Validating RDF Parser (VRP) – Analyzing and Parsing RDF.

Karsten Tolle

Technical Report FORTH-ICS-TR-270

Heraklion, April 2000

Table of Contents

1 Introduction....7
1.1 How to read this thesis?....8
2 RDF Presentation....9
2.1 History of RDF....9
2.2 Design objectives and goals of RDF....9
2.3 The notion of RDF resources....9
2.4 RDF M&S (data model)....10
2.4.1 RDF Graph....10
2.4.2 RDF in 3-tuples....13
2.4.3 RDF Syntax (XML)....13
2.5 RDF Schema....23
2.5.1 Classes in RDF Schema....24
2.5.2 Properties in RDF Schema....25
2.5.3 Errors in the RDF Schema....26
2.6 Pending Issues of RDF & RDFS....27
3 Towards an RDF Formalization....29
3.1 Definitions....29
3.2 Type Inheritance....30
3.3 Implicit RDF Instantiation....31
3.3.1 rdfs:ContainerMembershipProperties....31
3.3.2 rdfs:Resource....31
3.3.3 rdfs:Literal....32
3.4 RDF Constraints....32
3.5 Existing RDF Parsing Software....37
3.5.1 SiRPAC....37
3.5.2 Pro Solutions Perl Parser....38
3.5.3 New Ways?....39
4 Architecture and Implementation of VRP....41
4.1 Introduction to VRP....41
4.2 Evolution of VRP....41
4.3 The Java Packages of VRP....42
4.4 The Parser....44
4.4.1 The Lexical Analyzer....45
4.4.2 The Syntax Analyzer....46
4.5 The Model....46
4.6 The Validation....50
4.7 Namespace Connection....52
4.8 Error Messages....54
4.9 Compiler Generators for Java....56
4.9.1 Introduction....56
4.9.2 Lex/YACC....57
4.9.3 PCCTS....60
4.9.4 Conclusion....63
5 Comparison....65

5.1 VRP and SiRPAC 65
5.2 VRP for rdf2cb 66
6 Conclusions 69
Appendix A – Compact XML Serialization of the RDF Schema 71
Appendix B – Official Errata in RDF Schema 74
Appendix C – The RDF (syntax) Namespace 76
Appendix D – BNF of RDF 77
Appendix E – The VRP Validation Constraints 78
Appendix F – SiLRI Validating Rules 79
Appendix G – The compact VRP Java class descriptions 81
Appendix H – The VRP RDF/BNF Grammar 116
References 123
Index 127

Abstract

The Web provides a simple and universal infrastructure to exchange various kinds of information. In order to share, interpret, and manipulate information worldwide, the role of metadata is widely recognized. Indeed, metadata allow us to easily locate information available in the Web, by providing descriptions about the structure and the content of the various Web resources (e.g. data, documents, images, etc.) and for different purposes. The emergence of the Resource Description Framework (RDF) is expected to enable metadata interoperability across different communities or applications by supporting common conventions about metadata syntax, structure, and semantics.

More precisely, it provides a) a Standard Representation Language for Web metadata; and b) a Schema Definition Language (RDFS) to interpret (meta)data using specific class and property hierarchies (i.e. vocabularies). Moreover, RDF/RDFS offer a syntax for representing metadata and schemas in XML, enabling the creation and exchange of RDF descriptions in a both human readable and machine understandable form. Many information providers like ABC News, CNN and Time Inc., Web portals like Open Directory as well as Web browsers like Netscape, and search engines like Altavista, Yahoo and Webcrawler already support the RDF proposal.

RDF is based on a directed graph model that alludes to the semantics of resource description. The basic idea is that a Resource (identified by a URI) can be described through a collection of Statements forming a so-called RDF Description. A specific resource together with a named property and its value is an RDF statement. RDFS schemas are then used to declare vocabularies, i.e. collections of classes and properties, that can be used in resource descriptions for a specific purpose or domain.

Unfortunately, existing RDF parsers (e.g., SiRPAC) check only the well-formedness of RDF resource descriptions according the W3C RDF M&S specifications. For this reason, I have developed the ICS-FORTH Validating RDF Parser (VRP) allowing the validation of RDF resource descriptions against the associated RDFS schemas, as well as of the schemas themselves.

VRP is a tool to analyze, validate and process RDF descriptions based on standard compiler generator tools for Java, namely CUP/JFlex (similar to YACC/LEX). As a result, users do not need to install additional programs (e.g., XML Parsers) in order to run VRP while they can easily update or extend the VRP BNF grammar in case of changes in the RDF/RDFS specifications. VRP is a 100% Java™ development understanding embedded RDF in HTML or XML and providing full Unicode support. The quick LALR grammar parser (i.e. CUP) as well as the stream-based parsing support (i.e. JFlex) ensure good performance during the processing of large volumes of RDF descriptions.

The most distinctive feature of VRP is its ability to verify the constraints specified in the RDF Schema specification. This allows the validation of both the RDF descriptions against one or more RDFS schemas, and the schemas themselves. The VRP validation module relies on (a) a complete and sound algorithm to translate descriptions from an RDF/XML form into the RDF core model (i.e. triples) (b) an implementation of this model in Java to efficiently verify the RDFS constraints.

To favor metadata reusability, RDF supports a) the sharability of RDFS schemas using the XML namespace mechanism (i.e. provide only incremental modifications to a base schema in order to create a new variant); and b) the creation of RDF (meta)data using multiple schemas at the same time (i.e. merging different types of metadata). This implies to maintain for real scale applications, several interconnected RDF schemas than can be potentially used to describe Web resources. To meat these requirements VRP supports validation across several namespaces: we can connect to remote namespaces in order to import the external statements we need to validate our RDF descriptions. Note that the RDF and RDFS namespaces are necessary for every RDF description and therefore their statements are by default included into VRP.

In this thesis, I present and discuss the *RDF Model and Syntax Specification* (W3C Recommendation 22 February 1999) and the *RDF Schema Specification* (W3C Proposed Recommendation 03. March 1999). I figure out some errors and try to address a number of open questions of the RDF specifications in order to provide a complete set of RDF validation constraints. Based on these explanations I present the basic notions and structures of the implementation of VRP and compares it with existing RDF Parsing software.

VRP is free available including the Java™ source code at http://www.ics.forth.gr/isst/proj/RDF.

1 Introduction

In this section I first want to give a short introduction to the field of metadata. The most common way to transcribe *metadata* is 'data about data'. A more extensive definition of the term metadata is given by the DESIRE project [DESIRE]. They describe metadata as 'data associated with objects which relieves their potential users of having to have full advance knowledge of their existence and characteristics'. Examples for metadata could be information about authenticity, availability and accessibility, digital signatures, copyright reproduction, indexing terms, summaries, bibliographic information, etc.

Let us have a closer look what could be done with metadata. If we have a big collection of data like a book, a library, an Intranet or the World Wide Web (WWW) we need something to find things we are searching for. E.g. for a book we use the summary, the table of contents or the index to find the part we are interested at or to decide to buy the book. In the WWW we use search engines to find information we are looking for. These search engines are at the moment mostly based on a textual search in the document text without knowing the context or meaning. This would correspond to read the complete book to find the subject we are interested at instead of just having a look at the index or the summary. This means we could save a lot of time by using metadata. But there are also problems that can only be solved by a machinery that understands the meaning of the words like metadata could do:

- **Homonym**. Most words in English – and other languages as well – have multiple meanings. For example, if we are searching for an article that discusses types of springs and their uses, we might retrieve articles on freshwater springs or on the season of spring, as well as on spring mattress, main-springs, or coil springs. Or take the word „construct", which can be both a noun (in mathematics) and a verb.
- **Synonym**. Many words represent the same concept, although they may do it with different shades of meaning. Take the words „ball", „sphere", and „orb", or „diving-goggles" versus „diving-mask". If you look for diving-goggles, but the term used is diving-mask, you will miss materials you might otherwise find. Good metadata should draw these materials together, despite their use of different synonyms.
- **Ambiguity**. If we return to our example of springs, we can see that what differentiates these meanings is their context. It is unlikely that an article on coil springs will also discuss water quality. The other words used in the article, and the processes described, will be entirely different. A search engine must understand the meaning, not just be able to match the spelling of a word, if it is going to differentiate between different meanings of the same word. While new natural language processing search engines can often determine these differences by the context of the text in the document, Web and Boolean engines cannot. Just some probabilistic search engines like those on the Web do determine relevance through co-occurrence of query terms in the document. However, the addition of controlled vocabularies in a subject field can distinguish between different kinds of springs or constructs. The goal of a controlled vocabulary, then, is at least to have each term mean one thing, and one thing alone. By using these controlled terms, we can disambiguate a term, as a human or a natural language processing system world.

Metadata should provide the semantic meaning of the information in a machine-readable and machine-understandable form. This would enable to process the data and give the user the ability to query the information he is interested in like: "Who is the author of a page?" and "When was the last time the page was modified?"

The first very simple versions of metadata in the WWW came in 1994 with the HTML 3.2 standard [HTML 3.2]. It defined a META-tag as an option that could appear in the HEAD tag of HTML documents. The syntax can associate an arbitrary list of key-value pairs to the document in which they are contained. META-tags frequently gives a short description and attribute a document's authorship (see Figure 1.1) [META 1].

```
<HTML>
<HEAD>
    <TITLE="An example providing the META tag.">
    <META NAME="Author"       VALUE="Someone">
    <META NAME="Rights"       VALUE="My Company">
    <META NAME="Description" VALUE="A short description of the document. ">
    <META NAME="Date"         VALUE="November 23, 1999">
</HEAD>
....
</HTML>
```

Figure 1.1 The META-tag in HTML.

In the last five years, there had been a huge number of metadata projects. You can find a list of projects and standardization efforts in [META 2]. These metadata projects provide a more differentiated underlying model and enable machines to understand the semantic of and the relations between the data. Plans for the future could be a standardization of offers within the Electronic Commerce or the exchange of database contents between different database systems all over the WWW using metadata.

This thesis will treats the Resource Description Framework (RDF), developed by the World-Wide Web Consortium (W3C), that provides the foundation for metadata interoperability across different resource description communities. It is an infrastructure that enables the encoding, exchange and reuse of structured metadata in some standard ways. The main questions I want to clarify in this document is a) how to analyze the information out of a given RDF file and b) how to validate these information against given constraints. See chapter 1 for a more precise introduction to RDF.

1.1 How to read this thesis?

Chapter 1 provides a precise introduction to RDF including the model and syntax specifications in section 2.4. In section 2.4.3.2 I provide an algorithm to analyze the information of a given RDF file and section 2.4.3.3 shows the difference between RDF and XML. The RDF Schema Specification is explained in section 2.5. If you are familiar with the subject of RDF you could skip these parts. Note that in section 2.5.3 you can find a list of errors of the RDF Schema Specification (version of the 03 March 1999).

In chapter 3 I mention and try to solve the open questions of RDF in a formal way. One main remaining question is how to deal with instances of rdfs:Literal, see section 3.3.3. Especially section 3.4 dealing with the validation of RDF I would recommend to the reader. I think this subject had been neglected by other papers.

The chapters 4 and 5 are dealing with the implemented tool VRP. Chapter 4 gives an introduction to the basic implementation fundamentals. VRP is based on a compiler generator and therefore I provide in section 4.9 the comparison of some existing compiler generators for Java™. Chapter 5 compares VRP with the mainly used parser at the moment called SiRPAC and shows in an example how VRP could be used for other tools.

In the last chapter I give a short outlook into the future and list things that could be done to force the usage of RDF and to enrich VRP.

2 RDF Presentation

This chapter gives a short introduction of the Resource Description Framework (RDF). It is a foundation for processing metadata and describes the interrelationships among resources in terms of named properties and values. RDF provides three models for representing metadata. These models will be presented in the section 2.4. Furthermore RDF provides mechanisms to declare the properties and relationships between properties and resources. These mechanisms are described in the section 2.5.

You can find the complete specification in [RDF M&S and RDF Schema], other useful papers are [RDF 1, RDF 2, RDF 3 and RDF 4].

2.1 History of RDF

RDF was developed by the World-Wide Web Consortium (W3C) as an extension of the Platform for Internet Content Selection (PICS) developed by the W3C in 1995. PICS is a mechanism for communicating ratings of web pages from a server to clients. These ratings, or rating labels, contain information about the contents of web pages. For example, parents worried about their children's web usage could set their browsers to filter out any web pages not matching given criteria.

To address more general issues of resource description, the W3C formed a new working group, the Resource Description Framework working group. This working group contains several W3C Member companies working together. So, we can say RDF is a collaborative design effort. The main influences have come from the *Web standardization community* itself in the form of HTML META-tag and PICS, the *library community*, the *structured document community* in the form of SGML and more importantly XML [XML]. Also other metadata efforts, such as the Dublin Core [DC], the Warwick Framework [WF] and proposals from Microsoft [XMLDATA] or Netscape [MCFXML] influenced the design of the RDF.

2.2 Design objectives and goals of RDF

- **Independence**. It should be possible for anyone to define its own properties or classes and use and/or reuse them in a specific semantic way. Moreover, RDF based applications should be future-proofing, regardless on the evolution of the schemas (see below).
- **Interchange**. It should be easy to transport and storage the metadata described by RDF.
- **Scalability**. Even for a huge set of metadata it should be easy to handle and process them.

2.3 The notion of RDF resources

RDF allows descriptions of any object with a Uniform Resource Identifier (URI) in a machine understandable form. In RDF these objects having a URI are called *resources*. Anything can have a URI and therefore can be represented by a resource; the extensibility of URIs allows the introduction of identifiers for any entity imaginable.

Examples for resources:

- Web sites, e.g., http://www.csi.forth.gr, http://www.uni-hannover.de, ...
- specific HTML or XML elements within the documents source
- objects that are not accessible via the Web, e.g., mountains, persons, ...

To create unique identifiers for the declared resources and make it possible to reuse them, RDF operates with *namespaces* [Namespaces in XML]. A namespace in RDF is a file containing resources identified by a Unique Resource Identifier (URI) reference. This URI of a resource specified in a namespace consist of the Unique Resource Locator (URL) of the file, the anchor symbol '#' plus the name (ID) of the resource. In the text I use an abbreviated way of writing these URIs. A prefix identifies the URL of the namespace followed by an colon and the name (ID) of the resource. The

main important namespaces are the RDF namespace, I use the prefix ‘rdf’ for it, and the RDF Schema namespace, I use the prefix ‘rdfs’ for it.

2.4 RDF M&S (data model)

RDF describes the interrelationships among resources in terms of named *properties* and values. These named properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. One main difference is that properties defined in RDF only identified by their name (URI) and not like in other object models, where attributes are identified by their name plus the domain class they can describe. So, properties have a URI and therefore are also resources. The *value* of a property can be another resource or a *literal.* A literal is simple string or other primitive data type as defined by [XML].

RDF provides three representations of the RDF data model namely:

- RDF Graph – a syntax-neutral graphical description of the data
- RDF 3-tuples – the set of statements described in triples
- RDF Syntax – provides some standard ways for describing data using XML

The RDF constructs and further nomenclature will be explained within the section 2.4.1. For the other two models I just explain the way of thinking and their underlying notion.

2.4.1 RDF Graph

The RDF Graph is a syntax-neutral way of representing RDF expressions using directed labeled graphs. These graphs also called *nodes and arcs diagrams*. The arcs represent the named properties. Each property connects two nodes, coming from a node representing a resource (drawn as oval) and pointing to another resource or a literal (drawn as rectangle). So, we have the following two options, as shown in Figure 2.1:

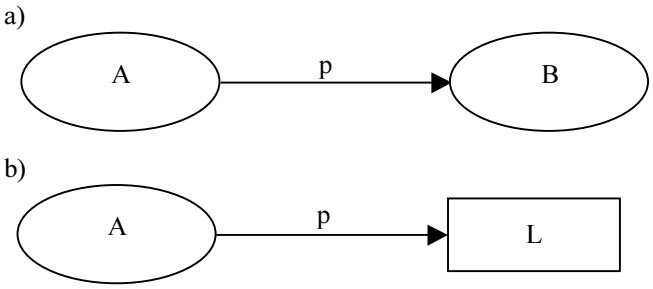


Figure 2.1 Statements in the RDF Graph representation.

a) The named property p connects the two resources A and B.

b) The named property p connects the resource A with the literal L.

Such a combination of a resource, a named property and its value is called a *statement* and define the resource as the *subject*, the property as the *predicate* and the value as the *object* of the statement.

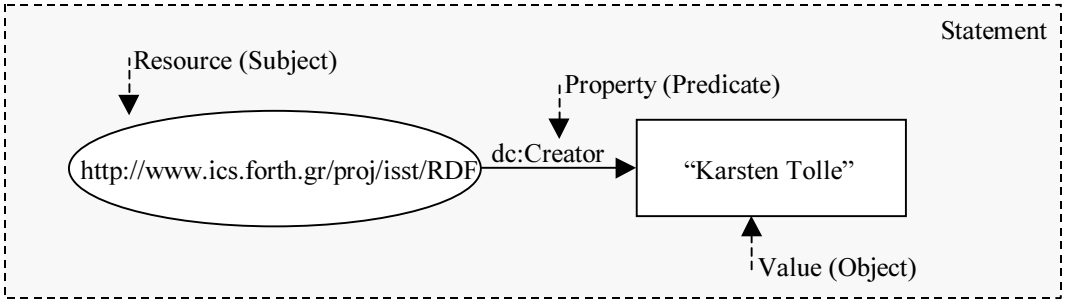


Figure 2.2 The elements of a statement: Subject, Predicate and Object.

This statement would represent the following sentence:

Karsten Tolle is the creator of the resource http://www.ics.forth.gr/proj/isst/RDF.

In some cases we generate new resources to express a sentence. These new resources do not have a predefined URI and we call them *anonymous.* Anonymous resources are represented in the RDF Graph as an empty oval. Intern a URI will be created for this resource, otherwise it would not be one. The internal URI is just not shown in this model and the choice how to generate it has to be made by the application database designer. So, we can distinguish between different anonymous resources. Consider as an example the following sentence, Figure 2.3 represents this in graph form:

http://www.ics.forth.gr/proj/isst/RDF was created by an individual whose name is Karsten Tolle, email tolle@irb.uni-hannover.de and the individual whose name is Mister X published it.

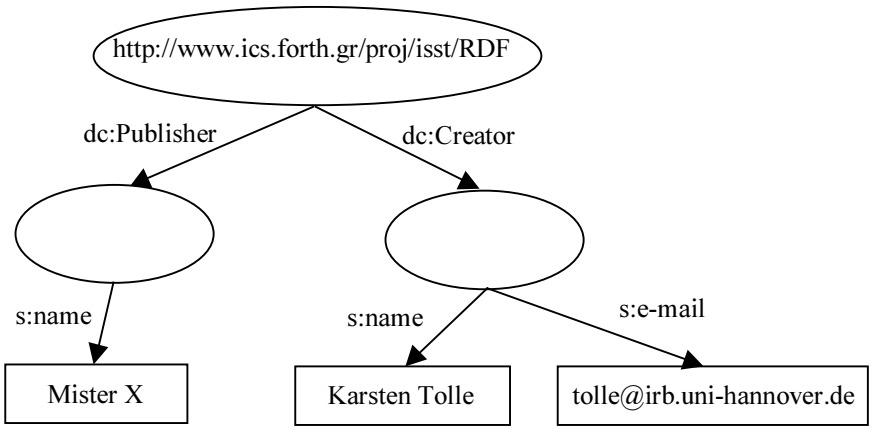


Figure 2.3 Use of anonymous resources in RDF Graph.

I use the ‘dc’ prefix for the Dublin Core namespace as shown in Appendix B of [RDF Schema] and the ‘s’ prefix for an imaginary namespace.

There are many cases where it is necessary to make statements about a collection of things, e.g., the members of a group or resources at a Web site. RDF provides three different containers to make references to a collection of resources or literal. A *Bag* representing a multi set of values, a *Sequence* representing an ordered list of values and an *Alternative* representing alternatives for a single value of a property. A container itself is a resource and must be instance of one of the predefined classes (see RDF Schema for classes) rdf:Bag, rdf:Seq or rdf:Alt by using the *rdf:type* property. The properties to instanciate values to the container are named simply „rdf:_1“, „rdf:_2“, „rdf:_3“, etc. For example, to represent the sentence

The students of the course s:Digital Design are „Peter“, „Paul“ and „Mary“.

The corresponding RDF Graph would be:

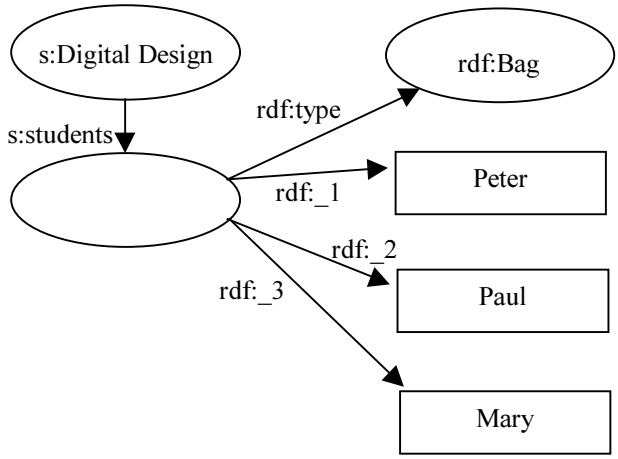


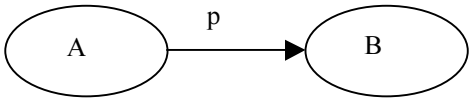
Figure 2.4 A bag as an example of how to use containers in RDF.

In order to enable RDF to make statements about other RDF statements it provides the ability to create a *model of a statement*. This process to make statements about statements is formally called *reification* in the Knowledge Representation community. So, a model of a statement is called a reified statement. A *reified statement* is a resource having the rdf:type = rdf:Statement and must define the following three properties to guarantee a unique reference:

- rdf:predicate The link to the predicate of the statement we refer to.
- rdf:subject The link to the subject of the statement we refer to.
- rdf:object The link to the object of the statement we refer to.

In Figure 2.5 you can see the abstract way of modeling a statement and in Figure 2.6 we have a more precise example. In both cases the resource for the reified statement is an anonymous resource. RDF provides a shorthand to create statements about statements by using the attribute 'rdf:bagID' (refer to [RDF M&S Chapter 4] for more examples).

The original statement:



The model of the statement:

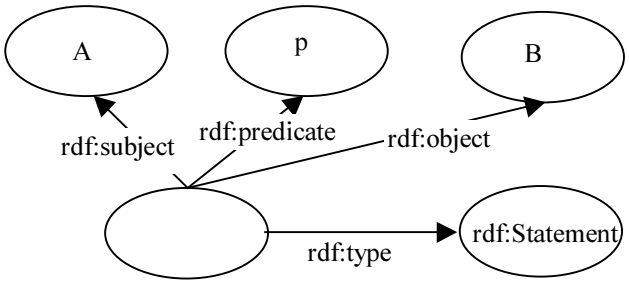


Figure 2.5 The formal remodel of a statement to a reified statement.

Using a reified statement we can model the sentence:
Vassilis says: "Karsten is the creator of the resource http://www.ics.forth.gr/proj/isst/RDF."

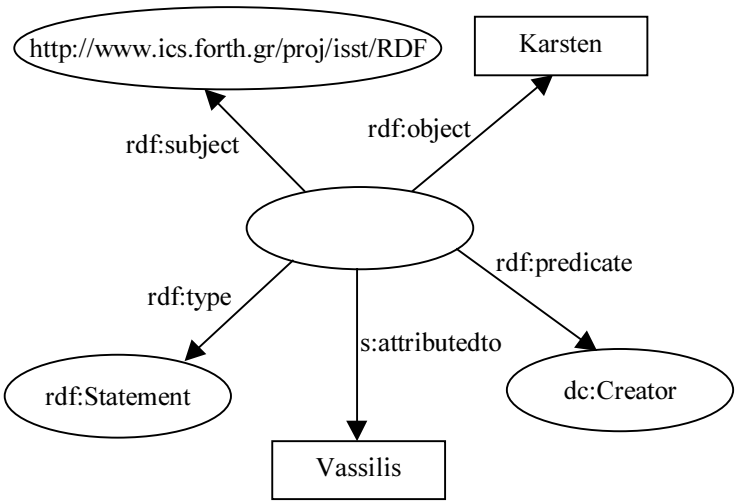


Figure 2.6 An example of a reified statement in the RDF Graph representation.

2.4.2 RDF in 3-tuples

Another way to represent a set of RDF statements is to use 3-tuples, also called triples. Each triple {p, s, o} correspond to an arc from the subject s to the object o, labeled by the predicate p. The example shown in Figure 2.2 would be written as:

{[http://purl.org/dc/...#Creator], [http://www.ics.forth.gr/.../RDF], „Karsten Tolle“}

The notation [*I*] denotes the resource identified by the URI *I* and quotation marks denote a literal. Actual this notation is only necessary for the object o of the statement, because the predicate p and the subject s must be resources.

This representation is an easy way to represent a very huge set of statements and process them and therefore provides the scalability that had been forced as one goal for RDF.

2.4.3 RDF Syntax (XML)

This representation was chosen for encoding and transporting the statements in a manner that maximizes the interoperability of independently developed Web servers and clients. The syntax presented here uses the eXtensible Markup Language (XML) [XML]. It consist out of elements having a name and an optional attribute list. The elements may contain some contents (other elements or strings). The elements are delimited a start-tag and an end-tag, or, for empty elements (no contents) by an empty-element tag.

Elements with contents:

'<' name attribute-list '>' // start-tag
contents
'</' name '>' // end-tag

Elements without contents:

'<' name attribute-list '/>' // empty-element tag

The [RDF M&S] provides a basic serialization syntax and three forms of abbreviation for it. These abbreviations are:

1. Properties used with a literal as value can be used in the same way as attributes in XML.
   Example:

```
<rdf:Description about=”http://www.ics.forth.gr/proj/isst/RDF”>
    <dc:Creator>Karsten Tolle<dc:Creator/>
</rdf:Description>
```

can also be written as

```
<rdf:Description about="http://.../isst/RDF" dc:Creator="Karsten Tolle"/>
```

2. Nested 'description' (see BNF) elements as the value of a property.

   Example:

```
<rdf:Description about="http://www.w3.org/Home/Lassila">
    <dc:Creator rdf:resource="http://www.w3.org/staffId/85740"/>
</rdf:Description>

<rdf:Description about=" http://www.w3.org/staffId/85740">
    <rdf:value>Ora Lassila</rdf:value>
</rdf:Description>
```

   can be written together as

```
<rdf:Description about="http://www.w3.org/Home/Lassila">
    <dc:Creator>
           <rdf:Description about="http://www.w3.org/staffId/85740">
                  <rdf:value>Ora Lassila</rdf:value>
           </rdf:Description>
    </dc:Creator>
</rdf:Description>
```

3. You can attach the type of a resource by the element name instead of using the rdf:type property.

   Example:

```
<rdf:Description ID="http://purl.org/metadata/dublin_core#Creator">
 <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Description>
```

   can be written as

```
<rdf:Property ID="http://purl.org/metadata/dublin_core#Creator"/>
```

You can find the complete Backus-Naur Form (BNF) for RDF containing all syntax forms in Appendix D. An explanation of the RDF attributes described in the BNF is given in section 2.4.3.1.

There are some syntactic features inherited from XML. These features are:

- Attributes do not have to follow an order and can be mixed with properties (literal as value) used in an attribute list..
- No attribute name may appear more than once in the same start-tag or empty-element tag (well-formed constraint).
- It is valid to use single quotes around attribute values.
- The special XML attributes 'xml:lang' and 'xmlns' can be entered in each attribute list.[1]
- The use of whitespace around attributes and the '='.

  The XML 1.0 Specification [XML] says:

  S ::= (#x20 | #x9 | #xD | #xA)+

  Eq ::= S? '=' S?
- Character references, e.g., the string '&' for '&'.

You can find the rules how to pick out the statements out of an RDF/XML-File in the section 2.4.3.2.

2.4.3.1 The RDF attributes

In this section we give a short explanation to the RDF attributes used in the BNF (the numbers in brackets refer to the BNF). Again the prefix 'rdf' denotes the namespace of RDF (syntax). The prefix 's' in the examples stands for any user defined namespace. We use just the names of new resources and not the full URIs inside the graphical representations of the examples.

```
[6.6] idAttr          ::= ' ID="' IDsymbol '"'
```

[1] The attributes 'xml:lang' and 'xmlns' do not represent triples. There is no specific data model representation for the namespace or language declarations.

The attribute for generating a new resource with the name “IDsymbol”. The full URI depends on the absolute path of the file containing these statement.

Example 1:

...<rdf:Description rdf:ID=”Entity”... inside the file http://www.ics.forth.gr/proj/isst/RDF/examples/CIDOC.rdf would create the resource with the URI http://www.ics.forth.gr/proj/isst/RDF/examples/CIDOC.rdf# Entity.

```
[6.7] aboutAttr      ::= ' about="' URI-reference '"'
```

This attribute denotes that the “description” or “typedNode” is describing the resource specified by the given “URI-reference”. We assume the existence of the given resource in this case.

```
[6.8] aboutEachAttr  ::= ' aboutEach="' URI-reference '"'
                       | ' aboutEachPrefix="' string '"'
```

The “aboutEach” attributes refers to each resource contained by the container specified in the “URI-reference”. The statements in a “description” that has the “aboutEach” attribute apply individually to each resource contained by this container.

