement about a specific one of these claims.

The problem is that the reification syntax of RDF does not provide the means to directly attach the reified statement to the statement which reifies it. For example, in Figure 2.23 the reified statement identified by `identifier1` could include all triples where the subject, the predicate and the object assume the values specified. All we can do is try to refine which specific statement we are talking about (which statement was made at this time and in this place etc.)

Because of this (and because the syntax for reification introduced above is quite complex), the RDF recommendation actually provides a another way to "name" an RDF triple. More specifically, we can use the `rdf:ID` construct (already described in Subsection 2.4.5) for this purpose. In the following example we specify an RDF triple stating that Rembrandt was born in 1606.

```
<rdf:Description rdf:about="#Rembrandt">
  <s:bornIn rdf:ID="triple42">1606</s:bornIn>                    (2.1)
</rdf:Description>
```

Note that we have given an attribute `rdf:ID` for the property element `s:bornIn`. This notation tells the RDF parser to automatically create the appropriate reification triples. In other words, the RDF description above is equivalent to the following triples (`baseURI` stands for the base URI of our example above):

```
{[baseURI#Rembrandt], [s:bornIn], "1606"}
{[baseURI#triple42],  [rdf:type], [rdf:Statement]}
{[baseURI#triple42],  [rdf:subject], [baseURI#Rembrandt]}
{[baseURI#triple42],  [rdf:predicate], [s:bornIn]}
{[baseURI#triple42],  [rdf:object], "1606"}
```

Here, however, `baseURI#triple42` identifies exactly the triple in question. In some other RDF triple this URI can then stand as a subject or object providing information about the triple itself.

We should also note that, when using `rdf:ID` for reification purposes, the original RDF triple is actually placed in the resulting RDF graph (in this case, the triple stating that Rembrandt was born in 1606).

Exercise 2.12: Check the above statement by invoking the W3C RDF validator [111] on the RDF snippet (2.1) and ask the validator to draw the RDF graph for you. Note: do not forget to embed the piece of RDF code in question into proper `<rdf:RDF>` elements.

Finally, let us note that reification can be used in a natural way to create statements such as "he claims that she claims…", since nothing forbids us from reifying a statement which uses a reified statement as its object.

2.5.4. Containers and collections

In this subsection we introduce built-in RDF classes which can be used to gather resources into groups and to make statements about such groups.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

We often consider a particular resource as a group of things. For example, the creators of a homepage or a program are such a group, one that has members which in this case are the creators themselves. Such groups are called *containers* in RDF. The RDF language allows the definition of containers and provides a set of built-in classes and properties for this.

The following three container classes are defined in RDF:

- *Bag* (`rdf:Bag`)*:* This is a group of resources or literals where the same item can occur more than once and the order of the members is not significant. Examples of data that can be stored in an `rdf:Bag` are the list of employees working in a department, the publications of one person and the contents of a shopping cart. The latter may contain more than one of the same item.
- *Sequence* (`rdf:Seq`)*:* This is an ordered `rdf:Bag`, i.e. a group where the same item can occur more than once but where the order of the items matters. Examples of data suitable for `rdf:Seq` are the list of employees working in a department (in alphabetical order), the contents of a shopping cart (in increasing order of the price of the items), and a person's publications (in order of appearance).
- *Alternative* (`rdf:Alt`)*:* This is an unordered group of resources or literals where the same item can occur more than once and the items are in some sense interchangeable, being alternatives of each other. Examples of `rdf:Alt` are a group of synonyms and the list of mirror sites of a file accessible through the web. In distinction from the two other container classes an `rdf:Alt` cannot be empty, i.e. it must contain at least one item. The first item of the container is considered the default item: therefore in fact the order of items is not significant, with the exception of the first item.

Note that RDF does not define container classes corresponding to the concept of a set as used in mathematics.

A resource can be declared to be a container by using the `rdf:type` attribute. It is worth noting that, with RDF, containers are never "created" in the sense that we create a new array, as for example in a classical programming language. There we usually specify how many elements the array can have, what types of elements it may contain etc. Here we are merely declaring about an already existing resource that it is a container. The following snippet, using the `rdf:type` property, states that a resource is an `rdf:Bag`:

```
<rdf:Description rdf:ID="Rucksack">
   <rdf:type rdf:resource=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"/>
</rdf:Description>
```

A container resource identifies the container itself. This resource can be used in arbitrary RDF statements. For example, we may state about the above rucksack what it cost, what colour it is etc. We might also determine the number of people in the group of creators of a program or specify in some way the average employee morale; however, although the morale of the group as a whole might be high that would not necessarily mean that *everyone* in the group is so enthusiastic about his or her work.

The members of a container can be defined with special RDF properties. The form of these is `rdf:_n`, where `n` is a base 10 natural number. Such properties are `rdf:_1`, `rdf:_2` etc. These properties are attached to the container; their values identify members of the container, as can be seen here (using the simplified syntax of instantiation):

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<rdf:Bag rdf:ID="Rucksack2">
  <rdf:_1 rdf:resource="#Apple"/>
  <rdf:_2 rdf:resource="http://swexpld.org/things/laptop"/>
  <rdf:_3 rdf:resource="http://swexpld.org/stationery#pen"/>
</rdf:Bag>
```

As we have already mentioned, containers are not "created" in RDF; rathers, existing resources are thereby declared to be containers (this is true in general for RDF instantiation). Consequently, we have not declared that our rucksack contains *only* the listed items and nothing else. All we have said is that *these* objects are in the rucksack, even though there could be many more. Another RDF source might place further resources in the container; in fact, there could be many more such RDF sources doing this.

An `<rdf:Description>` element describing a container can have an arbitrary number of property elements, not just the `rdf:type` and `rdf:_n` properties. Accordingly, the following snippet specifies the model of the rucksack and places several resources in it:

```
<rdf:Bag rdf:ID="Rucksack3">
   <rdf:_1 rdf:resource="#Cookie"/>
   <rdf:_2 rdf:resource="#Candy"/>
   <rdf:_3 rdf:resource="#Gum"/>
   <s:model>Deuter 28</s:model>
</rdf:Bag>
```

Observe that even though RDF provides a means of specifying containers, `rdf:Bag` is just an ordinary class as far as RDF is concerned. In an RDF graph the above `Rucksack3` would appear as a resource connected to a URI via the `rdf:type` property and to a literal as the value of the `s:model` property. We could create an isomorphic graph structure which has nothing to do with containers; what makes containers special is that the processing applications should recognise the `rdf:Bag`, `rdf:Seq`, `rdf:_n` etc. URIs and associate the right meaning with them, *as defined by* the RDF specification.

A typical use of containers is as the objects of statements. For example, to model in RDF that a given homepage was created by John Stiles and company, we would have to represent the group of creators as a container and specify this container as the object of the appropriate RDF statement. This can be seen in a more complicated example in Figure 2.25. This example specifies the creators and the registered users of a web portal. Part of the example can be seen in Figure 2.26 in graphical form. The groups of creators and of users are both identified by anonymous containers. The example also demonstrates the use of two further, as yet unseen, features.

The `rdf:li` property can be used instead of `rdf:_n`; its first occurrence is replaced by `rdf:_1`, the second by `rdf:_2` etc. The members of the container are still attached to the container via `rdf:_n` properties (as can be seen in Figure 2.26): `rdf:li` is merely syntactic sugar, used so that we do not have to number the items manually.

Another interesting feature is a yet unseen construct for representing a resource as the object of a statement. The use of the `<rdf:Bag>` element is nothing other than the simplified form of instantiation already introduced, but since the container is represented as an anonymous resource, i.e. there is no URI assigned to it, the `rdf:about` attribute is missing. Writing this `<rdf:Bag>` element as a child element of the `<dc:creator>` element

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://swexpld.org/utils#"
 xmlns:dc="http://purl.org/dc/elements/1.1/">

 <rdf:Description rdf:about="http://swexpld.org/portal">
  <dc:creator>
   <rdf:Bag>
    <rdf:li>John Stiles</rdf:li>
    <rdf:li>Mary Major</rdf:li>
    <rdf:li>Richard Roe</rdf:li>
   </rdf:Bag>
  </dc:creator>

  <s:users>
   <rdf:Seq>
    <s:comment>Ordered by time of registration</s:comment>
    <rdf:li rdf:resource="http://swexpld.org/people/id15"/>
    <rdf:li>Lou</rdf:li>
   </rdf:Seq>
  </s:users>
 </rdf:Description>
</rdf:RDF>
```

Figure 2.25. An example of the use of containers.

is another way of specifying that the given resource is the value of the `dc:creator` property (see Subsection 2.4.3). The situation is analogous for the `<rdf:Seq>` container in the example.[10]

If the container appears as the object of a statement, in some cases it can be substituted by as many statements as the number of items in the container. The graph in Figure 2.26 implies that the given three people are all creators of the portal; therefore the following RDF description could be viewed as semantically equivalent to it:

```
{[http://swexpld.org/portal],[dc:creator],"John Stiles"}
{[http://swexpld.org/portal],[dc:creator],"Mary Major"}
{[http://swexpld.org/portal],[dc:creator],"Richard Roe"}
```

However, this is not true in general. It might happen that such a transformation changes the meaning of the graph. A classical counter-example is the following: consider a group

[10] Note that if a container is represented as an anonymous resource then it is practically closed, since other sources cannot refer to it owing to the lack of a URI.

