Relax NG XML

Relax NG XML

Table of Contents

List of Figures

List of Tables

Part I. Preface

Preface

Chapter 1. Preface

The "X" in XML stands for "eXtensible" and XML by itself is so extensible that I can invent new elements and attributes as I write my XML documents. The main limit of this extensibility is that I need to keep track of these elements and attributes and that I often need to convey to the applications what I intend to accept in my documents. This is needed for validation purposes and also to automate some of the most time consuming --and boring-- programming tasks. And this is where XML schema languages come to play.

XML Schema languages are a nice idea... as long as they don't add to XML a weight so heavy that XML becomes uneXtensible and, unfortunately, that's what was likely to happen before Relax NG. XML Schema, the dominant XML Schema language, had become so overloaded that it's both difficult to learn, difficult to extend and that its expressive power is too limited to describe all the possibilities offered by XML. Even though we can expect that many applications will accept this overweight, a lightweight and simpler alternative was needed for those of us who want to preserve all the extensibilty of a free style XML.

That's what Relax NG is really:

- a XML schema language
- focused on validating the structure of XML documents
- lightweight enough to be easy to learn, read and write
- powerful enough to describe virtually any vocabulary which is based on well formed XML and conform to namespaces in XML.

There are a couple of reasons why Relax NG is so much easier than W3C XML Schema and both contribute to make it also more reliable and safer to use: Relax NG has a very sound mathematical ground and it has been kept focussed on doing perfectly well a single thing --validating the structure of XML documents. Relax NG won't do the coffee for you, but if you need a schema language easy to use and which won't block you in a labyrinth of obscure limitations this is the language you should be using. Furthermore, an excellent open source tool is available which will convert your Relax NG schemas into other languages including W3C XML Schema.

Who Should Read This Book?

Read this book if you want to:

- Create Relax NG schemas.Status: Need to be updated after a first set of reviews.
- Understand existing Relax NG schemas.
- Discover that XML schema languages can be simple.

To understand this book, you should already have a basic understanding of the structure of XML documents but do not need to know any other XML schema language.

Who Should Not Read This Book?

Do not read this book if you will only be using existing Relax NG schemas to validate XML documents.

Organization of this book

- Chapter 1: Relax NG In Perspective - This chapter gives more perspective on the many aspects of XML validation, what is a schema language and what makes Relax NG really unique.

- Chapter 2: Simple Is Beautiful - This chapter introduces the background of Relax NG itself and insists on the notion of `pattern` which is the elementary building brick on which the whole language is built.

- Chapter 3: First Schema - Following the structure of the instance document used all over this book this chapter builds, step by step, a first complete Relax NG schema using the XML syntax.

- Chapter 4: Non XML Syntax - The XML syntax is simple, natural and reads almost as plain English but is also verbose. In this chapter we see the alternative compact (non XML) syntax. In the rest of this book, each example will present both the XML and the compact syntaxes so that you can either focus on one of them or learn both of them in parallel.

- Chapter 5: Flattening Our First Schema - Our first schema had been following the structure of our instance document in what is called a "Russian doll" design. In this chapter we show how named patterns can be used to limit the depth of a schema, provide re-usability or mimic a DTD.

- Chapter 6: More Patterns - Up to now, we have only seen ordered sequences of elements and in this chapter we introduce new compositors to define alternatives and unordered (interleaved) content models.

- Chapter 7: Constraining Text Values - In this chapter we introduce the generic mechanism used to constrain text values and the two Relax NG built-in datatypes (namely string and token).

- Chapter 8: Datatype Libraries - In this chapter we see how external datatype libraries may be plugged into Relax NG schemas and spend some type exploring the two datatype libraries which are most frequently used: the W3C XML Schema datatype library and the DTD compatibility datatype library.

- Chapter 9: W3C XML Schema Regular Expressions - One of the most powerful facet which can be used to add constraints on the W3C XML Schema datatype library is the `pattern` facet which relies on its own flavor of regular expressions presented in this chapter.

- Chapter 10: Creating Building Blocks - Now that we have now all the building blocks, we see in this chapter how we can reuse and redefine them in "grammars" which can be merged.

- Chapter 11: Namespaces - We explain briefly what XML namespaces are and discover in this chapter how straightforward is their support in RELAX NG.

- Chapter 12: Writing Extensible Schemas - This chapter covers both the extensibility of the schemas themselves and the extensibility of the class of instance documents described by a schema (in other words, its openness).

- Chapter 13: Annotating Schemas - Documentation may be targeted to human users but also to other applications and we cover both aspects in this chapter showing for instance how Schematron rules may be embedded in Relax NG schemas, and covering other annotations systems such as Bob DuCharme's schema document pipeline proposal and my own xvif.

- Chapter 14: Generating Relax NG schemas - This chapter explores different sources from which Relax NG can be generated, such as instance documents (Examplotron), UML diagrams, spreadsheets and litterate programming.

- Chapter 15: Simplification And Restrictions - This chapter goes into the details of the simplification of Relax NG documents described as a normative part of the Relax NG specification which understanding is needed to understand some few obscure limitations.

- Chapter 16: Determinism and Datatype Assignment - One of the strengths of Relax NG is to allow non deterministic schemas. While this is extremely convenient for validation purposes, this is an issue for assigning datatypes to the nodes of the instance documents (a controversial feature out of the scope of Relax NG but used by some applications). This chapter presents the concepts of schema determinism and ambiguity and their impacts on the different ways to use Relax NG schemas.

- Chapter 17: Relax NG Elements - This chapter is a short reference guide describing all the elements of the XML syntax with their description, synopsis and example.
- Chapter 18: Non XML Syntax Reference - This chapter is a short reference guide describing all the elements of the compact syntax with their description, synopsis and example.
- Chapter 19: W3C XML Schema Datatypes - This chapter is a short reference guide to W3C XML Schema datatypes often used as an external datatype library in Relax NG schemas.
- Chapter 20: Glossary - This chapter is a glossary providing a short explanation of the terms used all over the book.
- Appendix A: DSDL - This appendix presents the ISO DSDL project which does include Relax NG as its part 2.

Acknowledgments

I would like to thank the Relax NG OASIS Technical Committee for having provided the subject of this book which would obviously never have been possible without their work and especially Murata Makoto, James Clark and John Cowan for the timely and higly accurate answers they have provided to my many questions.

My own implementation of Relax NG has proven to be most useful in gaining a deep understanding of the language and I would also like to thank Uche Ogbuji who has been my Python mentor during this project and again James Clark for the detailed instructions of how Relax NG can be implemented using the very nice so called "derivative algorithm".

This book is the result of a collaborative work and I thank all the people having contributing comments and annotations, including the tech reviewers, David Eisenberg, John Cowan and Dave Pawson who have extended their comments well beyond the scope of simple tech reviews and have significantly improved its level of quality. This collaborative work would never have started without my editor, Simon St.Laurent who has believed in this book since before its beginning and just made it happen.

Finally, I need to thank my wife and children for their patience and moral support while I was busy writing this book. Unlike I had done in the preface of my previous book I won't dare to promise that they will recover their husband and father now that this book is over fearing that a new other challenging project might swallow me in a near future!