Example 2:

Here we first generate a bag with the ID “pages” and then make a description about each member of this bag. Figure 2.7 shows the corresponding RDF Graph.

```
<rdf:Bag ID="pages">
    <rdf:li resource="http://foo.org/foo.html" />
    <rdf:li resource="http://bar.org/bar.html" />
</rdf:Bag>
<rdf:Description aboutEach="#pages">
    <s:Creator>Ora Lassila</s:Creator>
</rdf:Description>
```



Figure 2.7 Illustrating the usage of the “aboutEach” attribute in the RDF Graph representation.

The “aboutEachPrefix” attribute declares that there is a bag whose members are all the resources whose fully resolved resource identifiers begin with the character string given as the value of the attribute. The statements in a “description” that has the “aboutEachPrefix” attribute apply individually to each of the members of this bag.

Example 3:

If the two resources http://foo.org/doc/page1 and http://foo.org/doc/page2 exist then we can say that each of them has a copyright property by writing:

```
<rdf:Description aboutEachPrefix="http://foo.org/doc">
    <s:Copyright>© 1998, The Foo Organization</s:Copyright>
</rdf:Description>
```

If these are the only two resources whose URIs start with that string then the above is would be illustrated as shown in Figure 2.8:

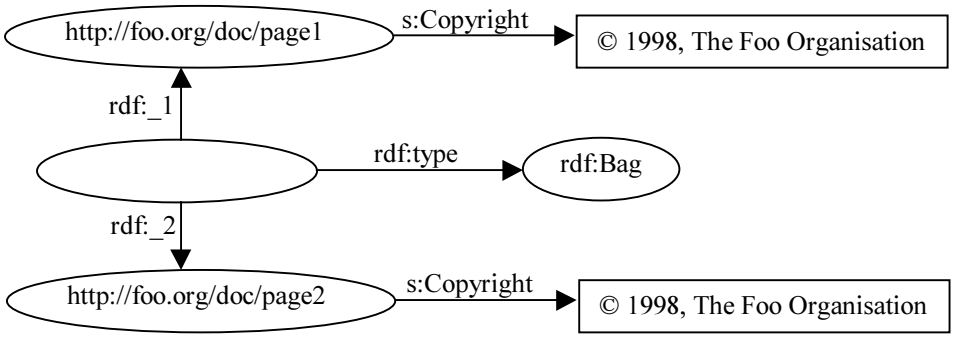


Figure 2.8 Illustrating the usage of the "aboutEachPrefix" attribute in the RDF Graph representation.

```
[6.9] bagIdAttr      ::= ' bagID="' IDsymbol '"'
```

The "bagID" attribute is a shorthand form for generating a reified statement. It creates a bag with the name specified in the value of the attribute. The members of this bag are reified statements representing the statements of the description containing the "bagID" attribute. This way the reified statements are new generated anonymous resources.

Example 4:

```
<rdf:Description about="http://www.w3.org/Home/Lassila" bagID="D_001">
      <s:Creator>Ora Lassila</s:Creator>
      <s:Title>Ora's Home Page</s:Title>
</rdf:Description>
```

would result in the graph shown in Figure 2.9.

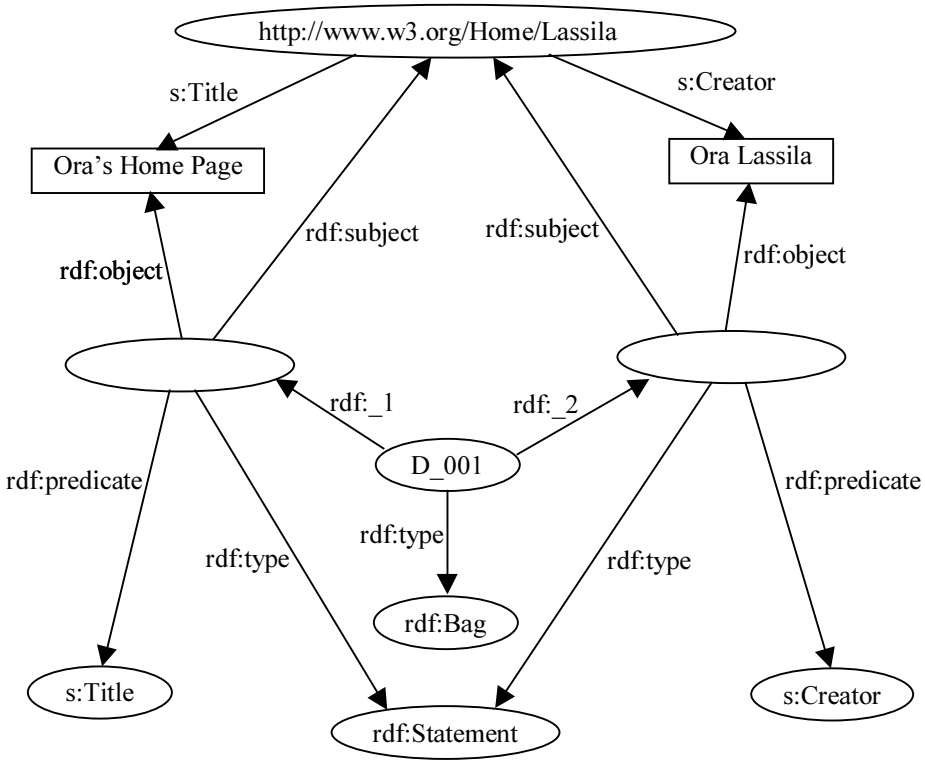


Figure 2.9 Illustrating the usage of the "bagID" attribute in the RDF Graph representation.

Note: Together with the "aboutEach" attribute referring to the created bag the "bagID" attribute provides a shorthand for making statements about statements. E.g.:

```
<rdf:Description aboutEach="#D_001" s:true="YES"/>
```

```
[6.11] typeAttr      ::= ' type="' URI-reference '"'
```

Indicates that a resource is a member of a class (see section 2.5). Can also be used as a property.

```
[6.18] resourceAttr   ::= ' resource="' URI-reference '"'
```
Only for property elements and container member elements to denote that the object we refer to is the resource given by the value of the attribute and not a literal.

```
[6.31] memberAttr     ::= ' rdf:_n="' string '"' (where n is an integer)
```
A shorthand version to specify the members of a container. This way we can only define a literal as value.

```
[6.32] parseLiteral   ::= ' parseType="Literal"'
```
Only for property elements and container member elements, the contents of the element will be treated as a literal (any well formed XML). Markups will not be further evaluated.

Example 5:

```
<rdf:Description about="http://www.ics.forth.gr/isst/proj/RDF">
    <s:Contents parseType="Literal">
          <s:Examples rdf:resource="http://www.ics.forth.gr/isst/proj/RDF/examples"/>
    </s:Contents>
</rdf:Description>
```
would result in the graph shown in Figure 2.10.

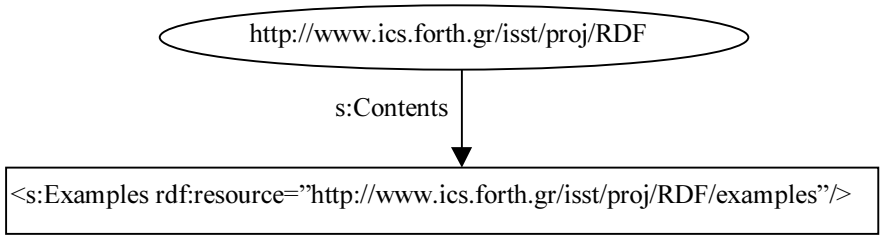


Figure 2.10 Illustrating the usage of the 'parseType="Literal" ' attribute in the RDF Graph representation.

```
[6.33] parseResource  ::= ' parseType="Resource"'
```
Only for property elements and container member elements, the contents of the element will be treated as if it were the contents of a "description" element.

Example 6:

```
<rdf:Description about="http://www.ics.forth.gr/isst/proj/RDF">
    <s:Contents parseType="Resource">
          <s:Examples rdf:resource="http://www.ics.forth.gr/isst/proj/RDF/examples"/>
    </s:Contents>
</rdf:Description>
```
would result in the graph shown in Figure 2.11.

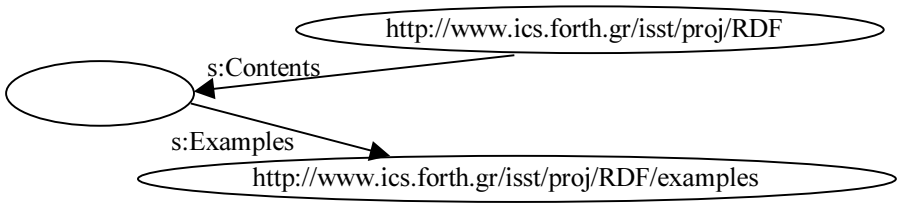


Figure 2.11 Illustrating the usage of the 'parseType="Resource" ' attribute in the RDF Graph representation.

2.4.3.2 Statement Creation

Let us see now how to create statements {P, S, O} (P – Predicate, S – Subject, O – Object) out of an RDF/XML-File. The following numbers 1-9 represent a algorithm I created based on [RDF M&S, chapter 6]. The expressions in double quotas refer to the BNF of RDF (see Appendix D).

1. For each „propertyElt" E contained by a „description" or a „typedNode" create {P, S, O}.

   P = The expansion of the namespace of E + „name" of E.

S = The resource given in „aboutAttr“ or in the „idAttr“. If those attributes do not exist it is an anonymous resource (the parser will generate an intern ID for it).

O = Depending of the following cases:

a) „parseLiteral“ then O is the contents of E (a well-formed literal).

Example:

The example 5 of section 2.4.3.1 would create the following statement:

{<namespace expansion for ‘s’> + Contents,
http://www.ics.forth.gr/isst/proj/RDF,
“<s:Examples rdf:resource=”http://www.ics.forth.gr/isst/proj/RDF/examples”/>”}

b) „parseResource“ → contents must be treated as a “description” contents.

Example:

The example 6 of section 2.4.3.1 would create the following statements:

{<namespace expansion for ‘s’> + Contents,
http://www.ics.forth.gr/isst/proj/RDF,
<anonymous resource A, URI depends on the application>}
{<namespace expansion for ‘s’> + Examples,
<anonymous resource A, URI depends on the application>,
http://www.ics.forth.gr/isst/proj/RDF}

c) Empty contents then O is the resource whose identifier is given by the „resourceAttr“ attribute of E. If the „resourceAttr” is not specified O is a new anonymous resource.

Example 7:

```
<rdf:Description about=”http://www.ics.forth.gr”>
       <rdf:value rdf:resource=”http://www.ics.forth.gr/isst”/>
</rdf:Description>
```

would create the following statement:

{http://www.w3.org/TR/1999/REC-rdf-syntax#value,
http://www.ics.forth.gr,
http://www.ics.forth.gr/isst}

d) „string“ as contents then O is the string representing a literal.

Example 8:

```
<rdf:Description about=”http://www.ics.forth.gr”>
       <rdf:value>http://www.ics.forth.gr/isst</rdf:value>
</rdf:Description>
```

would create the following statement (the same triple as in example 8 but with the object as an literal):

{http://www.w3.org/TR/1999/REC-rdf-syntax#value,
http://www.ics.forth.gr,
“http://www.ics.forth.gr/isst”}

Example 9: It is possible that the “string” corresponds to the empty-string “”.

```
<rdf:Description about=”http://www.ics.forth.gr”>
       <rdf:value></rdf:value>
</rdf:Description>
```

would create the following statement:

{http://www.w3.org/TR/1999/REC-rdf-syntax#value,
http://www.ics.forth.gr,
“”}

e) „obj“ as contents then O is the resource described by this „description“ or „container“.

Example 10:

```
<rdf:Description about=”http://www.ics.forth.gr”>
```

```
    <rdf:value>
        <rdf:Bag rdf:ID="Institutes"
                 rdf:_1="http://www.ics.forth.gr/isst"/>
    </rdf:value>
</rdf:Description>
```

would create the following statements:

{http://www.w3.org/TR/1999/REC-rdf-syntax#value,
 http://www.ics.forth.gr,
 <namespace expansion for this file> + Institutes}
{http://www.w3.org/TR/1999/REC-rdf-syntax#type,
 <namespace expansion for this file> + Institutes,
 http://www.w3.org/TR/1999/REC-rdf-syntax#Bag}
{http://www.w3.org/TR/1999/REC-rdf-syntax#_1,
 <namespace expansion for this file> + Institutes,
 "http://www.ics.forth.gr/isst"}

2. For each „propAttr" contained by „description" or „typedNode" not starting with xmlns or xml:lang create the statement {P, S, O}.

   P = The expansion of the namespace of the attribute + „propName".

   S = The resource given in „idAboutAttr". If this attribute does not exist it is an anonymous resource (the parser will generate an intern ID for it).

   O = The „string" value of the „propAttr" (a literal).

   Example 11:

```
<rdf:Description about="http://www.ics.forth.gr/isst/proj/RDF" s:Creator="Karsten Tolle"
/>
```

   would create the triple:

   { <namespace expansion of 's'> + Creator,
    http://www.ics.forth.gr/isst/proj/RDF,
    "Karsten Tolle"}

3. For each „propAttr" other than rdf:ID, rdf:resource or rdf:bagID contained by „propertyElt" E not starting with 'xmlns' or 'xml:lang' create the statements: {P, S_1, S_2}, {P_a, S_2, O_a} where

   P = The expansion of the namespace of E + „name" of E.

   S_1 = The resource being referred to by the element containing E.

   S_2 = The resource named by the rdf:resource attribute if present or a new resource. If the rdf:ID attribute is given its value is the identifier of this new resource.

   P_a = The expansion of the namespace-qualified attribute name.

   O_a = The corresponding attribute value.

   Example 12:

```
<rdf:Description about="http://www.ics.forth.gr/isst/proj/RDF">
   <rdf:value s:Creator="Karsten Tolle" />
</rdf:Description>
```

   would create the following triples:

   {http://www.w3.org/TR/1999/REC-rdf-syntax#value,
    http://www.ics.forth.gr/isst/proj/RDF,
    <anonymous resource A, URI depends on the application>}
   {<namespace expansion of 's'> + Creator,
    <anonymous resource A, URI depends on the application>,
    "Karsten Tolle"}

4. For each „typedNode“ create the statement {P, S, O} where

   P = The RDF namespace URL + 'type'.

   S = The resource whose identifier is given by the value of the „aboutAttr“ attribute or whose ID is given by the value of the „idAttr“ attribute value of the typedNode element, else it is a anonymous resource.

   O = Is the expansion of the namespace-qualified tag-name.

   Example 13:

   ```
   <rdf:Property rdf:ID="my_property"/>
   ```

   would create the statement:

   {http://www.w3.org/TR/1999/REC-rdf-syntax#type,
   <namespace expansion for this file> + my_property,
   http://www.w3.org/TR/1999/REC-rdf-syntax#Property}

5. When „aboutEach“ is specified in a „description“ or „typedNode“ the given property-value pairs (P_i, O_i) refer to each of the members of the container named by „aboutEach“ → create for each member M_j of the container the statements { P_i, M_j, O_i}.

   Example:

   The example 2 of section 2.4.3.1 would create the following statements:

   {http://www.w3.org/TR/1999/REC-rdf-syntax#type,
   <namespace expansion for this file> + pages,
   http://www.w3.org/TR/1999/REC-rdf-syntax#Bag}
   {http://www.w3.org/TR/1999/REC-rdf-syntax#_1,
   <namespace expansion for this file> + pages,
   http://foo.org/foo.html}
   {http://www.w3.org/TR/1999/REC-rdf-syntax#_2,
   <namespace expansion for this file> + pages,
   http://bar.org/bar.html}
   {<namespace expansion of ‘s’> + Creator,
   http://foo.org/foo.html,
   “Ora Lassila”}
   {<namespace expansion of ‘s’> + Creator,
   http://bar.org/bar.html,
   “Ora Lassila”}

6. When „aboutEachPrefix“ is specified in a „description“ or „typedNode“ the given property-value pairs (P_i, O_i) refer to each of the members of an anonymous container (Bag) whose elements are all the resources beginning with the string specified in the value of „aboutEachPrefix“ → create for each member M_j of these anonymous container the statements { P_i, M_j, O_i}.

   Example:

   The example 3 of section 2.4.3.1 would cause the following statements:

   {http://www.w3.org/TR/1999/REC-rdf-syntax#type,
   <anonymous resource A, URI depends on the application>,
   http://www.w3.org/TR/1999/REC-rdf-syntax#Bag}
   {http://www.w3.org/TR/1999/REC-rdf-syntax#_1,
   <anonymous resource A, URI depends on the application>,
   http://foo.org/doc/page1}

{http://www.w3.org/TR/1999/REC-rdf-syntax#_2,
<anonymous resource A, URI depends on the application>,
http://foo.org/doc/page2}
{<namespace expansion of 's'> + Copyright,
http://foo.org/doc/page1,
"© 1998, The Foo Organization"}
{<namespace expansion of 's'> + Copyright,
http://foo.org/doc/page2,
"© 1998, The Foo Organization"}

7. For each container „sequence" | „bag" | „alternative" create the statement {P, S, O}
   P = The RDF namespace URL + 'type'.
   S = The name of the container specified in „idAttr" or a new anonymous resource.
   O = The RDF namespace URL + ('Seq' | 'Bag' | 'Alt').

8. For each „li" element or „memberAttr" contained by „sequence" | „bag" | „alternative" create the statement {P, S, O}
   P = The RDF namespace URL + '_n' , where „n" is an integer-counter starting with 1 for each container.
   S = The name of the container specified in „idAttr" or a new anonymous resource.
   O = Depending of the following cases (compare with 1.):
   a) „parseLiteral" then O is the contents of E (a well-formed literal).
   b) „parseResource" → contents must be treated as a "description" contents.
   c) Empty contents then O is the resource whose identifier is given by the „resourceAttr" attribute of E.
   d) „string" as contents then O is the string representing a literal.
   e) „obj" as contents then O is the resource described by this „description" or „container".

9. If „bagIdAttr" is specified for reification, create a rdf:Bag named by the value of „bagIdAttr". For each statement $\{P_i, S_i, O_i\}$ (i = 1, ... , n) of the „description" or the „typedNode" create a new resource of type rdf:Statement with the properties rdf:subject, rdf:predicate and rdf:object and the corresponding values. Enter these new resources as members to the rdf:Bag.
   This will create the following statements:
   {<value of „bagIdAttr">, rdf:type, rdf:Bag}
   { rdf:type , <anonymous 1>, rdf:Statement}
   { rdf:_1, <value of „bagIdAttr">, <anonymous 1>}

   { rdf:subject, <anonymous 1>, $<S_1>$}
   { rdf:predicate, <anonymous 1>, $<P_1>$}
   { rdf:object, <anonymous 1>, $<O_1>$}

   { rdf:type , <anonymous 2>, rdf:Statement}
   { rdf:_2, <value of „bagIdAttr">, <anonymous 2>}
   { rdf:subject, <anonymous 2>, $<S_2>$}
   ....
   { rdf:object, <anonymous n>, $<O_n>$}

Note: I left the prefixes here for a better readability. The real statements would contain the string ‘http://www.w3.org/TR/1999/REC-rdf-syntax#’ instead of ‘rdf:’.

2.4.3.3 Difference between XML and RDF

The question is „What does RDF provide we can not do with XML?“, XML uses similar structures, provides information about the document and XML DTDs can be compared with the schemas in RDF. But if we take a closer look we can find major differences between XML and RDF due to their different design objectives. XML was manly created for the structural organization of documents. Therefore XML allows a large number of ways in which a statement can be expressed. For example the sentence we used before:

Karsten Tolle is the creator of the resource http://bsp.html.

This sentence could be expressed in XML by:

```
<creator>
       <uri>http://bsp.html</uri>
       <name>Karsten Tolle</name>
</creator>
```

or maybe

```
<document href="http://bsp.html" >
       <creator>Karsten Tolle</creator>
</document>
```

or maybe

```
<document>
       <details>
              <uri>href="http://bsp.html"</uri>
              <creator>
<name>Karsten Tolle</name>
</creator>
       </details>
</document>
```

or maybe as an empty element

```
<document href="http://bsp.html" creator="Karsten Tolle" />
```

We can say XML in general provides a large number of ways mapping a statement. If we want to query information in the WWW we would need to query in an independent way. So, the biggest work for a query would be to convert the set of different representations of a fact into one statement. In XML the Document Type Definition (DTD) tells us which way of expressing the statement should be used and also fixes the order in which elements appear in an XML document. For example in XML the DTD could determine whether the name of the author should appear before or after the title of the document. Without knowing this order we could not say anything about the relations between the elements and their semantics.

Furthermore XML provides the ability to mix elements and character strings as the contents of an element. This is useful for the structural organization of a document, but makes it hard to handle. For example the following construct would be valid in XML but not in RDF:

```
<Description>The value of this property contains some text, mixed up with child properties
such as its temperature (<temp>27</temp>) and longitude (<longitude>101</longitude>).
</Description>
```

RDF gives some standard ways of writing the statements and the order has no meaning. So a parser can easily extract the statements. We only need the used schemas to learn about the semantic of the used properties and classes. Even if there are some schemas we could not connect, we might have statements we understand. This is the philosophy of RDF saying: “May be we do not understand the whole stuff, but we still can do things with it!”

2.5 RDF Schema

The RDF data model specification (as defined in [RDF M&S]) provides no mechanisms for declaring properties, nor does it provide any mechanisms for defining the relationships between properties and other resources. That is the role of the *RDF Schema* as specified in [RDF Schema]. The RDF Schema defines resources using the mechanism it describes. It can be expressed in the RDF data model (with some exceptions). The RDF Schema also describes resources of the RDF namespace. This RDF

(syntax) namespace (see Appendix C) defines the properties described in [RDF M&S] and uses the mechanisms of the RDF Schema.

A namespace describing resources is called a *schema*. The RDF Schema is such a schema. It is simply a set of resources and properties and implicitly part of every RDF model using the RDF schema machinery.

Note that there are some constraints on certain RDF Schema resources which are themselves not fully expressible using the embedded RDF of [RDF Schema]. For example, it does not tell us that rdfs:subClassOf arcs should not (to use terminology from the nodes and arcs representation) form loops in any RDF model. I will discuss all constraints of RDF Schema resources in chapter 3. In this way the RDF Schema differs from other *user defined* Schemas.

2.5.1 Classes in RDF Schema

The RDF Schema defines a class-system in a hierarchical fashion. You can think of a class as a set of resources, except rdfs:Literal that represents the set of literal. A class itself is a resource and therefore has a URI. A resource is a class if it is an instance of the class *Class* of RDF Schema (*rdfs:Class*). Therefore the class rdfs:Class is an instance of itself. To instanciate a resource to a class we use the property *type* of the RDF namespace (*rdf:type*). You can find a copy of the RDF namespace in Appendix C. To organize these classes in a hierarchical fashion we use the property *rdfs:subClassOf* saying a class is a sub class of another.

The RDF Schema vocabulary namespace includes the following classes:

- rdfs:Resource – The most general class, containing all resources.
- rdfs:Class – The concept of class, containing all classes.
- rdfs:Literal – Representing the set of atomic values, e.g., textual strings.
- rdfs:ConstraintResource – Resources used to express RDF Schema constraints.
- rdfs:ConstraintProperty – Properties used to express RDF Schema constraints.
- rdfs:Container – The class represents the set of containers. Containing the classes rdf:Bag, rdf:Seq, rdf:Alt.
- rdfs:ContainerMembershipProperty – The class containing the properties rdf:_1, rdf:_2, ... that are used to indicate the membership to a container.

and describes the following classes defined in the RDF namespace:

- rdf:Property – The concept of properties, containing all properties.
- rdf:Bag – The class represents the set of bags. Instances are interpreted as a multi set of values (resources or literal).
- rdf:Seq – The class represents the set of sequences. Instances are interpreted as an ordered list of values.
- rdf:Alt – The class represents the set of alternatives. Instances are interpreted as alternatives for the value of a property.
- rdf:Statement – The class represents the set of reified statements.

In Figure 2.12 you can see the relationship of these classes. The arc goes from the superset to the subset.

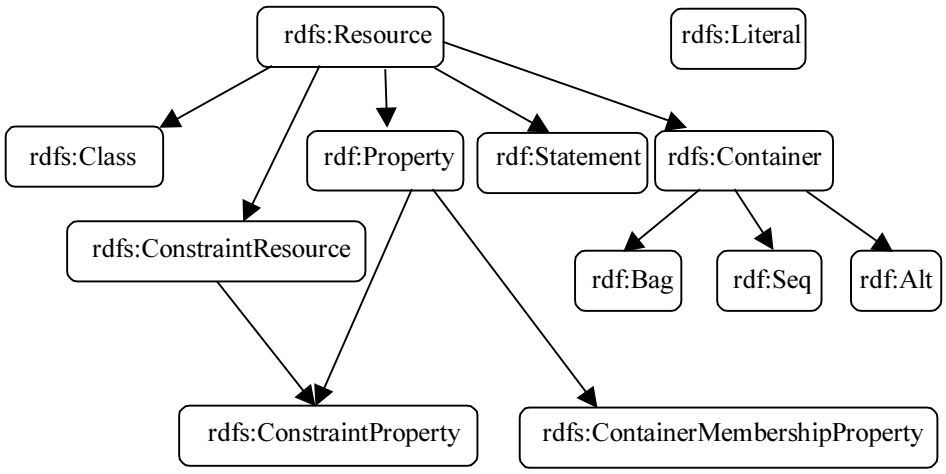


Figure 2.12 Superset relationship of the classes.

2.5.2 Properties in RDF Schema

The RDF Schema defines and describes basic properties that are used for describing resources, their relationships and constraints. Each application should understand the semantic meaning of these properties.

The RDF Schema vocabulary namespace defines the following properties:

- rdfs:subClassOf – Indicates specialization of classes.
- rdfs:subPropertyOf – Indicates specialization of properties.
- rdfs:seeAlso – Indicates a resource that provides information about the subject resource.
- rdfs:isDefinedBy – Indicates a resource containing and defining the subject resource. Defined as a sub property of rdfs:seeAlso.
- rdfs:comment – Provides a human-readable description of the subject resource.
- rdfs:label – Provides a human-readable version of a resource name.
- rdfs:range – For properties to indicate the class its instances can be used as objects for the property. Defined as an instance of the class rdfs:ConstraintProperty.
- rdfs:domain – For properties to indicate the classes their instances can be used as subjects for the property. Defined as an instance of the class rdfs:ConstraintProperty.

It also includes the following property defined in the RDF namespace:

- rdf:type – Indicates membership of a class.
- rdf:predicate – Identifies the predicate of a statement when representing the statement in reified form.
- rdf:subject – Identifies the subject of a statement when representing the statement in reified form.
- rdf:object – Identifies the object of a statement when representing the statement in reified form.
- rdf:value – Identifies the principal value (usually a string) of a property when the property value is a structured resource.

Table 2.1 Properties of the RDF namespace with their range and domain definitions (defined in the RDF Schema namespace). In the 'additional constraints' column you can find valid domain constraints not specified in the RDF Schema.

RDF Property	domain	range	additional constraints
type	unconstraint	rdfs:Class	domain = rdfs:Resource
subject	rdf:Statement	rdfs:Resource	
predicate	rdf:Statement	rdf:Property	
object	rdf:Statement	unconstraint	
value	unconstraint	unconstraint	domain = rdfs:Resource
_n [n is integer]	unconstraint	unconstraint	domain = rdfs:Container

In the case a property is unconstraint it can be used with any resource or literal. This fact corresponds to be constraint to the union of the classes rdfs:Resource and rdfs:Literal.

Table 2.2 Properties of the RDF Schema namespace with their range and domain definitions.

RDF-Schema Property	domain	range
comment	rdfs:Resource	rdfs:Literal
label	rdfs:Resource	rdfs:Literal
subClassOf	rdfs:Class	rdfs:Class
subPropertyOf	rdf:Property	rdf:Property
seeAlso	rdfs:Resource	rdfs:Resource
isDefinedBy	rdfs:Resource	rdfs:Resource
domain	rdf:Property	rdfs:Class
range	rdf:Property	rdfs:Class

The properties rdf:_1, rdf:_2, rdf:_3, ... belong implicit to the namespace of RDF. It is an infinite set of properties and can not be described by the mechanism RDF provides. They are also implicit instances of the class rdfs:ContainerMembershipProperties.

2.5.3 Errors in the RDF Schema

In this chapter I list up all errors I found in RDF Schema Specification. References that apply to [RDF Schema] are denoted by double quotation marks.

1. In "Figure 1" and "Figure 2" the resource rdfs:Literal is just shown as a resource. In "Figure 3", the "Basic XML Serialization", the "Compact XML Serialization" and the "embedded RDF" it is described as a class. In contrast to the others rdfs:Literal is defined in the "Compact XML Serialization" and the "embedded RDF" as a rdfs:subClassOf rdfs:Resource.

   This subject is further discussed in section 3.3.3 on page 32 of this document.
2. In "Figure 3" we should read the property rdf:object instead of the property rdf:value. The property rdf:value is unconstraint according to the rdfs:range and rdfs:domain declarations. It is the property rdf:object that is used to identify the object of a statement when representing the statement in reified form and therefore has the rdfs:domain = rdf:Statement declaration.
3. In the chapter "A Prototype PICS/RDF" we can read:

   <rdf:Property ID="xy" comment="..."> ...