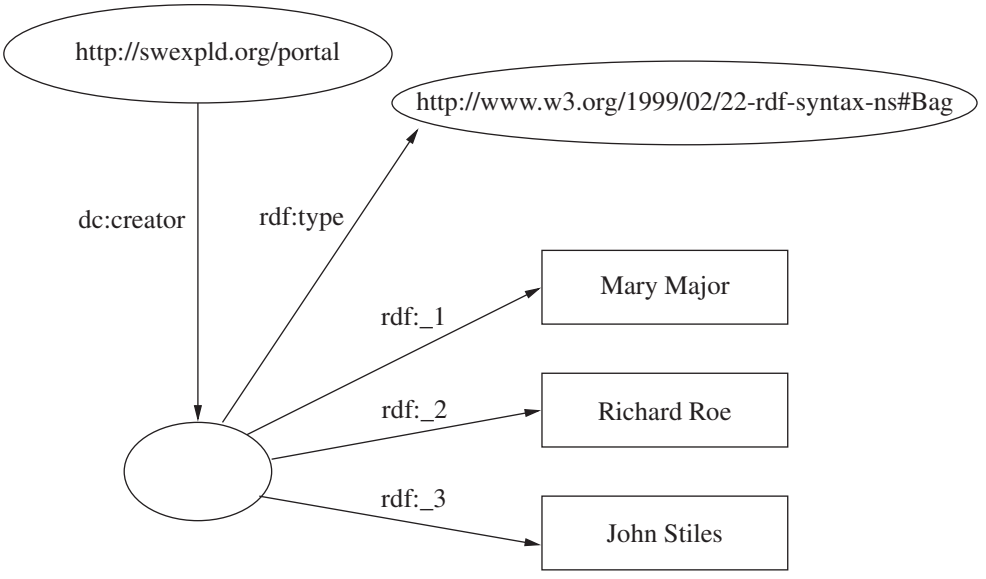

Figure 2.26. Containers presented as graphs.

of people who pass a resolution by plain majority voting. This can be written as follows in RDF:

```
<rdf:Description rdf:ID="Resolution">
   <n:acceptedBy>
      <rdf:Bag>
         <rdf:li>Gandalf</rdf:li>
         <rdf:li>Lord Elrond</rdf:li>
         <rdf:li>Frodo Baggins</rdf:li>
      </rdf:Bag>
   </n:acceptedBy>
</rdf:RDF>
```

If this is transformed in the above manner then we get three separate RDF statements declaring that the resolution has been accepted by all members of the group individually. However, this is not necessarily true in majority voting.

Finally we mention that RDF also allows the creation of groups of resources which are closed and contain only the listed items. Such a group is called an *RDF collection*, which is nothing other than a *list* of resources. RDF defines URIs for building such lists. A specific collection is an instance of the `rdf:List` class. The first item is given using the `rdf:first` property and the rest is *another* collection, attached to the main collection via the `rdf:rest` property. This second collection can also have a first item, and so on. The end of the collection is denoted by the `rdf:nil` resource. The container of the creators shown in Figure 2.25 can be closed, according to this, as follows:

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<rdf:Description rdf:about="http://swexpld.org/portal">
   <dc:creator>
      <rdf:List>
         <rdf:first>John Stiles</rdf:first>
         <rdf:rest>
            <rdf:List>
               <rdf:first>Mary Major</rdf:first>
               <rdf:rest>
                  <rdf:List>
                     <rdf:first>Richard Roe</rdf:first>
                     <rdf:rest rdf:resource=
                      "http://.../22-rdf-syntax-ns#nil">
                  </rdf:List>
               <rdf:rest>
            </rdf:List>
         </rdf:rest>
      </rdf:List>
   </dc:creator>
</rdf:Description>
```

In some cases RDF permits the specification of collections in a simpler form very similar to that for declaring containers. In order to do this one must use the `rdf:parseType` attribute with the value `Collection`, described in more detail in the RDF specification [11]. It is worth knowing that for this kind of use of the `rdf:parseType` attribute, only collections containing RDF resources can be specified; so, those containing literals (such as the collection in our example) cannot.

Exercise 2.13: How many elements are there in the container called `Rucksack3` shown above?

In the following subsection we introduce the concept of data types in RDF, describe the two types of literal and show how these can be specified.

2.5.5. Data types in RDF

We distinguish two types of literals in RDF. Until now, we have seen only *plain literals*. These have the common trait of representing a character sequence. We have seen that it is often beneficial to store such data in a structured manner instead. As an example we suggested the splitting up of an address stored as a literal into logically separable segments and the use of an intermediate resource to connect them. Such segments were the postal code, the street name and the house number.

Even though adding structure to literals is a significant step forward, it is still true that literals are viewed as character sequences. This is unfortunate in the case of house numbers, for example, or in the case of numbers in general. The literal `12` in RDF is simply the characters `1` and `2` written side by side. It would be good if this could be interpreted as a number. However, then there would be the question of in which base it is interpreted.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

All in all, it would be beneficial to introduce typed values for RDF literals. Literals with types are called *typed literals*. Sadly, RDF does not define built-in types, not even the simplest ones such as integers, floating point numbers etc. Nonetheless, it provides means to assign a URI to a literal which identifies its type. This can be done using the `rdf:datatype` attribute. This special attribute is added to the property element which contains the literal in question. For example, the literal value of the property element

```
<n:age>23</n:age>
```

"obtains" a type as follows:

```
<n:age rdf:datatype="Int">23</n:age>
```

(where `Int` is a URI which identifies a type – in this case the integer type).

The question remains: where can we find a URI which identifies the type that we need and which is recognised in a circle wide enough to contain the application which is going to process our RDF source? Fortunately the *XML schema* specification defines such types. We have seen that the type of the content of an XML element can be prescribed using XML schemas. The XML schema defines a number of primitive data types (integer, floating point etc.) and complex data types (date, time interval etc.). A data type defined in an XML schema can be referred to by appending a fragment identifier to the URI `http://www.w3.org/2001/XMLSchema`. For example, integers are identified by the following URI:

```
http://www.w3.org/2001/XMLSchema#int
```

It is important to note that as far as RDF is concerned the URI given as the value of the `rdf:datatype` attribute is irrelevant, even though usage of the XML schema is recommended. Actually, we could specify a URI which identifies something that is not a type or something that contradicts the representation of the literal (e.g. we could specify "integer" as the type of a literal containing characters). However, RDF ignores all this. All it does is to provide a way of attaching an external type to RDF literals in a standard manner. Everything else is the responsibility of the processing application. If the application in question "comprehends" the given URI then it will verify the correctness of the literal.

As a summary of the above consider Figure 2.27, which is the XML representation of the graph seen in Figure 2.14 in which we specify the date of birth of the painter "typed" as a date, this time correctly.

Exercise 2.14: Run the W3C RDF validator [111] on the RDF source using the typed literals shown in Figure 2.27 and also on the earlier version of the same knowledge base given in Figure 2.15. How does the use of typed literals change the resulting triples?

2.5.6. Summary and open issues

In the above we introduced the RDF framework, which allows us to make statements about resources identified by URIs. RDF descriptions can be represented in XML syntax, which thus presents meta-information in a form that can be easily processed – at least syntactically – by standard and automated methods. Metadata provided in RDF can be used by Internet search engines to facilitate a more intelligent and human-centric search.

The RDF approach is still very fresh and many questions naturally arise about its usability. A primary concern regarding RDF descriptions is that nothing guarantees or even requires

```
<?xml version="1.0" encoding="iso-8859-1"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:s="http://swexpld.org/utils/">

   <rdf:Description rdf:about="http://...night_watch.htm">
      <s:paintedBy rdf:parseType="Resource">
         <s:name>Rembrandt van Rijn</s:name>
         <s:bornIn rdf:datatype=
          "http://www.w3.org/2001/XMLSchema#date">
         1606-07-15</s:bornIn>
      </s:paintedBy>
   </rdf:Description>