Powered by WikiML

Most if this book has been edited in a WikiWikiWeb powered by PhpWiki, a PHP implementation of the concept of WikiWikiWeb invented by Ward Cunningham in 1995 and famous for the simplicity of its text based markup. The WikiWikiWeb pages have been converted to XHTML pages using the parser developed by the WikiML project and these pages have been transformed through XSLT into DocBook for production at O'Reilly.

This is probably one of the first attempts to leverage on something as simple to use as a WikiWikiWeb to produce something as complex as a whole book markuped as DocBook and I have been surprised by the smoothness of the whole process.

To learn more about these subjects:

- http://c2.com/cgi-bin/wiki?WikiWikiWeb (WikiWikiWeb)
- http://phpwiki.sourceforge.net/ (PhpWiki)
- http://wikiml.org/ (WikiML)
- http://www.w3.org/TR/xslt (XSLT)

- http://www.oasis-open.org/docbook (DocBook)

By the way, why is it called Relax NG?

Relax stands for "Regular Language description for XML" and of course, it's also a joke from its author, Murata Makoto who used to advertise his language as: "Tired of complicated specifications? You just RELAX !". Despite its humourous name, Relax has been a very serious candidate as a XML schema language and it has been published as an ISO/IEC Technical Report in 2001 under the title: "ISO/IEC DTR 22250-1, Document Description and Processing Languages -- Regular Language Description for XML (RELAX) -- Part 1: RELAX Core".

Relax has then been merged with TREX (Tree Regular Expressions for XML), another XML schema language proposed by James Clark in 2001 under the name Relax NG (NG standing for "New Generation") the wish of both Murata Makoto and James Clark being that users of Relax and TREX gradually migrate to Relax NG.

Part II. User Guide

The user's guide is a tutorial showing all the features of RELAX NG through a gentle progression and many examples.

Chapter 2. Chapter 1: Relax NG In Perspective

XML is about diversity

I have heard people jest that XML was standing for "eXcellent Marketing Language" and I often felt that, unfortunately, this had become a very accurate definition. Nevertheless, the official meaning of the XML is "eXtensible Markup Language" and this one is still more accurate!

XML is not extensible in the sense that the XML specifications themselves would be extensible and many experts think that both the XML recommendation itself and the pile of XML related specifications have already become legacy and are very hard if not impossible to update and extend.

XML is extensible in the sense that it lets you define your own sets of elements and attributes which can express virtually any hierarchical structure. And it's not only accurate but real: the extensibility of XML is been used, some would even say overused, and we've lost the count of the different sets of XML elements and attributes (let's call them XML vocabularies) used by different people for different applications. And, of course, all of these vocabularies need to be validated which means that there is a need for validation tools easy to adapt to each of these vocabularies.

XML is about the independence of documents over applications

I have also heard many people elaborate on the tight relationship or parallel between XML and object orientation; saying that XML is the same paradigm for data than object orientation for programs and that XML is a perfect serialization format for object systems. That's not untrue, but we can also see XML as anti-object oriented, or maybe post object oriented, because it's reintroducing a clean separation between data and program which is the complete opposite of the basic object oriented principle which says that objects encapsulate both data and treatments.

In the XML world, XML documents live their own lives independently of programs: they can be edited, read, displayed and transformed using generic tools independent of any application and it's vitally important that they can also be validated independently of any application.

It's not only important that XML documents can be validated independently of any application because XML documents themselves have become independent of applications but it's also difficult because of the extensibility of XML. The diversity of the XML vocabulary is virtually infinite. It's one of the main assets of XML and that's something we certainly do not want to limit with the tools used to validate XML documents. It's also difficult because of the diversity of what we can call "validation".

There is more than one aspect in validation

Validation can be about checking the structure of XML documents, it can be about checking the content of each text node and attribute independently of each other (that's often called datatyping), it can be about checking constraints between nodes, it can be about checking constraints between nodes and external information such as lookup tables or links, it can be about checking "business rules" and it can be anything else such as spell checking.

Validation is key to improve the level of quality of our XML based information systems and that's something which appears to be most needed. I have recently followed two presentations about two independent projects in very different domains and both came out with an alarming ratio of one out of ten real world XML document containing errors. With such a high proportion, validation is not

only useful but indispensable! Can you imagine a fund transfer system where 10% of the transactions would contain errors?

Relax NG is the best tool to validate the structure of XML documents

Relax NG won't solve this issue all by itself: it's not designed to solve the whole issue. Relax NG is designed to be the best tool available to solve two pieces of the problem and validate the structure of XML document by itself and provide a plug to datatype libraries which validate the content of text nodes and attributes. It's also designed to be used as a part of the ISO DSDL framework which has the vocation to deal with the issue of validation at large (see Appendix A: DSDL for more information about DSDL).

This strong focus is what makes Relax NG so different from its main rival, W3C XML Schema. One of the reasons of the complexity of W3C XML Schema is that it includes many features which have been kept out of the perimeter of Relax NG. W3C XML Schema doesn't only care about validating the structure of XML documents, but also to validate the content of text nodes and attributes and the integrity between keys and references. Furthermore, W3C XML Schema doesn't only care about validation but also attempts to be a modeling language for XML document which can classify the elements and attributes of XML documents, identify their semantic and may be used to perform automatic binding between XML documents and objects. All these goals may be fair, but there is always a risk to loose focus when trying to solve too many different problems with a single technology.

On the contrary, focus has always been kept on XML structure validation and no compromise has been made to any other feature during the development of the Relax NG specification and the result is that Relax NG can pretend to be the logical successor of XML DTDs and the best tool available to validate the structure of XML documents. Relax NG is powerful and its expressive power is such that virtually any XML vocabulary may be described with Relax NG which isn't the case of W3C XML Schema not even of DTDs. Maybe still more important for people having to write schemas, Relax NG is also very simple: because it doesn't try to model XML documents, validate too many things and brew coffee, the syntax is intuitive, it has been kept simple and isn't cluttered by limitations complex to learn and difficult to remember.

Unexpected uses of Relax NG

This focus doesn't mean that Relax NG is a niche language meant to stay limited to its original goal. Relax NG may well follow the path of XSLT (also developed by James Clark): XSLT which development has been focused on document transformation has become the Swiss army knife of XML developers and its unpredictable success shows that it's being used for much more than what had been originally forcasted.

The same will likely happen with Relax NG and I can give a couple of examples.

The other day, I had to write a converter for a flat non XML format into XML. The structure of the resulting document was described by a non Relax NG schema and after various attempts to find hacks to map the 400 different information items of this flat structure into as many elements and attributes, I have found that the easiest way was through a Relax NG Schema. I have transformed the schema of the XML vocabulary into a Relax NG schema as simple as possible. A trivial program (written in Python in that case but any other language could have been used) can just walk through this structure while parsing the flat document and dispatch the information items where they belong. This was made easy by the uncluttered simplicity of the syntax of Relax NG and it would have taken me much more time with any other schema language.