   In this case the attributes ID and comment would inherit the namespace of the start-tag (rdf), but comment does not belong to this namespace.

   In "C. Sample PICS Rating System" of the same chapter we can see the namespace declarations within the first Description element. The namespace declarations are not valid outside this

element. So, we should read the string „</Description>" at the end of the example. This would also make it to a well-formed document, otherwise there would be no root element.

4. In the "Basic XML Serialization", the "Compact XML Serialization" and the "embedded RDF" we can read for the property rdfs:subClassOf the same rdfs:comment as for the property rdf:type: „Indicates membership of a class". For rdfs:subClassOf we should read something like: „Indicates specialization of classes".
5. In the "Basic XML Serialization" the rdfs:domain and rdfs:range declarations are missing.
6. In the "Basic XML Serialization", the "Compact XML Serialization" and the "embedded RDF" the class Property is redefined (ID="Property"). Instead we it should describe the class rdf:Property (about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property").
7. In the "Compact XML Serialization" we can find in the definition for rdfs:Resource:

   ... rdfs:comment="The most general class/">

   Here the end sign '/>' of the element is mixed with the value of the attribute. It should say:

   ... rdfs:comment="The most general class"/>.
8. In the "Compact XML Serialization" and the "embedded RDF" there is a misspelling of the attribute rdfs:label during the definition of rdfs:label.
9. In the "Compact XML Serialization" there is a misspelling of the attribute rdfs:label in the description of the class rdf:Statement.
10. In the "Compact XML Serialization" and the "embedded RDF" we can find in the descriptions of rdf:Statement, rdf:Bag, rdf:Seq and rdf:Alt:

    <rdfs:Class about="...

    Doing so the 'about' attribute would inherit the namespace of the start tag. So, we need to use the prefix rdf to denote that the attribute belongs to the RDF Syntax namespace. We should read:

    <rdfs:Class about="...

All these errors had been send to Ralf Swick of the W3C. The errors 6, 8 and 10 are already reported in the official error page at http://www.w3.org/TR/1999/PR-rdf-schema-19990303/errata. The current page can be found in Appendix B. In Appendix A you can find a corrected version of the Compact XML Serialization for the RDF Schema.

2.6 Pending Issues of RDF & RDFS

In this section I will mention some open questions and pending issues regarding RDF. Some of these issues are further discussed in the archive of the mailing list that can be found at http://lists.w3.org/Archives/Public/www-rdf-interest/.

- The semantic of rdfs:subClassOf and rdfs:subPropertyOf does not fit to the names of the properties, e.g., when the property rdfs:subClassOf is used it points from the sub class to the super class as shown in Figure 2.13, where Class A is defined as a sub class of Class B. This would be expressed by the triple: {rdfs:subClassOf, ClassA, Class B}. If we would translate this to a normal sentence using Class A as the subject and Class B as the object we would have:

  The rdfs:subClassOf of Class A is Class B.

  This sounds more like Class A being a super class of Class B. Therefore the property rdfs:subClassOf should be renamed to rdfs:superClass.

  For the same reason rdfs:subPropertyOf should be renamed to rdfs:superProperty.

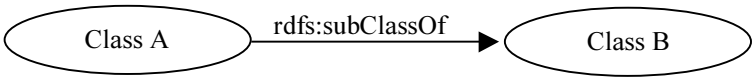


Figure 2.13 The Graph Model for defining Class A as a sub class of Class B.

This subject first discussed in: http://www.mailbase.ac.uk/lists/rdf-dev/1999-07/0007.html

- It is still a question what are and how to deal with identical statements. In the set of statements it is possible that the same statement appears, like {pred_1, subj_1, obj_1} and {pred_1, subj_1, obj_1}. In the case the obj_1 is a resource these statements are identical, in the case it is a literal there exist different opinions. The question is how can we say that two literal are equal. RDF M&S gives no rule for it (under progress).

  A set of statements is a set of facts, and since any set by standard definition of the term can not contain the same element twice, multiple identical members of the statements should be ignored by an RDF processor.

  This subject is discussed in: http://lists.w3.org/Archives/Public/www-rdf-comments/1999AprJun/0000.html

- There is no way the core RDF could know if an URI is a document that could be accessed for the extraction of more data or not.

  Examples:

  ... <rdf:Description about="http://www.isc.forth.gr/something#else"> ...

  in this case the resource may be just an anchor in some text, an XML element or an RDF property or class.

  ... <rdf:Description about="http://www.isc.forth.gr/something"> ...

  in this case the resource may be one document or an entire Web site.

  It might be that the resource ceases to exist or becomes unavailable. Furthermore the RDF M&S says: "A resource may also be an object that is not directly accessible via the Web, e.g. a printed book."

  This subject is discussed in: http://lists.w3.org/Archives/Public/www-rdf-interest/1999Nov/0106.html

- It might be useful to lay the RDF (syntax) namespace (see Appendix C) and the embedded RDF of RDF Schema (see Appendix A) together into one namespace. At the moment the embedded RDF of RDF Schema already describes all resources defined in the RDF (syntax) namespace. Both namespaces contain the rdfs:comment property with different values. This might force errors of even confusion and is at least a redundant storage of information.

- Using RDF in the described way we will never be able to enumerate the elements of a class. There is no way to constraint users not to enter there resources to a class.

  Example:

  If you define a class called 'Digit' and enter resources representing the numbers 0-9 there is always the possibility for other users to reuse your class and enter further resources to it.

3 Towards an RDF Formalization

There are still open questions how to use RDF, what can be done with it and what it means. In this chapter I will point out some of these questions and will give possible solutions. My view is based on the definitions as described in section 3.1. I want to give special attention to section 3.4, this subject is hardly discussed so fare in existing papers.

Section 3.5 gives an overview about some existing RDF applications. You might find some more applications at the reference page of [VRP Home].

3.1 Definitions

In section I will introduce a model of the basic classes included in the RDF Schema. I will introduce some semantic useful extra constraints not explicitly stated by the RDF specifications [RDF M&S] and [RDF Schema].

In the following we think of the property rdf:type with the meaning of 'element of' or 'instance of', the property rdfs:subClassOf as 'sub set of' (Script: $\subset$) and think of classes as sets. But we need to distinguish between the set itself and the resource representing this set. Otherwise we would have in Figure 3.1 an example of a sub set containing its super set. In the most cases it is obvious if I mean the set or the resource. So, I kept the same name for it.

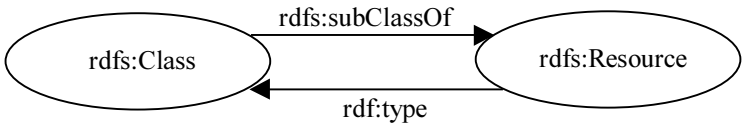


Figure 3.1 The relation between rdfs:Class and rdfs:Resource.

Based on the classes defined in RDF and RDF Schema as shown in Figure 2.12 we can make the following definitions:

Definition 1 *rdfs:Resource* is the set of resources.

Definition 2 *rdfs:Literal* is the set of literal.

Definition 3 *rdfs:Class* is the set of classes.

Definition 4 *rdf:Property* is the set of properties.

Definition 5 *rdf:Statement* is the set of reified statements.

Definition 6 *rdf:Bag* is the set of bags.

Definition 7 *rdf:Seq* is the set of sequences.

Definition 8 *rdf:Alt* is the set of alternatives.

RDF allows multiple classification. Semantically we are forced not to mix elements of rdf:Property, rdfs:Class, rdf:Bag, rdf:Seq, rdf:Alt, rdf:Statement and rdfs:Literal. This brings us to the following 'Type Mix' constraints:

C1: $\text{rdf:Property} \cap (\text{rdfs:Class} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C2: $\text{rdfs:Class} \cap (\text{rdf:Property} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C3: $\text{rdf:Bag} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C4: rdf:Seq ∩ (rdf:Property ∪ rdfs:Class ∪ rdf:Bag ∪ rdf:Alt ∪ rdf:Statement ∪ rdfs:Literal) = ∅

C5: rdf:Alt ∩ (rdf:Property ∪ rdfs:Class ∪ rdf:Seq ∪ rdf:Bag ∪ rdf:Statement ∪ rdfs:Literal) = ∅

C6: rdf:Statement ∩ (rdf:Property ∪ rdfs:Class ∪ rdf:Bag ∪ rdf:Seq ∪ rdf:Alt ∪ rdfs:Literal) = ∅

C7: rdfs:Literal ∩ (rdf:Property ∪ rdfs:Class ∪ rdf:Bag ∪ rdf:Seq ∪ rdf:Alt ∪ rdf:Statement) = ∅

These constraints are visualized in Figure 3.2, where classes are shown by rounded rectangles and sub classes are completely enclosed by their super class. The possible intersection between rdfs:Literal and rdfs:Resource in our view is denoted by two question marks to show that it depends on further evolutions of RDF and on the view of the user to allow this intersection. This subject will be discussed in section 3.3.3.

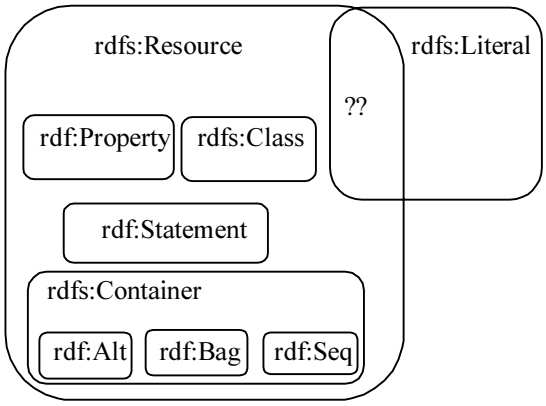


Figure 3.2 An overview of the main classes of RDF and RDF Schema.

3.2 Type Inheritance

In RDF types can be inherited in two ways.

1. Type inheritance for classes. Classes inherit all type declarations of their super classes.
2. The second is the type inheritance for instances of classes. Instances of a class (a resource is an instance of a class C if it has rdf:type = C) inherit the types of the super classes of the class they belong to (they are also instances of the super classes).

Figure 3.3 shows an example for type inheritance for instances. For the classes rdf:Property, rdfs:ConstraintResource and rdfs:ConstraintProperty it would not be necessary to declare them to be rdf:type = rdfs:Class as it is done in RDF Schema. This type could be inherited by the super class rdfs:Resource and would be an example for the type inheritance for classes.

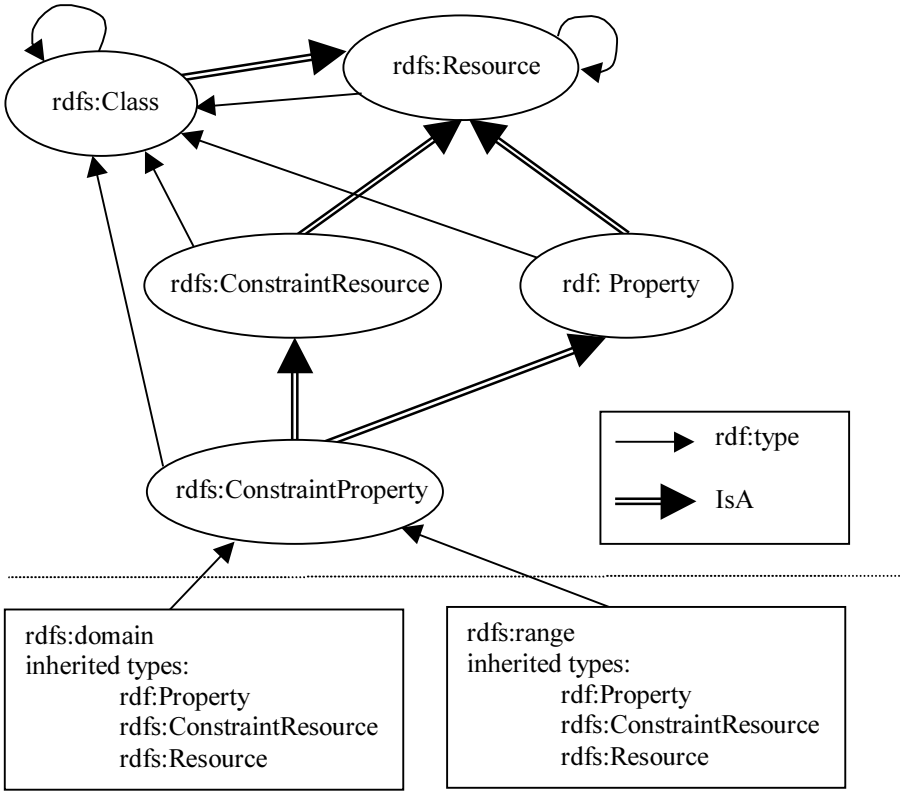


Figure 3.3 An Example for type inheritance. The classes rdfs:ConstraintProperty, rdfs:ConstraintResource, rdf:Property and rdfs:Class will inherit the type rdfs:Resource from their super class rdfs:Resource, not illustrated above.

3.3 Implicit RDF Instantiation

The RDF Schema provides three classes with implicit instances:
rdfs:ContainerMembershipProperties, rdfs:Resource and rdfs:Literal.

3.3.1 rdfs:ContainerMembershipProperties

The properties rdf:_1, rdf:_2, rdf:_3, ... belong implicit to the namespace of RDF. It is an infinite set of properties and can not be described by the mechanism RDF provides. They are implicit instances of the class rdfs:ContainerMembershipProperties.

These properties only make sense in the use with a container. Therefore I constrained the rdfs:domain of these properties to be rdfs:Container (see Table 2.1). The rdfs:range is unconstraint.

3.3.2 rdfs:Resource

During the creation of the statements we do not always create the information if something is a resource. In those cases I implicit say, that the resource is an instance of the class rdfs:Resource. For example:

```
<rdf:Description about="Es">
       <dc:Creator>Stephen King</dc:creator>
```

```
</rdf:Description>
```

This RDF description (the namespace declarations left apart) would create the following statement (if we assume it is saved in a file called ‘xy’):

(http://purl.org/metadata/dublin_core#Creator, file:xy#Es, Stephen King)

In this case the subject 'file:xy#Es' would be implicit an instance of rdfs:Resource.

3.3.3 rdfs:Literal

In the [RDF Schema] rdfs:Literal can be found in Figure 1 and Figure 2 shown just as an rdfs:Resource and not as a rdfs:Class. In the ‘Basic XML Serialization’ of [RDF Schema] it is defined as a rdfs:Class and in the ‘Compact XML Serialization’ it is defined as a rdfs:Class being a sub class of rdfs:Resource. The rdfs:Literal is representing a set and is used in the rdfs:range definitions for rdfs:comment and rdfs:literal. This makes it rich in meaning to define it as a rdfs:Class. In the case it would be defined as a sub class of rdfs:Resource all elements of the class would inherit the type rdfs:Resource, but a textual string (literal) has no URI and therefore is no resource. So, in the following we will use the definition of the ‘Basic XML Serialization’ saying rdfs:Literal is a class without being sub class of rdfs:Resource. In the same way I already introduced rdfs:Literal in chapter 2.5.1.

A *literal* is a simple string or other primitive data type (any XML text with „<“, „>“ and „&“ escaped) or any well-formed XML that is not further evaluated [RDF M&S]. (Well-formed: For all elements, if the start-tag is in the contents of another element, the end-tag is in the contents of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other [XML].) Such a literal does not have a URI and therefore is not a resource. This makes it impossible to use the rdf:type property (domain of rdf:type = rdfs:Resource) to say a literal is an instance of a class. Therefore it is necessary to define a literal implicit as an instance of the class rdfs:Literal. This makes the class rdfs:Literal to a special class, representing not only a set of resources, while all other classes represent only a set of resources.

There are some efforts to create classes containing resources to represent primitive data types like integer, real, date, ... (see http://www.mailbase.ac.uk/lists/rdf-dev/1999-08/0033.html). Consequently these classes could be entered into the class hierarchy as a specialization (sub class) of the class rdfs:Literal. Instances of these classes would inherit the type definition of all super classes they belong to. This would mean to allow resources to be instances of the class rdfs:Literal. (Of course the user of RDF is also free to enter a resource direct to the class rdfs:Literal.) This circumstance makes it hard to differentiate between a literal and a resource.

If we allow resources to have the type rdfs:Literal the use of rdfs:Literal to constraint the domain of a property would make sense. Otherwise the use of rdfs:Literal as domain should be disallowed.

There need to be further discussions on this subject.

3.4 RDF Constraints

In this section I provide a complete list of all RDF constraints and their reasons I see (except the Type Mix constraints C1-C7 I already stated in section 3.1). You can find the complete list of constraints in Appendix E.

An RDF/XML-File represents a set of statements. These set of statements contains the statements of the file plus statements form used namespaces (recursive, Example 1) plus

inherited statements in the case of type inheritance (Example 2). We will call this set of statements the *statement-model of the file*.

During this section we use the shorthand way of writing for the namespaces.

Example 1:

(Notation: If r is a resource, then r.URI represents the URI of the resource r. A literal is denoted by quotation marks.)

If the file would contain the following statement about a resource r:

(rdf:type, r.URI, rdf:Bag).

So, the statement-model of the file would also contain the statements from the RDF namespace for the resources rdf:type and rdf:Bag. These are (Can be created out of the RDF namespace as printed in Appendix C.):

(rdf:type, rdf:Bag, rdfs:Class),

(rdfs:comment, rdf:Bag, „ An unordered collection“),

(rdf:type, rdf:type, rdf:Property),

(rdfs:comment, rdf:type, „ Identifies the Class of a resource“).

Then of course we also need to enter the statements for the resources: rdf:Property, rdfs:comment and rdfs:Class and so on.

Example 2:

If the file contain the resource rdfs:domain, then this resource would inherit the types rdfs:Resource, rdf:Property and rdfs:ConstraintResource as described in Figure 3.3. So, the statement-model of the file would also contain the following statements pointing these types to the resource rdfs:domain:

(rdf:type, rdfs:domain, rdfs:Resource),

(rdf:type, rdfs:domain, rdfs:ConstraintResource),

(rdf:type, rdfs:domain, rdf:Property).

For the validation we suppose to have the statement-model of the file called M as described above. So, M is a set of statements. The things need to be tested for the validation are listed below. Additional the Type Mix constraints C1 - C7 need to the tested as described in 3.1.

Property Check:

For all statements s=(predicate, subject, object) of M the predicate must be a property (a direct or indirect instance of the class rdf:Property) and the subject must be a resource. As described in 3.3.2 the subject is always a resource and we do not have to test it here. We need to test:

C8: $\forall s \in M,\ s = (p, s, o)\ \exists\ x \in M,\ x = (\text{rdf:type}, p, \text{rdf:Property})$

Loop sCO:

For the subclass definitions the property rdfs:subClassOf is used. It defines a class to be a sub class of another. Semantically this corresponds to a specialization. For example you could make a specialization for Animals creating the classes Mammals, Birds and Flying Animals, as shown in Figure 3.4.

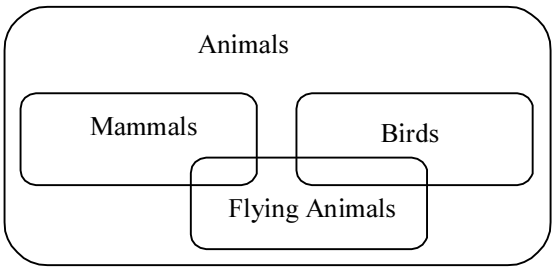


Figure 3.4 Example of a specialization for classes.

The property rdfs:subClassOf (Script: <) is *transitive*. If class A is sub class of class B and B is sub class of class C then A is also sub class of C. This transitivity is expressed by inherit the type definition as described in 3.2. So, we do not really need to enter it as a constraint, but to be complete:

C9: $A<B \wedge B<C \Rightarrow A<C$ | A,B,C are classes

A class can not be a rdfs:subClassOf of itself, otherwise it would not be a specialization. So, rdfs:subClassOf is *anti-reflexive*.

C10: $A<B \Rightarrow A \neq B$ | A,B are classes

Out of the fact of rdfs:subClassOf being transitive and anti-reflexive follows the constraint that there are no loops allowed in the rdfs:subClassOf definitions (a loop would correspond to a circle in the directed graph representing rdfs:subClassOf). Mathematically this would correspond to be *anti-symmetric*. To check for these loops it is important to follow the rdfs:subClassOf graph for each class to the root. This is done by generating the statement-model of the file.

Loop sPO:

The property rdfs:subPropertyOf is used to define a sub property of a property. Semantically this corresponds to a specialization. For example you could specialize the property parent-of by defining a property father.

The property rdfs:subPropertyOf (Script: <) is *transitive*. If property p is sub property of property q and q is sub property of property q' then p is also sub property of q'. This fact can not be expressed by type inheritance like the rdfs:subClassOf can. Therefore we enter it to the constraints.

C11: : $p<q \wedge q<q' \Rightarrow p<q'$ | p,q,q' are properties

A property can not be a rdfs:subPropertyOf of itself, otherwise it would not be a specialization. So, rdfs:subPropertyOf is *anti-reflexive*.

C12: $p<q \Rightarrow p \neq q$ | p,q are properties

Out of the fact of rdfs:subPropertyOf being transitive and anti-reflexive follows the constraint that there are no loops allowed in the rdfs:subPropertyOf definitions (a loop would correspond to a circle in the directed graph representing rdfs:subPropertyOf). Mathematically this would correspond to be *anti-symmetric*. To check for these loops it is important to follow the rdfs:subPropertyOf graph for each property to the root. This is done by generating the statement-model of the file.

Range Check:

The value of a property whose range is A must be an instance of class A. In the following we will think of a property p as of a relationship. In this way of thinking a property represents a set of relations between resources and values (resources or literal) (Script: [p] := {(subject, object) | subject is a resource and object is a *resource* or a *literal*}). With the property rdfs:range the set of allowed objects can be further constraint and must be an instance of the class specified as the value of the rdfs:range property (Script: p has range A | p a property and A a class). So, the following constraint can be stated:

C13: p has range A $\Rightarrow$ [p] = {(subject, object) | subject is a resource and object $\in$ A}

Unique Range:

A property can have at most one range property.

C14: p has range A $\wedge$ p has range B $\Rightarrow$ A = B | p is property and A,B are classes

Domain Check:

With the property rdfs:domain the set of subjects for a property is further constraint. It must be instance of the class specified by the value of rdfs:domain specification given for this property. (Script: p has domain A | p is a property and A is a class). If there is more than one rdfs:domain specified for one property the used subjects must be instance of the union of the specified classes. So, the following constraint can be stated:

C15: p has domain $\{A_i\}$ $\Rightarrow$ [p] = {(subject, object) | subject $\in \cup_i\{A_i\}\}$ | A_i are classes and p is property

Sub Property Check:

If we define a property to be a specialization for another property we do not want to be able to use this sub property with values the super property is not allowed to be used with. Therefore the domain and range definitions of a sub property must fit to the domain and range definitions of the super property.

C16: p < q $\wedge$ p has range A $\wedge$ q has range B $\Rightarrow$ A = B $\vee$ A < B | p,q properties and A,B classes

Note: This is not explicitly stated by RDF Schema!

There is also nothing stated in the RDF specifications about the inheritance of range and domain definitions for sub properties, but I think it would be very useful. There might be the case that the user did not specify the domain and range of a sub property. I decided to inherit the domain and range of the super property in this case. To do so without making mistakes it is useful to have the correct order of the hierarchy of the properties. Otherwise we could have the following case illustrated in Figure 3.5: P3 is sub property of P2 and P2 is sub property of P1. Lets say for P3 and P2 are no domain and range declared and P1 has as domain the class A and as range the class B. If we first inherit from P2 to P3 and then from P1 to P2, we would have P3 without range and domain declaration. So, it would be allowed for P3 to connect objects not being allowed to be connect by P2 or P1 and this is not in the sense of a sub property.

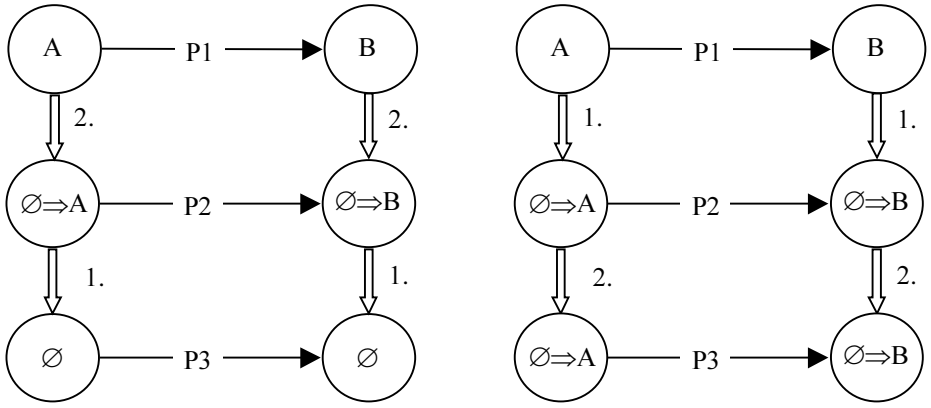


Figure 3.5 The order to inherit domain and range for sub properties.

Without knowing the topological order of the properties we are forced to go each time to the bottom not to create an error.

It is allowed for a property to have more than one super property. In the case that two of the set of this super properties are not related by the rdfs:subPropertyOf relation and the domain and range of those properties are not the same, the semantic right thing to do would be to inherit the intersection of the two classes. In RDF we can not create an intersection of two classes. Only if the two classes are related with rdfs:subClassOf (one of them is the sub class of the other), we know the intersection is the sub class. If this is not the case we could say it is still valid if we might find something less, like just a subclass of both (see Figure 3.6).

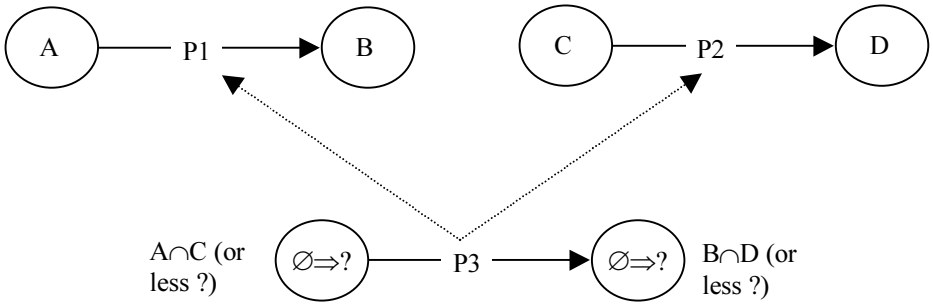


Figure 3.6 Domain and range inheritance for properties with more than one super property.

As you can see we could get in some trouble if we allow the inheritance of the domain and range definitions for sub properties, but I still think it would be worth doing so. Just for properties with more than one super property the user should be forced to explicitly state the range and domain constraints (if they should be used in a constraint way) to solve some trouble.

Statement Check:

A reified statement models a statement. To make the model unique for each statement that is reified we need exactly one (Script: $\exists'$) subject, predicate and object using the properties: rdf:subject, rdf:predicate and rdf:object. Lets say S is the set of reified statements contained in M ($s \in S \Leftrightarrow \exists\, x \in M \mid x = (\text{rdf:type}, s.\text{URI}, \text{rdf:Statement})$). Then we have:

C17: $\forall\, s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:predicate}, \beta = s.\text{URI}$

C18: $\forall s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:subject}, \beta = s.URI$

C19: $\forall s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:object}, \beta = s.URI$

3.5 Existing RDF Parsing Software

Various software and tools for RDF can be found at the reference page of [VRP Home] or at Dave Beckett's reference page [DB RES]. In this chapter we only present the main existing RDF Parsers (SiRPAC and Pro Solutions Perl Parser) and some validate applications of the SiRPAC. VRP will be presented in chapter 4. In Figure 3.7 you can see an overview of the programs introduced here and their relations between each other (for SiRPAC and rdf2cb see also chapter 5).

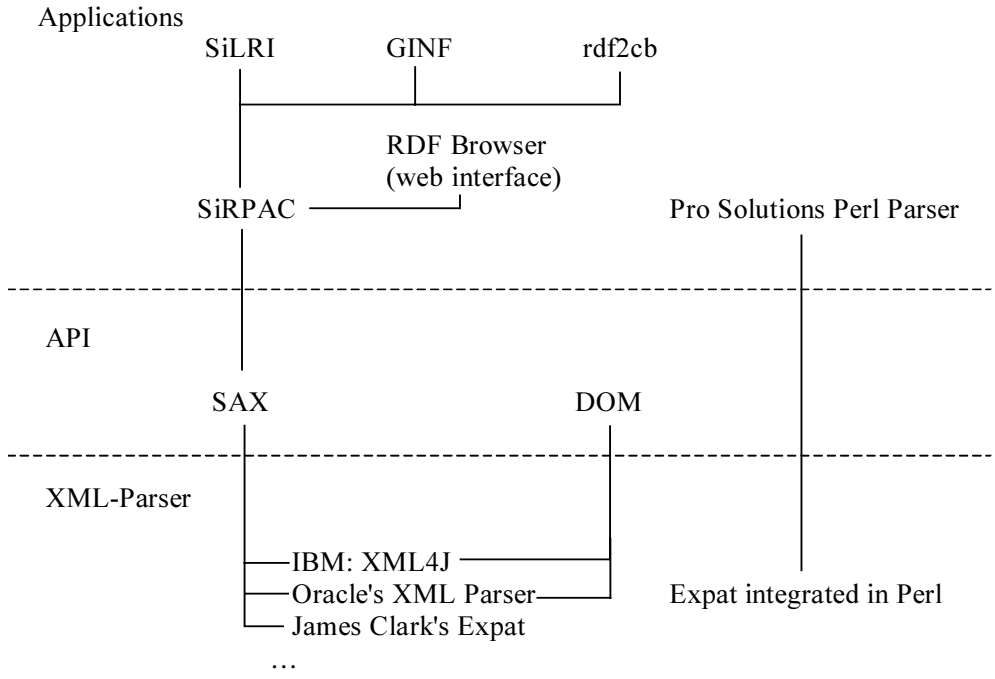


Figure 3.7 An overview of the existing programs and their relations.