</rdf:RDF>
```

Figure 2.27. The date of birth of Rembrandt given as a typed literal.

that the creators of various web pages provide truthful and valid meta-information about their resources. It often happens even today that a page "lies" about what it contains in the hope that it will occur more often and in more distinguished positions in the result sets of search engines (there are even web pages dedicated to giving detailed tips and tricks to achieve such effects). We have also mentioned that web pages exist where the web server, detecting that a web crawler is trying to access a particular page, returns a page different from that which it would present for an ordinary page request.

Nonetheless, we saw in Subsection 1.2.4 that the question of the reliability of information found on the web is important from more than just the Semantic Web point of view. We believe that a future direction for the Internet is in developing its potential for distinguishing reliable and unreliable sources. Various techniques which are hard to manipulate, such as the PageRank algorithm, can help (and already do so today) to give a higher value to meta-information originating from more reliable pages than those from pages of low esteem. This idea also helps to answer the question what to do with contradictory meta-information. Namely, what should happen when someone claims that a particular file contains music while someone else states that it is the source code of a program? This question will have to be answered by future research.

Independently, solutions for extracting meta-information automatically have started to spread, which in part replaces the need to provide meta-information manually on an RDF basis and thus reduces any potential intentional imprecision. Such solutions are the so-called RDFizers [98], which try to extract semantically important information from different kinds of information source. There is RDFizer for converting pictures, email data, BibTEX, CSV files, GPS data, BitTorrent files etc. into RDF format. The extraction process itself may work with or without human intervention, depending on the source type. These tools may greatly increase the amount of RDF data available on the web, thus providing a sufficient amount of meta-information for the search engines to work with.

2.6. Representing background knowledge: RDF schema

RDF schema is a lightweight ontology language [22]. It provides constructs to build domain-specific classes and properties as well as to specify their characteristics and the hierarchical relations between them. An example could be the introduction of the class `Human` and the property `hasChild`. Such classes and properties could then be used in ordinary RDF descriptions, such as one stating `{[Mary],[hasChild],[Bill]}` and `{[Mary], [rdf:type],[Human]}`.

RDF schemas are actually written in RDF syntax, i.e. using triples. This means that RDF schema constructs are actually special RDF resources. This idea is outlined below.

2.6.1. RDF schema as a collection of RDF resources

We have seen before that the fundamental idea of the RDF framework is to construct triples from RDF resources identified by URIs. For this RDF provides some predefined resources, such as the `rdf:type` and the `rdf:value` properties or the `rdf:Alt` container class; RDF associates with these resources a meaning, which is understood by all RDF-aware applications and which thus can be designed to handle such resources. If, for example, a processing application sees a triple where the predicate is `rdf:type`, it will know that the URI in the subject position identifies a resource which is an *instance* of the object (and as a consequence, the resource in the object position must be a class). Similarly, if it encounters a triple where the predicate is `rdf:type` and the object is `rdf:Alt`, it will know that the subject of the triple in question is a resource that models a group of resources. Consequently it might treat this resource somewhat differently, e.g. it could answer a question which queries the default item in the group. As we will see, much depends on whether the processing applications have ample knowledge about the meaning associated with particular resources.

The Semantic Web is an Internet-based infrastructure facilitating *reasoning*. However, it is often difficult, if not impossible, to reason merely on RDF statements. Let us consider the following example. We are able to describe with RDF that a person is a friend of someone else. We can do this by constructing an RDF statement where the subject and the object are two people and the predicate is a resource about which we *know* that it somehow identifies the "friendship" relation. The RDF triple in question could be the following (assuming that the two people and the predicate are identified by the given URIs):

```
{[U1],[http://swexpld.org/rel#hasFriend],[U2]}
```

Given this triple, can we determine whether one person *knows* the other? (2.2)

For us human beings the answer is trivial, since we know for a fact that everyone knows their own friends, but notice that this is actually an implication for which we use some background knowledge. That is, two people cannot be friends if they do not know each other.

For a computer, stored knowledge is only available as RDF statements. In our example there is only one statement, which merely states that one resource is in an `http://swexpld.org/rel#hasFriend` relation with another. Since the "knows" relation is certainly identified by a *different URI*, the machine will return a negative answer to question (2.2). This is in fact the correct behaviour, since without additional knowledge

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

the computer has no way of knowing what the resources identified by various URIs actually mean (that *different URIs* could easily identify a resource representing "hates").

In the field of artificial intelligence the meaning of various entities is usually determined by formally specifying their relation with other entities. In the present case we would need to be able somehow to determine the relation between the `n:hasFriend` and the `s:knows` properties. We need a statement such as "between any two resources, which are in `n:hasFriend` relation, the `s:knows` relation also holds". With this added knowledge, the computer would be able to figure out that the above two people know each other even though this information would still not have been given explicitly. It is very important that we should be able to specify this background knowledge in RDF, since we have stressed many times that RDF descriptions are fundamental in the Semantic Web. The most obvious solution is to make an RDF statement:

```
{[http://.../rel#hasFriend],
 [V1],
 [http://.../a#knows]}
```

where the intended meaning of the resource identified by the URI `V1` is exactly what we have just described. The only remaining questions are whether there is a resource which has this semantics and whether the resource itself is widely known.

In a different example we might want to formally describe a different relation between properties or classes as part of the background knowledge. It can also be important to enable the specification of various qualities of properties, such as that they are transitive, symmetric etc. Furthermore, we should be able to define our own classes and properties and to define their features. This leads us to the concept of the *RDF Schema*.

So, in a sense, the RDF Schema is nothing other than a handful of resources, with well-defined semantics, extending the base RDF library. These resources can be used in RDF statements, just like those we have already seen. For example, such a resource helps us to describe the relation between the "hasFriend" and "knows" RDF properties in the example above. But the reason why the special resources of RDF schema are not considered as base RDF resources is that their purpose is to describe information at a different *meta-level*. As an analogy, we could mention the difference between the object and model layers in the Unified Modeling Language (UML).

Before we introduce the constructs of the RDF Schema in detail, let us first present an example that demonstrates some possible uses of the schemas.

2.6.2. A simple example

Figure 2.28 shows an RDF description that uses some constructs of the RDF Schema. In this example we define two classes (the second definition uses the simplified syntax): the classes of buildings (1) and of bungalows (2). We also declare that the class "bungalows" is a *subclass* of the class "buildings", i.e. that all bungalows are buildings (3). Furthermore, we define an RDF property (4) which only applies to buildings (5), and its value is an integer (6). Following the conventions, class names have initial capitals while property names begin with lowercase letters. The base URI valid throughout the document is declared at the beginning of the RDF document: thus the URIs referring to parts of the schema are independent of where the schema is stored. For example, the class of buildings may be referred to by the URI `http://swexpld.org/houses#Building`. The `rdfs` namespace is also declared

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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:base="http://swexpld.org/houses">

   <rdf:Description rdf:ID="Building">                    (1)
      <rdf:type rdf:resource=
       "http://www.w3.org/2000/01/rdf-schema#Class"/>
   </rdf:Description>

   <rdfs:Class rdf:ID="Bungalow">                         (2)
      <rdfs:subClassOf rdf:resource="#Building"/>         (3)
   </rdfs:Class>

   <rdf:Property rdf:ID="buildYear">                      (4)
      <rdfs:domain rdf:resource="#Building"/>             (5)
      <rdfs:range rdf:resource=
       "http://www.w3.org/2001/XMLSchema#integer"/>       (6)
   </rdf:Property>
</rdf:RDF>
```

Figure 2.28. A simple RDF schema description.

at the start of the document; this URI is used to access the RDF Schema constructs. The name of the namespace does not have to be `rdfs`, but this is the convention that we will follow throughout this book.

Observe that the example does not contain instances, only class and property definitions. This separation often increases the readability of RDF sources, and we will use this approach in the rest of this book. However, a source often contains both the schema and data-level RDF triples. In either case we get a regular RDF graph, since schema descriptions are also standard RDF triples.

Exercise 2.15: Use an ontology editor to visualise the RDF Schema shown in Figure 2.28. We suggest Protégé [104] for this purpose. Note how the editor displays the subclass relation and the fact that the RDF property `buildYear` is only applicable to the class `Building`.

2.6.3. Classes and their hierarchical relations

A resource can be declared to be a class using the `rdf:type` property. For this, the value (object) of the `rdf:type` property must be the `rdfs:Class` resource (in other words, a class is an instance of the `rdfs:Class` class, i.e. the `rdfs:Class` resource might be considered as a class of classes). The RDF language does not "create" new classes. It

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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:base="http://www.w3.org/2000/03/example/classes">