The second example is taken from Relax NG itself. As we will discover in "Chapter 4: Non XML Syntax", a non XML compact syntax is available for Relax NG and in its specification this syntax is described by an EBNF grammar. Knowing James Clark, I was sure he hadn't written it by hand but

had generated it from XML and when I have written the reference guide for this syntax ("Chapter 18: Non XML Syntax Reference"), I have asked him to send me the source of this grammar as XML. I was expecting a format such as the DocBook EBNF module and guess what I received? A Relax NG schema of course! The syntax of Relax NG is flexible enough to describe the productions of an EBNF grammar and Chapter 18: Non XML Syntax Reference is generated from this schema. It's only a summary and the semantics and restrictions of Relax NG are not fully respected, but Relax NG is still a nice way to describe this EBNF.

Relax NG as a pivot format

These two examples are a little bit extreme, and more to the point Relax NG appears to be the perfect pivot format for XML schema related task. The first time I started to think of Relax NG as a pivot format was attending the presentation about the Sun multi-schema validator (MSV) by Kohsuke Kawaguchi at XML 2001. During his talk, Kohsuke Kawaguchi explained that the grammar based different schema languages supported by MSV (DTDs, Relax NG, Relax and W3C XML Schema) were translated into a common data model by the validator and that the validation algorithm relied on this data model. After his talk, I asked him what was this common data model and he answered that it was Relax NG. This is the proof that the expressive power of Relax NG is such that 99% of the constraints which can be described with other schema languages can be described with Relax NG.

This advantage could be a major drawback: if the expressive power of Relax NG is so much important than the expressive power of other languages, that could mean that a schema written with Relax NG would be impossible to translate into other languages. This issue happens to be more theoretical than practical and even if when you know both Relax NG and W3C XML Schema you can imagine Relax NG that cannot be translated into W3C XML Schema, this doesn't happen often in real life schemas and when this happen, you can always ponder the need to express such a schema against your need to be able to publish a W3C XML Schema schema. And the reason why I can be fairly confident when I say that most Relax NG schemas can be translated into other schema languages is that I have seen it! James Clark has developed Trang, a magic tool that takes a Relax NG schema and converts it into W3C XML Schema or a DTD.

Simpler to write by hand, Relax NG is also simpler to generate and that's something important too since a growing number of applications, especially but not only those having huge schemas with hundreds of elements and attributes tend to generate their schemas from logical models with a higher level of abstraction rather than create them from scratch by hand. Whether you are using UML as your design tool, a simple spreadsheet like the OASIS UBL project or sample documents like my examplotron, it is easier to derive a Relax NG schema from this model than any other schema language.

Why should anyone use any other schema language?

Having converters to and from other schema language, easier to write, easier to generate, easier to use by applications, why would anyone even consider using any other schema language as its main pivot schema language? As far as validation only is concerned, I can really not see any good reason.

The only area where Relax NG is still a little bit behind is for data type assignment and data binding. Datatype assignment appears to be getting increasingly important for a whole set of applications; for instance, many new features of the XPath 2.0, XSLT 2.0 and XQuery 1.0 family of future W3C recommendations. Because data type assignment was out of the scope of Relax NG itself, Relax NG is very permissive about "non deterministic" schemas which could lead to non deterministic type assignment and this is something to keep in mind when writing Relax NG schemas which will be transformed into W3C XML Schema schemas. I will present the latest updates on this subject in "Chapter 16: Determinism and Datatype Assignment".

Chapter 3. Chapter 2: Simple Is Beautiful

The exploration of a foreign country or language deserves some preliminary explanations on its particularities to save lot of time and trouble and avoid lots of misunderstandings. Relax NG doesn't escape this rule so we'd better try to highlight its profound differences with other XML schema languages.

XML Infoset

One of the few things that all the XML schema languages have in common is that they define constraints to apply to a logical view on the XML documents (called the XML Infoset) rather than to the document which you can read as a text file. This is how they differentiate from other techniques such as regular expressions which you might used at the level of XML documents considered as text files.

This may be a surprised if you're not familiar with the concept, but in XML, what you see is not what you get and when you write "<book id='b0836217462' available='true'/>" XML applications do not see the string "<book id='b0836217462' available='true'/>". Most of them do not even see a "tag" named `book` but they all see an `element` named book with its two attributes `id` and `available` and the vast majority of them do not care about the way you've written this element in a text document nor even if you have ever written it in a ext document: what they really care about is the element `book` and its two attributes.

The set of the information considered significant in a XML document -such as our element `book` and its two attributes- is what is called the XML Infoset and it has been published as a W3C Recommendation. This Infoset defines an abstract model of XML documents which has a hierarchical structure and is described in terms generic and neutral enough to be acceptable for specifications with different backgrounds and goals such as XPath or the DOM.

The different schema languages work at the level of the XML Infoset and their main goal is to let us define constraints on a subset of the XML Infoset. Because they work at that level, they can't be used to express constraints on things which do not belong to the XML Infoset, such as the order of the attributes or the number of spaces between them. In addition, most of them won't let you define constraints on XML comments or Processing Instructions nor on the use of entities.

Different types of schema languages

That being said, the different XML schema languages have chosen different ways of defining those constraints:

- The constraints may be expressed as rules, like it's the case with Schematron: you give sets of rules such as "the element named book must have an attribute named id and this attribute must match this and this rule, ...".
- They may be expressed as a thorough description of each element and attribute like DTDs and W3C XML Schema and say: "it's an element, named book and it has two attributes named id and available which look like this and this".
- They may be expressed as "patterns" similar in their principle to regular expressions adapted to match XML infosets rather than text documents and we will cover this third way of defining constraints in detail over this book since it's the way that has been chosen by Relax NG.

The first XML schema language ever used was the DTD. Of course DTDs cover more than schema features and include the definition of internal and external entities, but their schema features focus on describing elements: each element must be described and for each element, a list of nodes must be defined listing whether text nodes are allowed, the list of allowed child elements and the list of

its attributes. Pieces of content model may be defined, by using special entity types (the parameter entities) which work like a kind of macro-processing.

W3C XML Schema has extended this principle and defines several kind of "components" allowing to manipulate not only elements, but also attributes, datatypes which are containers describing the content of elements or attributes and even groups of elements and groups of attributes. The approach is still very focused on elements and attributes and which are clearly differentiated.

Relax NG, on the contrary is based the generic concept of `pattern` which is more or less symmetrical to the XPath concept of "node set": in first approximation, a pattern could be defined as the description of a set of valid nodesets.

The difference may be difficult to perceive, but when we define an element with a DTD or W3C XML Schema, we try to give a description of the element itself while when we define the same element with Relax NG, we define a pattern which will be checked against elements like a regular expression to see if they match. The difference is dim, but the later option gives us a much wider flexibility to write, maintain and combine schemas.