3.5.1 SiRPAC

The first RDF-Parser on the market (since 28-Jul-98) was SiRPAC – Simple RDF Parser & Compiler by Janne Saarela, W3C. It was the only RDF Parser for a long time. The program written in Java™ compiles RDF/XML documents into triples of the data model in the form:

triple ("predicate", "subject", "object").

SiRPAC allows RDF being contained into HTML or XML documents. It transfers the RDF part of the document to the referring 3-tuples. SiRPAC on its one does not check the constraints specified in the RDF Schema Specification. It is not a validating parser and only tests the syntax for RDF as specified in [RDF M&S]. SiRPAC is build on top of the Simple API to XML documents (SAX), which can be used by various XML Parsers. It first builds up the parse tree

and then extracts the statements. To run SiRPAC you first need to install an XML Parser supporting the SAX interface.

There is an interactive compilation service with an additional Java applet based on the SiRPAC that allows one to browse through the RDF data model. At this RDF Browser [Browser] at the W3C site you can enter any RDF and send it to the server to show the triple and visually browse them (requires Java).

To check the constraints specified in RDF Schema you have to run additional applications on top of SiRPAC like:

- SiLRI – the Simple Logic-based RDF Interpreter [SiLRI]. A simple deductive database, written by Stefan Decker and Juergen Angele (University of Karlsruhe) implemented in Java™. SiLRI is able to reason with metadata in the XML serialization of RDF using SiRPAC and was developed in the context of the Ontobroker-project. Latest release V1.1.1, 9 May 1999.

  You can find a the list of validating rules supported by this version in Appendix F. These rules are written in Frame Logic [F-Logic] and only contain the Domain and Range Check as described in 3.4 and provide the type inheritance.

- RDF parser / serializer / validator from the Generic Interoperability Framework [GINF] project by Sergey Melnik, Digital Libraries Project, Database Group, Stanford University, USA. First announcement 23 June 1999. (Uses SiRPAC and AElfred XML parser)

  It performs validation according to the following algorithm:

```
foreach triple in model {
  if triple.predicate == RDF.type
    /* check whether triple.object is an RDFS.Class */
  if triple.predicate is a RDFS.ContainerMembershipProperty
    /* check whether triple.subject is an RDFS.Container --- [RDFSchema] specs seems
not to require it */
  foreach d in validDomains(triple.predicate)
    /* check whether triple.subject is RDF.type of one such d */
  if validRange(triple.predicate) != null    // there can be only single
                                             // range specification
    /* check whether triple.object is RDF.type of validRange(triple.predicate) */
}
```

  Currently, unsupported constraints are ignored. This means the program only supports the Domain and Range Check.

So, at the moment none of these validation extensions of SiRPAC support a full validation against all constraints specified in RDF M&S and RDF Schema.

There is a big set of further applications using SiRPAC. In this draft we will only refer to:

- rdf2cb converting the RDF Schema into O-Telos frames, written by Christian Capelle (University of Hannover) implemented in Perl [rdf2cb] (see 5.2 for more details).

3.5.2 Pro Solutions Perl Parser

On 27 Oct 1999 Pro Solutions Ltd. announced a production quality Resource Description Framework (RDF) Model & Syntax Specification (22-Feb-99) parser. The parser is implemented as a Perl module and is available for download at the Pro Solutions Warehouse section at URI: http://www.pro-solutions.com/download.

The parser has a similar architecture to that of SiRPAC i.e. it first builds the parse tree using the Expat XML parser shipped with the XML::Parser Perl module and then starts the translation.

Pro Solutions Ltd. is currently investigating the adaptation of a stream-based parsing approach. The implementation does not support the ‘aboutEachPrefix’ feature of RDF, otherwise the implementation is complete.

3.5.3 New Ways?

For all existing software a XML-Parser is necessary to run the program and the RDF-Parser is build on top of it written from scratch. This way has two main drawbacks:

1. It is hard to update or extend the RDF-Parser in the case of changes in the RDF grammar.
2. You need to install an extra XML-Parser and the features of the RDF-Parser depends on the features of the XML-Parser.

With the Validating RDF Parser (VRP), we will introduce in 4 chapter, we want to explore a new way of building an RDF-Parser by using the standard tools JFlex/CUP similar to LEX/YACC. This way the parser can be updated or extended by changing the grammar for the parser instead of changing a program written from scratch. The parser does not need an extra XML-Parser and provides all necessary features that are needed to parse a RDF/XML-File. The non important XML parts of a RDF/XML-File will just be tested for the well-formed constraint and not further evaluated. This makes VRP very quick and the whole program does not contain useless code.

Additional VRP is the first RDF-Parser that provides a full validation module with no need to install further programs. It validates against all constraints described in section 3.4 (a compact list of the constraints can be found in Appendix E). These include all constraints described by [RDF M&S and RDF Schema].

4 Architecture and Implementation of VRP

This chapter gives an introduction to the basic implementation fundamentals of the Validating RDF Parser (VRP). VRP is based on a compiler generator and therefore I provide in section 4.9 the comparison of some existing compiler generators for Java™.

4.1 Introduction to VRP

The ICS-FORTH Validating RDF Parser (VRP) contains a parser and a validator module. The parser analyses the statements of a given RDF/XML document according to the RDF M&S specification. These statements are represented by a RDF object model implemented in Java™ (not persistent). The validator access the generated object model in order to validate the information against the RDF Schema constraints. Several output options are supported by VRP (see [VRP Home]. In Figure 4.1 you can see an overview of the system of VRP. The program (including source code) is free available and can be downloaded at [VRP Home].

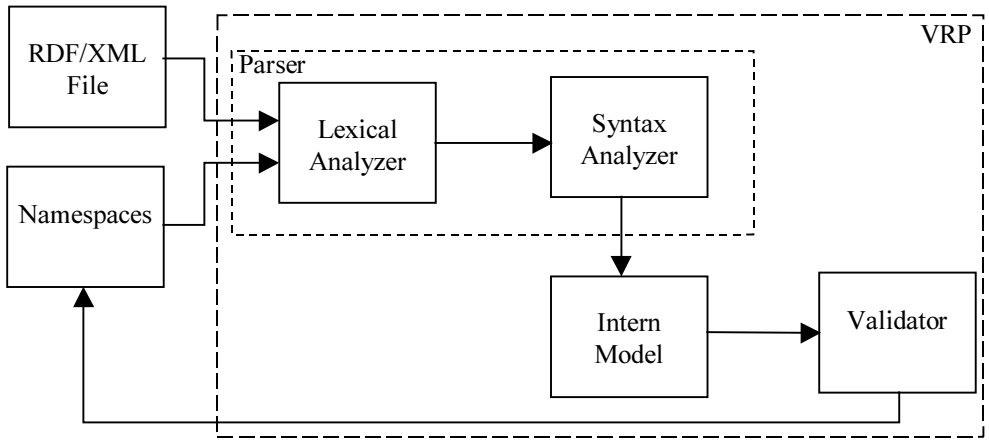


Figure 4.1 Overview of the VRP system.

Main Features

- 100% Java™
- you only need Java™ 1.2 or higher
- understands embedded RDF in HTML or XML
- full Unicode support
- easy to install
- based on compiler generator tools for Java CUP/JFlex (similar to the standard C tools YACC/LEX)

4.2 Evolution of VRP

The motivation for VRP was based on building a repository for metadata using the Semantic Index System [SIS]. The first idea was to use a parser that analyses statements and send them during the parsing direct (via a Loader, creating SIS/Telos classes/properties out of the statements) into the SIS system and save them in a model that could express the RDF model, as shown in Figure 4.2. We then wanted to use the existing constructs inside the SIS system to validate the entered statements.

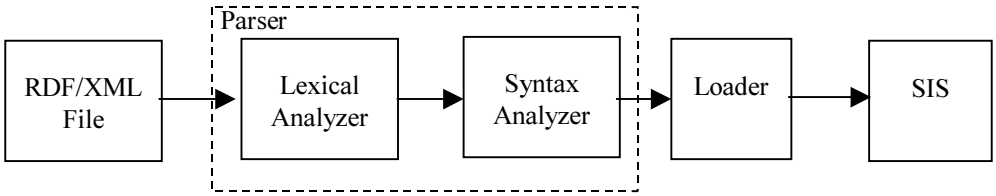


Figure 4.2 Evolution of VRP (first).

In RDF the order of statements has no meaning and you can use properties classes or resources and define them later in the same file or in a namespace. In SIS and similar systems you are not allowed to do so. To solve these problems I decided to create an intermediate model to be able to collect statements that belong together and enter them in any order I like (see Figure 4.3).

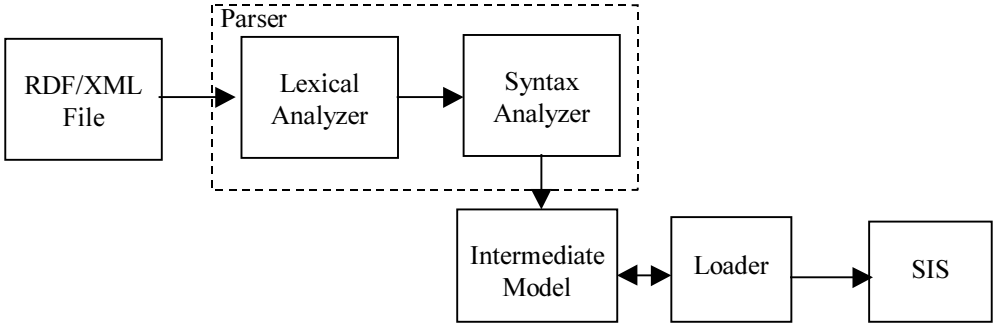


Figure 4.3 Evolution of VRP (second).

With the need of an intermediate model some questions obtruded:

- What kind of validation can be done during the creation of the intermediate model?
- What kind of validation must be done afterwards?
- What is the best structure for the intermediate model?

I will discuss these questions in the paragraphs 4.5 and 4.6. Not to separate the validation parts too much and to enrich the parser, I decided to use the intermediate model for the whole validation as described in 3.4. Doing so we now have an application for RDF independent of SIS that analyses the statements of a RDF/XML file and validates them against RDF and RDF Schema constraints. The VRP system as shown in Figure 4.1.

4.3 The Java Packages of VRP

The Java™ classes of VRP are collected in the *package VRP*. It has two sub packages the *packages Parser* and *Model*. An overview of these packages and their classes can be found in Figure 4.4. The functionality of the package Parser is described in chapter 4.4 and for the package Model in 4.5 and 4.6. For further information about single methods of the classes please have a look at the Java™ Documentation. It can be found online at [VRP Home] and is included in the downloading version of VRP on the same page.

The source codes for the classes can be found in a separate directory of VRP called ‘src’ containing subdirectories for the single packages. It also contains the files ‘parser.cup’ and ‘lexer.jflex’. These files contain the source code to run CUP and JFlex for the generation of the Java™ files for the syntactical and lexical analyzers (see 4.4).

To run VRP the *package java_cup* is necessary. I do not discuss the classes of this package here, it belongs to the CUP [CUP] program.

- VRP
  - Class VRP.Main (containing the main method and variables for RDF and RDF Schema elements)
  - Class VRP.Output
  - Class VRP.RDF_Error
  - Model
    - Class VRP.Model.Link (providing an object for storing the subject-object pairs for properties)
    - Class VRP.Model.Model (main class for creating the model)
    - Class VRP.Model.RDF_Class
    - Class VRP.Model.RDF_Container
    - Class VRP.Model.RDF_DAG (for representing a Direct Acyclic Graph)
    - Class VRP.Model.RDF_Property
    - Class VRP.Model.RDF_Resource
    - Class VRP.Model.RDF_Statement
    - Class VRP.Model.RDF_Validator (main class for validating a model)
    - Class VRP.Model.Resource
  - Parser
    - Class VRP.Parser.CUP$parser$action (action code for the parser)
    - Class VRP.Parser.Lexer
    - Class VRP.Parser.parser
    - Class VRP.Parser.UpCall (helper class for storing property-value pairs)
    - Class VRP.Parser.UpCall_aE (helper class for storing the information to create the statements if the ‘rdf:aboutEach’ attribute is specified)

Figure 4.4 Summary for the package VRP with the sub packages VRP.Model and VRP.Parser.

The generation of the sub packages is for a better modularization of the program. This enables the user to reuse the parsing or validation module. For example to reuse the validation module with other RDF-Parsers you would just need to implement the method ‘VRP.Model.Model.addStatement(String pred, String subj, String obj, boolean res)’ to enter the statements to the model and make some changes in the following classes:

- Class VRP.Main that coordinates the whole program.
- Class VRP.Model.RDF_Validator that connects to the namespaces.

The Figure 4.5 illustrates the relations between the most important classes. The Figure does not show the connections to the classes VRP.Output and VRP.RDF_Error. These classes are used by the most classes of all packages for reporting information or errors.

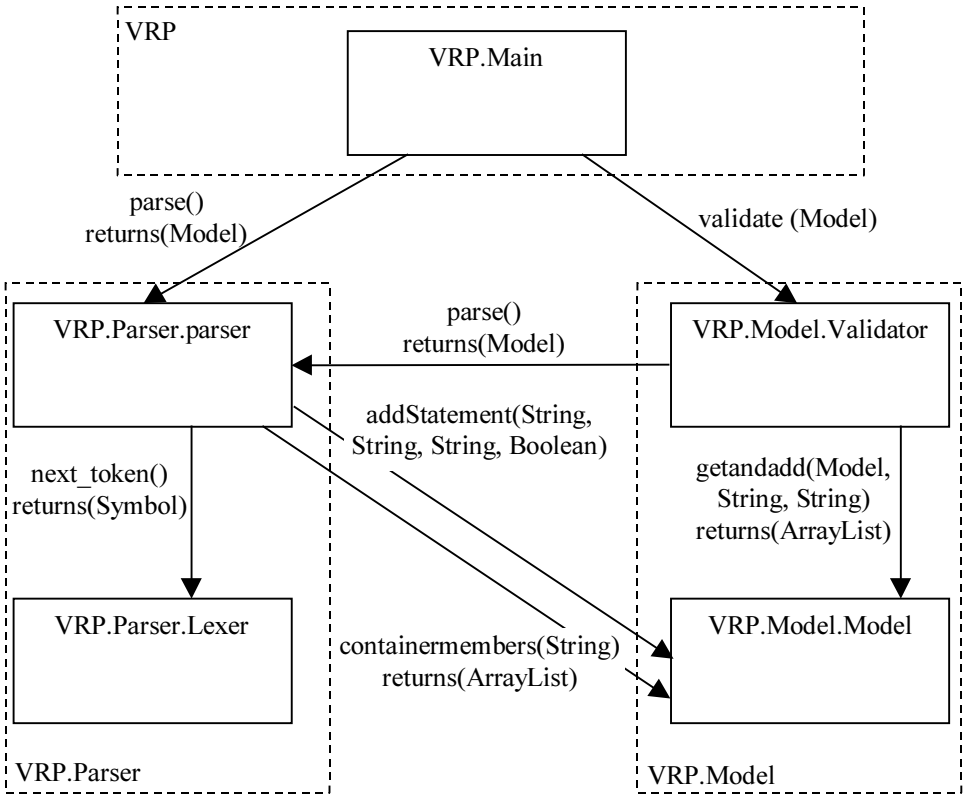


Figure 4.5 The relations between the most important classes. The arcs are named by the called methods. The possible feedback is shown in the 'return' statement.

For the extensions that are implemented inside the package VRP.Model to realize the model please see Figure 4.7. The complete lists of variables, constructors and methods for all classes are enumerated in Appendix G.

4.4 The Parser

The parser is build using the CUP (version 0.10j, July 24, 1999) parser generator[2] for Java™ and the lexical analyzer generator[3] JFlex (version 1.2.2, August 23, 1999). Both programs are free available (including source code) in the Web ([CUP and JFlex]), but are not necessary to run the program.

The reasons for creating a new parser instead of using the existing parser SiRPAC are further discussed in the chapter 5. The advantages for creating a new parser using standard compiler generators instead of existing XML parsers can also be found there. The decision for using CUP and JFlex is based on the chapter 4.9.

Limitations of the parser

- The parser does not support the RDF attribute "aboutEachPrefix". With this attribute we would refer to all resources starting with the value of the attribute (see 2.4.3.1). For a efficient use of this feature sitemaps are needed to describe the hierarchical description of Web sites.
- For the RDF attribute "aboutEach" the parser does not support references to other files.

[2] similar to the standard C parser generator YACC

[3] similar to the standard C lexical analyzer generator LEX

4.4.1 The Lexical Analyzer

The lexical analyzer (lexer) is responsible for creating the token stream out of the RDF/XML file and send it to the syntax analyzer. The RDF descriptions may be contained within XML or HTML files. Therefore the lexer must be able to distinguish whether to read over the elements or to create tokens. Furthermore the XML attribute 'xmlns' might be important for the use of prefixes inside the RDF descriptions even if they are declared outside. For example in [RDF Schema] we can find in the beginning of the source code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
           "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML   xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs = "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
        xmlns:dc = "http://purl.org/dc/elements/1.0/"  >
<HEAD>
   <TITLE>Resource Description Framework (RDF) Schema Specification</TITLE>
 <STYLE TYPE="text/css">
     .EXAMPLE { margin-left: 1em }

   </STYLE>
   <LINK rel="stylesheet" type="text/css" media="screen"
      href="/StyleSheets/TR/W3C-PR">
   <rdf:RDF>
   <rdf:Description about=""
       xmlns:ddc="http://purl.org/net/ddc#"
       dc:Title="Resource Description Framework (RDF) Schema Specification"
       dc:Description=" ... "
       ...
```

In the HTML element are the prefixes 'rdf', 'rdfs' and 'dc' declared. They are used inside the RDF description. The definitions of these prefixes and the element structures (the prefixes are only valid inside the element they are defined) must be send to the syntax analyzer, while the rest is not important. VRP just test for the well-formed constraint outside the RDF descriptions. For HTML there are some special elements with optional or forbidden end-tags. A list of the HTML 4.0 elements can be found at [HTML 4.0]. These special elements will be completely ignored by VRP.

To solve these problems and not to get conflicts in coordinating the incoming character stream to the tokens the lexer has different states. These states are shown in Figure 4.6. The arcs representing the possibility to change from one state to another. The conditions for doing so are shown beside the arcs. Strings inside double quotas must be matched, the other strings represent token names. The definition of the tokens can be found in the source code of *VRP/Parser/Lexer.java*. The '*' represents any valid token for this state, except those mentioned at another arc coming from this state.

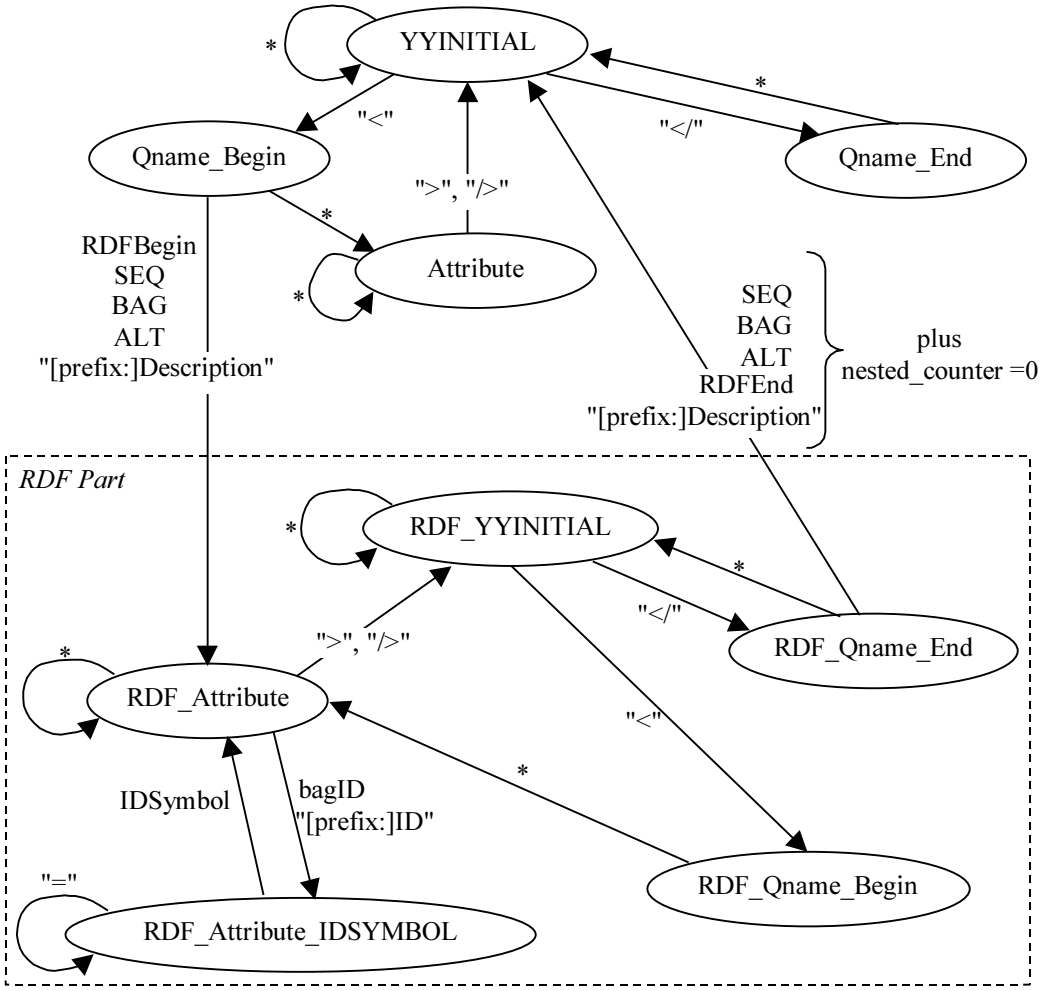


Figure 4.6 The states of the lexical analyzer.

With the *nested_counter* (see Figure 4.6) we make sure not to leave the RDF Part too early, e.g., in the case of a container contained by a Description.

To realize this lexical analyzer I used the tool JFlex [JFlex] that is using a stream-based reading of the file and so allows a stream-based parsing. This enables high speed and less usage of memory.

4.4.2 The Syntax Analyzer

The syntax analyzer (SA) analyses the statements of a given RDF/XML document according to the RDF M&S specification.

He causes the lexical analyzer to send the tokens and compares them with the given grammar. In the action code of the SA the statements are created and send to the model. A statement contains the information about the predicate, subject and object in a string form plus a Boolean saying whether the object should be interpreted as a resource or as a literal.

4.5 The Model

The Model is used to store all the information of an RDF/XML file and the important information of relevant namesapces.

I explored different ways of representing the RDF model:

Set of Triples

The statements could be represented as a set of triples (predicate, subject, object).

- The easiest way of storing the statements.
- There would be no semantic validation during the creation of the model.
- For the validation it would be necessary to traverse the set frequently.
- High costs to traverse the model.

Adjazence Lists

Adjazence Lists are a common way to represent a directed graph. Each node of the graph would be represented by a list containing all nodes you could reach from this node. To represent RDF we would need to represent a labeled graph and therefore would be forced to save the list as a list of tuples. There are two ways either to define the subjects of the statements as the nodes of the graph or the properties. However for both representations:

- Only a few semantic validation possible during the creation of the model.
- Still it would be necessary to traverse the set frequently.

We will use this structure only for special purpose testing, e.g. cycles in sub class/property definitions.

Our Solution

Based on the model described in chapter 3 we created different Java™ classes to represent the basic RDF elements. The idea is to include the knowledge of the embedded RDF Schema (see Appendix A) and the RDF namespace (see Appendix C) and save the corresponding possible annotations within each object. In Figure 4.7 you can see an overview of the classes and their possible annotations. For subclasses the annotations are inherited. In the following Table 4.1 you can see what RDF elements the classes represent and the corresponding type definition.

Table 4.1 The Java™ Classes and their corresponding RDF Elements.

Java™ Class	**RDF Element**	**Corresponding Type Definition**
Resource	a resource without annotation	rdfs:Resource
RDF_Resource	a resource with annotation	rdfs:Resource
RDF_Class	a class	rdfs:Class
RDF_Property	a property	rdf:Property
RDF_Statement	a statement	rdf:Statement
RDF_Container	a bag, sequence or alternative	rdf:Bag, rdf:Seq, rdf:Alt

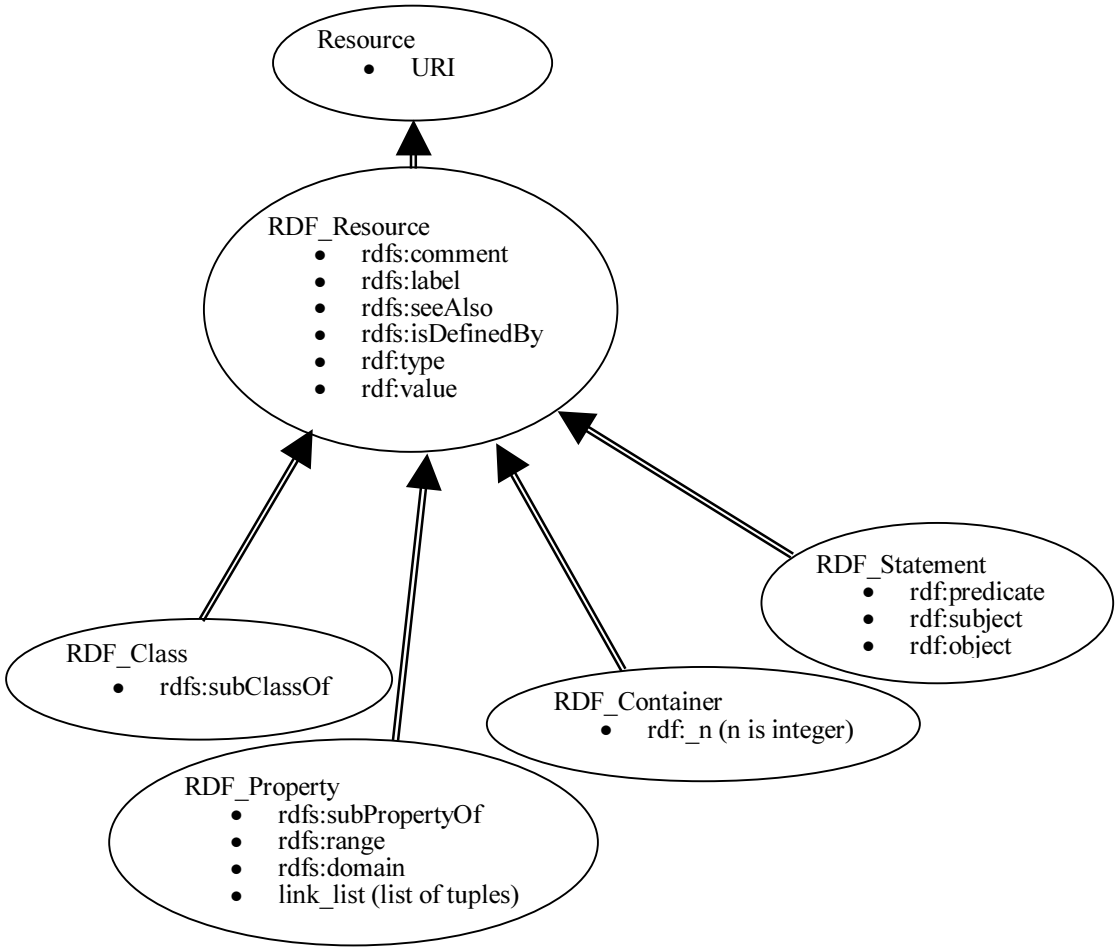


Figure 4.7 The Java™ classes of our model and their extensions. The arcs pointing from a class to a super class.

The parser will enter the statements one after another. Doing so we will try to create for each resource the corresponding Java™ object, based on the knowledge of the RDF and RDF Schema namespace. This means for a statement s=(predicate, subject, object) we would know that the predicate should be a property. In the case the predicate belongs to the RDF or RDF Schema namespace we would know their domain and range definitions as defined in Table 2.1 and Table 2.2 and could enter the subject corresponding to the domain class, the object corresponding to the range class.

Let us take for example the statement s=(http://www.w3.org//TR/PR-rdf-schema#subClassOf, X, Y). We would enter the resources X and Y as RDF_Classes, because the range and the domain of rdfs:subClassOf is rdfs:Class.

In RDF the order of the statements of the file and the order we connect to the namespaces has no meaning. So, we must be able to enter the statements in any order. Doing so, we can not always say in forehead if a Resource is later used as a Property, Class, Statement or Container. In the case we later get the information that the Resource is, e.g., a Property as shown in the example below, we would need to cast from the Resource to Property. The problem in Java™ is, like in every object oriented language, we are only able to cast from a subclass to a super class. In this case we need the other way round.

For example we have the following statements:

s1=(http://www.w3.org/1999/02/22-rdf-syntax-ns#value, file:a#b, Y)

s2=(http://www.w3.org/1999/02/22-rdf-syntax-ns#type, file:a#b,
http://www.w3.org/1999/02/22-rdf-syntax-ns#Property)

To solve this casting problem we use a hash map, containing the URIs of the Resources as keys and connect them with the corresponding object. If we want to cast an object, we need to transfer the

information of the super class object to a new object of the subclass and replace the old object with the new one.

After the casting we can change the pointer in the model for the Resource as you can see in Figure 4.8. So, if we would save connections between two Resources by using the pointer we would be forced to check and change these links each time we make such a casting. To go around these time spending procedure VRP saves every reference to a Resource by saving its URI as a String. To get information from the Resource we need to access the hash map to find the pointer to the Resource.

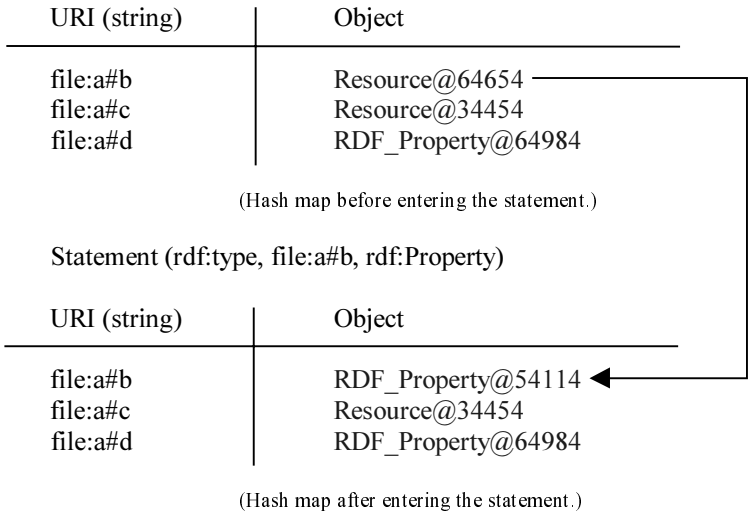


Figure 4.8 Example of casting a resource to a property.

The next question is where and how to represent a link between two resources. There are two opportunities, the first is to save the link with each subject (→ link as a tuple of predicate and object) of a statement and the second with each property (→ link as a tuple of subject and object). By representing the links at the properties we have a great benefit during testing the domain and range constraints (C:13, C:15) of the properties. We do not have to search for each property where it is used, we just have to check the right types of the subject (domain) and the object (range) of the links. Therefore I decided to represent the links with each property.

Let us take the example in 3.3 of the RDF M&S:

(I just added the namespace declatations!)

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:s="http://purl.org/metadata/dublin_core#">

       <rdf:Bag ID="pages">
              <rdf:li resource="http://foo.org/foo.html"/>
              <rdf:li resource="http://foo.org/bar.html"/>
       </rdf:Bag>

       <rdf:Description about="#pages">
              <s:Creator>Ora Lassila</s:Creator>
       </rdf:Description>
</rdf:RDF>
```