   <rdf:Description rdf:ID="Animal">
      <rdf:type rdf:resource=
       "http://www.w3.org/2000/01/rdf-schema#Class"/>
   </rdf:Description>
</rdf:RDF>
```

Figure 2.29. The class of animals.

merely allows us to state with regard to a resource that it is a class, just as when we state that a resource is a container, a person or a computer. Figure 2.29 shows an example where a resource is declared to be a class. The class defined this way can be referred to as `#Animal` within the scope of the same base URI. For external access, the full URI must be specified, in this case by `http://www.w3.org/2000/03/example/classes#Animal`.

A class `A` can be declared to be a *subclass* of class `B` using the `rdfs:subClassOf` property. This means that all instances of `A` are also instances of `B`. For example, such a subclass–parent class relation exists between the classes of dogs and animals (all dogs are animals), mobile phones and telephones (all mobile phones are telephones) and squares and rectangles (all squares are rectangles). In RDF a class can have an arbitrary number of parent classes and an arbitrary number of subclasses (also known as child classes). The `rdfs:subClassOf` property is transitive, i.e. if `A` is a subclass of `B` and `B` is a subclass of `C` then `A` is also a subclass of `C`. For example, if we know that cats are mammals and that mammals are animals then we can figure out that cats are animals.

Figure 2.30 shows definitions of the classes of reptiles, mammals or humans. It also demonstrates the use of the `rdfs:subClassOf` and `rdfs:comment` properties. The latter allows the attachment of a short textual comment to any resource (including classes). The example uses the simplified syntax for class definitions.

Figure 2.30 calls attention to a further important detail. A schema, for example the definition of the class of reptiles, can refer to a class defined in another schema. This demonstrates how two schemas can be connected. As a generalisation, a schema might refer to a larger number of schemas, those schemas might refer to yet others, and so on, similarly to how HTML pages refer to each other using hyperlinks. Eventually, schemas can interweave the entire Internet.

The possibility of referring to another schema is very useful. The schema seen in Figure 2.30 reuses an existing class instead of defining another for the same purpose, thus avoiding redundancy and potential inconsistency. It can still happen that independent sources define similar, if not identical, classes without knowing about each other. A goal of the Semantic Web is to create the necessary connections between these classes (e.g. to be able to express the equivalence of two classes in two distinct schemas).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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:base="http://swexpld.org/thing/rdf/schemas/simple">

   <rdfs:Class rdf:ID="Reptile">
      <rdfs:subClassOf rdf:resource=
       "http://www.w3.org/2000/03/example/classes#Animal"/>
      <rdfs:comment>Latin: Reptilia</rdfs:comment>
   </rdfs:Class>

   <rdfs:Class rdf:ID="Mammal">
      <rdfs:subClassOf rdf:resource=
       "http://www.w3.org/2000/03/example/classes#Animal"/>
      <rdfs:comment>Latin: Mammalia</rdfs:comment>
   </rdfs:Class>

   <rdfs:Class rdf:ID="Human">
      <rdfs:comment>Latin: Homo Sapiens</rdfs:comment>
      <rdfs:subClassOf rdf:resource="#Mammal"/>
   </rdfs:Class>

</rdf:RDF>
```

Figure 2.30. Connected RDF schemas.

We have seen earlier that a resource can be declared to belong to a class through instantiation. We also mentioned that RDF instantiation closely resembles the "instance of" relation of the object-oriented universe; therefore the following snippet can be read as follows: Willy is a mammal and his age is 23 years (the snippet refers to the class defined in the schema in Figure 2.30):

```
...
<rdf:Description rdf:ID="Willy">
   <rdf:type rdf:resource=
    "http://swexpld.org/thing/rdf/schemas/simple#Mammal"/>
   <s:age rdf:datatype=
    "http://www.w3.org/2001/XMLSchema#integer">
   23
   </s:age>
</rdf:Description>
...
```

This also specified that the literal value of the `s:age` property describing the age of Willy is an integer.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

Notice that it is possible to do some reasoning about Willy. For example, we are able to determine that Willy is an animal, even though this is not stated explicitly,[11] thanks to the RDF schema. The reason is that we know that Willy is a mammal, and the schema in Figure 2.30 tells us that the class of mammals is a subclass of the class of animals.

In the rest of this book we use the term "class hierarchy" to refer to a directed graph consisting of RDF classes with the arcs of `rdfs:subClassOf` properties stretching between them.

2.6.4. Properties and their hierarchical relations

An RDF property is an instance of the `rdf:Property` class. Accordingly, we may declare a resource to be a property via instantiation. RDF allows us to define a hierarchy of properties just as in the case of classes. This is the purpose of the `rdfs:subPropertyOf` property. Thus, if `A` is a subproperty of `B` then, for all cases where `A` holds between two resources, `B` also holds. The `rdfs:subPropertyOf` property is obviously transitive, just like the `rdfs:subClassOf` property. Comments may also be attached to properties using the `rdfs:comment` property, giving us the ability to write RDF schemas which are easier for humans to understand. This is illustrated in Figure 2.31.

Defined in this way, RDF properties can be freely used in RDF sources as predicates of RDF statements. For example, using them we could express that two people are friends of each other (Figure 2.32). The example refers to two schemas specified earlier. One defined

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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:base="http://swexpld.org/rel">

   <rdf:Description rdf:ID="knows">
      <rdf:type rdf:resource=
       "http://www.w3.org/2000/01/rdf-schema#Property"/>
      <rdfs:comment>Social relation</rdfs:comment>
   </rdf:Description>

   <rdf:Property rdf:ID="hasFriend">
      <rdfs:comment>Close acquaintances</rdfs:comment>
      <rdfs:subPropertyOf rdf:resource="#knows"/>
   </rdf:Property>
</rdf:RDF>
```

Figure 2.31. RDF properties and their hierarchical relations.

[11] More precisely, it is not stated that `Willy` is an instance of the `http://www.w3.org/2000/03/example/classes#Animal` class.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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:n="http://swexpld.org/thing/rdf/schemas/simple#"
  xmlns:s="http://swexpld.org/rel#">

  <n:Human rdf:ID="Eric_Clapton">
    <rdfs:comment>guitarist, AKA "Slowhand"</rdfs:comment>
  </n:Human>

  <n:Human rdf:ID="Mark_Knopfler">
    <rdfs:comment>Dire Straits lead</rdfs:comment>
    <s:hasFriend rdf:resource="#Eric_Clapton"/>
  </n:Human>
</rdf:RDF>
```

Figure 2.32. Using defined RDF properties.

the class of humans (Figure 2.30), the other defined the `hasFriend` property (Figure 2.31). These schemas are identified by qualified names. For example, the `s:hasFriend` qualified name corresponds to the URI `http://swexpld.org/rel#hasFriend`, and this URI indeed identifies the desired property, according to the base URI seen in Figure 2.31. Also notice that this example – both in its syntax and semantics – is an RDF source just like those we encountered before we said anything about schemas. At that time we requested the reader simply to assume that this or that class was identified by the URI used. Such a resource was the class of humans, in the case of the example seen in Figure 2.9. Now we see that we ourselves are able to state with regard to certain resources that they are in fact properties or classes,[12] and we can also define the hierarchical relations between them. The latter is very important since this allows us to deduce that, for example, `Eric Clapton` is in the `s:knows` relation with `Mark Knopfler`, using the RDF information in Figures 2.31 and 2.32. Thus, we finally see that RDF schemas are truly suitable for representing the necessary background knowledge required to answer the question (2.2) posed in the introduction of this Section (see page 98).

2.6.5. Property restrictions

In addition to hierarchical relationships, RDF schemas allow us to express background knowledge by using *property restrictions*. Namely, the *domain* and/or the *range* of a property can be specified using schemas. If, for example, we specify the range of an RDF property to be the class `n:Person` then, for any specific RDF statement referring to this property, the object (value) will be known to be an instance of the `n:Person` class. Similarly, specifying

[12] Furthermore, we are able to do this in a standard RDF/XML syntax; therefore the processing application will also be able to figure out that some resources should be considered as classes.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

the domain of a property implies that the subject of a statement is an instance of the class specified as the domain.

For example, let the domain of the `t:hasStudent` property (which means that the given person is a student of a specified university) be the `p:University` class, and let its range be the `n:Person` class. This definition signifies that the `t:hasStudent` property holds between universities and people: i.e. in the triple

```
{[E1],t:hasStudent,[E2]}
```

the `[E1]` resource is an instance of `p:University` and `[E2]` is an instance of `n:Person`.

RDF intentionally does not determine for what the domain and range information should be used. Certain applications may use this information to look for *inconsistencies* in RDF data. A good example of this is a statement where an instance of the `o:Television` class is connected to someone via a `t:hasStudent` property.[13] Other applications, e.g. interactive RDF editors, may use this information to provide default values during editing, facilitating efficiency. Reasoning engines may use domain and range specifications to figure out indirectly details which are not specified explicitly. For example, even if we do not know anything about the `[E2]` resource, we can reason that it must be a person. In these cases property restrictions are considered as background knowledge, which helps data-level reasoning.

The domain and range of an RDF property can be specified using the `rdfs:domain` and `rdfs:range` properties respectively. In the example in Figure 2.28 we specified that the `buildYear` property connects buildings with integers.

The domain and range of a property can be specified as classes. If the property does not have an `rdfs:domain` specification then its domain defaults to `rdfs:Resource`, which contains all existing resources. In other words, if a property has no domain restriction then it may be applied to resources belonging to arbitrary classes. Similarly, if we do not specify the range of a property then its value could be any resource or literal. A property may have multiple domain (and range) specifications; in such cases its domain (and range) will be the *intersection* of the specified classes. In the following example, a resource may have a `hasMaidenName` property if it is a person and a female at the same time:

```
<rdf:Property rdf:ID="hasMaidenName">
   <rdfs:domain rdf:resource="#Person"/>
   <rdfs:domain rdf:resource="#Female"/>
</rdf:Property>
```

It is worth noticing that multiple property restrictions are special in the sense that they indirectly allow us to speak about classes as intersections of other classes. There is no other way to do that in RDF. One might think that this can be achieved using an intermediate class, defined as a subclass of both the `Person` and `Female` classes. However, this would not represent the exact intersection class, but only a class which is a *subset* of the intersection.