A simple example:

Let's take a first look at the example which we will using throughout this book and look at this `book` element with its two attributes and four different sub-elements:

Figure 3.1. full-pattern

```
<book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
 - <author id="CMS"></author>
 - <character id="PP"></character>
 - <character id="Snoopy"></character>
 - <character id="Schroeder"></character>
 - <character id="Lucy"></character>
</book>
```

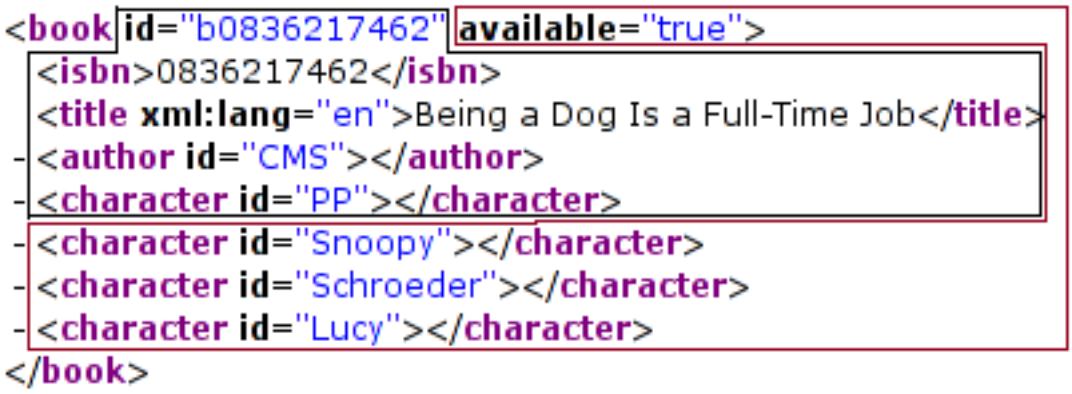
With a DTD and to a lesser attempt with W3C XML Schema, we are pretty much stuck to define lists of attributes and elements and cannot mix and combine them together. W3C XML Schema has introduced the concept of `type` which is an abstract object that has no match in the XML documents and is the description of the content of an element or an attribute, but still, types can't be freely combined together. This means that we can split the description of this elements into blocks such as:

Figure 3.2. xsd-full-pattern

```
<book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
 - <author id="CMS"></author>
 - <character id="PP"></character>
 - <character id="Snoopy"></character>
 - <character id="Schroeder"></character>
 - <character id="Lucy"></character>
</book>
```

Relax NG patterns on the contrary can freely mix different type of nodes (elements, text and attributes) and if we have a need for this, Relax NG is flexible enough to split the definition of the book element into a first pattern composed of the `id` attribute, the `isbn`, `title`, `author` and first `character` element and a second one composed of the `available` attribute and the other `character` elements:

Figure 3.3. rng-full-pattern



This flexibility is not only useful for combining complex patterns but also a source of simplicity for the designers of Relax NG schemas who do not need to learn a long list of limitations which must be checked when they write and combine their schemas.

This generic concept of patterns is powerful enough to replace the specialized containers of the DTD and W3C XML Schema. Relax NG has no need (and no notion) of reusable element, attribute or type definition: to reuse an element or attribute, this element or attribute is embedded in a pattern where it will be left alone; to reuse a type, a pattern is created to contain the content definition. These patterns are the reusable building blocks of Relax NG. They can be named, reused and even redefined at will, combined through operators to group them or provide alternatives between them.

The benefit of having non specialized patterns is an increased flexibility: this is well known in the construction industry where reusing a small number of generic parts provides more flexibility and a higher number of possible combinations than using more specific pieces and this is true for XML schema languages too...

A strong mathematical background

This notion of patterns is both new an ancient. New in the way it has been applied to XML in Relax first and now in Relax NG and ancient since it is the adaptation to XML of techniques and theories developed for Regular Expressions in the 60s and the name "Relax" stands for REgular LAnguage for XML. It relies on a strong mathematical theory and on works done by Murata Makoto to adapt the mathematical concept of "hedges" to XML.

When Murata Makoto has kindly pointed me to his work to answer my first questions, I have been horrified to see that all the maths I had learned at school seemed to have left me and that I couldn't understand the first word of it and I can insure you that you won't need to understand the maths behind Relax NG to use it and shouldn't worry about this. On the contrary, it's very comforting to know that the schema language you are using has such a background and it's a guarantee that its design is flawless.

The Relax NG specification is based on this mathematical background and the Relax NG patterns are defined as logical operations performed on sets of XML structures. This gives to the specification a formalism which removes any possibility of ambiguity for its interpretation and this is most important for insuring the interoperability of the different implementations of Relax NG.

And a strong experimental basis

This strong mathematical background doesn't mean that everything need to be reinvented for Relax NG implementers. On the contrary, the so-called "derivative algorithm" used by James Clark in his processor named Jing has been inspired by works done in 64 on the "derivation" of regular expressions and simply recursively removes from the patterns the nodes found in the instance documents: the document is valid if the patterns left after the last node are all optional.

Murata Makoto on his side, has adapted the ancient and well know algorithm of finite state machines to cope with the level of non determinism accepted by Relax NG and has developed a Relax NG validator lightweight enough to be used in a mobile phone.

Beside the fact that it is implementable with well known and documented algorithm, developers of Relax NG processors also appreciate the the simplicity of this underlying model and this should also guarantee a strong interoperability between implementation which is unfortunately not the case with more complex schema languages.

Patterns and only patterns

In the history of science, strong theories based on simple and basic particles have proven to have an almost infinite potential and can be used as a foundation for the most complex applications. There is no doubt that Relax NG is, in its domain, one of these applications both easy to explain, easy to implement and generic and flexible enough to meet the most stringent requirements.

We will present the Relax NG patterns throughout this book but can't leave this chapter before we give a list of some of them.

The three basic patterns match the three types of XML nodes in the scope of Relax NG (which do not pay attention to XML Processing Instructions and comments):

- Text nodes --which can be specialized into `data` which can carry "datatypes" and split into list items.
- Elements
- Attributes.

These patterns can be combined into ordered or non ordered groups and into choices defining alternatives between several patterns. Their cardinality, i.e. the number of time that can appear in instance documents, can also be controlled using cardinality patterns and, finally, a whole set of features are provided to build reusable libraries of patterns. Similar to patterns, name classes define set of element and attribute names that can be used to open a schema and control where elements and attributes with unknown names may be included in the instance documents.

Some of these features have been defined to facilitate the work of writing Relax NG schemas and are not basic "atomic" patterns. To avoid to overloading and omplicating the basic model with these "cosmetic" features, the Relax NG specification describes a "simplification algorithm" applied internally by Relax NG processors to transform a full schema into a simple form with fewer and simpler patterns. This algorithm is presented in "Chapter 15: Simplification And Restrictions".

Chapter 4. Chapter 3: First Schema

Getting started

Throughout the book, we will be using variations of the same document describing a library:

```
<?xml version="1.0"?>
<library>
 <book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
  <author id="CMS">
   <name>Charles M Schulz</name>
   <born>1922-11-26</born>
   <died>2000-02-12</died>
  </author>
  <character id="PP">
   <name>Peppermint Patty</name>
   <born>1966-08-22</born>
   <qualification>bold, brash and tomboyish</qualification>
   </character>
  <character id="Snoopy">
   <name>Snoopy</name>
   <born>1950-10-04</born>
   <qualification>extroverted beagle</qualification>
  </character>
  <character id="Schroeder">
   <name>Schroeder</name>
   <born>1951-05-30</born>
   <qualification>brought classical music to the Peanuts strip</qualification>
  </character>zeroOrMore
  <character id="Lucy">
   <name>Lucy</name>
   <born>1952-03-03</born>
   <qualification>bossy, crabby and selfish</qualification>
  </character>
 </book>
</library>
```

Our first patterns

In plain English, we could describe this document as being:

- a `library` element composed of one of more
- `book` elements having
- `id` and `available` attributes and
- an `isbn` element composed of text
- a `title` element with a xml:lang attribute and a text node
- one or more `author` elements with

- an `id` attribute
- a `name` element
- an optional `born` element
- an optional `died` element
- zero or more `character` elements with
- an `id` attribute
- a `name` element
- an optional `born` element
- a `qualification` element.