In the case it would be saved in a file called „xy“, the parser would create the following statements:

1.(http://purl.org/metadata/dublin_core#Creator, http://foo.org/foo.html, Ora Lassila, false)

2.(http://purl.org/metadata/dublin_core#Creator, http://foo.org/bar.html, Ora Lassila, false)

3.(http://www.w3.org/1999/02/22-rdf-syntax-ns#type, file:xy#pages, rdf:Bag, true)

4.(http://www.w3.org/1999/02/22-rdf-syntax-ns#_1, file:xy#pages, http://foo.org/foo.html, true)

5.(http://www.w3.org/1999/02/22-rdf-syntax-ns#_2, file:xy#pages, true)

These statements would create the following model:

URI	Object
s:Creator	RDF_Property URI = s:Creator links = (http://foo.org/foo.html, Ora Lassila), (http://foo.org/bar.html, Ora Lassila)
file:xy#pages	RDF_Resource URI = file:xy#pages types = rdf:Bag
rdf:_1	RDF_Property URI = rdf:_1 links = (http://foo.org/foo.html, Ora Lassila)
rdf:_2	RDF_Property URI = rdf:_2 links = (http://foo.org/bar.html, Ora Lassila)
http://foo.org/bar.html	Resource URI = http://foo.org/bar.html
http://foo.org/foo.html	Resource URI = http://foo.org/foo.html

(for a better reading we use the prefixes 's' and 'rdf' here instead of using the absolute paths)

As you can see a literal like ‘Ora Lassila’ is just entered within the property in the list of links as the object. For our model there is no special way to say something is a literal. We just say that every object in a list of links that can not be found in the column of URIs is a literal.

4.6 The Validation

The validation process comprise two phases: a) which can be done during the creation of the model and b) afterwards. The Table 4.2 shows what kind of validation will be done during which phase. The first column correspond to the constraints defined in chapter 3 (also can be found in Appendix E). Below I will discuss the validation for VRP according to the constraint names given in Table 4.2.

Table 4.2 Overview of the phases the constraints are tested.

Constraint Name	Check for constraint	during model creation	after model creation
Type Mix	C1-C7	+	-
Property Check	C8	+	+
Loop sCO	C9, C10	-	+
Loop sPO	C11, C12	-	+
Range Check	C13	+	+
Unique Range	C14	+	-
Domain Check	C15	+	+
Sub Property Check	C16	-	+
Statement Check	C17 -C19	+	+

Type Mix:

These constraints that the types rdf:Property, rdfs:Class, rdf:Bag, rdf:Seq, rdf:Alt, rdfs:Literal and rdf:Statement are not mixed is tested during creation of the model by creating an object of the corresponding Java™ class.

Property Check:

Each statement must start with a property. These constraint is guarantied by entering each predicate as an RDF_Property. After the creation of the model we just need to test if each element of the class RDF_Property has the type rdf:Property.

Loop sCO:

These constraints correspond to a loop in the rdf:subClassOf graph. Testing for loops is an expensive test, so, we decided to run these test separate from the model creation. Each time we enter a statement starting with the rdfs:subClassOf property, we enter the connection to a directed graph provided by an object of RDF_DAG.class. The graph is represented by using adjazence lists as described above. We do not need to represent a labeled graph here because we only use the rdfs:subClassOf property for this object. To make sure to enter all necessary information from all namespaces VRP follows the rdfs:subClassOf path to the root class. For more details about the namespace connection see section 4.7.

Loop sPO:

These constraints correspond to a loop in the rdf:subPropertyOf graph. We use the same mechanism as described in 2. to test for these constraint, we just use another object of RDF_DAG.class to represent this graph.

Range Check:

For the known properties we enter the objects according to the rdfs:range definitions of the properties. After the creation of the model we need to test if these objects have the corresponding type.

Example:

The property rdfs:subClassOf has the rdfs:range = rdfs:Class. Therefore objects used with this property are entered as a rdfs:Class. We need to test afterwards if the corresponding resource really has the type rdfs:Class.

The test for the rdfs:range definitions of other properties is made after the creation of the model.

The model does not include the type inheritance. It would be an redundant information. So, we might need to traverse the model.

Unique Range:

In the case a second rdfs:range is defined the system will report an error during the fly.

Domain Check:

For the known properties we enter the subjects according to the rdfs:domain definitions of the properties. After the creation of the model we need to test if these subjects have the corresponding type (see 4. for an similar example). The test for the rdfs: domain definitions of other properties is made after the creation of the model.

The model does not include the type inheritance. It would be an redundant information. So, we might need to traverse the model.

Sub Property Check:

The method for testing this constraint is called by the program following the topological order of the subProperty graph. This means, if ‘a’ is subProperty of ‘b’ and ‘b’ is subProperty of ‘c’ the method is fist called for the highest superProperty, in this example the property ‘c’ then ‘b’ and last but not least ‘a’. The method itself creates a list of all direct superProperties. It then compares the domain(s)/range of each of these direct superProperty with the domain(s)/range of the subProperty (must be equal or subclasses). In the case the domain/range of the

subProperty is not defined the method inherits the first unconstraint domain/range definition[4] of the direct superProperties (if exists, otherwise they are all unconstraint and therefore valid). If the domain/range definitions of the direct superProperties are equal this will not cause an error. If they are not equal and we have no predefined domain/range definitions for the subProperty the method might report an error even if the definitions are valid. This depends on the order in the direct superProperty list, but we are sure to report an error if it is not valid. In general the domain/range definitions should be defined for subProperties to avoid validation errors.

Example (just looking at the range):

Property p_3 supProperty of p_1 and p_2 and no range definition for p_3

Property p_2 range B

Property p_1 range A

Class B subClass of class A

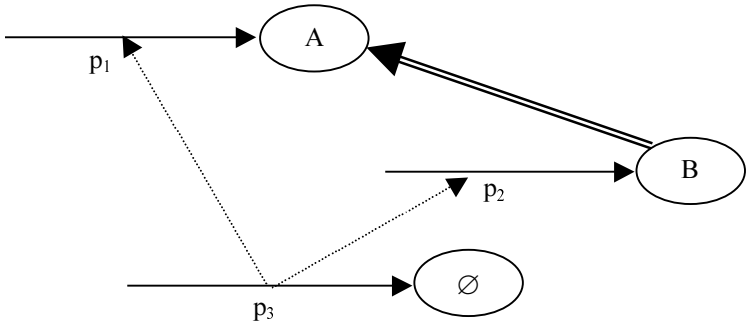


Figure 4.9 Example for a sub property having two super properties.

If p_2 appears in the internal representation before p_1 in the direct superProperty list it is no error – p_3 would inherit class B as range and B is a subClass of A.

If p_1 appears before p_2 in the direct superProperty list it is an error – p_3 would inherit class A as range and A is not a subClass of B.

Statement Check:

During the creation of the model we test if the rdf:subject, rdf:predicate and rdf:object annotations are unique. Afterwards we test if they exist for all objects of RDF_Statement.

4.7 Namespace Connection

As described RDF uses namespaces to reuse defined resources. This is the case in a) Metadata referring to Schemata or b) Schemata referring to other Schemata.

The philosophy of VRP is to connect only to namespaces if we need to. This means if we need more information about a resource specified in a schema to validate the set of statements we explore.

Example:

```
<rdf:RDF
       xmlns:rdf="http://www.w3.org/TR/1999/REC-rdf-syntax#"
       xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
       xmlns:dc="http://purl.oclc.org/metadata/dublin_core#">
  <rdf:Description about="http://www.uni-hannover.de">
       <rdfs:comment>Homepage of the University</rdfs:comment>
       <dc:Creator>Someone</dc:Creator>
       <rdf:value>Something</rdf:value>
  </rdf:Description>
</rdf:RDF>
```

In this case we would need to connect to the Dublin Core namespace specified by the 'dc' prefix to get the information that dc:Creator is really a property (has the type rdf:Property). For the two other properties rdfs:comment and rdf:value we do not need to connect to the

[4] For the domain definition this might be a list of classes.

namespaces, because the information of the embedded RDF Schema and the RDF (syntax) namespaces are included to VRP. For the resource “http://www.uni-hannover.de” we do not need further information to validate.

There are three cases we are forced to connect to namespaces:

1. For type checks during the Domain and Range Check (constraints C13 and C15) and for the Property Check as described in the previous example (C8).
2. For loop tests in rdfs:subClassOf definitions (C9 and C10).
3. For loop tests in rdfs:subPropertyOf definitions (C11 and C12).

Not to make our model of the file we explore to big VRP just enters the information of the requested resource instead of entering the complete set of statement contained by the namespace. The information we enter are restricted to the following definitions:

- rdfs:comment – to have a human readable description of the resource.
- rdfs:label – for graphical presentations.
- rdfs:seeAlso – to provide better semantics by including cross references.
- rdfs:isDefinedBy – to provide better semantics by including cross references.
- rdf:type – important for the validation.
- rdfs:subClassOf (only for classes) – important for the validation.
- rdfs:domain (only for properties) – important for the validation.
- rdfs:range (only for properties) – important for the validation.
- rdfs:subPropertyOf (only for properties) – important for the validation.

VRP does not enter user defined properties nor the list of links. These information are not necessary for the validation or the understanding of the resources. To query the information VRP parses the namespaces and builds up the internal model of them. These internal model will not be further validated but will be kept in the memory for further access. Entering the information might cause the need of entering further information of the same or other namespaces as shown in Figure 4.10.

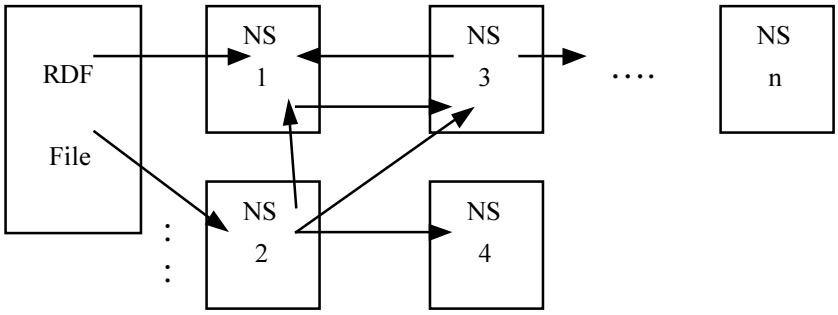


Figure 4.10 Possible namespace relations.

Especially for the properties rdfs:subClassOf and rdfs:subPropertyOf VRP will recursively enter the information of the object to go to the root class or property. This is necessary to test for loops in this definitions as described in section 3.4 (point 2 and 3).

Example:

In the case we have the following triple in the set of statements we validate (again we use the abbreviated form to write the namespaces, the resource without a prefix belong to the RDF file we explore):

{rdfs:subClassOf, A, NS1:B},

we would enter the information about resource B from the namespace ‘NS1’. If these information would contain the statement {rdfs:subClassOf, NS1:B, NS3:C} and the

namespace ‘NS3’ would contain {rdfs:subClassOf, NS3:C, NS1:B} it would cause an error by VRP.

4.8 Error Messages

For a parser the error reporting is one of the main features. Therefor VRP provides a huge list of differentiated error messages refined with additional information. The system provides for each error an unique error number. These error numbers start with the number of the type the error belongs to followed by a running number for each type. There are five error types provided by the system:

0. Lexical error – errors called by the lexical analyzer, if the input does not match the specifications.
1. Syntax error – errors against the BNF of RDF, called by the syntax analyzer.
2. Semantic/Syntax error – errors against the syntax, called by the action part of the syntax analyzer.
3. Semantic error – errors against the RDF Schema, called during creation of the model and validating it.
4. System error – system errors like read-write or connection errors.

It exists an extra Java™ class (RDF_Error) to handle errors. The method to emit an error is called ‘emit_error’. The method needs four parameters:

1. The number of the type the error belongs to.
2. The number of the error message.
3. Additional information like parameters of the method the error is called of in from of a string.
4. The Java™ class name and the name of the method the error appeared, separated by a dot.

For example the following line would emit the error number 25 of the type 3 (Semantic error), called by the method RDF_Property_test of the Java™ class Model.

```
e.emit_error(3, 25, str, "Model.RDF_Propterty_test");
```

It could cause the following error report:

```
Semantic error
       3025: The element is not declared in the RDF-Schema namespace.
       http://www.w3.org/TR/1999/PR-rdf-schema-19990303#test
       Error called by: Model.RDF_Property_test
       at line 16(2) of input during parsing: </
```

First comes the type in a textual form, followed by the number and corresponding error message. The next line shows the additional information. Then comes the name of the method the error appeared. In the last line of the error report we can see a hint where to search for the error inside the input file. Because of the bottom up parsing this is not the exact position of the error but points to the position in the file where the element that contains the error is reduced by the parser. This information can only be provided for errors detected during parsing.

The full list of error messages sorted by error types (0-4):

Lexical error (type 0)

There is no error recovery supported for the lexical analyzer. If an lexical error appears and the system would not be stopped we would get in the most cases a huge list of errors caused by the first error. This is not very useful and therefore I decided to exit the system after an lexical error appears. For these errors there will be an error position report.

```
0000: A valid XML, RDF or HTML statement expected.
0001: A qualified name expected [NSprefix:]QName). See http://www.w3.org/TR/1999/REC-xml-
      names-19990114/ for more info about QName.
0002: A qualified name expected [NSprefix:]QName). See http://www.w3.org/TR/1999/REC-xml-
      names-19990114/ for more info about QName.
0003: A valid XML Attribute expected.
0004: A valid RDF element expected.
0005: A qualified name expected. See http://www.w3.org/TR/1999/REC-xml-names-19990114/ for
      more info about QName.
0006: A qualified name expected [NSprefix:]QName). See http://www.w3.org/TR/1999/REC-xml-
      names-19990114/ for more info.
0007: A valid XML/RDF Attribute expected.
0008: A qualified name in quotas expected ="QName"). See http://www.w3.org/TR/1999/REC-xml-
      names-19990114/ for more info about QName.
```

Syntax error (type 1)

Syntax errors are detected during parsing by mismatching the given grammar. Therefore CUP provides an ‘error’ token. If an incoming token does not fit to the grammar and an error token is specified in the procedure an syntax error will be reported and the system will try to recover the error. Only if this error recovery fails the system will be stopped. It is possible that previous errors affect later once. For these errors there will be an error position report.

```
1001: See RDF M&S [6.1].
1002: Error in a description. Compare with [6.3] RDF M&S.
1003: Error in an attributelist.
1004: Error in a propertyele [6.12] RDF M&S.
1005: Error in a sequence, see [6.25] of RDF M&S.
1006: Error in a bag, see [6.26] of RDF M&S.
1007: Error in an alternative, see [6.27] of RDF M&S.
1008: Error in a member, see [6.28]-[6.30] of RDF M&S.
1009: This is not a well-formed XML [6.34] RDF M&S.
```

Semantic/Syntax error (type 2)

These errors will not stop the parsing and they are not affecting each other. There will be an error position report.

```
2001: You are not allowed to enter other attributes than ID-attr at this position.
2002: ID should be unique.
2003: Attribute used twice in a attribute list.
2004: Only one of the attributes ID, about, aboutEach and aboutEachPrefix are allowed in an
      attr_list.
2006: You need to specify an alternative.
2007: You can not define the same prefix more than one time.
2008: The prefix of the tag name is not declared.
2009: The start tag does not match end tag.
2010: You are not allowed to enter a RDF:ID and a RDF:resource within one propertyEle.
```

Semantic error (type 3)

These errors will not stop the parsing and they are not affecting each other. There may be an error position report.

```
3001: You are not allowed to change a rdfs:Class to a rdf:Property.
3002: You are not allowed to change a rdfs:Class to a rdf:Statement.
3003: You are not allowed to change a rdfs:Class to a rdf:Container.
3004: You are not allowed to change a rdf:Property to a rdfs:Class.
3005: You are not allowed to change a rdf:Property to a rdf:Statement.
3006: You are not allowed to change a rdf:Property to a rdf:Container.
3007: You are not allowed to change a rdf:Statement to a rdfs:Class.
3008: You are not allowed to change a rdf:Statement to a rdf:Property.
3009: You are not allowed to change a rdf:Statement to a rdf:Container.
3010: You are not allowed to change a rdf:Container to a rdfs:Class.
3011: You are not allowed to change a rdf:Container to a rdf:Property.
3012: You are not allowed to change a rdf:Container to a rdf:Statement.
3013: The rdfs:subPropertyOf is not allowed to use with a class as subject.
3014: The rdfs:subPropertyOf is not allowed to use with a property as object.
3015: The rdfs:range property is not allowed to use with a class as subject.
3016: The rdfs:range property is not allowed to use with a property as object.
3017: The rdfs:domain property is not allowed to use with a class as subject.
3018: The rdfs:domain property is not allowed to use with a property as object.
3019: The object for this triple must be a Literal.
3020: This properties are not allowed to use with a Literal as object.
3021: The object for the property rdf:type can not be a property.
3022: The rdf:type property is not allowed to be used with a Literal as object.
3023: The property of the statement had not been inserted in the model as a property.
3024: Type mismatch, it is not allowed to mix the types: Class, Property and Statement.
3025: The element is not declared in the RDF-Schema namespace.
3026: The element is not declared in the RDF namespace.
3027: The element is not declared in the corresponding namespace.
3028: The Container you are asking for is not a Container.
3029: Loop detected!!
3030: There is only one range declaration per rdf:Property allowed.
3031: There is only one subject declaration per rdf:Statement allowed.
3032: There is only one predicate declaration per rdf:Statement allowed.
3033: There is only one object declaration per rdf:Statement allowed.
3034: There need to be for each rdf:Statement exactly one predicate, subject and object.
3035: Where not able to find the type:Statement for this rdf:Statement.
```

```
3036: Where not able to find the type:Class for this rdfs:Class.
3037: Where not able to find the type:Property for this rdf:Property.
3038: Where not able to find one of the types:Bag, Seq or Alt for this rdf:Container.
3039: The range definition of the sub property does not fit to the range definition of the
      super class.
3040: The domain definition of the sub property does not fit to the domain definition of the
      super class.
3041: Error in range check.
3042: Error in domain check.
3043: The namespace does not contain the element you requested.
```

System error (type 4)

System errors are fatal errors and will cause in the most cases an exit of the system. There is no error position report for these errors.

```
4001: Output file does not exists.
4002: Can't read output file.
4003: Can't write to output file.
4004: Error in creation or connection with the output file.
4005: Could not go on parsing or validate.
4006: Could not connect to file.
4007: Invalid option. Valid options are: d, f, g, o, s, t and v.
4008: To many arguments! Run program with no arguments to get the usage-text.
4009: Expected '-' at the beginning of the options.
```

4.9 Compiler Generators for Java

This section is an overview about existing compiler generators for Java™. The decision to choose JFlex and CUP for the VRP is based on this knowledge (see 4.9.4). For a basic introduction to the subject of compiler generators please see [APPEL].

4.9.1 Introduction

The most widely used compiler compilers today fall into two main families: Lex [LEX] / YACC [YACC] (Yet Another Compiler Compiler) and PCCTS [PCCTS] (Purdue Compiler Construction Tool Set). The most used languages are C, C++ and more recently Java™. C is probably most used because of its popularity. This popularity is due to its relative simplicity and mostly, its speed performance. C is fast because it has been designed as a portable intermediate-level programming language for implementing the Unix operating system. C++ follows, having gained most of its popularity from its complete backward compatibility to the C language. C++ adds object-oriented elements to the C language.

Object-oriented programming has the advantage of simplifying the maintenance of a compiler over time. The interest in building compilers in the Java language lies in its platform independence, its robustness from a software-engineering point of view and its popularity among defectors from C++. Java is sometimes called „C *plus plus minus minus*“ because it lacks many undesirable features of C and C++ like pointer arithmetic.

In the following subsections, we study both tool families (YACC and PCCTS) and look at their most popular Java implementations. Figure 4.11 shows an overview of programs we will discuss. A more detailed list of compiler generators for Java™ can be found at: http://www.first.gmd.de/cogent/catalog/java.html.

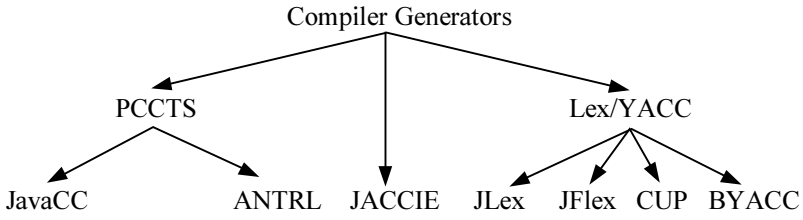


Figure 4.11 Classifying the Java compiler generators.

In a short form the main difference:

Lex/YACC – building bottom up LALR(1) (Look Ahead Left-to-right parse, Right most-derivation) parser

PCCTS – building top down LL(1) (LL(k)) (Left-to-right parse, Left most-derivation) parser

4.9.2 Lex/YACC

Lex [LEX] and YACC [YACC] are a pair of tools that can be used together to generate a compiler or its front-end (just lexical scanner and parser). Many variations on these tools are in use today. Among the most popular versions are the Open Software Foundation's GNU system Flex and Bison tools. These tools use the C language to specify the *action code*. (We use the term *actions* to refer to the code written by a programmer to be executed at specific points of a lexer and/or parser execution).

Lex is normally used to partition a stream of characters into tokens. It takes as input a specification that associates regular expressions with actions. From this specification, Lex builds a function implementing a deterministic finite automaton (DFA) that recognizes regular expressions in linear time. At runtime, when a regular expression is matched, its associated action is executed.

YACC is a parser generator. Like lex, YACC reads a specification that contains both the grammar of the compiled language and actions associated with each alternative of a production of the grammar. It then generates a parser that will execute the action code associated with each alternative as soon as discovered. YACC generates LALR(1) parsers, and has a few options to deal with ambiguous grammars and operator precedence.

The combination of Lex/YACC allows a programmer to write a complete one-pass compiler by simply writing two specifications: one for Lex and one for YACC.

4.9.2.1 JLex

JLex [JLex] is a lexical analyzer generator (also known as scanner generator), written for Java™, in Java™. JLex was developed by Elliot Berk at Princeton University. It is now maintained with the assistance of Martin Dirichs, University of Oldenburg, Germany.

Advantages

- JLex DFA based lexers are usually faster than hand written lexers.
- JLex supports macros to simplify the specification of complex regular expressions.
- JLex is available in source code form.

Drawbacks

- JLex supports only 8 bit characters. But, Java has adopted 16 bits Unicode characters as its native character set. JLex has a *%unicode* directive, but it is not yet implemented.
- JLex still has known bugs in presence of complex macros.

- JLex macros are treated much like C macros. This means that they are textually replaced in regular expressions. This can lead to very hard to find bugs similar to those found in C in presence of unparenthesized macros[5].
- JLex and CUP have not been specially designed to work together. So, it is the programmer's job to build the links between the code generated by both tools. JLex has *%cup* directive, but it has no effect.

4.9.2.2 JFlex

JFlex [JFlex] is a lexical analyzer generator for Java™, written in Java™. It is a rewrite of JLex. JFlex is developed by Gerwin Klein in 1998. As Vern Paxon states for his C/C++ tool Flex: They do not share any code though.

JFlex is designed to work together with the LALR(1) parser generator CUP by Scott Hudson, but can also be used together with other parser generators or as a standalone tool.

Advantages

All features of JLex, and additionally:

- Faster generated scanners
- Faster Scanner generation (independent of the input character set)
- Three different kinds of generated code for best performance/size
- Full Unicode support
- Predefined character classes
- Macro definitions are regular expressions, not just text
- Cycles in macro definitions are detected and reported at generation time
- Platform independent end of line operator „$“
- Beginning of line operator „^“ works as expected (not consuming newlines)
- Grouping of rules with same lexical states
- Built-in support for the CUP parser generator
- Standalone scanners (as with C/C++ flex)
- Debugging support

4.9.2.3 CUP

A Java version of YACC is called CUP [CUP] (*Constructor of Useful Parsers*). It has been developed by Scott E. Hudson, Graphics Visualization and Usability Center, Georgia Institute of Technology. It is very similar to YACC, but actions are written in the Java language.

Advantages

- Cup generates LALR(1) parsers and can deal with some ambiguous grammars using options to resolve LALR conflicts.
- There are examples of LL(1) grammars that are not LALR(1) grammars, but in general we can say that LALR parsers are more powerful than LL(1) or even LL(k) parsers.
- LALR(1) grammars can be left recursive whereas LL(k) grammars can not.

[5] If a macro M is defined as a|b , then the regular expression aMb will be interpreted as aa|bb (aa)|(bb), not a(a|b)b as intended.

- LALR(1) parsers are usually faster than equivalent PCCTS LL(k) parsers for the same language, because PCCTS uses costly syntactic predicates to resolve parsing conflicts.
- Cup is available in source code form.

Drawbacks

- CUP options for handling ambiguous grammars can be quite dangerous in the hand of a novice, because it is hard to clearly determine the recognized grammar.
- *Action code* embedded in CUP parsers can be quite difficult to debug. The abstraction required to clearly understand the operation of a table-based LALR parser is the source of this difficulty for casual users of CUP.
- With today's low memory prices and faster processors, many programmers prefer to work on an *Abstract Syntax Tree (*AST*)* representation of parsed programs. CUP offers no support for building ASTs. So, the programmer has to write the appropriate action code to build nodes for every production alternative of the grammar.
- The lack of support for ASTs renders CUP ill suited for multiple pass compilers.
- The fact that actions are embedded in the specification is a big software-engineering problem. It means that resulting specifications are often enormous. Since CUP does not allow the specification to be broken into multiple files, the specification may result in one huge file. Furthermore, the duplication of action code in the specification and in the resulting program is quite bad. It leaves to the programmer the responsibility of keeping the specification and resulting program consistent. So safely debugging JLex/CUP action code involves the following tedious cycle:

  Repeat

  1. Writing or modifying action code in the specification file.
  2. Compiling the specification.
  3. Compiling the resulting code.
  4. Executing the resulting program to find errors.
  5. Locating the errors in the program.
  6. Looking back in the specification for the related erroneous action code.
  7. Until success
- Taking a shortcut in the previous cycle by fixing errors directly in the generated program can result in unsynchronized specification and working code, if the programmer does not take the time to update the specification accordingly. This is most likely to happen when a programmer debugs actions in an Integrated Development Environment.