[13] One must be very cautious in this case, since it may be that there is no contradiction even though the given resource is specified to be an instance of the class different from `p:University`. For example, if we have `q:Institute` instead of `o:Television` and we know that universities are institutes, then there is no contradiction.

A URI identifying a data type may also be specified as a range restriction, e.g. when we expect a property to connect people with numbers. However, when using such a property we still need to specify the type of the value using the `rdf:datatype` attribute.

The `rdfs:domain` and `rdfs:range` properties can also be used to specify formally the restrictions of other built-in RDF properties. For example, the domain and range of the `rdfs:subPropertyOf` property are both `rdf:Property` class resources. This is a formal way of saying that the subproperty relation holds between properties. Similarly, the domain and range of the `rdfs:subClassOf` property are both `rdfs:Class` resources. Consequently the subclass relation may exist between resources which are instances of the `rdfs:Class` class (in short, between classes). The `rdfs:domain` and `rdfs:range` properties also have a domain and a range. The domain of these properties is `rdf:Property`, and their range is `rdfs:Class`.

We can describe so-called *global* property restrictions using the `rdfs:domain` and `rdfs:range` properties. They are global because, once specified, the domain and/or range of a property are unchangeable and hold under all circumstances for that property.

Global property restrictions have some so far unmentioned drawbacks. For example, we might want to specify that the range of the `n:size` property must be the set of integers when the property is applied to instances of the class `c:Shoe`. However, we cannot declare that the value of the *same* property should be a literal when it is applied to T-shirts (e.g. one of the literals `S`, `M`, `L`, `XL`). Similarly, it would be beneficial to be able to say that if the value of a property is of a certain type or larger than a particular value then it can be applied only to instances of a given class. Generally speaking, there is no way to obtain domain-dependent range restrictions from global property restrictions; furthermore, there is no way to make domain restrictions depend on the value of the property.

The so-called *local* property restrictions, which are included in several schema languages that are richer than RDF schema, offer a solution to these problems. Such a language is OWL, which will be described in Chapter 8. The OWL language also provides various class construction operators missing in RDF including class intersection, discussed above.

Exercise 2.16: To what type of resources can we apply the property `p2` using the following (stand alone) RDF description?

```
...
  <rdfs:Property rdf:ID="p1">
    <rdfs:domain rdf:resource="#Person"/>
  </rdfs:Property>

  <rdfs:Property rdf:ID="p2">
    <rdfs:subPropertyOf rdf:resource="#p1">
    <rdfs:domain rdf:resource="#Tall"/>
  </rdfs:Property>
...
```

---

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.6.6. Other RDF schema constructs

The RDF schema language provides a handful of generic resources in addition to the resources used for building class hierarchies and specifying property restrictions. The `rdfs:comment` property has already been used in an earlier example. This property allows us to attach a textual comment to an arbitrary resource. This can improve the readability for humans.

The `rdfs:label` property can be used to declare the verbose name of a resource in a way suited to human readers. The following example demonstrates how we can present for humans the resource identified by `http://swexpld.org/tantras/p12` as Richard Roe.

```
<rdf:Description rdf:about="http://swexpld.org/tantras/p12">
  <rdfs:label>Richard Roe</rdfs:label>
  <n:hasEyeColour
         rdf:resource="http://swexpld.org/colours#brown"/>
</rdf:Description>
```

The `rdfs:isDefinedBy` property helps us to specify an additional resource which more precisely defines the resource used as the subject of a statement (either formally or in plain text). An example occurs when we refer to a Wikipedia page to specify more elaborately the class `Human`. Additional information can also be linked to resources using the `rdfs:seeAlso` property. The full set of elements defined by RDF schema can be found in its specification [22].

Exercise 2.17: Design an RDF schema covering your own topic of interest. Try to use as many different RDF schema constructs as possible: class and property hierarchies, property restrictions, comments and labels. Make sure that your ontology also contains instances, i.e. data-level RDF statements such as `Bill loves Mary`. Load your RDF description into an ontology editor, Protégé [104] for example, to visualise it.

2.6.7. RDF schemas and the object-oriented world

This subsection compares the class, object, property and miscellaneous other concepts of the RDF schema language with their rough equivalents in the object-oriented world (such as UML).

2.6.7.1. Property as a primary modelling primitive

At first glance, UML shows many similarities with the RDF Schema language. Both have a concept of classes, and these classes can have instances. One of the most important differences is that there is no real UML equivalent of the property–predicate–relation concept in RDF schema,[14] even though it closely resembles UML associations and attributes.

RDF properties have an interesting character in that they are bound to resources only at "run time". That is, we define the properties with respect to classes which contain the instances to which the given property might be applied, and not vice versa. This is in contrast

[14] UML Standard 2.0 appears to make this gap smaller.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

with the object-oriented world, where one has to decide at *modelling time* what attributes and associations a given class can have.

The following example is an attempt to demonstrate what we have said so far. Let us consider an RDF property `publishedBy`, without domain restriction and with `Publisher` as its range. This means that the `publishedBy` property connects two objects, the first of which can be practically anything and the second of which has to be a `Publisher`. For example, we can apply this property to an instance of the class `Book`. An object-oriented system, however (if we forget about associations for now), would first define the class of `Book`s, which would have an attribute called `publishedBy`, of the type `Publisher`.

This difference stems from the distinct fundamental philosophies of modelling languages. On the one hand, in an object-oriented system, such as UML, in accordance with the object-oriented approach the primary modelling primitive is the class. On the other hand, property is the *primary modelling primitive* in RDF. The latter is not the only property-oriented language: there are also DAML+OIL and OWL, presented in Chapter 8, and numerous further knowledge-representation languages. These property-oriented languages model the world as if it consists of instances which have various characteristics. More precisely, an instance can be related to other instances, and these relations are the information providers about the instances themselves. Classes are "created" by specifying what common features characterise their respective instances. For example, the class of parents could be defined as follows: *a parent is someone who has at least one child*. After that, all instances that are known to have a child (i.e. they are on the appropriate side of a `hasParent`/`hasChild` relation in another instance) are also known to be *parent*s.

These property-oriented languages are very efficient when it comes to describing instances, since, contrary to the approach of object-oriented systems, there is no need to "allocate" all attributes of an instance at instantiation time. In the previous example, it is not necessary to store an array of children for each instance. We do not even know in the first place whether the given instance is a *human*, i.e. whether it can have children at all. It is possible that the particular instance is a flower, and that later someone will specify its colour.

This is closely connected to the other major benefit of the property-oriented languages (and thus RDF), namely that, rather like humans, they are able to efficiently store "knowledge crumbs" and perform reasoning on these. As we said, instances can be categorised into classes only through their attached properties: there is nothing to know about an instance in itself except its identifier. In the object-oriented world, however, an object can only exist as an instance of a specific class (or several classes), and this has two implications. First, the instance has only those and exactly those properties (attributes) which were declared with foresight during the definition of the class. This is fairly inflexible, since how are we able to place in memory a piece of *unexpected but potentially useful information*, such as that someone's favourite food is fried chicken, if there is no predetermined attribute in which to store it? Furthermore, when an object "appears", we must be able to categorise it, i.e. recognise to which class the instance belongs.

In many cases, however, there is not enough information available to do this categorisation. If, for example, we learn that an existing RDF resource (instance) `loves` another resource then the question arises whether we can recognise to which class this other resource belongs. The answer is no, since even if we know that the first resource represents a person, the second resource could still be a piece of music, a pet, a car, a book or another human

being. Later, we might learn that the second resource has a child. This detail can be associated with the resource without any problem: furthermore, we could reason that this resource must be a *parent* and, if we also know that parents are people, then we know that the second resource is a *person*.

It should be clear that reasoning is often necessary to tell to which class a particular instance belongs. Accordingly, it is computationally more expensive to determine the class of an instance in a property-oriented language than in UML (this is a price to be paid for flexibility). In UML this class information is always explicitly present while in RDF, as we have seen, a resource can be a member of the class of humans without being explicitly declared as such.

Note that, in property-oriented languages which allow local property restrictions (such as DAML+OIL and OWL), one can be very "economical" with properties. The reason for this is that the same property can be used in multiple "roles"; for example, `size` could characterise a pair of shoes, a piece of clothing, a flat screen display etc. In UML an attribute or an association is always linked to a class and, even though two classes could have identically named attributes, these are always stored and treated separately by object-oriented systems.

2.6.7.2. Property-based approach and the web

The processing of the above-mentioned "unexpected but useful knowledge" is analogous to the fundamental notion of the Semantic Web, that is the association of meta-information with resources. According to what we have said so far, if we want to state that the server (resource, object) identified by the `http://128.30.52.45` URI[15] was made by IBM then this can be done even if the given object does not have an attribute named "brand". The corresponding RDF description can be seen in Figure 2.33.

In summary, property-oriented languages are much more suitable for Internet, a system famous for its flexibility and dynamics, than the more static object-oriented systems.