The good news -and what makes Relax NG so easy to learn- is that in its simplest flavor,it's pretty much a XML formalization of this statement with simple matching rules:

- "library element" translates into <element name=`library`>...</element>
- "id attribute" translates into "<attribute name="id"/>
- "one or more" becomes: <oneOrMore>...</oneOrMore>
- "zero or more" becomes: <zeroOrMore>...</zeroOrMore>
- `text` is: <text/>
- `optional` is: <optional>...</optional>

We've seen in "Chapter 2: Simple Is Beautiful" that almost everything is a pattern for Relax NG and each of these Relax NG elements are patterns. Let's now spend some time to introduce each of them.

<text/>

This pattern is the simplest we can think of and simply matches a text node. More exactly, it matches zero or text nodes and we'll see in "Chapter 6: More Patterns" that the `text` it makes a difference in the context of mixed content models (i.e. to define elements which may have both sub elements and text nodes) but until then we can think of it as matching a text node.

By extension the `text` pattern also matches any attribute value even if attribute values are not considered as nodes by the XML infoset.

The XML expression for `text` patterns is just:

```
<text/>
```

<attribute/>

As expected, the `attribute` pattern matches attributes. The name of these attributes are defined in the `name` attribute of the `attribute` pattern and the content of these attributes is defined as a child element of the `attribute` pattern (don't worry, that's easier to write and read than to explain!).

To define the `id` attribute, we could thus write:

```
<attribute name="id">
 <text/>
</attribute>
```

This would read as: "an attribute named id with a text value". Since any attribute can have a value and that the `text` pattern isn't restrictive in this case, it has been made optional and this definition is strictly equivalent to the following one:

```
 <attribute name="id"/>
```

The last thing we need to mention about this pattern is that even if in most of the cases the name of the attributes are defined by the `name` attribute or the `attribute` pattern, it is also possible to define sets of possible names for an attribute. This feature will be explained in detail in "Chapter 12: Writing Extensible Schemas".

<element/>

Just as the `attribute` pattern matches attributes, the `element` pattern matches elements. To define the `name` element, we will write:

```
<element name="name">
 <text/>
</element>
```

As mentioned for the `attribute` pattern, it is also possible to replace the `name` attribute of the `element` patter by a set of names as explained in detail in "Chapter 12: Writing Extensible Schemas".

Not all elements accept text nodes and for that reason, the `text` pattern isn't implicit within elements. In fact there is no implicit content for elements and the content of each element must be explicitly described even when the element is always empty.

The fact that a `text` pattern matches zero or more text nodes means that this definition of the `name` element would match empty elements such as:

```
<name/>
```

as well as more traditional names such as:

```
<name>Charles M Schulz</name>
```

We will see in "Chapter 7: Constraining Text Values" how we can add additional restrictions to text nodes to avoid empty elements and in "Chapter 8: Datatype Libraries" how to use the datatypes from W3C XML Schema to add more specific restrictions such as being a valid date.

Attributes can be added within elements and to define the `title` element we will write:

```
<element name="title">
 <attribute name="xml:lang"/>
 <text/>
</element>
```

We will see the support of namespaces in "Chapter 11: Namespaces" but we can already note how straightforward it is. Here we had to define an attribute (xml:lang) from the XML namespace and we've just added the description of this attribute straight away in out schema. In the case of the XML namespace which is considered as predeclared in any XML document conform to the Namespaces in XML recommendation and implicitly assign the "xml:" prefix to "http://www.w3.org/XML/1998/ namespace", we do not need to declare the namespace. In the general case, we would have needed to declare the namespaces using mechanisms described in "Chapter 11: Namespaces".

Note that Relax NG is clever enough to know that attributes are always located in the start tag of XML elements and that their order is not considered as significant. This means that the `attribute` pattern may be located anywhere in the definition of elements and that it would have made no difference if we had written:

```
<element name="title">
 <text/>
 <attribute name="xml:lang"/>
</element>
```

In addition to text nodes and attributes, elements can also include sub elements and we could define the `author` element as:

```
<element name="author">
 <attribute name="id"/>
 <element name="name">
  <text/>
 </element>
 <element name="born">
  <text/>
 </element>
 <element name="died">
  <text/>
 </element>
</element>
```

That's not exactly what we want, though since the `born` and `died` elements should have been made optional. To do so, we need to introduce a new pattern:

<optional/>

The `optional` pattern just makes its content optional. To specify that the `born` and `died` elements are optional, we will write:

```
<optional>
 <element name="born">
  <text/>
 </element>
</optional>
```

```
<optional>
 <element name="died">
  <text/>
 </element>
</optional>
```

Note that this is different from

```
<optional>
 <element name="born">
  <text/>
 </element>
 <element name="died">
  <text/>
 </element>
</optional>
```

And also different from

```
<optional>
 <element name="born">
  <text/>
 </element>
 <optional>
  <element name="died">
   <text/>
  </element>
 </optional>
</optional>
```

In the first case, each element is embedded in its own `optional` pattern. The two elements are thus independently optional and I can include both of them, none of them or one of them in valid instance documents. In the second case, both elements are embedded in the same `optional` pattern and we can only include either none of them or both of them in instance documents. In the third case, a first optional pattern includes the `born` element and an optional `died` element meaning that we can find either both of them or none of them in an instance document or the `born` element only but that we forbid the `died` element if the `born` element isn't there.

None of the combinations is "right" or "wrong", they are just different pattern combinations allowing different element combinations in the instance documents and corresponding to different use cases. What's nice with Relax NG is that there are so few restrictions that almost any combination to which you can think of is allowed. To be honest I must admit that there are some few restrictions and they will be covered in "Chapter 15: Simplification And Restrictions".