4.9.2.4 BYACC/Java

BYACC/Java [BYACC] is an extension of the Berkeley v 1.8 YACC-compatible parser generator. Developed by Bob Jamison the newest version is *Java extension v 0.92* 1999. It is the standard YACC tool there is just added a „-j" flag, which will cause BYACC to generate Java source code, instead of C++.

So, you can use your 'old' C-YACC files with YACC and your new Java™-BYACC files with BYACC. To port the C-YACC files to BYACC or the other way round you have to change a few thinks and at least transform the action-code in the other language.

Several benefits are derived from a Java parser-generator of this sort:

Advantages

- BYACC/Java can be executed from existing Makefiles and IDE's.
- BYACC/Java is coded in C, so the generation of Java code is extremely fast.
- The resulting byte code is small – starting at about 11 Kbytes.

- Only one or two class files are included. If you need only a single type or an Object class, then one class file is generated. If you need a simple generic type, a simple data class is generated for you, making another small file.
- No additional runtime libraries are required. The generated source code is the entire parser.
- It can parse existing YACC grammars, enabling the 'Javanizing'; of a large installed base of YACC source code (of course, your 'actions' need to be in Java).
- Many developers are already very familiar with the workings of YACC.
- It is absolutely free; no license, no royalties, free!

Drawbacks

- BYACC/Java is coded in C, so you will need an additional C compiler.
- For further drawbacks see Drawbacks of Cup.

4.9.3 PCCTS

PCCTS [PCCTS] stands for *Purdue Compiler Construction Tool Set*. It has been developed mainly by Terence Parr. PCCTS was written in the C++ language to generate compilers written in C++.

PCCTS has been developed in reaction to the complexity of using Lex/YACC to resolve compilation problems. Mainly, the table based bottom-up parsing of YACC resulted in very hard to debug compilers. Instead, PCCTS builds *LL(k) recursive-descent* parsers. An LL(1) recursive-descent parser is constituted of a set of functions called recursively to parse the input. Each function is responsible for a single production. The function gets a token from the lexer and determines the appropriate alternative based on it. The problem is that LL(1) grammars are very restrictive. Many common programming constructs are not easily written as an LL(1) grammar. The most common problems are:

1. LL(1) grammars require two alternatives of a production to be distinguishable by looking only at the next available token. This means that the following grammar is not allowed: A = aA|a.
2. LL(1) grammars cannot be left recursive. So, this is not allowed: A = Aa|a.

PCCTS offers LL(k) parsing and many powerful features. The most important ones are (1) semantic predicates, (2) syntactic predicates, (3) Extended Backus-Naur Form[6] (EBNF) syntax, and (4) AST building and Tree-parsers.

A *semantic predicate* specifies a condition that must be met (at run-time) before parsing may proceed. In the following example, we use a semantic predicate to distinguish between a variable declaration and an assignment alternative, using PCCTS syntax:

```
Statement:
 {isTypeName(LT(1))}? ID ID ‘;’ | // declaration ‘type varName;’
 ID ‘=’ expr ‘;’; // assignment
```

The parsing logic generated by PCCTS for this example is:

```
if(LA(1)==ID && isTypeName(LT(1))) {
        match alternative one
        }
else if (LA(1)==ID) {
        match alternative two
        }
else error
```

[6] EBNF is BNF augmented with the regular expression operators: (), *, ? and +.

This example showed us how one can influence the parsing process with semantic information using semantic predicates.

Syntactic predicates are used in cases where finite LL(k) for k>1 is insufficient to disambiguate two alternatives of a production. A syntactic predicate executes the parsing function calls of the predicate, but it does not execute any action code. If the predicate succeeds, then the alternative is chosen and the parsing calls of the alternative are executed along with action code. If the predicate fails, the next matching alternative (or predicate) is executed.

One of the most popular features of PCCTS is its directive to build abstract syntax trees (AST) automatically. The resulting AST has a single Java node type shared by all the nodes. One gets specific information about the parsing type of the nodes by calling methods on the node. Additionally, PCCTS allows for building tree-parsers. A tree-parser is a parser that scans the AST for patterns. If it finds a specified pattern, it executes the action code associated with the pattern. A tree-parser is very similar in its specification and implementation to normal parser. It uses the same LL(k) EBNF and predicates as normal parsers and it is built using recursive functions.

4.9.3.1 ANTLR

PCCTS 1.33 has been ported to Java and renamed ANTLR 2.xx [ANTLR].

ANTLR 2.xx is a single tool combining together all three parts of PCCTS. PCCTS 1.33 consisted of:

- ANTLR (ANother Tool for Language Recognition),
- DLG (DFA-based Lexical analyzer Generator) and
- SORCERER, a tool for generating tree parsers and tree transformers

So, ANTLR is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions [You can use PCCTS 1.xx to generate C-based parsers].

Note: ANTLR lexers do not support 16 bits Unicode character input.

4.9.3.2 JavaCC

JavaCC[JavaCC] Developed by Sun Microsystems inc., very similar to ANTRL. There are no major differences between the two products but for the copyright and source code availability. While ANTLR is in the public domain, JavaCC is a commercial product, not available in source code format (Program is free).

JavaCC consists out of three parts:

- JavaCC (parser generator)
- JJTree (tree builder pre-processor)
- JJDoc (document generator)

Note: JavaCC DFA-based lexers accept 16 bits Unicode characters.

Advantages

- Integration of the lexer and the parser.
- JavaCC DFA-based lexers accept 16 bits Unicode characters.
- ANTLR lexers are LL(k)-based (with predicates) and share the same syntax as parsers.
- Action code is much easier to debug with ANTLR and JavaCC than with LALR(1) table-based parsers, due to the natural behavior of recursive-descent parsers.
- Both tools support EBNF.
- Both tools have options to generate ASTs automatically.
- Both tools support recursive-descent tree-parsers.
- The support for ASTs is very convenient for multiple-pass compilers.

- The range of languages that can be parsed by these tools is much bigger than LL(1), and is relatively comparable to LALR(1). The use of semantic predicates enables the parsing of *context-sensitive* grammars.
- Both tools are free. Additionally, ANTLR is provided in source code form and is in the public domain.
- Both tools are relatively well supported, with dedicated newsgroups on the Internet.

Drawbacks

- ANTLR lexers do not support 16 bits Unicode character input.
- The exibility of predicates comes with a very high cost in performance if used poorly by a programmer.
- While semantic predicates allow the parsing *context-sensitive* grammars, they also are a software-engineering problem. Semantic verifications must happen along with parsing in order to enable semantic predicates. Furthermore, the predicates are somehow anintegral part of the resulting grammar. This obscures the grammar.
- Syntactic predicates are very expensive in computation time.
- A majority of predicate uses would not be needed by an LALR(1) parser to recognize the same grammar.
- LL(k) grammars cannot be left recursive. This handicap can be fixed by widely know grammar transformations and the use of EBNF syntax. A = Aa|a = A = (a)+.
- ANTLR/JavaCC specifications suffer from the same important software engineering problems as JLex/CUP. They tend to become huge, and debugging action code involves the same tedious cycle.
- As with JLex/CUP, the responsibility of keeping the specification and the generated code synchronized is left to the programmer. So taking a shortcut in the debugging cycle by fixing errors in the program directly can result in unsynchronized specification and working code. This is most likely to happen when debugging semantic predicates in an Integrated Development Environment.
- The integrity and the correctness of the AST is left in the hands of the programmer. There will be no warning if a transformation on the AST results in a degenerated tree. Such bugs are extremely difficult to track, because they may result in a null pointer exception, or some other error condition in unrelated code thousands of instructions after the transformation occurred. This is comparable to C and C++ array out of bound problems.

4.9.3.3 JACCIE

JACCIE[JACCIE] (JAva-based Compiler-Compiler with Interactive Environment) is an educational tool for visualizing modern compiling techniques. It automatically generates demonstration compilers, which continuously display their internal states and are controlled with the mouse to move back and forth, in single or larger steps. In an integrated environment JACCIE provides special editors for scanner and parser definitions, tools for viewing derived information like first / follow sets or parsing automata, and tools for debugging scanners and parsers.

Technically speaking, the major components of JACCIE are a scanner generator and a variety of parser generators covering all parsing strategies (LL(1), SLR(1), LALR(1)) typically found in modern compilers, and more. In debugging mode, JACCIE parsers can be operated non-deterministically: Ambiguities are then resolved by backtracking or by prompting the user for advice. In contrast to conventional parsers, input may contain non-terminal symbols, too.

4.9.4 Conclusion

The RDF Specification is in the status of an Recommendation and therefore changes may be possible in the near future. Furthermore I where not sure in forehead to be able to express the grammar of RDF in an unambiguous way, especially because of some inherited features of XML.

When we want to use RDF for data exchange we need to be able to process huge sets of data. It is necessary to provide a fast parser. LALR(1) parsers are usually faster than equivalent LL(k) parsers.

These had been the main reasons for our choice using an LALR(1) parser.

I decided to use the JFlex tool because of the full Unicode support. Because of the hand in hand working of JFlex an CUP I decided to use CUP instead of BYACC.

5 Comparison

In this chapter I compare the VRP with SiRPAC, the most used parser at the moment. I only mention the main differences between SiRPAC and VRP here. Please see section 3.5.1 for an introduction to SiRPAC. In section 5.2 I will give an example how to use the benefits of VRP. rdf2cb is an application based on SiRPAC and will show how VRP could be used for rdf2cb instead of SiRPAC to enrich the program.

5.1 VRP and SiRPAC

In this section I list up the main drawbacks of SiRPAC compared to VRP. Table 5.1 provides a summary of these comparison.

- It is necessary to install an XML-Parser before you can run SiRPAC. This is a time spending process for the user. He has to chose and install the right XML-Parser. Features like full Unicode support depend on this choice.

  VRP on the other hand does not need an extra XML-Parser. It is not necessary to install further programs (see Figure 5.1) to run VRP. The parser included in VRP as described in section 4.4 provides a full Unicode support and only tests for the well-formed constraint outside RDF and this is all we need.

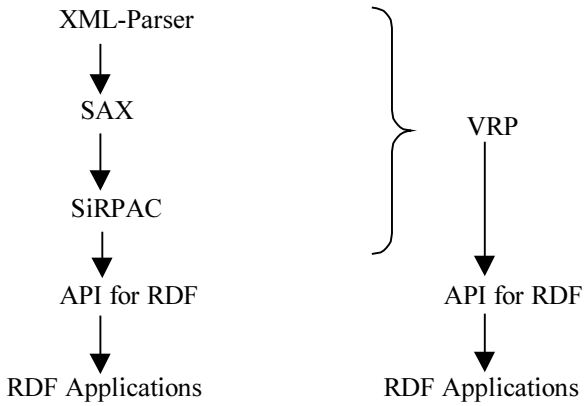


Figure 5.1 Comparing the SiRPAC and VRP structure.

- SiRPAC is written from scratch without using standard tools. This makes it difficult to update SiRPAC in the case of changes in the RDF grammar and it is hard to extend it. The user needs to understand the whole program and must be firm in using Java™.

  The lexer and parser part of VRP is written by using the standard tools JFlex and CUP. Therefore VRP can easily be updated in case of changes in the RDF syntax. The user just needs to change the grammar 'parser.cup' file (the CUP program is needed to compile it). For new resources defined in the RDF (syntax) namespace or the RDF Schema the user just needs to change the variables in the file 'Main.java'. Only to enter special semantics for new resources the user needs to understand the Java™ methods.

- There are still bugs in SiRPAC and the program is not longer supported from the W3C.
- There is no semantic validation provided by SiRPAC.
- SiRPAC compiles RDF/XML files into triples of the form:

  triple("*predicate*", "*subject*", "*object*").

  In this form SiRPAC does not distinguish between a triple having a resource as object and a triple having a literal as object. As you can see in the example below, SiRPAC would create in both cases the same triple.

  Example:

a)

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
        <rdf:Description about="xyz">
                <rdf:value>http://www.w3.org</rdf:value>
        </rdf:Description>
</rdf:RDF>
```

b)

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
        <rdf:Description about="xyz">
                <rdf:value rdf:resource="http://www.w3.org"/>
        </rdf:Description>
</rdf:RDF>
```

SiRPAC would create in both cases the following triple (in the case the description is saved in the file 'test'):

triple("http://www.w3.org/1999/02/22-rdf-syntax-ns#value",
"file:test#xyz",
"http://www.w3.org")

VRP would create different internal models for these examples.

a)

URI	Object
file:test#xyz	RDF_Resource value = "http://www.w3.org"

b)

URI	Object
file:test#xyz	RDF_Resource value = "http://www.w3.org"
http://www.w3.org	Resource

Table 5.1 Summary of the main differences between VRP and SiRPAC

VRP	SiRPAC
provides validation according to the RDF Schema specification (may be deactivated)	no validation according to the RDF Schema specification
based on standard tools and therefore easy to modify them in case of changes in the RDF syntax	written from scratch and therefore difficult to modify them in case of changes in the RDF syntax
you do not need to install further programs	necessary to install an XML-Parser before you can run SiRPAC
VRP distinguish in the internal model between resources and literal	SiRPAC does not distinguish between resources and literal

5.2 VRP for rdf2cb

rdf2cb is a tool to convert RDF statements to a special data-model using ConceptBase. ConceptBase is a deductive object management system for saving O-Telos frames, see [ConBase] for more information. rdf2cb is based on SiRPAC and analyses O-Telos frames out of the set of statements

created by SiRPAC. These O-Telos frames are saved in a file and can be used to load the annotations of the RDF/XML file direct to ConceptBase. Figure 5.2 illustrates the steps during running rdf2cb using SiRPAC.

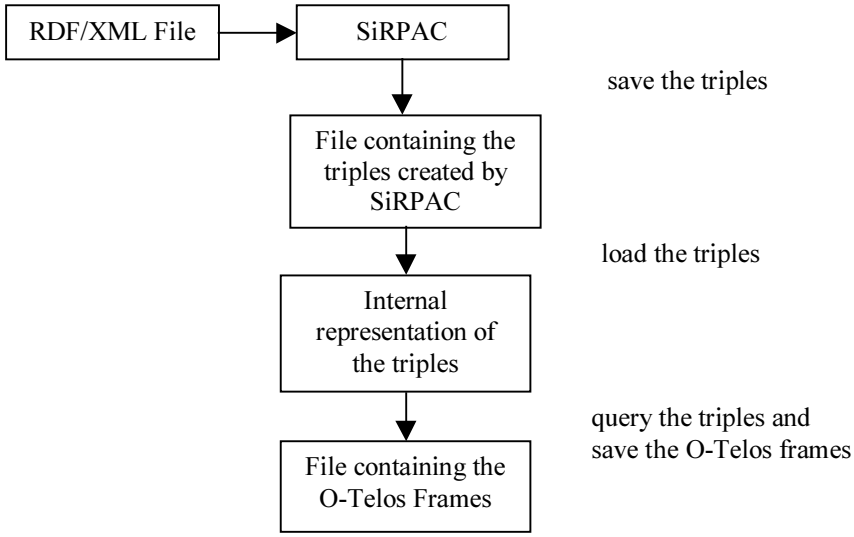


Figure 5.2 Steps during running rdf2cb using SiRPAC.

Especially the saving of the triples to a file and the reloading to an internal representation is highly redundant and costs a lot of time. The intern representation is used to collect the necessary information for building the O-Telos frames. These internal representation is based on a set of triples. For each rdfs:Class, rdf:Property and rdfs:Container rdf2cb traverse these triples and collects the important information to build the corresponding O-Telos frames.

By using VRP instead of SiRPAC the process could be much easier. The internal model of VRP could be used instead of saving and reloading the triples. In these internal model the information for each rdfs:Class, rdf:Property and rdfs:Container is already selected. There would be no need to traverse the whole set of triples each time. The reduced steps have to be done for running rdf2cb are illustrated in Figure 5.3.

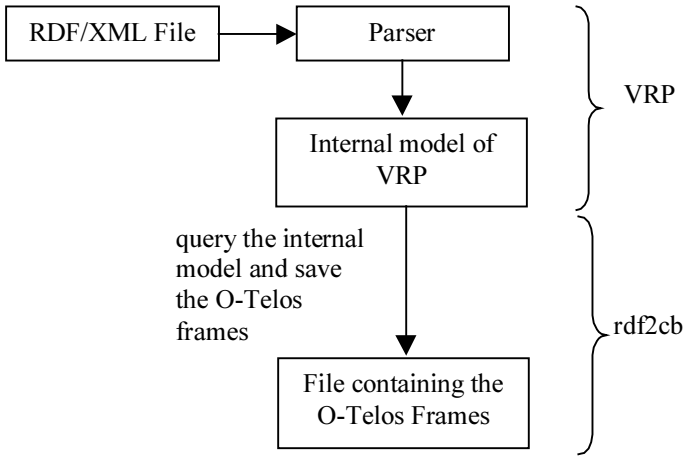


Figure 5.3 Steps during running rdf2cb using VRP.

There are further benefits for rdf2cb using VRP. These benefits are:

- VRP provides a semantic validation against the RDF Schema constraints. This feature is not provided by SiRPAC and is one of the main missing things specified by the author of rdf2cb.

- To run rdf2cb at the moment the user is forced to include all the relevant data and namespaces to one file. This is a time intensive procedure and the user needs to know RDF and the way URI references are made.

  VRP is able to connect and include namespaces to the internal model.
- The internal model of VRP supports rdf:Statement as a special type and the topological order of the rdfs:subPropertyOf relation. Not necessary for this version of rdf2cb but this makes it easy to extend it.

6 Conclusions

The World Wide Web contains a very, very huge set of data and the supply and demand of information will raise in the future. Therefore it will be necessary to provide metadata. RDF as an collaborative design effort has a great chance to become a standard for these metadata. But this is only the first little step on a long way. It is important that the different user groups build up and use their own schemas for their needs. Probably there will be competing schemas and it will not be easy for a user to decide which he should use, but I am sure that in a kind of evolution process the best schemas will survive. With the increasing use of metadata the providers of search engines and similar systems will be forced to deal with metadata. If this happens and the search engines are based on metadata the users are forced to use them, otherwise their information would get lost in the mass of data.

The Validating RDF Parser (VRP) presented in this paper (see chapter 4) is an example for a module that could be integrated in systems that try to understand and/or validate the RDF annotations. These systems could be search engines or as described in section 5.2 converters/data-loaders for databases. It also could be used as a standalone tool to parse and validate RDF. In the case the internal model would be saved in a persistence way VRP also could be used as an standalone repository for RDF annotations. There are some more remaining extensions that can be made to enrich VRP. These extensions are:

- A graphical user interface for an easy handling of the program for a standalone usage.
- A graphical visualization of the internal model to provide a better human readability.
- A query module on top of the internal model.
- An API for an convenient integration of VRP to other systems.

To create a standard Application Programming Interface (API) for RDF is in my opinion the biggest challenge at the moment. Once we have this API we could combine the different existing modules for the users needs. There are various APIs proposed from different communities but there is no standardization jet. The proposals are:

- RDF API 1.0 Draft (related to [GINF]), by Sergey Melnik (can be found at http://www-db.stanford.edu/~melnik/rdf/api.html)
- RADIX: A proposal for an RDF API [RADIX], by Ron Daniel
- Mozilla RDF API [MOZ] (C++)
- IBM's RDF for XML Java API & implementation [IBM]

For parsers like SiRPAC, Pro Solutions Perl Parser and VRP there is also the task to find a standardization in the error messages during the parsing. It would make it more easy for the user to understand them.

Appendix A – Compact XML Serialization of the RDF Schema

This is a corrected version of the Compact XML Serialization that can be found in [RDF Schema]. The corrected errors correspond to the error list that can be found in 2.5.3.

```
<rdf:RDF xml:lang="en"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">

<rdfs:Class rdf:ID="Resource"
        rdfs:label="Resource"
        rdfs:comment="The most general class"/>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
        rdfs:label="type"
        rdfs:comment="Indicates membership of a class">
  <rdfs:range rdf:resource="#Class"/>
</rdf:Property>

<rdf:Property ID="comment"
        rdfs:comment="Use this for descriptions"
        rdfs:label="comment">
  <rdfs:domain rdf:resource="#Resource"/>
  <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>

<rdf:Property ID="label"
        rdfs:label="label"
        rdfs:comment="Provides a human-readable version of a resource name">
 <rdfs:domain rdf:resource="#Resource"/>
 <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>

<rdfs:Class rdf:ID="Class"
        rdfs:label="Class"
        rdfs:comment="The concept of Class">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdf:Property ID="subClassOf"
        rdfs:label="subClassOf"
        rdfs:comment=" Indicates specialization of classes">
  <rdfs:range rdf:resource="#Class"/>
  <rdfs:domain rdf:resource="#Class"/>
</rdf:Property>

<rdf:Property ID="subPropertyOf"
        rdfs:label="subPropertyOf"
        rdfs:comment="Indicates specialization of properties">
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property ID="seeAlso"
        rdfs:label="seeAlso"
        rdfs:comment="Indicates a resource that provides information about the subject
resource">
  <rdfs:range rdf:resource="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Resource"/>
  <rdfs:domain rdf:resource="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Resource"/>
</rdf:Property>

<rdf:Property ID="isDefinedBy"
        rdfs:label="isDefinedBy"
        rdfs:comment="Indicates a resource containing and defining the subject resource.">
  <rdfs:subPropertyOf rdf:resource="#seeAlso"/>
  <rdfs:range rdf:resource="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Resource"/>
  <rdfs:domain rdf:resource="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Resource"/>
</rdf:Property>

<rdfs:Class rdf:ID="ConstraintResource"
  rdfs:label="ConstraintResource"
  rdfs:comment="Resources used to express RDF Schema constraints.">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdfs:Class rdf:ID="ConstraintProperty"
```

```
        rdfs:label="ConstraintProperty"
        rdfs:comment="Properties used to express RDF Schema constraints.">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:subClassOf rdf:resource="#ConstraintResource"/>
</rdfs:Class>

<rdfs:ConstraintProperty rdf:ID="domain"
        rdfs:label="domain"
        rdfs:comment="This is how we associate a class with  properties that its instances can
have">
  <rdfs:range rdf:resource="#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:ConstraintProperty>

<rdfs:ConstraintProperty rdf:ID="range"
  rdfs:label="range"
  rdfs:comment="Properties that can be used in a
  schema to provide constraints">
  <rdfs:range rdf:resource="#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:ConstraintProperty>

<rdfs:Class about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"
        rdfs:label="Property"
        rdfs:comment="The concept of a property.">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Literal"
  rdfs:label="Literal"
  rdfs:comment="This represents the set of atomic values, eg. textual strings."/>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"
        rdfs:label="Statement"
        rdfs:comment="This represents the set of reified statements.">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#subject"
        rdfs:label="subject">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  <rdfs:range rdf:resource="#Resource"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate"
        rdfs:label="predicate">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#object"
               rdfs:label="object">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
</rdf:Property>

<rdfs:Class rdf:ID="Container"
  rdfs:label="Container"
  rdfs:comment="This represents the set Containers.">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"
        rdfs:label="Bag">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq"
        rdfs:label="Sequence">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt"
        rdfs:label="Alt">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:ID="ContainerMembershipProperty"
```

```
        comment="This is the class that the properties _1,_2, ... that are used to represent
lists and are an instance of">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#value"
  rdfs:label="value"/>
</rdf:RDF>
```

Appendix B – Official Errata in RDF Schema

The newest version of this page can be found at:
http://www.w3.org/TR/1999/PR-rdf-schema-19990303/errata.

Errata in PR-rdf-schema-19990303

This documents contains the known errors in [RDF Schema]. Further comments on the original document or these errata may be found in [RDF Archive] the archive of the comment mailing list.

Basic XML Serialization

The description of the 'Property' resource puts the class into the wrong namespace. The description should read:

```
<rdf:Description about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">
  <rdfs:label xml:lang="en">Property</rdfs:label>
  <rdfs:label xml:lang="fr">Propriété</rdfs:label>
  <rdfs:comment>The concept of a property.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Resource"/>
  <rdf:type resource="#Class"/>
</rdf:Description>
```

(*thanks to Karsten Tolle*)

Compact XML Serialization

In the definition of the 'label' property, the label property was mis-spelled. The definition should read:

```
<rdf:Property ID="label"
   rdfs:label="label"
   rdfs:comment="Provides a human-readable version of a resource name">
 <rdfs:domain rdf:resource="#Resource"/>
 <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>
```

This error also appears in the RDF embedded within the document. (*thanks to Karsten Tolle*)

The description of the 'Property' resource puts the class into the wrong namespace. The description should read:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"
    label="Property"
    comment="The concept of a property.">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>
```

This error also appears in the RDF embedded within the document. (*thanks to Karsten Tolle*)

The description of the 'Statement' resource omits a required namespace prefix on the about attribute and mis-spells the 'label' property. It should read:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"
  label="Statement"
  comment="This represents the set of reified statements.">
<rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>
```

The first of these errors also appears in the RDF embedded within the document. (*thanks to Karsten Tolle*)

The descriptions of the 'Bag', 'Seq', and 'Alt' classes omit a required namespace prefix on the about attribute. They should read:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"
    label="Bag">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq"
    label="Sequence">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt"
    label="Alt">
  <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>
```

This error also appears in the RDF embedded within the document.

Ralph R. Swick
Last updated: 1999-10-13T18:43:46Z CVS $Date: 1999/10/13 18:45:04 $

Appendix C – The RDF (syntax) Namespace

The RDF namespace as it can be found at http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

```
<?xml version="1.0"?>
<RDF
  xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://www.w3.org/TR/WD-rdf-schema#">

<!-- This is the RDF Schema for the RDF data model as described in the Resource Description
Framework (RDF) Model and Syntax Specification http://www.w3.org/TR/REC-rdf-syntax -->

<s:Class rdf:ID="Statement" s:comment="A triple consisting of a predicate, a subject, and an
object." />

<s:Class rdf:ID="Property" s:comment="A name of a property, defining specific meaning for the
property" />

<s:Class rdf:ID="Bag" s:comment="An unordered collection" />

<s:Class rdf:ID="Seq" s:comment="An ordered collection" />

<s:Class rdf:ID="Alt" s:comment="A collection of alternatives" />

<Property ID="predicate" s:comment="Identifies the property used in a statement when
representing the statement in reified form">
  <s:domain rdf:resource="#Statement" />
  <s:range rdf:resource="#Property" />
</Property>

<Property ID="subject" s:comment="Identifies the resource that a statement is describing when
representing the statement in reified form">
  <s:domain rdf:resource="#Statement" />
</Property>

<Property ID="object" s:comment="Identifies the object of a statement when representing the
statement in reified form" />

<Property ID="type" s:comment="Identifies the Class of a resource" />

<Property ID="value" s:comment="Identifies the principal value (usually a string) of a
property when the property value is a structured resource" />

</RDF>
```

Appendix D – BNF of RDF

The formal grammar of RDF as it can be found in chapter 6 of [RDF M&S].