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ns="http://swexpld.org/useful/thing#">

   <rdf:Description about="http://128.30.52.45">
      <rdf:type rdf:resource=
      "http://swexpld.org/computers#Server"/>
      <ns:brand>IBM</ns:brand>
   </rdf:Description>

</rdf:RDF>
```

Figure 2.33. The brand of the given computer is IBM.

[15] An IP address without the `http://` prefix is not a URI (not even a relative one) according to the specification; therefore it cannot be used as an RDF resource identifier.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

2.7. Examples of RDF use

In the following, we present a few examples of the use of RDF sources in real, practical, applications. First we introduce the widely known and acknowledged web-based concept vocabulary called Dublin Core Metadata Initiative, which was created before the RDF era but is now accessible in RDF Schema form as well. This is followed by the description of a very promising, freely accessible, web catalogue, the Open Directory Project, which publishes its data in RDF form. Next we introduce the MusicBrainz meta-database, which was designed for the storage of additional information (CD track titles, album title etc.) about pieces of music and for serving this type of data to clients. The client–server communication is done via an RDF-based interface. Subsequently, we briefly describe the RDF Schema called RDF Site Summary, which can be used to describe information in web pages offered for other peers. Finally we introduce a web-based thesaurus which answers queries using RDF descriptions, automatically associating a URI to tens of thousands of expressions.

2.7.1. Dublin Core

For RDF to be used in a particular field one needs a commonly accepted RDF schema, describing resources which are used by all RDF sources in that field. RDF schemas give a means for everyone to prepare their own vocabularies, but this raises several problems. First, it is clear that it is inefficient if everyone reinvents the wheel and defines the properties of basic concepts such as author, address and name. Instead, a shared vocabulary accepted by all should be created. This can then be extended by specific local elements. For instance, one can declare a class called `Chief Executive Officer` (CEO) and state that it is a subclass of `Person`, which is acknowledged by everyone. As a consequence, those RDF-processing clients which know about this commonly accepted schema can reason about Peter, whom we declare to be a CEO, even though these clients do not know anything about chief executive officers.

Vocabularies accepted by many people have to be at a fairly high level. This means that they define generic things, thus providing building blocks for constructing lower-level schemas. The Dublin Core [32] is an initiative defining such high-level concepts. Using RDF terminology, Dublin Core is an RDF Schema which mainly defines properties.

The history of Dublin Core reaches back before the dawn of the RDF era. Dublin Core was constructed in 1995, on a workshop on metadata held in Dublin, Ohio. The goal was to create an element set which facilitates the automated discovery of electronic documents and resources in general [109]. The result was a collection of 13 entries, which describe certain characteristics of documents. This set was later slightly rearranged and extended by two further elements. This set is the foundation of the ISO Standard 15836 from 2003 as well. The 15 elements are the following:

Title: The title of the selected resource.

Creator: The name of the entity which is responsible for creating the contents of the resource. This can be a person, an organisation or a service.

Subject: The topic of the contents of the resource. Generally this is described by keywords, but the recommended procedure is to use a code defined by some sort of classification system.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

Description: The summary of the contents of the resource. Typically this is free text, perhaps an abstract or the table of contents.

Publisher: The name of the entity which is responsible for the accessibility of the resource. This can be a person, an organisation or a service.

Contributor: The name of the entity which contributed to the preparation of the contents of the resource. It can be a person, an organisation or a service.

Date: The date of an event which is in some way connected to the resource, typically the date of preparation. It is recommended that the format of the date adheres to the ISO 8601 Standard.

Type: The type of resource or its contents (i.e. image, movie, service, software etc.). It is recommended that a word from a commonly accepted vocabulary is used here.

Format: The physical parameters of the resource. Examples are the size, length or MIME type of the resource.

Identifier: The unambiguous identifier of the resource. Typically URIs are used, but other unique identifiers such as the ISBN numbers of books may be given here as well.

Source: Reference to a resource during used as a source at the preparation of the contents.

Language: The language of the contents of the resource.

Relation: Reference to a resource which is in some way related to the selected resource.

Coverage: The coverage, i.e. scope, of the resource, often in the sense of geographical coverage.

Rights: Information about user and other rights of the resource and its contents.

The main benefit of an easily understandable element set like this is that through its simplicity it promotes its own spread and use. Nowadays many applications use the above fields to provide meta-information about several resources.

The goal of the Dublin Core vocabulary is not to cover all fields but, as mentioned earlier, to fulfil the role of a high-level vocabulary. Various communities might require new elements; furthermore, they would probably like to connect these to the existing Dublin Core elements in a standard manner.

This is greatly aided by the XML and RDF schema form of the Dublin Core vocabulary. The XML schema supports the use of Dublin Core elements in arbitrary XML documents. In Subsection 2.7.4 we will give an example of an XML file containing meta-information, published on a well-known web page, in which certain properties were defined using Dublin Core elements.

The RDF schema of the Dublin Core, however, offers even more ways of usage with respect to the RDF framework. Within an RDF description, Dublin Core elements can be used as building blocks in the definition of one's own RDF classes and properties, by simply referring to the RDF resources of the Dublin Core RDF schema. However, in the most widespread usage one simply refers to Dublin Core elements as RDF properties in RDF documents. An example of the latter can be seen in the RDF source shown in Figure 2.34, which gives meta-information about a file stored on a computer.

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/">
   <rdf:Description rdf:about=
                        "http://last.fm/...money.mp3">
      <dc:creator>Dire Straits</dc:creator>
      <dc:title>Money for Nothing</dc:title>
      <dc:description>
      The theme song of the album "Money for Nothing"
      </dc:description>
      <dc:date>1987-11-30</dc:date>
   </rdf:Description>
</rdf:RDF>
```

Figure 2.34. Using Dublin Core in an RDF source.

2.7.2. Open Directory Project

The *Open Directory Project* (ODP) [89] is a web catalogue. Web catalogues were described in detail in Subsection 1.4.4. The ODP differs significantly from other catalogues in that it is not affiliated to any profit-oriented company; the editorial and maintenance work is performed by unpaid volunteers and so the entire contents of the catalogue are freely accessible and usable.

The inclusion of one's own web site can be requested from the ODP editors by filling in a form, naming the appropriate category and providing a short summary of the site contents. If it is accepted, the site will be listed in the catalogue within a couple of weeks and in roughly the same time it will most probably also appear in the result pages of various search engines. This is due to the fact that many large non-catalogue-based engines use the data of the Open Directory Project as a starting point for their crawlers, and some offer it directly, in catalogue form, as a secondary service. Google, AOL Search and Excite are all examples of such engines.

The ODP editors are volunteers responsible for a specific field extensively known by them. Within this field, using their own experience they add new sites to the catalogue and edit and refine the descriptions of already listed sites. It is also the duty of the editors to review requests for the addition of new sites to the catalogue. Such an addition has very serious quality requirements: the editors of the ODP are very proud that their catalogue lists only particularly useful sites.

Anyone can become an editor. All one needs to do is to fill in an application form, where, apart from answering a couple of questions and specifying the field of expertise (a category), the applicant must list two or three web sites which they would add to that category. Applications are reviewed by more experienced editors, among other factors on the basis of how relevant the listed web sites are in that field. Editors have a great responsibility, so at first they work on a smaller category and only later receive the credentials to rename or move categories, create new ones etc. A number of guidelines have been created to support the

editorial work, which are readable by everyone (thus one can see, for example, what considerations apply when one is reviewing a site addition request, what it is necessary to pay particular attention to etc.)

The ODP data can be downloaded in RDF form by anyone. However, one needs to be prepared for the huge size of the catalogue. The smaller of the two most important data files describes the structure of the ODP, the hierarchical layout and the characteristics of the individual categories; the size of this RDF source exceeds 300 megabytes. The larger of the two data files lists the web sites themselves, along with their descriptions, ordered into categories. This file is larger than 1 gigabyte. It first lists the various sites for each categories and then details the sites themselves. The following example shows part of the `Top/Arts/Movies/Titles/1984_-_1984` category, which lists links to pages about the film version of the famous novel by George Orwell.

```
...
<Topic r:id="Top/Arts/Movies/Titles/1/1984_-_1984">
  <catid>460423</catid>
  <link r:resource="http://.../aaronbcaldwell/1984.html"/>
  <link r:resource="http://orwell.ru/.../m84_01.htm"/>
  <link r:resource="http://.../.../filmography/014.html"/>
  <link r:resource="http://...batcave.net/1984.htm"/>
  <link r:resource=
            "http://www.filmtracks.com/titles/1984.html"/>
  <link r:resource="http://www.imdb.com/title/tt0087803/"/>
</Topic>

<ExternalPage
  about="http://.../aaronbcaldwell/1984.html">
  <d:Title>Top 100 Movie Lists: 1984</d:Title>
  <d:Description>Photos, sounds...</d:Description>
  <topic>Top/Arts/Movies/Titles/1/1984_-_1984</topic>
</ExternalPage>

<ExternalPage
   about="http://orwell.ru/a_life/movies/m84_01.htm">
  <d:Title>George Orwell's Movies - 1984</d:Title>
  <d:Description>Review.</d:Description>
  <topic>Top/Arts/Movies/Titles/1/1984_-_1984</topic>
</ExternalPage>
...
```

Both RDF files use the Dublin Core schema. In the above example, the `d:Title` URI refers to the `Title` property of the Dublin Core schema.[16]

The RDF sources of ODP (in most cases the RDF source describing the categories) are often converted into various other formats, e.g. into HTML or a relational database. Applications performing these conversions can be found readily on the web.