<oneOrMore/>

The `oneOrMore` pattern specifies that its content may appear one or more time. The use case for this pattern in our example is to define that a book must have one or mote authors:

```
<oneOrMore>
 <element name="author">
  <attribute name="id"/>
```

```
      <element name="name">
       <text/>
      </element>
      <element name="born">
       <text/>
      </element>
      <optional>
       <element name="died">
        <text/>
       </element>
      </optional>
     </element>
    </oneOrMore>
```

<zeroOrMore/>

The last pattern needed in our example is `zeroOrMore` and you'll have guessed that it tells that its content may appear zero or more time like for our `character` elements:

```
    <zeroOrMore>
     <element name="character">
      <attribute name="id"/>
      <element name="name">
       <text/>
      </element>
      <optional>
       <element name="born">
        <text/>
       </element>
      </optional>
      <element name="qualification">
       <text/>
      </element>
     </element>
    </zeroOrMore>
```

Complete schema

We have now in hand all the patterns needed to write a full schema expressing all we've said for this example:

```
<?xml version = '1.0' encoding = 'utf-8' ?>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library">
 <oneOrMore>
  <element name="book">
   <attribute name="id"/>
   <attribute name="available"/>
   <element name="isbn">
    <text/>
   </element>
   <element name="title">
    <attribute name="xml:lang"/>
    <text/>
```

```
    </element>
    <oneOrMore>
     <element name="author">
      <attribute name="id"/>
      <element name="name">
       <text/>
      </element>
      <optional>
       <element name="born">
        <text/>
       </element>
      </optional>
      <optional>
       <element name="died">
        <text/>
       </element>
      </optional>
     </element>
    </oneOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id"/>
      <element name="name">
       <text/>
      </element>
      <optional>
       <element name="born">
        <text/>
       </element>
      </optional>
      <element name="qualification">
       <text/>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

Constraining number of occurrences

Those of you familiar with W3C XML Schema will probably have noted that the control over the number of occurrences is specified ala DTD and doesn't have the fine granularity of W3C XML Schema's minOccurs and maxOccurs. Relax NG has been designed in this way because these four cases (exactly once which is the default, optional, zero or more and one or more) are the most common and also because if applications need a finer granularity, they can create it using these four basic occurrence constraints. If for instance, we needed to define that each book's description should include between two and six characters, we could write it as two mandatory characters followed by four optional ones and write:

```
    <!-- 1 -->
    <element name="character">
     <attribute name="id"/>
     <element name="name">
      <text/>
     </element>
```

```
 <optional>
  <element name="born">
   <text/>
  </element>
 </optional>
 <element name="qualification">
  <text/>
 </element>
</element>
<!-- 2 -->
<element name="character">
  .../...
</element>
<!-- 3 -->
<optional>
 <element name="character">
 .../...
  </element>
 </element>
</optional>
<!-- 4 -->
<optional>
 <element name="character">
 .../...
 </element>
</optional>
<!-- 5 -->
<optional>
 <element name="character">
 .../...
 </element>
</optional>
<!-- 6 -->
<optional>
 <element name="character">
 .../...
 </element>
</optional>
```

This is certainly verbose but we will see how we can define and reuse patterns to reduce the verbosity.

Russian doll schemas

Let's look side by side at the schema and the instance document:

Figure 4.1. russian-doll

We see that even though information have been added in the schema to describe the content of the text nodes and the number of occurrences, the schema keeps the same hierarchical structure than the instance document.

This type of schema in which the different definitions are embedded in each other (the definition of the `library` element physically contains the definition of the "author element" which physically contains the definition of the `name` element) are often called "Russian doll schemas". We will see in " Chapter 5: Flattening Our First Schema" how Russian doll schemas may be broken into independent patterns combined together to reproduce the structure of the instance document. Before this, we'll have a look in nxet chapter, "Chapter 4: Non XML Syntax" to an alternative, equivalent, compact and non XML syntax for Relax NG.

Chapter 5. Chapter 4: Non XML Syntax

Getting started

Although the schema shown in the previous chapter was simple, its XML representation was rather verbose. This is not surprising nor even uncommon for XML vocabularies. In fact it even conforms quite well to the basic principles of XML, a Recommendation whose design goals state that "XML documents should be human-legible and reasonably clear" but also "Terseness in XML markup is of minimal importance". Our schema is a good example of a document which is "human-legible and reasonably clear" and... verbose!

The principal goal of the XML syntax is to be a serialization of Relax NG schemas which is easily processable by computers using a standard XML toolkit and James Clark has introduced a second syntax, strictly equivalent to the XML syntax, which is both more concise and easier for humans to read and write. Relax NG processors are free to support this "compact" syntax or not; if a Relax NG processor doesn't users can use existing translators to translate the XML syntax to and from the compact syntax. Since these two forms are strictly equivalent, there is no loss of information during translation and even comments and the annotations which will be presented in "Chapter 13: Annotating Schemas" are preserved in the process. This is, of course, not true of the "syntactical details" of XML such as entity expansion or Processing Instructions which are lost when the XML syntax is translated into compact but this is a limitation of the XML processing architecture rather than a limitation of Relax NG itself.

The compact syntax has been published as official OASIS Relax NG committee specification but not yes submitted as to the ISO.

We'll see that this syntax is a strange mix between concepts borrowed from the definition of structures in programming languages, some notations from XML DTDs and the Relax NG patterns which, of course, we'll find there: element and attribute patterns look like Java declarations with their curly brackets preceded by a reserved word (`element` or `attribute`) which is their Relax NG pattern name. On the other hand, optional, one or more and zero or more are represented by DTD qualifier suffixes (`?` for optional, "+" for one or more and `*` for zero or more). But don't worry, the result is incredibly good and intuitive and you'll find this syntax both simple and familiar before the end of this book!

Our first compact patterns

Let's see how our first patterns translate into the compact syntax.

text

This was the simplest pattern in the XML syntax and this is still the simplest with the compact syntax! The `text` pattern is just... `text`:

```
text
```

In this definition, the word `text` identifies the text pattern.

And of course, since both syntaxes are equivalent, all we've said about "<text/>" is true for `text`.

attribute

For the compact syntax, the `attribute` pattern borrows Java's curly brackets:

```
attribute id { text }
```

In this definition, the first word `attribute` identifies the `attribute` pattern, the second one `available` is the name of the attribute (we will see in "Chapter 12: Writing Extensible Schemas" how this could be extended to define sets of names) and the curly brackets "{...}" delimit the definition of the content of the attribute.

Since empty curly brackets "{}" would have looked weird and would have kind of imply `empty` attributes rather than attributes containing a text value, the convention of the XML syntax making a `text` pattern an implicit content for attributes has not been carried on to the compact syntax. The definition of the content of attributes is thus mandatory and must be explicitly made when you're using the compact syntax. In other word,

```
<attribute name="id"/>
```

translates into:

```
attribute id { text }
```

and

```
attribute id { }
```

is just a syntax error for the compact syntax.

The compact syntax is position sensitive and words such as `text` and `attribute` are reserved words only when they appear in first position. In practice, this is very convenient when we need to define attributes (or elements) which name are reserved words. For instance, I could define attributes named `text` or even `attribute` without any precaution such as:

```
attribute text { text }
attribute attribute { text }
```

Because the compact syntax is position sensitive, it wouldn't be confused by the fact that I have used reserved words as attribute names. This would be the same for the `element` pattern which we'll see in the next section.

Element

You'll have already guessed that the definition of the name element would be:

```
element name { text }
```

To add an attribute to our elements we need a delimiter between the different content within a single element. We'll see more delimiters and their meaning in "Chapter 6: More Patterns" and for the moment we'll use a comma (",") as delimiter between content and this will have the same effect than what we've seen with the XML syntax:

```
element title {
  attribute xml:lang { text },
  text
}
```

White spaces (i.e. spaces, tabulations, line feeds and carriage return) are not significant for the compact syntax and this could have been written:

```
element title {attribute xml:lang { text }, text}
```

As with programming languages, many people tend to prefer to split definitions so that there is only one per line and tend to consider the first form as more readable but a Relax NG processor won't get confused and will treat both as equivalent.

We can add more content and write our author element as:

```
element author {
  attribute id { text },
  element name { text },
  element born { text },
  element died { text }
}
```

Again, all what we've said about the properties of the `element` pattern in the XML syntax is true for the compact syntax: these are just two equivalent syntaxes for the same pattern.

To meet our requirements, we need to introduce the `optional` pattern

optional

Here is where DTDs have been asked to contribute! The optional pattern is formalized as a trailing `?` added after a definition. To define that the attribute `id` is optional we would write:

```
attribute id { text }?
```

and to define that the `born` element is optional we write:

```
element born { text }?
```

Note that this qualifier (?) must be added after the definition of the pattern but before the delimiter. The definition of our author element would thus be:

```
element author {
  attribute id { text },
  element name { text },
  element born { text }?,
  element died { text }?
```

```
    }
```

In "Chapter 3: First Schema", we had mentioned that other combinations could be described using the `optional` pattern as a container. In the compact syntax, the `optional` pattern is a qualifier and we need a container to define the same combinations. This container are the parenthesis "(...)". They are neutral in that their effect depends on the delimiter used within the parenthesis and on the optional qualifier following them. The definition of our author could be written:

```
    element author {(
      attribute id { text },
      element name { text },
      element born { text }?,
      element died { text }?
    )}
```

or

```
    element author {
      (attribute id { text }),
      (element name { text }),
      (element born { text })?,
      (element died { text }?)
    }
```

```
 without changing its meaning at all. They are more useful (and actually even r
```

```
 <optional>
  <element name="born">
   <text/>
  </element>
  <element name="died">
   <text/>
  </element>
 </optional>
```

would translate as:

```
      (element born { text }, element died { text })?
```

While

```
 <optional>
  <element name="born">
   <text/>
  </element>
```

```
 <optional>
  <element name="died">
   <text/>
  </element>
 </optional>
</optional>
```

Would translate as:

```
      (element born { text }, element died { text }? )?
```

oneOrMore

The oneOrMore pattern is also a qualifier and, in the DTD tradition it is a "+":

```
    element author {
      attribute id { text },
      element name { text },
      element born { text }?,
      element died { text }?
    }+
```

zeroOrMore

Last but not least, the `zeroOrMore` pattern is the `*` qualifier:

```
    element character {
      attribute id { text },
      element name { text },
      element born { text }?,
      element qualification { text }
    }*
```

Full schema

```
element library {
  element book {
    attribute id { text },
    attribute available { text },
    element isbn { text },
    element title {
      attribute xml:lang { text },
      text
    },
    element author {
      attribute id { text },
      element name { text },
      element born { text }?,
```