```
[6.1] RDF            ::= ['<rdf:RDF>'] obj* ['</rdf:RDF>']
[6.2] obj            ::= description | container
[6.3] description    ::= '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '/>'
                         | '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '>'
                              propertyElt* '</rdf:Description>'
                         | typedNode
[6.4] container      ::= sequence | bag | alternative
[6.5] idAboutAttr    ::= idAttr | aboutAttr | aboutEachAttr
[6.6] idAttr         ::= ' ID="' IDsymbol '"'
[6.7] aboutAttr      ::= ' about="' URI-reference '"'
[6.8] aboutEachAttr  ::= ' aboutEach="' URI-reference '"'
                         | ' aboutEachPrefix="' string '"'
[6.9] bagIdAttr      ::= ' bagID="' IDsymbol '"'
[6.10] propAttr       ::= typeAttr
                         | propName '="' string '"' (with embedded quotes escaped)
[6.11] typeAttr       ::= ' type="' URI-reference '"'
[6.12] propertyElt    ::= '<' propName idAttr? '>' value '</' propName '>'
                         | '<' propName idAttr? parseLiteral '>'
                              literal '</' propName '>'
                         | '<' propName idAttr? parseResource '>'
                              propertyElt* '</' propName '>'
                         | '<' propName idRefAttr? bagIdAttr? propAttr* '/>'
[6.13] typedNode      ::= '<' typeName idAboutAttr? bagIdAttr? propAttr* '/>'
                         | '<' typeName idAboutAttr? bagIdAttr? propAttr* '>'
                            propertyElt* '</' typeName '>'
[6.14] propName       ::= Qname
[6.15] typeName       ::= Qname
[6.16] idRefAttr      ::= idAttr | resourceAttr
[6.17] value          ::= obj | string
[6.18] resourceAttr   ::= ' resource="' URI-reference '"'
[6.19] Qname          ::= [ NSprefix ':' ] name
[6.20] URI-reference  ::= string, interpreted per [URI]
[6.21] IDsymbol       ::= (any legal XML name symbol)
[6.22] name           ::= (any legal XML name symbol)
[6.23] NSprefix       ::= (any legal XML namespace prefix)
[6.24] string         ::= (any XML text, with "<", ">", and "&" escaped)
[6.25] sequence       ::= '<rdf:Seq' idAttr? '>' member* '</rdf:Seq>'
                         | '<rdf:Seq' idAttr? memberAttr* '/>'
[6.26] bag            ::= '<rdf:Bag' idAttr? '>' member* '</rdf:Bag>'
                         | '<rdf:Bag' idAttr? memberAttr* '/>'
[6.27] alternative    ::= '<rdf:Alt' idAttr? '>' member+ '</rdf:Alt>'
                         | '<rdf:Alt' idAttr? memberAttr? '/>'
[6.28] member         ::= referencedItem | inlineItem
[6.29] referencedItem ::= '<rdf:li' resourceAttr '/>'
[6.30] inlineItem     ::= '<rdf:li' '>' value </rdf:li>'
   | '<rdf:li' parseLiteral '>' literal </rdf:li>'
   | '<rdf:li' parseResource '>' propertyElt* </rdf:li>'
[6.31] memberAttr     ::= ' rdf:_n="' string '"' (where n is an integer)
[6.32] parseLiteral   ::= ' parseType="Literal"'
[6.33] parseResource  ::= ' parseType="Resource"'
[6.34] literal        ::= (any well-formed XML)
```

Appendix E – The VRP Validation Constraints

The list of constraints need to be taken in consideration for validation. I use the abbreviated form to represent the RDF and RDF Schema namespaces. M represents a statement model of a file, as defined in chapter 3. S is the set of statements contained in M ($s \in S \Leftrightarrow \exists\ x \in M \mid x = (\text{rdf:type}, s.URI, \text{rdf:Statement})$). The notation r.URI represents the URI of the resource r.

C1: $\text{rdf:Property} \cap (\text{rdfs:Class} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C2: $\text{rdfs:Class} \cap (\text{rdf:Property} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C3: $\text{rdf:Bag} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C4: $\text{rdf:Seq} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Bag} \cup \text{rdf:Alt} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C5: $\text{rdf:Alt} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Seq} \cup \text{rdf:Bag} \cup \text{rdf:Statement} \cup \text{rdfs:Literal}) = \varnothing$

C6: $\text{rdf:Statement} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdfs:Literal}) = \varnothing$

C7: $\text{rdfs:Literal} \cap (\text{rdf:Property} \cup \text{rdfs:Class} \cup \text{rdf:Bag} \cup \text{rdf:Seq} \cup \text{rdf:Alt} \cup \text{rdf:Statement}) = \varnothing$

C8: $\forall s \in M,\ s = (p, s, o)\ \exists\ x \in M,\ x = (\text{rdf:type}, p, \text{rdf:Property})$

C9: $A<B \wedge B<C \Rightarrow A<C \mid A,B,C$ are classes

C10: $A<B \Rightarrow A \neq B \mid A,B$ are classes

C11: : $p<q \wedge q<q' \Rightarrow p<q' \mid p,q,q'$ are properties

C12: $p<q \Rightarrow p \neq q \mid p,q$ are properties

C13: p has range $A \Rightarrow [p] = \{(\text{subject}, \text{object}) \mid \text{subject is a resource and object} \in A\}$

C14: p has range $A \wedge$ p has range $B \Rightarrow A = B \mid p$ is property and A,B are classes

C15: p has domain $\{A_i\} \Rightarrow [p] = \{(\text{subject}, \text{object}) \mid \text{subject} \in \cup_i\{A_i\}\} \mid A_i$ are classes and p is property

C16: $p < q \ \wedge$ p has range $A \wedge$ q has range $B \Rightarrow A = B \vee A < B \mid p,q$ properties and A,B classes

C17: $\forall\ s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:predicate}, \beta = s.URI$

C18: $\forall\ s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:subject}, \beta = s.URI$

C19: $\forall\ s \in S \Rightarrow \exists' (\alpha, \beta, \gamma) \in M \mid \alpha = \text{rdf:object}, \beta = s.URI$

Appendix F – SiLRI Validating Rules

The rules are written using the syntax as described in [F-Logic].

```
FORALL Message,Resource,Predicate,Correct <-
  problem(Message,Resource,Predicate,Correct).

FORALL Predicate,Correct,Resource
  problem("Domain constraint violation", Resource, Predicate, Correct) <-
  domainSet(Resource,Predicate,Correct),
  not domainOK(Predicate).

FORALL Predicate,Correct,Resource
  problem("Range constraint violation", Resource, Predicate, Correct) <-
  rangeSet(Resource,Predicate,Correct),
  not rangeOK(Predicate).

FORALL Predicate
  domainOK(Predicate) <-
    EXISTS Resource,Domain,Class,Value
    Predicate["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#domain" ->> Domain],
    Resource["http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ->> Class;
          Predicate->>Value],
    Class :: Domain.

FORALL Resource,Predicate,Class
  domainSet(Resource,Predicate,Class) <-
    EXISTS Value
    Resource[Predicate->>Value],
    Predicate["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#domain" ->> Class].

FORALL Predicate
  rangeOK(Predicate) <-
    EXISTS Resource,Range,Class,Value
    Predicate["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#range" ->> Range],
    Resource[Predicate->>Value],
    Value["http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ->> Class],
    Class :: Range.

FORALL Resource,Predicate,Class
  rangeSet(Resource,Predicate,Class) <-
```

```
    EXISTS Value
    Resource[Predicate->>Value],
    Predicate["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#range" ->> Class].

FORALL Object
  problem ("User-defined constraint not validated", Object, "-", "-") <-
     Object["http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ->>
          "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#ConstraintProperty"].

FORALL Object
  problem ("User-defined constraint not validated", Object, "-", "-") <-
     Object["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#subClassOf" ->>
          "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#ConstraintProperty"].

FORALL X,Y
   X :: Y <- X["http://www.w3.org/TR/1999/PR-rdf-schema-19990303#subClassOf" ->> Y]
   OR unify (X,Y).

FORALL X,Y
  X : Y <-
     X["http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ->> Y]
  OR unify (X,Y).
```

Appendix G – The compact VRP Java class descriptions

In this appendix you can find the tables of the variables, constructors and methods contained by the Java classes of the Java packages VRP, VRP.Parser and VRP.Model. It is a compact version of the Java_Doc documentation that can be found at [VRP Home].

Class VRP.Main

```
java.lang.Object
    |
    +----VRP.Main
```

public class **Main**

extends java.lang.Object Main.java - Containing the main method and the parameters for the RDF and the Schema namespaces.

Variable Index

debug	When this option is set the lexer will print the token names and values.
fetch	Flag wether to conntact to namespace or not.
graph	Flag to print textual representation of the model.
only	Not creating the model, this means no validation and just creating the triples by the parser.
OutputFile	String containing the name of the output file.
p_all	Will be used mainly by the RDF_Error.class.
R_about	Attribute from the RDF namespace: RDF_NS+"about"
R_aboutEach	Attribute from the RDF namespace: RDF_NS+"aboutEach"
R_aboutEachPrefix	Attribute from the RDF namespace: RDF_NS+"aboutEachPrefix"
R_Alt	Element from the RDF namespace: RDF_NS+"Alt"
R_Bag	Element from the RDF namespace: RDF_NS+"Bag"
R_bagID	Attribute from the RDF namespace: RDF_NS+"bagID"
R_Classes	The set of Classes from the RDF namespace.
R_Description	Attribute from the RDF namespace: RDF_NS+"Description"
R_ID	Attribute from the RDF namespace: RDF_NS+"ID"
R_msp	Element from the RDF namespace: RDF_NS+"_" (will be used to test the start of rdf:_n | n integer)
R_object	Element from the RDF namespace: RDF_NS+"object"
R_predicate	Element from the RDF namespace: RDF_NS+"predicate"
R_Properties	The set of Properties from the RDF namespace.
R_Property	Element from the RDF namespace: RDF_NS+"Property"
R_resource	Attribute from the RDF namespace: RDF_NS+"resource"
R_Seq	Element from the RDF namespace: RDF_NS+"Seq"
R_Statement	Element from the RDF namespace: RDF_NS+"Statement"
R_subject	Element from the RDF namespace: RDF_NS+"subject"
R_type	Element from the RDF namespace: RDF_NS+"type"
R_value	Element from the RDF namespace: RDF_NS+"value"
RDF_NS	The RDF namespace URL: "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

S_Class	Element from the RDF Schema namespace: Schema_NS+"Class"
S_Classes	The set of Classes from the RDF Schema namespace.
S_comment	Element from the RDF Schema namespace: Schema_NS+"comment"
S_ConstraintProperty	Element from the RDF Schema namespace: Schema_NS+"ConstraintProperty"
S_ConstraintResource	Element from the RDF Schema namespace: Schema_NS+"ConstraintResource"
S_Container	Element from the RDF Schema namespace: Schema_NS+"Container"
S_ContainerMembershipProperty	Element from the RDF Schema namespace: Schema_NS+"ContainerMembershipProperty"
S_domain	Element from the RDF Schema namespace: Schema_NS+"domain"
S_isDefinedBy	Element from the RDF Schema namespace: Schema_NS+"isDefinedBy"
S_label	Element from the RDF Schema namespace: Schema_NS+"label"
S_Literal	Element from the RDF Schema namespace: Schema_NS+"Literal"
S_Properties	The set of Properties from the RDF Schema namespace.
S_range	Element from the RDF Schema namespace: Schema_NS+"range"
S_Resource	Element from the RDF Schema namespace: Schema_NS+"Resource"
S_seeAlso	Element from the RDF Schema namespace: Schema_NS+"seeAlso"
S_subClassOf	Element from the RDF Schema namespace: Schema_NS+"subClassOf"
S_subPropertyOf	Element from the RDF Schema namespace: Schema_NS+"subPropertyOf"
Schema_NS	The RDF Schema namespace URL: "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
statements	Flag to print the triples of the model.
triples	Flag to print also the triples of the Parser.
verbose	Printing a feedback about the things VRP does to standard output.

Constructor Index

Main()	

Method Index

clean_file(String)	Clear the output file and enter time of generation.
init_R_Classes()	The classes of RDF namespace.
init_R_Properties()	The properties of RDF namespace.
init_S_Classes()	The classes of RDF-Schema namespace.
init_S_Properties()	The properties of RDF-Schema namespace.

main(String[])	The main method.
set_options(String)	Method to set the options: count, debug, fetch, lexer, print, triples and statements.
start(String)	Create and start the parser and the validator.

Class VRP.Output

```
java.lang.Object
    |
    +----VRP.Output
```

public class **Output**
extends java.lang.Object Output.java - Writing to the output file.

Constructor Index

Output()	

Method Index

out(String)	Printing the String to the OutputFile specified in 'Main.java'.

Class VRP.RDF_Error

```
java.lang.Object
    |
    +----java.lang.Throwable
            |
            +----java.lang.Exception
                    |
                    +----VRP.RDF_Error
```

public class **RDF_Error**
extends java.lang.Exception RDF_Error.java - Containing all error messages.

Constructor Index

RDF_Error()	Create a new RDF_Error.

Method Index

emit_error(int, int, String, String)	This method is called whenever an error is found.

get_text(int, int)	Contains the collection of all error messages.

Class VRP.Parser.Lexer

```
java.lang.Object
    |
    +----VRP.Parser.Lexer
```

class **Lexer**
extends java.lang.Object
implements java_cup.runtime.Scanner This class is a scanner generated by JFlex 1.2.2

Variable Index

Attribute	
nested_counter	The nested_counter helps to know when we are allowed to leave a RDF description in case we have a embedded RDF.
ns	Boolean set if we have a namespace declaration.
NS_Prefix	NS_Prefix is for saving the prefixes of the RDF namespace.
QName_Begin	
QName_End	
RDF_Attribute	
RDF_Attribute_IDSYMBOL	
RDF_prefix_set	RDF_prefix_set containing the prefixes for the RDF namespace.
RDF_QName_Begin	
RDF_QName_End	lexical states
RDF_YYINITIAL	
yy_atBOL	yy_atBOL == true <= the scanner is currently at the beginning of a line
yy_atEOF	yy_atEOF == true <= the scanner has returned a value for EOF
YY_ATTRIBUTE	YY_ATTRIBUTE[aState] contains the attributes of state `aState`
yy_buffer	this buffer contains the current text to be matched and is the source of the yytext() string
yy_currentPos	the current text position in the buffer
yy_endRead	endRead marks the last character in the buffer, that has been read from input
yy_eof_done	denotes if the user-EOF-code has already been executed
YY_ERROR_MSG	
YY_ILLEGAL_STATE	
yy_lexical_state	the current lexical state

yy_markedPos	the text position at the last accepting state
YY_NO_MATCH	
yy_packed	The packed transition table of the DFA
YY_PUSHBACK_2BIG	
yy_pushbackPos	the text position at the last state to be included in yytext
yy_reader	the input device
yy_rowMap	Translates a state to a row index in the transition table
yy_startRead	startRead marks the beginning of the yytext() string in the buffer
yy_state	the current state of the DFA
YY_UNKNOWN_ERROR	
yychar	the number of characters up to the start of the matched text
yycmap	Translates characters to character classes
yycmap_packed	Translates characters to character classes
yycolumn	the number of characters from the last newline up to the start of the matched text
YYEOF	this character denotes the end of file
YYINITIAL	
yyline	number of newlines encountered up to the start of the matched text
yytrans	The transition table of the DFA

Constructor Index

Lexer(InputStream)	Creates a new scanner.
Lexer(Reader)	Creates a new scanner There is also a java.io.InputStream version of this constructor.

Method Index

attributename()	Recognize the attribute "ID" and "xml..." and the special prefix "xmlns".
hextodec(String)	Transforming the hexadecimal number given by the parameter to the corresponding decimal int.
next_token()	Resumes scanning until the next regular expression is matched, the end of input is encountered or an I/O-Error occurs.
nsattributename()	Recognize the attribute "ID" and "xml..." and the special prefix "xmlns".
RDF_ns()	Testing if the current yytext() containing a prefix that belong to the RDF namespace.
Ref_replace(String)	Replace a reference with the corresponding string.
symbol(int, Object)	Creating the Symbol that will be send to the parser.
yy_advance()	Gets the next input character.
yy_do_eof()	Contains user EOF-code, which will be executed exactly once, when the end of file is reached
yy_ScanError(int)	Reports an error that occured while scanning.
yy_unpack_cmap(String)	Unpacks the compressed character translation table.
yy_unpack(String)	Unpacks the compressed DFA transition table.

yybegin(int)	Enters a new lexical state
yyclose()	Closes the input stream.
yylength()	Returns the length of the matched text region.
yypushback(int)	Pushes the specified amount of characters back into the input stream.
yystate()	Returns the current lexical state.
yytext()	Returns the text matched by the current regular expression.

Class VRP.Parser.parser

```
java.lang.Object
    |
    +----java_cup.runtime.lr_parser
            |
            +----VRP.Parser.parser
```

public class **parser**
extends java_cup.runtime.lr_parser CUP v0.10j generated parser.

Variable Index

_action_table	Parse-action table.
_production_table	Production table.
_reduce_table	`reduce_goto` table.
action_obj	Instance of action encapsulation class.
error_count	Counting reported errors.
filename	Containing the name of the file we are parsing.
lexer	The lexer the parser works together with.
syn_err_message	Message for Syntax errors.
this_model	The model we want to store the triples at.
warn_count	Counting reported warnings.

Constructor Index

parser()	Default constructor.
parser(Scanner)	Constructor which sets the default scanner.
parser(String)	Parser called with String or Reader.

Method Index

action_table()	Access to parse-action table.
do_action(int, lr_parser, Stack, int)	Invoke a user supplied parse action.
EOF_sym()	`EOF` Symbol index.
error_sym()	`error` Symbol index.

init_actions()	Action encapsulation object initializer.
production_table()	Access to production table.
reduce_table()	Access to `reduce_goto` table.
report_error(String, Object)	Report a non fatal error (or warning).
report_fatal_error(String, Object)	Report a fatal error.
report_warning(String, Object)	Printing a warning and counting them.
report(String, Object)	
scan()	Scan to get the next Symbol.
start_production()	Indicates start production.
start_state()	Indicates start state.
syntax_error(Symbol)	This method is called when a syntax error has been detected and recovery is about to be invoked.
user_init()	User initialization code.

Class VRP.Parser.CUP$parser$actions

```
java.lang.Object
    |
    +----VRP.Parser.CUP$parser$actions
```

class **CUP$parser$actions**
extends java.lang.Object Cup generated class to encapsulate user supplied action code.

Variable Index

about_Each_list	Containing the aboutEach statements that will be done at the end.
attr_hashmap	Hash-map containing the attributes and their values.
attr_ns_list	List of prefixes used in a attribute list to check if the prefix exists in the refering prefix_hashmaps.
deep_count	Counting how deep inside an element we are.
e	For reporting an error.
ID_hashset	Hash set containing the IDs for RDF:bagID and RDF:ID.
member_list	List of Members in a container.
parser	
prefix_hashmap	Hash map containing hash maps with the prefix and namespaces.
propertyElt_list	EropertyElt_list property elements inside a typedNode.
propertyElt_stack	A stack containing the propertyElt_lists.
tripel_count	Counting the created triples.

Constructor Index

CUP$parser$actions(parser)	Constructor

Method Index

alternative_test(Object)	Tests if an alternative is declared.
attr_hashmap_absolut(Object)	Returns the actual valid attribute list with the absolut path expressions.
attr_ns_enter(Object)	Enters the namespace-prefix in the attribute namespace list.
attr_ns_prefix_test()	Tests if the namespace prefixes in the attr_ns_list are inside the corresponding namespace maps.
attribut_list_test(Object)	Tests if other atributes than ID are specified in the attribut_list.
bagid_create(String, String, String, String, boolean, int)	Is called when the attribute 'rdf:bagID' is inside the attribute list.
container(String, Object)	Called by a container to create the corresponding statements.
creat_orig_1(Object, Object)	For the 'rdf:parseType="Literal"' we need to recreate the original text.
creat_orig_2(Object, Object, Object, Object)	For the 'rdf:parseType="Literal"' we need to recreate the original text.
creat_orig_str(Object, Object)	For the 'rdf:parseType="Literal"' we need to recreate the original text.
CUP$parser$do_action(int, lr_parser, Stack, int)	Method with the actual generated action code.
cut_quotas(String)	Returns the string without " or ' at beginning and end.
emit_error(int, int, String, String, Object)	Printing the error and counting them.
emit_warning(String, Object)	Printing the warning and counting it.
enter_attr(Object, Object)	Enters a attribute name in the valid map in <attr_hashmap.
enter_ID(Object)	Enters the ID into the <ID_hashset.
enter_maps()	This Method is called to raise the deep count counter and to enter new maps to <prefix_hashmap and <attr_hashmap.
get_current_attrlist()	Returns the current valid attribute list.
get_ns(Object)	Returns the namespace of an QName.
has_prefix(String)	If qname == {QName} ":" {QName} this method returns 'true' otherwise it will return 'false'.
init_hashmap()	Enters the hashmap 'map_0' in a new hashmap and returns this map.
init_propertyElt_stack	Init the propertyElt_stack by entering an empty array list.
name_compare(Object, Object)	Comparing two object.
new_triple(String, String,	Triple creation

String, boolean)	
ns_enter(Object, Object)	Enters the namespace and the prefix in the map in <prefix_hashmap.
one_name(Object)	This Method is called with every element having only a start tag.
org_attr()	For the 'rdf:parseType="Literal"' we need to recreate the original text.
out(String)	Printing the message to the standard output.
prefix_test(Object)	Tests if a prefix is defined.
prop_lit(Object, Object)	Called by property elements having the attribute 'rdf:parseType="Literal"'.
prop_val(Object, Object)	Called by property elements having a value as child.
propAttr(Object)	Called by property elements not having children to create the corresponding statements.
propRes(Object)	Called by property elements having the attribute 'rdf:parseType="Resource"' to create the statements for it.
rdf_ns(Object)	If the corresponding namespace is not Main.RDF_NS it prints a warning.
remove_maps()	Removes the corresponding hashmap at the end of an element out of the <prefix_hashmap hashmap and <attr_hashmap.
result()	Creating the result and output for the parsing prozess.
toabsolut(Object)	Transforms the name to the absolut path name by replacing the prefix with the ns.
two_names(Object, Object)	This Method is called with every element having a start and end tag.
typedNode(Object)	Is called by descriptions to create the corresponding statements.

Class VRP.Parser.sym

```
java.lang.Object
    |
    +----VRP.Parser.sym
```

public class **sym**
extends java.lang.Object CUP generated class containing symbol constants.

Variable Index

ALELEND	
ALEND	
ALT	
ATTRIBUTENAME	
ATTRIBUTEVALUE	
BAG	

BAGID	
ELEND	
ELSTART	
EOF	
EQUAL	
error	
IDNAME	
LIST	
MEMBERATTR	
PARSETYPELIT	
PARSETYPERES	
QNAME	
RDFBEGIN	
RDFEND	
RESOURCEATTR	
SEQ	
STRING	
XMLATTR	
XMLLANG	
XMLNS	
XMLQNAME	
XMLSPEC	

Constructor Index

sym()	

Class VRP.Parser.UpCall

```
java.lang.Object
    |
    +----VRP.Parser.UpCall
```

public class **UpCall**
extends java.lang.Object UpCall.java - A class to transport the information to wrapped elements.

Variable Index

object	The object of the triple we want to create later.
predicate	The predicate of the triple we want to create later.
res_res	Boolean saying if the object is a resource or not.

Constructor Index

UpCall(Object, Object, boolean)	Create a new UpCall (for the parser actions).

Method Index

getObject()	Get the object of the UpCall.
getPredicate()	Get the predicate of the UpCall.
getRes_Lit()	In case the object is a Literal it returns false else true.

Class VRP.Parser.UpCall_aE

```
java.lang.Object
    |
    +----VRP.Parser.UpCall_aE
```

class **UpCall_aE**
extends java.lang.Object UpCall_aE.java - Containing information about the RDF "aboutEach" statements. We collect this info and create the triples before returning the result, so, the container should be created.

Variable Index

attr_val	Array containing specified properties, objects and the Boolean if the object is a Resource (in the form of UpCalls) of the Description.
container	The name of the container we refer to.

Constructor Index

UpCall_aE()	Create a new UpCall_aE element.

Method Index

add_attr_val(UpCall)	Enter predicate, subject and boolean of the triples.
get_attr_val()	Get the list of predicates, subjects and booleans of the triples.
get_container()	Get the container name.
set_container(String)	Set the container name.

Class VRP.Model.Link

```
java.lang.Object
    |
    +----VRP.Model.Link
```

public class **Link**

extends java.lang.Object Link.java - A class for links containing subject and object of a statement.

Variable Index

object	Containing the object of the Link.
subject	Containing the subject of the Link.

Constructor Index

Link()	Create a new Link.
Link(String, String)	Create a new Link and set the subject and object.

Method Index

getobject()	Get the object of a link.
getsubject()	Get the subject of a link.
setobject(String)	Set the object of a link.
setsubject(String)	Set the subject of a link.

Class VRP.Model.Model

```
java.lang.Object
    |
    +----VRP.Model.Resource
             |
             +----VRP.Model.Model
```

public class **Model**
extends Resource Model.java - All we can do to a model.

Variable Index

anonymos	Counter for the creation of the generic generated IDs.
BaseURI	Containing the pathname of the corresponding file.
counter	Counter to count the created triples.
e	For reporting errors.
existing	Hash map containing the model.
subClassOf	Representing a Directed Acyclic Graph of Classes.
subPropertyOf	Representing a Directed Acyclic Graph of Properties.

Constructor Index

Model()	Create a new model.

Method Index

addStatement(String,	Entering a statement to the model.

String, String, boolean)	
containermembers(String)	Is called by parser in case of the attribute "aboutEach".
dump()	Dump the content of the model and print it to the outputfile defined in RDF_PAV/Main.
enter_domain_range(int, Object, Object, String, String)	Entering a domain (1) or range (2) statement to the <subject>and making a Class out of the <object>.
enter_msp(Object, String, String)	Entering a member to the <subject>and make a RDF_Container out of it.
enter_Property(String)	If <pred>is not already entered to the model as a Property it will be done by this method.
enter_s_p_o(int, Object, Object, String, String)	Entering a subject (1), predicate (2), object (3) statement to the <subject>.
enter_sA_iDB_C_L_V(int, Object, String, String)	Entering a seeAlso (1), isDefinedBy (2), comment (3), label (4), value (5) statement to the <subject>.
enter_subClassOf(Object, Object, String, String)	Entering a subClassOf statement to the <subject>and making a Class out of the <object>.
enter_subPropertyOf(Object, Object, String, String)	Entering a subPropertyOf statement to the <subject>and making a Propery out of the <object>.
enter_type(Object, Object, String, String)	Entering a type statement to the <subject>and making a Class out of the <object>.
genSysID()	Whenever a system-generated identifier is needed for statements or nodes.
getandadd_classinfo(RDF_Class, Model)	Get and enter the typical information of a RDF_Class for <rdf_class and enter them to the model <model>.
getandadd_propertyinfo(RDF_Property, Model)	Get and enter the typical information of a RDF_Property for <rdf_class and enter them to the model <model>.
getandadd_rdf_resourceinfo(RDF_Resource, Model)	Get and enter the typical information of a RDF_Resource for <rdf_class and enter them to the model <model>.
getandadd(Model, String, String)	Get the information of the resource with the URI <about>from this.model and enter the statements to the <to_model.model.
getBaseURI()	Get the base URI for the model.
getStatements()	Enumerate through the statements in this model.
make_class(String)	Making a Class out of the <object>.
make_container(String)	Making a Container out of the <object>.
make_property(String)	Making a Property out of the <object>.
make_statement(String)	Making a Statement out of the <object>.
printStatement(String, String, String)	Printing the statements to the output file defined in RDF_PAV/Main.
RDF_Property_test(String)	Testing in the case that <pred>starts with RDF or RDF-Schema namespace if they belong to the namespace and is a Property.
rdf_res_dump(RDF_Resource)	Returning the information of a RDF_Resource in a String.
RDF_test(String)	Testing in the case that <str>starts with RDF or RDF-Schema namespace if they belong to the namespace.
setBaseURI(String)	Set the base URI for the model.

Class VRP.Model.RDF_Class

```
java.lang.Object
    |
    +----VRP.Model.Resource
            |
            +----VRP.Model.RDF_Resource
                    |
                    +----VRP.Model.RDF_Class
```

public class **RDF_Class**
extends RDF_Resource RDF_Class.java

Variable Index

subClassOf	List containing the super classes of the Class.

Constructor Index

RDF_Class()	Create a new RDF_Class.
RDF_Class(RDF_Resource)	Cast the RDF_Resource to RDF_Class by creating a new RDF_Class.
RDF_Class(Resource)	Cast the Resource to RDF_Class by creating a new RDF_Class.
RDF_Class(String)	Create a new RDF_Class and set the URI.
RDF_Class(URL)	Create a new RDF_Class and set the URI.

Method Index

addsubClassOf(String)	Enter a subClassOf statement to the class.
getsubClassOf()	Get all subClassOf statements of the class.

Class VRP.Model.RDF_Container

```
java.lang.Object
    |
    +----VRP.Model.Resource
            |
            +----VRP.Model.RDF_Resource
                    |
                    +----VRP.Model.RDF_Container
```

public class **RDF_Container**
extends RDF_Resource RDF_Container.java

Variable Index

member	Containing the URIs or Literals of the members of the Container.

Constructor Index

RDF_Container()	Create a new RDF_Container.
RDF_Container(RDF_Resource)	Cast a RDF_Resource to a new RDF_Container.
RDF_Container(Resource)	Cast a Resource to a new RDF_Container.
RDF_Container(String)	Create a new RDF_Container and set the URI.
RDF_Container(URL)	Create a new RDF_Container and set the URI.

Method Index

addmember(String)	Enter a new member to the member_list.
getmember()	Get the subject of the statement.

Class VRP.Model.RDF_DAG

```
java.lang.Object
    |
    +----VRP.Model.RDF_DAG
```

public class **RDF_DAG**
extends java.lang.Object RDF_DAG.java - DAG stands for Directed Acyclic Graph.

Variable Index

dag	Containing the elements of the adjazence list.
no_loop	In case a loop occures during the method loopcheck it will be set to false.
order	Containig the order of the graph (available after running loopcheck).

Constructor Index

RDF_DAG()	Create a new RDF_DAG.

Method Index

enter(String, String)	Enters a link from 'sub' to 'superResource' in the DAG.
enumerate()	Print the DAG-Graph to Std IO.
loopcheck()	Checking the graph for loops and creating the topological order list.

output(List)	Prints the topological order of the DAG to the <OutputFile>.
paint_white(HashMap)	Paint the whole graph white by entering the elements in the color hash map having the color "w".
remove(Resource, Resource)	Tries to remove the link from 'sub' to 'superResource' out of the DAG.
visit(Object, HashSet, HashMap, HashMap)	Make a deepsearch for the element <actual>.

Class VRP.Model.RDF_Property