[16] The Open Directory Project uses version 1.0 of the Dublin Core, for some reason, in which property names begin with capital letters.

2.7.3. MusicBrainz

MusicBrainz (MB) is a musical meta-database edited by volunteers. It can be accessed either on the web (`http://musicbrainz.org/`) or by using some client application. Many CD and audio players are able to connect to the servers of MB, when the user inserts a CD into the drive or starts playing an MP3 file which does not contain any tags, and to request the necessary information. This information usually contains the title of the CD tracks and the album itself, the genre of the song etc. In the case of CDs, the identification is fairly simple because of the characteristics of the CD format. For MP3 and Ogg Vorbis files, identification is performed using a digital fingerprint of the file, which is extracted from a number of attributes (length, frequency distribution etc.).

Here we discuss briefly an earlier implementation of MusicBrainz (see `http://wiki.musicbrainz.org/RDF`) based on RDF. The present implementation uses relational databases and also makes the information available using a music ontology [97].

The RDF-based version of MusicBrainz defines three RDF schemas named `mm`, `mq` and `mem`.

The `mm` schema defines the basic classes and properties. It declares three classes, `Artist`, `Album` and `Track`. Furthermore, it defines exactly one dozen RDF properties such as `shortName`, `trackNum` (i.e. track number), `duration` etc. Some properties expect containers for values. For example, the value of `albumList` is an `rdf:Bag`, which contains all the albums of the given artist or band.

The `mq` schema defines those RDF classes and properties that are required for the questions proposed during a conversation between the MusicBrainz server and its clients. An instance of the `Result` class represents a specific answer, and the instances of the various query classes can be used to look for tracks, CDs, albums, artists etc. The following example issues a search for the band U2 and also specifies the search depth (which allows the client to specify the level of information that should be returned by the server):

```
<mq:FindArtist>
   <mq:depth>2</mq:depth>
   <mq:artistName>U2</mq:artistName>
</mq:FindArtist>
```

The above object is in fact an anonymous resource. The type of this resource is `mq:FindArtist`, and it has two properties. As an answer for this query, we receive an anonymous instance of `mq:Result` in RDF form, which informs us that the search was successful and that there are seven hits (the string U2 was found in the name of seven different formations, as in "Massive Attack and U2"):

```
<mq:Result>
 <mq:status>OK</mq:status>
 <mm:artistList>
  <rdf:Bag>
    <rdf:li rdf:resource=
     "http://musicbrainz.org/artist/a3cb23fc-acd3..."/>
    <rdf:li rdf:resource=
     "http://musicbrainz.org/artist/8ba42c7f-1ac8..."/>
    ...
```

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
  </rdf:Bag>
 </mm:artistList>
 </mq:Result>
```

The answer also contains the albums recorded by the various formations and the details of the songs on each such album.

The mem schema was set aside for future development; we will find properties here which can be used to attach lyrics to individual tracks and sound files.

According to the above, an RDF source using the schemas offered by MusicBrainz could look like this (this source does not use the mem schema but it does use the Dublin Core vocabulary):

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc  = "http://purl.org/dc/elements/1.1/"
    xmlns:mq  = "http://musicbrainz.org/.../mq-1.1#"
    xmlns:mm  = "http://musicbrainz.org/.../mm-2.1#">

 <mm:Artist rdf:about=
    "http://musicbrainz.org/artist/23d8426c-18c7...">
    <dc:title>Tangerine Dream</dc:title>
    <mm:sortName>Tangerine Dream</mm:sortName>
  </mm:Artist>
...
```

2.7.4. RSS: RDF site summary

Nowadays, so-called *web syndication* is the most widespread use of RDF. By syndication we mean that some kind of data is made available for external users. In television programming, it is often the case that various soap operas are not directly produced or ordered by the network; instead the broadcaster buys and airs some ready-made series. The creators of such television series expect that the television companies will buy their products, and so they offer them for buying. Similarly, the comic strips appearing in columns of daily newspapers are usually drawn not by the employees of the newspaper but by specialised studios and artist groups, from whom several newspapers buy their daily strips.

In web syndication parts of a web portal, normally including the most important news and events of the site, are made available for others, usually in XML format. These descriptions then are collected by RSS-reader client applications and presented to the user at the user's request. In this way, somebody can easily read the headlines of the web pages in which she is interested without the need to actually visit these sites one by one via her browser.

The most commonly used web syndication format is *RSS*, which has almost a dozen, partially incompatible, versions (the other popular format is *Atom*, supported by Google). In some versions RSS stands for *rich site summary*; elsewhere it is an acronym of *RDF site summary* or of *really simple syndication*.

Version 1.0 of RSS uses an RDF "dialect" similar to the first widely used version, 0.9, which was introduced by Netscape in 1999. Interestingly, the creator of RSS, Dan Libby, originally proposed a format very similar to version 1.0, but Netscape for some reason did

```
<rdf:RDF ...>
 <channel rdf:about="http://hardware.slashdot.org/">
  <title>Slashdot: Hardware</title>
  <link>http://hardware.slashdot.org/</link>
  <description>News for nerds, stuff that matters
  </description>
  <dc:language>en-us</dc:language>
  <dc:rights>Copyright 1997-2006, OSTG ...</dc:rights>
  <dc:date>2006-07-26T17:40:59+00:00</dc:date>
  <dc:publisher>OSTG</dc:publisher>
  <dc:creator>pater@slashdot.org</dc:creator>
  <dc:subject>Technology</dc:subject>
  <syn:updatePeriod>hourly</syn:updatePeriod>
  <syn:updateFrequency>1</syn:updateFrequency>
  ...
  <item rdf:about="http://hardware.slashdot.org/...">
   <title>Output Mouse</title>
   <link>http://rss.slashdot.org/...</link>
   <description>An anonymous reader writes "Combining
   unusual items together can sometimes produce interesting
   results. Over at MetkuMods one can find a computer mouse
   embedded with a cell phone display to show all sorts of
   information. For some people one TFT on the desktop just
   isn't enough."</description>
   <dc:creator>ScuttleMonkey</dc:creator>
   <dc:date>2006-07-25T22:57:00+00:00</dc:date>
   <dc:subject>inputdev</dc:subject>
   ...
  </item>
...
```

Figure 2.35. RSS in RDF form.

not support his proposal until later. ScriptingNews, the precursor of RSS, was designed by Dave Winer. It is he who has also been responsible for the more recent versions of RSS.

In spite of the differences, all versions agree in that they use XML syntax. Figure 2.35 shows a snippet of the RSS metadata published by the hardware section of the well-known nerd news syndication site Slashdot (`http://slashdot.org`) in RDF/XML form (using the default namespace). The example shows the most important details of the site, the title of a news entry, a link to the full entry and a short summary of its contents.

An RSS file can be linked from a page, and even referred to from the head of the HTML page, as follows (in this way RSS readers can easily reach it):

```
<link rel="alternate" type="application/rss+xml" title="RSS"
 href="http://www.w3.org/2000/08/w3c-synd/home.rss" />
```

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

The RDF version of RSS is an RDF Schema, which defines classes and properties that can be used in the definition of RDF data. The RDF variant, just like basic XML, can easily be extended later. For example, one schema defines the following three properties, where the `syn` namespace is associated with the URI `http://purl.org/rss/1.0/modules/syndication/` URI:

```
syn:updatePeriod
syn:updateFrequency
syn:updateBase
```

These properties are used to inform web crawlers collecting RSS data how often the published data are updated. Such RSS crawlers, for example, are RSSOwl [100] or AmphetaDesk [51]. Web syndication can be considered analogous to newsgroups, where subscribers regularly receive news updates in groups to which they have subscribed. In our case the users can select certain web sites which are regularly visited by their RSS crawlers, which then present the collected information in some, usually very stylish, format to them.

2.7.5. Wordnet

Wordnet [35] is a freely available online lexical reference system, which was developed in the Cognitive Science Laboratory at Princeton University. Wordnet contains more than a hundred thousand expressions, and it is unique in the sense that it also describes certain relations between words. If we search for the word *cat*, we first see that it has eight meanings as a noun plus two as a verb. Then we can ask, among other things, what parts a cat has (claws, fur...), to which taxonomy groups the cat belongs (feline, mammal...), and what kinds of cat there are (Persian, Egyptian...). Similarly, in the case of adjectives we can see that the antonym of high is low, the antonym of pretty is ugly etc. We can look up synonyms, words with similar meanings and many other things.

In the introduction to the Dublin Core schema in Subsection 2.7.1 we mentioned that one of the most difficult tasks when creating an RDF source is to use classes and properties accepted by the processing applications as well. What makes Wordnet particularly interesting for us is that it tries to provide a common terminology and, based on it, we may build RDF descriptions understood by others in the same way as by us.

As an experimental service, some Wordnet nouns have been associated with URIs. Different versions of Wordnet use different URIs. For example, version 1.6 uses the form of `http://xmlns.com/wordnet/1.6/XXX`, which identifies the entity described by `XXX`, where `XXX` must be an English noun listed in Wordnet. Wordnet 3 uses URIs of the form `http://purl.org/vocabularies/princeton/wn30/synset-XXX-noun-N`, where the number `N` distinguishes the different uses of the same word.

For example, a human being can be identified by the URI `http://purl.org/vocabularies/princeton/wn30/synset-person-noun-1.rdf`. This means that one can replace the probably undefined URI that we used in Figure 2.10 by the above link. The result can be seen in Figure 2.36. In this way, even if a particular processing application does not understand the `s:name` and `s:address` properties, it can easily notice that the given resource identifies a person.

Furthermore, even though we said that URIs do not necessarily denote actual downloadable documents in RDF, Wordnet URIs in fact do this! For example, the above

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://swexpld.org/utils#">

 <rdf:Description rdf:about=
  "http://freemail.org/~doe/#about">
    <s:name>John Doe</s:name>
    <s:address rdf:resource="mailto:johndoe@freemail.org"/>
    <rdf:type rdf:resource="http://purl.org/vocabularies/
                synset/princeton/wn30/person-noun-1.rdf"/>
 </rdf:Description>

</rdf:RDF>
```

Figure 2.36. Using a Wordnet concept in an RDF source.

link `http://purl.org/.../wn30/person-noun-1.rdf`. leads to an actual RDF document describing the notion of "person".

2.8. Summary

In this chapter we have presented the concept of the Semantic Web, the basic idea of which is to associate *meta-information* with Internet-based *resources* and to perform reasoning on them. We detailed the RDF and the RDF schema languages, the use of which is a standardised and recommended way of presenting meta-information. Below we summarise the most important points of the chapter.

- Describing the *meaning* of the information on the web is normally done through associating meta-information with it. Meta-information is present in the web languages; the HTML element called `META` is, for example, specifically used for providing meta-information although its usability is rather restricted. It is an important requirement in the Semantic Web approach that meta-information can be linked to an arbitrary resource in a uniform way that is easily processable by computers. Such a resource may be anything that can be identified uniquely.
- To support computer processing, meta-information is stored in *XML form*. An XML document is a text file designed to be capable of storing data in a structured form. The most important parts of an XML document are the *elements* and the *attributes* linked to them. An XML element may be complex, mixed or simple. Every XML document contains at least one element, the *root*. It is another feature of XML documents that they store the data in a hierarchical structure; a given document defines a *tree*, the root of which is the root element.

  The element and attribute names used in an XML description are only unique within the given document; these are called *local names*. If one merges several XML documents

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

then the local names could clash, which might cause problems. Such problems can be eliminated by the use of XML *namespaces*. This means that the element and attribute names used in the document will be given a prefix. The prefixes are URIs; a local name supplemented by a prefix is called a *qualified name*. The only role of the URIs is to function as unique names.

One of XML's strong points is that it allows the user to check whether an XML document is an instance of the given language. This requires a standardised description of the *grammar* itself, which is made possible by the *XML schema* language. An XML schema describes the elements and attributes which can be used in XML documents belonging to it, as well as the contents of the elements and the relations between them.

Documents in XML in most cases focus on the contents; they do not contain formatting information. Presenting XML documents is done by converting them into a form suitable for human consumption. This conversion is carried out by the use of an appropriate *transformation language*, the XSLT, for example. In this way the data and their presentation can be separated. The CSS style sheets related to XHTML pages serve a similar purpose.

By itself, XML is not suitable for supporting communication between applications without prior agreement on the meaning of the language used during the transfer. The aim of the Semantic Web and the related RDF concept is, among other things, to eliminate the need for such a prior agreement. The *RDF language* is capable of linking metadata to any *URI-identified* resources. A fundamental role is played by URIs in the Semantic Web approach; they make it possible to *formulate assertions* unambiguously and to combine fragments of meta-information coming from different sources.

- The basic idea of the RDF language is to connect URI-identified resources with other resources or just with plain *literals* using *properties*. Essentially, RDF helps to describe binary relations between things identified in a given way. Accordingly, *RDF statements*, also known as *triples*, consist of a subject, a predicate and an object. An RDF description is a set of such triples. The meaning of *description* is that the statements it contains are true.
- Using RDF triples one can also build a *graph* where the nodes represent the subjects and objects appearing in descriptions, while the edges represent predicates. Identical resources, i.e. those referred to by the same URI, are denoted by the same node; this is the way in which the various component assertions make an organic whole. The RDF graph may also contain blank nodes representing resources to which no URIs are associated, for example because they were unknown at the time when the description was made. Blank nodes are also often used to make the given information better structured.
- The RDF language has multiple representations; among these the XML form is the most widely used. We therefore presented the XML format of RDF first. We introduced descriptions involving a *shared subject* as well as *intermediate* and *object-positioned* resources. We also showed the use of *relative URIs* in XML form and introduced the `rdf:ID` attribute.
- Subsequently, we dealt with the various special RDF constructs in turn. In the RDF language, non-binary relations can be represented by using the `rdf:value` property. With the help of the `rdf:type` property one can state that a resource is an instance of

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

an RDF class. The *simplified XML form* of instantiation has the advantage that it strongly resembles the syntax of pure XML representation. This, too, may help to achieve in the near future the replacement of the XML-based storage of data in certain cases, by RDF-based representation.

- RDF has a few built-in facilities which can be used to create instances. One built-in class, for example, contains the instances of *higher-order statements*. We consider an RDF statement to be of higher order if it speaks about another statement. In order for it to do this the RDF statement should be modelled as a specific resource. This process is called *reification*, and the specific resource is a *reified statement*.
- Our second example of the use of built-in classes involved *containers*. RDF supports the use of three types of container classes, identified by the following URIs: `rdf:Bag`, `rdf:Seq` and `rdf:Alt`. RDF containers are *open*, i.e. the user can state only that certain resources are elements of a container. RDF also makes it possible to describe an aggregation of resources which is *closed*, i.e. it contains only the elements supplied. Such an aggregation is called an *RDF collection*, which, in fact, is a list.
- At the end of our presentation of the RDF language, we showed how to use *typed literals* in RDF descriptions.
- The RDF language alone is not sufficient to support reasoning during Internet queries, because it cannot express meta-knowledge. For example, using plain RDF it is not possible to find someone's friends when we are looking for whom that person knows, because the inference system does not have the knowledge that friends know each other. This problem is solved by introducing *RDF schema*, in which the meaning of a resource is made more specific by formally describing its relationship with other entities. For example one can declare that, between any two resources for which the `friendOf` relation holds the `knows` relation also holds.

  At first glance, RDF schema looks like nothing more than a few new resources, with well-defined meaning, added to the vocabulary of RDF. These resources can be used in RDF statements (e.g. for instantiation) in exactly the same way as before. So, it is important that no new language is introduced: the RDF notation is used. Looking more closely, however, one notices that the RDF schema extension allows users to define their "own" application-specific classes and properties as well as to describe the features of these resources and to specify the hierarchical relationships between them.
- A user-specific class can be defined by instantiating `rdfs:Class`. By using the property `rdfs:subClassOf`, one can state that class `A` is a subclass of class `B`. This means that any instance of `A` is an instance of `B` as well.
- A new property can be defined as an instance of the `rdf:Property` class. As in the case of classes, RDF schema allows hierarchical relations between properties. Such specifications can be made by using the `rdfs:subPropertyOf` property. RDF schema also supports the description of *data-level relations* between classes and properties. Namely, by using schemas one can describe the *domain* and *range* of a property. This information can be used for consistency checking as well as for reasoning or even to make the behaviour of an RDF editor more intelligent.

https://doi.org/10.1017/CBO9781139194129.003 Published online by Cambridge University Press

- The concepts used by RDF schema show similarities to those in object-oriented languages. RDF properties, however, have the interesting feature that they are only linked to resources at "run time". In object-oriented systems a class definition contains all the attributes of the objects which are created by the instantiation of the class. In contrast, in the world of RDF a property is defined in terms of the classes to which it is applicable, while a class is specified by the features of its elements. This shows that property-oriented languages such as RDF schema can nicely accommodate the anyone-can-say-anything philosophy of the Semantic Web.
- Finally, we presented a few real applications where the RDF language plays an important part. The first was the widely used Dublin Core dictionary, which is available as an RDF schema as well. We next presented one of the most popular web catalogues, the Open Directory Project, the data of which can be downloaded in RDF form. Then we examined a musical meta-database named MusicBrainz, which stores music-related data in RDF form and offers an RDF query interface. After that we described the RSS, one of the most widespread formats of web-based syndication, which has both XML and RDF versions. Finally, we briefly introduced the free Wordnet dictionary. As part of an experimental service the entities linked to the nouns in Wordnet are associated with URIs, which can be used in RDF descriptions.