```
java.lang.Object
    |
    +----VRP.Model.Resource
             |
             +----VRP.Model.RDF_Resource
                      |
                      +----VRP.Model.RDF_Property
```

public class **RDF_Property**
extends RDF_Resource RDF_Property.java - The class for the Properties.

Variable Index

domain	The domains of the property.
link_list	A list of subjects and objects the property connects.
range	The range of the property.
subPropertyOf	Containing all direct superproperties of this property.

Constructor Index

RDF_Property()	Create a new RDF_Property.
RDF_Property(RDF_Resource)	Casting a RDF_Resource to a new RDF_Property.
RDF_Property(Resource)	Casting a Resource to a new RDF_Property.
RDF_Property(String)	Create a new RDF_Property and set the URI.
RDF_Property(URL)	Create a new RDF_Property and set the URI.

Method Index

adddomain(String)	Set the domain(s) of the property.
addlink(String, String)	Add a link(subject, object) to the property.
addsubPropertyOf(String)	Entering a subPropertyOf statement to the property.
getdomain()	Getting the domain(s) of the property.
getlink()	Getting all links of the property.
getrange()	Get the range of the property.

getsubPropertyOf()	Getting all subPropertyOf statements of the property.
link_contains(String, String)	Tests if the list of links already contains the link(subject, object).
setdomain(ArrayList)	Setting the domain(s) of the property.
setrange(String)	Set the range of the property.

Class VRP.Model.RDF_Resource

```
java.lang.Object
    |
    +----VRP.Model.Resource
            |
            +----VRP.Model.RDF_Resource
```

public class **RDF_Resource**
extends Resource RDF_Resource.java - For Resources containing RDFS or RDF info.

Variable Index

comment	List containing the comments of the Resource.
isDefinedBy	List containing the isDefinedBy statements of the Resource.
label	List containing the comments of the Resource.
seeAlso	List containing the seeAlso statements of the Resource.
type	List containing the types of the Resource.
value	List containing the values of the Resource.

Constructor Index

RDF_Resource()	Create a new RDF_Resource.
RDF_Resource(Resource)	Cast a Resource to a RDF_Resource.
RDF_Resource(String)	Create a new RDF_Resource and set the URI.
RDF_Resource(URL)	Create a new RDF_Resource and set the URI.

Method Index

addcomment(String)	Add a comment to the resource.
addisDefinedBy(String)	Add a isDefiniedBy statement to the resource.
addlabel(String)	Add a label to the resource.
addseeAlso(String)	Add a seeAlso statement to the resource.
addtype(String)	Add a type to the resource.
addvalue(String)	Add a value statement to the resource.
getcomment()	Get the comments of the resource.
getisDefinedBy()	Get the isDefinedBy statements of the resource.
getlabel()	Get the labels of the resource.
getseeAlso()	Get the seeAlso statements of the resource.

gettype()	Get the type of the resource.
getvalue()	Get the value statements of the resource.
isClass()	Returns 'true' if the type rdfs:Class is specified for this resource.
isProperty()	Returns 'true' if the type rdf:Property is specified for this resource.
isStatement()	Returns 'true' if the type rdf:Statement is specified for this resource.
setcomment(ArrayList)	Set the comments of the resource.
setisDefinedBy(ArrayList)	Set the isDefinedBy statements of the resource.
setlabel(ArrayList)	Set the labels of the resource.
setseeAlso(ArrayList)	Set the seeAlso statements of the resource.
settype(ArrayList)	Set the type of the resource.
setvalue(ArrayList)	Set the value statements of the resource.

Class VRP.Model.RDF_Statement

```
java.lang.Object
    |
    +----VRP.Model.Resource
             |
             +----VRP.Model.RDF_Resource
                      |
                      +----VRP.Model.RDF_Statement
```

public class **RDF_Statement**
extends RDF_Resource RDF_Statement.java - The class for Statements.

Variable Index

object	The object of the RDF_Statement.
predicate	The predicate of the RDF_Statement.
subject	The subject of the RDF_Statement.

Constructor Index

RDF_Statement()	Create a new RDF_Statement.
RDF_Statement(RDF_Resource)	Cast a RDF_Resource to a new RDF_Statement.
RDF_Statement(Resource)	Cast a Resource to a new RDF_Statement.
RDF_Statement(String)	Create a new RDF_Statement and set the URI.
RDF_Statement(URL)	Create a new RDF_Statement and set the URI.

Method Index

getobject()	Get the object of the statement.

getpredicate()	Get the predicate of the statement.
getsubject()	Get the subject of the statement.
setobject(String)	Enter the object to the statement.
setpredicate(String)	Enter the predicate to the statement.
setsubject(String)	Enter the subject to the statement.
valid()	Returns true if all three (subject, predicate and object) are set (not null).

Class VRP.Model.RDF_Validator

```
java.lang.Object
    |
    +----VRP.Model.RDF_Validator
```

public class **RDF_Validator**
extends java.lang.Object RDF_Validator.java

Variable Index

basis_model	The model we want to validate.
done_list	Containing URIs for elements we entered the information from the namespace.
model_list	Hash map containing models of namespaces.

Constructor Index

RDF_Validator()	Create a new RDF_Validator.

Method Index

compare_classes(String, String)	The class with URI should be a subset of the class with URI <super>.
domain_check(String, ArrayList)	Helper method for 'domain_range_check' in case domain is set.
domain_range_check()	The domain and range check.
fetch_information(String)	In the case the URI <next>belongs to a namespace, we try to get the information about <next>and enter them to our basis_model.
has_type(String, String)	Returning true if the Resource has type or if the type can be inherit.
namespaces()	Try to get all information of RDF_Properties and RDF_Classes we entered to the Model.
range_check(String, String)	Helper method for 'domain_range_check' in case range is set.
sPO_range_domain(RDF_Property)	Testing if the domain and range of a sub property fit to the domain and range of the super properties.
subClassOf_check()	Testing for loops in the subClassOf Statements.

subProperty_check()	Tests for loops in the subPropertyOf Statements and calls the methode sPO_range_domain to compare the range and domain definitions of sub and super property.
type_check()	Testing if each element of the classes RDF_Class, RDF_Property, RDF_Statement and RDF_Container have the corresponding type.
type_inherit(int, ArrayList, String)	Type inheritance for (1) classes (2) instances.
validate(Model)	The method starting all the jobs for the validation.

Class VRP.Model.Resource

```
java.lang.Object
    |
    +----VRP.Model.Resource
```

public class **Resource**
extends java.lang.Object Resource.java - A node in an RDF Model. Also the base class that all RDF things are derived from. We don't actually DO much with a resource, other than get or set its URI.

Variable Index

uri	The URI of a Resource.

Constructor Index

Resource()	Create a new Resource.
Resource(String)	Create a new Resource and set the URI.
Resource(URL)	Create a new Resource and set the URI.

Method Index

getID()	Get the URI of the resource.
setID(String)	Set the URI of the resource.

Appendix H – The VRP RDF/BNF Grammar

This grammar is contained by the file *parser.cup* and implements the grammar for RDF as shown in Appendix D plus features inherited from XML. The numbers in brackets in the comments of this grammar correspond to the numbers in the RDF grammar.

```
//
// SYMBOL LISTS
//
/* Terminals (tokens returned by the scanner). */
terminal ELSTART, ALEND, ELEND, ALELEND, QNAME;
terminal RDFBEGIN, RDFEND, STRING, IDNAME;
terminal ATTRIBUTENAME, EQUAL, ATTRIBUTEVALUE;
terminal SEQ, ALT, BAG, XMLQNAME;
terminal MEMBERATTR, PARSETYPELIT, PARSETYPERES;
terminal XMLNS, XMLLANG, XMLSPEC, XMLATTR;
terminal LIST, RESOURCEATTR, BAGID;

/* Non terminals */
non terminal   rdf, obj, description, container, propertyele, propertyele2;
non terminal   sequence, bag, alternative, resourceattr;
non terminal   attributes, member1, member, containerAttr1, containerAttr;
non terminal   containerAttr2, attributename, attributelist, xmlAttr;
non terminal   idname, xmldekl, nsdekl, xmldekl_list, all_attr_list;
non terminal   literal, parseLiteral, parseResource, value, xmlresAttrxml;
non terminal   xmlidxml2, xmlidxml, xmlidxml1, parslitattr_list;
non terminal   begin, xmlparseResxml, xmlparseResxml1, xmlparseLitxml;
non terminal xmlparseLitxml1, parsresattr_list, xmlresAttrxml1, langdekl;
non terminal   root, element, element2;

//
// THE GRAMMAR
//
begin ::= XMLSPEC rdf
          {: RESULT = result(); :}
        | rdf
          {: RESULT = result(); :}
        | XMLSPEC root
          {: RESULT = result(); :}
        | root
          {: RESULT = result(); :}
        ;
/* [6.1] RDF M&S + optional you can start with an XML-Spec.
   like "<?xml version=1.0 ?>"
   If we have RDFBEGIN then there must be the RDF-NS as default NS.
   Otherwise the prefix given by RDFBEGINNS must be inside a NS deklaration
   refering to the RDF-NS */
rdf ::= ELSTART RDFBEGIN:n1 xmldekl_list ALEND obj ELEND RDFEND:n2 ALEND
               {: name_compare(n1, n2);
                  rdf_ns(n1);
               :}
        | description
        | container
        | error:e {: emit_error(1, 1, "", "parser", e); :}
        ;
////////////////////////////////////////////////////////////////////////
// eingebunden in HTML/XML !! HTML must be well-formed for this parser!!
root ::= ELSTART XMLQNAME:n1 xmldekl_list ALEND element ELEND XMLQNAME:n2 ALEND
        {: name_compare(n1, n2); :}
        ;
element ::= element2 element
        | /* Nil */
        ;
element2 ::= ELSTART XMLQNAME:n1 xmldekl_list ALEND element ELEND XMLQNAME:n2 ALEND
        {: remove_maps();
           name_compare(n1, n2); :}
        | ELSTART XMLQNAME xmldekl_list ALELEND
        {: remove_maps(); :}
        | rdf
```

```
        ;
////////////////////////////////////////////////////////////////////////
// [6.2] RDF M&S
obj ::= description obj
        | container obj
        | /* Nil */
        ;
// [6.3] RDF M&S + [6.13] RDF M&S
description ::= ELSTART QNAME:n1 attributelist ALEND propertyele ELEND QNAME:n2 ALEND
                {: RESULT = typedNode(n1);
                   two_names(n1,n2);
                   /* out("in a description 1"); */ :}
        | ELSTART QNAME:n attributelist ALELEND
                {: RESULT = typedNode(n);
                   one_name(n); :}
        |  ELSTART QNAME:n error:e {: emit_error(1, 2, "", "parser", e); :}
        ;
// [6.4] RDF M&S
container ::= sequence:res
        {: RESULT = res; :}
        | bag:res
        {: RESULT = res; :}
        | alternative:res
        {: RESULT = res; :}
        ;
attributelist ::= attributes
        {: attr_ns_prefix_test();
           enter_maps(); :}
        ;
// [6.5]-[6.11] RDF M&S
attributes ::= /* NIL */
        | attributename:attr EQUAL ATTRIBUTEVALUE:aval attributes
                {: attr_ns_enter(attr);
                   enter_attr(attr, aval); :}
        // [6.6] idAttr
        | IDNAME:n EQUAL ATTRIBUTEVALUE:ID attributes
                {: attr_ns_enter(n);
                   enter_attr(n, ID);
                   enter_ID(ID);  :}
        // [6.9] bagIdAttr
        | BAGID:n EQUAL ATTRIBUTEVALUE:ID attributes
                {: attr_ns_enter(n);
                   enter_attr(n, ID);
                  enter_ID(ID);  :}
        | nsdekl attributes
                {: :}
        | langdekl attributes
        | xmlAttr attributes
        | error:e {: emit_error(1, 3, "", "parser", e); :}
        ;
// special Attributes: (At the moment only ATTRIBUTENAME and RESOURCEATTR in use.)
attributename ::=  ATTRIBUTENAME:a
                {: RESULT = a; :}
        // [6.18] resourceAttr
        | RESOURCEATTR:a
                {: RESULT = a; :}
        ;
// [6.6] RDF M&S
idname ::= IDNAME:n EQUAL ATTRIBUTEVALUE:ID
                {: attr_ns_enter(n);
                   enter_ID(ID);
                   RESULT = ID;
                   /* out("idnamens"); */ :}
        ;
// [6.12] RDF M&S
propertyele ::= propertyele2 propertyele
        | /* Nil */
        ;
// This splitting is needed to reduce the property elements in the right order.
// Otherwise the maps for the valid namespaces would not fit.
propertyele2 ::=  ELSTART QNAME:n1 attributelist ALEND value:val ELEND QNAME:n2 ALEND
```

```
        // In this attributelist are only xml declarations and the idattr allowed.
        // To check this we call the method attribut_list_test.
        // The attributelist can be found in the hashmap
'attr_hashmap.get("map_"+deep_count)'.
            {: attribut_list_test(n1);
                attr_ns_prefix_test();
                prop_val(n1, val);
                two_names(n1, n2); :}
        | ELSTART QNAME:n1 parslitattr_list ALEND literal:lit ELEND QNAME:n2 ALEND
          {: attribut_list_test(n1);
                prop_lit(n1, lit);
                two_names(n1,n2); :}
        | ELSTART QNAME:n1 parsresattr_list ALEND propertyele ELEND QNAME:n2 ALEND
            {: attribut_list_test(n1);
                propRes(n1);
                two_names(n1,n2); :}
        | ELSTART QNAME:n attributelist ALELEND
            {: propAttr(n);
//              out("property");
               one_name(n); :}
        | ELSTART QNAME:n error:e {: emit_error(1, 4, "", "parser", e); :}
        ;
parslitattr_list ::= attributes parseLiteral attributes
        {: attr_ns_prefix_test();
           enter_maps(); :}
        ;
parsresattr_list ::= attributes parseResource attributes
        {: attr_ns_prefix_test();
           enter_maps(); :}
        ;
// [6.17] RDF M&S
// any XML text with '<', '>' and '&' escaped.
value ::= STRING:str
        {: RESULT = new UpCall(null, str, false); :}
        | // empty string
        {: RESULT = new UpCall(null, "", false); :}
        | container:cont
        {: RESULT = new UpCall(null, cont, true); :}
        | description:des
        {: RESULT = new UpCall(null, des, true); :}
        ;
// [6.18] RDF M&S
// resourceAttr ::= 'resource="URI-reference"'
resourceattr ::= RESOURCEATTR:r EQUAL ATTRIBUTEVALUE:attr
            {: attr_ns_enter(r);
               RESULT = attr; :}
        ;
// [6.25] RDF M&S
sequence ::= ELSTART SEQ:n1 xmlidxml:ID ALEND member ELEND SEQ:n2 ALEND
            {: two_names(n1,n2);
               RESULT = container(Main.R_Seq, ID); :}
        |  ELSTART SEQ:n containerAttr:ID ALELEND
            {: one_name(n);
               RESULT = container(Main.R_Seq, ID);
               /* out("seq_2"); */ :}
        |  ELSTART SEQ error:e {: emit_error(1, 5, "", "parser", e); :}
        ;
// [6.26] RDF M&S
bag ::= ELSTART BAG:n1 xmlidxml:ID ALEND member ELEND BAG:n2 ALEND
            {: two_names(n1,n2);
               RESULT = container(Main.R_Bag, ID);:}
        |  ELSTART BAG:n containerAttr:ID ALELEND
            {: one_name(n);
               RESULT = container(Main.R_Bag, ID); :}
        |  ELSTART BAG error:e {: emit_error(1, 6, "", "parser", e); :}
        ;
// [6.27] RDF M&S
// Test auf mindestens eine Alternative!!!!!
alternative ::= ELSTART ALT:n1 xmlidxml:ID ALEND member:mem ELEND ALT:n2 ALEND
            {: two_names(n1,n2);
               RESULT = container(Main.R_Alt, ID);
```

```
            alternative_test(mem); :}
       |  ELSTART ALT:n containerAttr:ID ALELEND
          {: one_name(n);
             RESULT = container(Main.R_Alt, ID);:}
       |  ELSTART ALT error:e {: emit_error(1, 7, "", "parser", e); :}
       ;
containerAttr ::= containerAttr1:ID
              {: attr_ns_prefix_test();
                 enter_maps();
                 RESULT = ID; :}
       ;
// IDNAME ONLY ONCE
containerAttr1 ::= MEMBERATTR:m EQUAL ATTRIBUTEVALUE:attr containerAttr1
              {: prefix_test(m);
                  member_list.add(new UpCall(null, attr, false)); :}
       | idname:ID containerAttr2
              {: RESULT = ID; :}
       | nsdekl containerAttr1
       | langdekl containerAttr1
       | xmlAttr containerAttr1
       | /* Nil */
       ;
containerAttr2 ::= MEMBERATTR:m EQUAL ATTRIBUTEVALUE:attr containerAttr2
              {: prefix_test(m);
                  member_list.add(new UpCall(null, attr, false)); :}
       | nsdekl containerAttr2
       | langdekl containerAttr2
       | xmlAttr containerAttr2
       | /* Nil */
       ;
// [6.28]-[6.30] RDF M&S
// important for ns-maps
member ::= member1:mem member
              {: RESULT = mem; :}
       | /* Nil */
       ;
member1 ::= ELSTART LIST:n xmlresAttrxml:attr ALELEND
         {: one_name(n);
            RESULT = n;
            member_list.add(attr); :}
       | ELSTART LIST:n1 xmldekl ALEND value:val ELEND LIST:n2 ALEND
         {: enter_maps();
            two_names(n1,n2);
            RESULT = n1;
            member_list.add(val);:}
       | ELSTART LIST:n1 xmlparseLitxml ALEND literal:lit ELEND LIST:n2 ALEND
         {:two_names(n1,n2);
            RESULT = n1;
           member_list.add(new UpCall(null, lit, false));:}
       | ELSTART LIST:n1 xmlparseResxml ALEND propertyele ELEND LIST:n2 ALEND
         {: two_names(n1,n2);
            RESULT = n1;
            member_list.add(propRes(n1)); :}
       | ELSTART LIST error:e {: emit_error(1, 8, "", "parser", e); :}
       ;
// [6.32] RDF M&S
// '[prefix:]parseType="Literal"'
parseLiteral ::=  PARSETYPELIT:pt
              {: prefix_test(pt); :}
       ;
// [6.33] RDF M&S
// '[prefix:]parseType="Resource"'
parseResource ::= PARSETYPERES:pt
              {: prefix_test(pt); :}
       ;
// [6.34] RDF M&S
// WELL-FORMED XML
// As the RESULT we we return the original test.
// parseResource and parseLiteral are not allowed in attributelist jet !!!!!!
// therefore we need an all_attr_list.
```

```
literal ::=  ELSTART QNAME:n1 all_attr_list ALEND literal:lit1 ELEND QNAME:n2 ALEND
literal:lit2
                  {: RESULT = creat_orig_2(n1, n2, lit1, lit2);
                     two_names(n1,n2); :}
        | ELSTART QNAME:n attributelist ALELEND literal:lit1
                  {: RESULT = creat_orig_1(n, lit1);
                     one_name(n); :}
        | STRING:str literal:lit1
          {: RESULT = creat_orig_str(str, lit1); :}
        | /* Nil */
           {: RESULT = ""; :}
        | error:e {: emit_error(1, 9, "", "parser", e); :}
        ;
all_attr_list ::= attributelist
        | parseLiteral
          {: enter_maps(); :}
        | parseResource
          {: enter_maps(); :}
        ;
//============================================================================
// Productions for xml statements.
//============================================================================
xmldekl_list ::= xmldekl
                {: enter_maps();
                   /* out("xmldekl_list"); */ :}
        ;
xmldekl ::= nsdekl xmldekl
        | langdekl xmldekl
        | xmlAttr xmldekl
        | /* Nil */
        ;
nsdekl ::= XMLNS:pre EQUAL ATTRIBUTEVALUE:ns
                {: /* out("Prefix: "+pre+" NS: "+ns); */
                   ns_enter(pre, ns);
                   enter_attr("xmlns:"+pre.toString(), ns); :}
        ;
langdekl ::= XMLLANG
                {: /* out("xmllang");
                   methode mit enter_attr fuer literal! */ :}
        ;
xmlAttr ::=  XMLATTR:xml EQUAL ATTRIBUTEVALUE:val
                {: enter_attr(xml, val); :}
        ;
xmlidxml ::= xmlidxml1:ID
                {: attr_ns_prefix_test();
                   enter_maps();
                   RESULT = ID; :}
        ;
xmlidxml1 ::= idname:ID xmlidxml2
            {: RESULT = ID; :}
        | nsdekl xmlidxml1
        | langdekl xmlidxml1
        | xmlAttr xmlidxml1
        | /* Nil */
        ;
xmlidxml2 ::= nsdekl xmlidxml2
        | langdekl xmlidxml2
        | xmlAttr xmlidxml2
        | /* Nil */
        ;
xmlparseResxml ::= xmlparseResxml1
        {: enter_maps(); :}
        ;
xmlparseResxml1 ::= xmldekl parseResource xmldekl
        {: /* out("parseType=Resource"); */ :}
        ;
xmlresAttrxml ::= xmlresAttrxml1:attr
        {: enter_maps();
           RESULT = attr; :}
        ;
xmlresAttrxml1 ::= xmldekl resourceattr:attr xmldekl
```

```
                {: attr_ns_prefix_test();
                   RESULT = attr; :}
        ;
xmlparseLitxml ::= xmlparseLitxml1
        {: enter_maps(); :}
        ;
xmlparseLitxml1 ::= xmldekl parseLiteral xmldekl;
```

References

ANTLR	ANTLR 2.0, Magelang Institute, 1997; http://www.antlr.org/
APPEL	"Modern Compiler Implementation in Java", by Andrew W. Appel, 1998, Cambridge University Press, for information see http://www.cs.princeton.edu~appel/modern
Browser	http://jigsaw.w3.org:8000/description
BYACC	BYACC/Java, by Bob Jamison, v.092 1999; http://www.lincom-asg.com/~rjamison/byacc/
CUP	CUP parser generator for Java, by Scott E. Hudson, 1997; http://www.cs.princeton.edu/ appel/modern/java/CUP/
DB RES	"Dave Beckett's Resource Description Framework (RDF) Resources"; http://www.cs.ukc.ac.uk/people/staff/djb1/research/metadata/rdf.shtml
DC	The Dublin Core initiative; http://purl.oclc/netadata/dublin_core
DESIRE	DESIRE Project; http://www.ukoln.ac.uk/metadata /desire/overview/rev_ti.html
DOM	"What is the Document Object Model?" J. Robie; http://www.w3.org/TR/REC-DOM-Level-1/introduction.html
F-Logic	http://www.informatik.uni-freiburg.de/~dbis/Publications/
GINF	http://www-diglib.stanford.edu/diglib/ginf/
HTML 4.0	http://www.w3.org/TR/REC-html40/index/elements.html
HTML 3.2	http://www.w3.org/TR/REC-html32.html
IBM	http://www.alphaworks.ibm.com/formula/rdfxml
JACCIE	JACCIE, by Lothar Schmitz; http://www2.informatik.unibw-muenchen.de/Research/Tools/JACCIE/jaccie.html
JavaCC	Java Compiler Compiler, by Sun Microsystems; http://www.suntest.com/JavaCC/.
JFlex	http://www.informatik.tu-muenchen.de/~kleing/jflex
JLex	"JLex: A lexical analyzer generator for Java™", by Elliot Berk, 1997; http://www.cs.princeton.edu/appel/modern/java/JLex/.
LEX	"Lex -a lexical analyzer generator", by M. E. Lesk, 1975, Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J.
MCFXML	"Meta Content Framework using XML", by R.V. Guha and Tim Bray, 1997; http://www.w3.org/TR/NOTE-MCF-XML/

META 1	"The skinny on metadata", Giovanni Flammia, IEEE Intelligent Systems Volume 14, Number 4 July/August 1999; http://computer.org/intelligent
META 2	"Metadata: Cataloging by Any Other Name ...", Jessica Milstead and Susan Feldman, Online, January 1999
MOZ	http://lxr.mozilla.org/seamonkey/source/rdf/base/idl/
PCCTS	"Language Translation Using PCCTS & C++", by Terence J. Parr, 1997, Automata Publishing Company, ISBN: 0962748854.
PerlPar	http://www.pro-solutions.com/download
PICS	"PICS Label Distribution Label Syntax and Communication Protocols", Version 1.1, W3C Recommendation 31-October-1996; http://www.w3.org/TR/REC-PICS-labels
RADIX	http://www.mailbase.ac.uk/lists/rdf-dev/1999-06/0002.html
RDF 1	"An Idiot's Guide to the Resource Description Framework" by Renato Iannella; http://archive.dstc.edu.au/RDU/reports/RDF-Idiot/
RDF 2	"An Introduction to the Resource Description Framework", Eric Miller,D-Lib Magazine May 1998; http://www.dlib.org/dlib/may98/miller/05miller.html
RDF 3	"RDF - What's in it for us?", by Janus Boye; http://www.irt.org/articles/js086/index.html
RDF 4	"RDF Made Easy", by John Cowan; http://www.ccil.org/~cowan/XML/RDF-made-easy.html
RDF Archive	http://www.w3.org/RDF/Interest
RDF M&S	"Resource Description Framework (RDF) Model and Syntax Specification", Ora Lassila and Ralph R. Swick, W3C Recommendation 22 February 1999; http://www.w3.org/TR/REC-rdf-syntax/
RDF Schema	"Resource Description Framework (RDF) Schema Specification", Dan Brickley and R. V. Guha, W3C Proposed Recommendation 03 March 1999; http://www.w3.org/TR/PR-rdf-schema/
rdf2cb	http://www.kbs.uni-hannover.de/theses/99/capelle/diplom.ps (German)
SAX	http://www.microstar.com/XML/SAX/
SiLRI	SiLRI, the Simple Logic-based RDF Interpreter; http://www.aifb.uni-karlsruhe.de/~sde/rdf/
SiRPAC	SiRPAC, the Simple RDF Parser and Compiler, by Janne Saarela http://www.w3.org/RDF/Implementations/SiRPAC
SIS	Semantic Index System, by Panos Constantopoulos and Martin Doerr; http://www.ics.forth.gr/proj/isst/Systems/SIS/index.html
Unicode	"The Unicode Standard – A Technical Introduction"; http://www.UNICODE.org/unicode/standard/principles.html

URI	"Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, R. Fielding, U.C. Irvine and L. Masinter, August 1998, RFC2396; http://www.ietf.org/rfc/rfc2396.txt
VRP Home	http://www.ics.forth.gr/proj/isst/RDF/
WF	"The Warwick Framework: A Container Architecture for Aggregating Sets of Metadata" ; Carl Lagoze, Clifford A. Lynch and Ron Daniel Jr.; http://cs-tr.cs.cornell.edu/Dienst/UI/2.0/Describe/ncstrl.cornell/TR96-1593
XML	"Extensible Markup Language (XML) 1.0", T. Bray, J. Paoli and C.M. Sperberg-McQeen, W3C Recommendation 10-February-1998; http://www.w3.org/TR/REC-xml
XML Names	"Namespaces in XML", Tim Bray, Dave Hollander and Andrew Layman, W3C Recommendation 14-January-1999; http://www.w3.org/TR/REC-xml-names
XMLDATA	"XML-Data", A. Layman, E. Jung, E. Maler, H.S. Thompson, J. Paoli, J. Tigue, N.H. Mikula and S. D. Rose, W3C Note 05-January-1998; http://www.w3.org/TR/1998/NOTE-XML-data
YACC	"YACC -yet another compiler compiler", by S. C. Johnson, 1975, Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.

Index

A

Abstract Syntax Tree (AST) 59
Adjazenz List 47
Alternative.....11, 29
Ambiguity 7
ANother Tool for Language Recognition (ANTLR).....61
Application Programming Interface (API)69

B

Bag 11, 29
BYACC.....59

C

Casting 48
Class.....24, 29
Compiler Generator.....56
Constructor of Useful Parsers (CUP).....58
Container.....11

D

Document Type Definition (DTD).....23

E

eXtensible Markup Language (XML).....13

G

Generic Interoperability Framework (GINF) 38

H

Homonym.....7

I

Implicit Instances 31
Independence 9
Interchange.....9

J

JAva-based Compiler-Compiler with Interactive Environment (JACCIE).....62
JavaCC 61
JFlex.....58
JLex.....57

L

Literal 10, 29

M

Metadata 7
META-tag 7

N

Namespace.....9
  Connection.....52
Nodes and Arcs Diagram 10

O

Object 10

P

Platform for Internet Content Selection (PICS) 9
Predicate 10
Property 10, 29
  Anti-Reflexive 34
  Transitive.....34
Purdue Compiler Construction Tool Set (PCCTS) 56, 60

R

rdf2cb 38, 66
Reification 12
Resource 9, 29
  Anonymous.....11
Resource Description Framework (RDF)...9
  3-tuples 13
  Alternative 11, 29
  Bag.....11, 29
  Class 24, 29
  Container 11
  Graph 10
  Literal 10, 29, 32
  Object 10
  Predicate 10
  Property 29
  Reified Statement.....29
  Resource 29
  Schema 23
  Sequence.....11, 29
  Subject 10
  Syntax 13
  Value.....10

S

Scalability.....9
Schema 23
Sequence.....11, 29
SiLRI 38
SiRPAC 37, 65

Statement....10
Creation....17
Reified....12, 29
Statement-Model of the File....33
Subject....10
Synonym....7

T

Triple....13
Type Inheritance....30

U

Unique Resource Identifier (URI)....9
Unique Resource Locator (URL)....10

V

Validating RDF Parser
Error Reporting....54
Error Types....54
Java Packages....42
Lexical Analyser....45
Syntax Analyser....46
Validation....50
Validating RDF Parser (VRP)....41, 65, 66
Validation....33
Value....10

Y

Yet Another Compiler Compiler (YACC)56