```
      element died { text }?
    }+,
    element character {
      attribute id { text },
      element name { text },
      element born { text }?,
      element qualification { text }
    }*
  }+
}
```

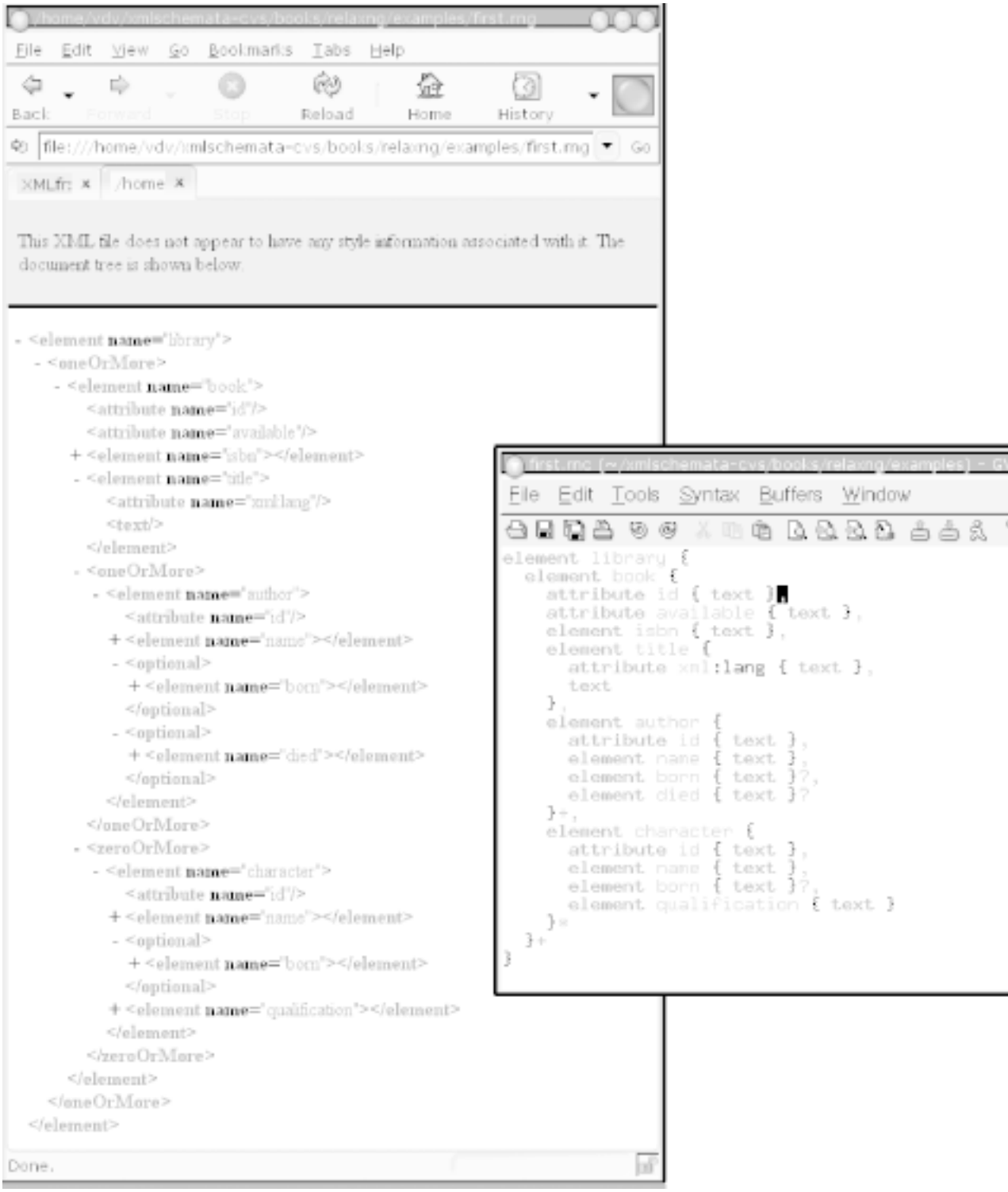
In the following chapters, for each example we will give both the XML and the compact syntax and we will have plenty of opportunities to get familiarized with both.

Note that there should be no confusion between the simple form of a Relax NG schema described in "Chapter 15: Simplification And Restrictions" and the compact syntax. These two notions are orthogonal and work at different level: the simple form is the result of simplifications internally done by Relax NG processors on the data model resulting from the parsing of a Relax NG "full" document; the compact syntax is another way to represent or serialize a Relax NG "full" document. The data models resulting from the parsing of a Relax NG full schema are thus the same if the schema has been written using the XML or the compact syntax and it will simplified into the same simple schema.

XML or compact?

Let's take a look at both syntaxes side by side:

Figure 5.1. xml-comp



There are two things which we immediately notice:

- The compact syntax is much more... compact.
- The XML syntax is... XML and plays fine with generic XML tools (here a web browser) while the compact syntax isn't XML and must be used with other tools (here the text editor VIM with a plug-in to highlight Relax NG's compact syntax).

These two findings summarize pretty well why they are both needed. The compact syntax is nice to work with and you'll probably find it soon more pleasant to edit your schemas and document your vocabularies. On the other hand, the XML syntax is wonderful if you want to either generate Relax NG schemas as we will see in "Chapter 14: Generating Relax NG schemas" or generate anything out of your Relax NG schemas using XML tools which will be covered in "Chapter 13: Annotating Schemas". And the fact that they can be translated without information loss is a guarantee that we can use both.

Chapter 6. Chapter 5: Flattening Our First Schema

Why do we need flat schemas?

If we look at the structure of our first schema, we see that it follows the structure of the instance document. The effort involved in writing this first schema has been pretty much limited to inserting `element`, `attribute` and `text` elements in the schema each time we've seen an element, attribute or text node in the instance document! This flavor of our schema can almost be seen as some kind of XML serialization of the XML infoset (i.e. of the information available in the document) and could be easily automated. Actually, it has been automated and this is the principle behind Examplotron which will be described in "Chapter 14: Generating Relax NG schemas".

However, there are a couple of downsides for modeling documents with this style of "Russian doll" schemas: first they are not modular and get difficult to read and maintain when documents are complex and second they cannot represent recursive models.

The lack of modularity can be seen even for a document as simple as our first example: we have a `name` element which is used with the same model both within the `character` and the `author` elements. In our first schema we need to give the definition of what a `name` is in both contexts:

Figure 6.1. 2names

One might think that this is not a big deal, but that's not completely true. The additional verbosity here is low because the definition of the `name` element is simple, but the principle would have been the same if the definition had been longer. As with any redundancy this makes the maintenance of the schema more error prone: if I need to update the definition of the name element, I need to update it twice. The rules of common sense used in any programming language do apply to XML schema languages as well!

An other rule which can be borrowed from programming languages is about modeling recursive models, i.e. models such as XHTML in which the `div` elements may be embedded within other `div` elements without any restriction in the number of levels. To be able to model such recursive models, it is clear that I can't just copy the definition of the `div` element again and again and that I need a way to define and reference the content model of the `div` element recursively. In the course of this chapter we will give different examples covering both modularity and recursive models.

Defining named patterns

For Relax NG, the answer to these issues is of course through patterns (haven't we learned that Relax NG is about patterns and patterns only?) which can hold a name and be referenced through this name.

In the XML syntax, the definition of named patterns is done through `define` elements. To define a named pattern containing our `name` element we would thus write:

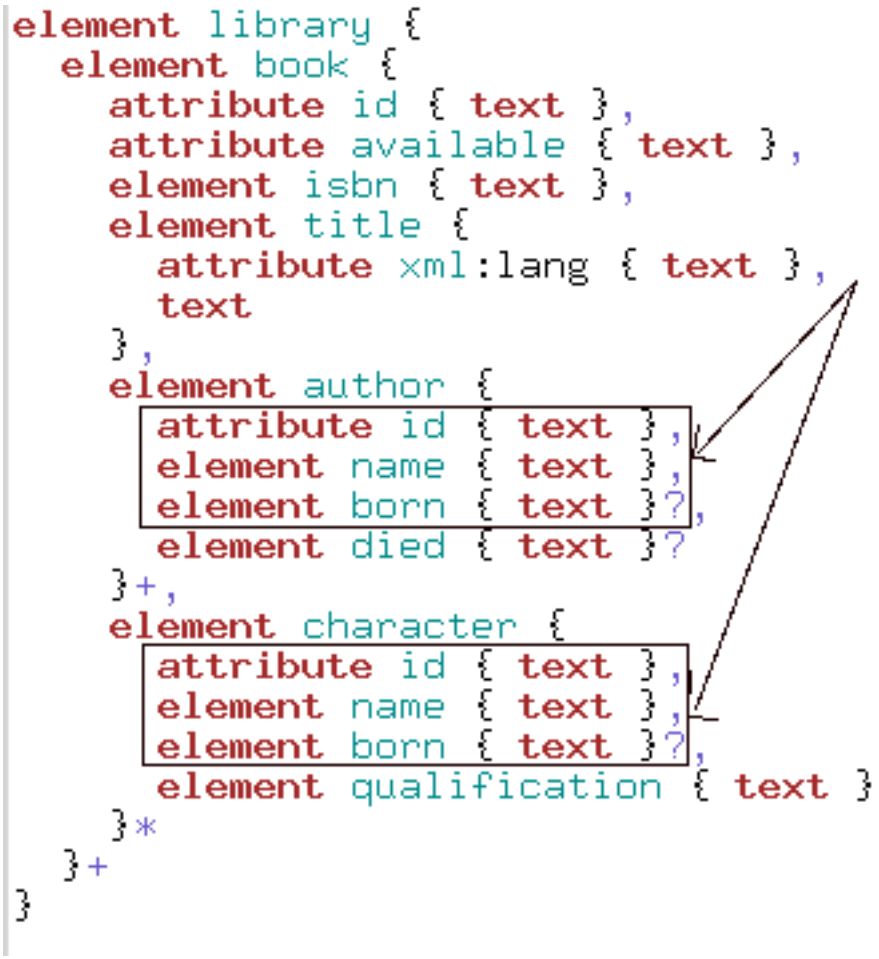
```
<define name="name-element">
 <element name="name">
  <text/>
 </element>
</define>
```

The compact syntax uses a construction similar to a programming language format and the same definition would be written as:

```
name-element = element name {text}
```

We are not limited to embedding a single element (or attribute definition in a named pattern and we could note that a common group composed of an `id` attribute, a `name` element and an optional `born` element are present in the same order and with the same definition in both the `author` and the "character element":

Figure 6.2. named2




```
<define name="common-content">
 <attribute name="id"/>
 <element name="name">
  <text/>
 </element>
 <optional>
  <element name="born">
   <text/>
  </element>
  </optional>
</define>
```

or:

```
common-content =
  attribute id { text },
  element name { text },
  element born { text }?
```

Referencing named patterns

Defining a named pattern has been easy but referencing it is still simpler!

Using the XML syntax, references are done using a `ref` element, for instance to define the `author` element using a reference to `name-element`:

```
  <element name="author">
   <attribute name="id"/>
   <ref name="name-element"/>
   <optional>
    <element name="born">
     <text/>
    </element>
   </optional>
   <optional>
    <element name="died">
     <text/>
    </element>
   </optional>
  </element>
```

To reference a named pattern in the compact syntax, you just name it:

```
element author {
  attribute id { text },
  name-element,
  element born { text }?,
  element died { text }?
}
```

The same would be applied to referencing the "common-content" named pattern:

```
  <element name="author">
   <ref name="common-content"/>
   <optional>
    <element name="died">
     <text/>
    </element>
   </optional>
  </element>
```

Or:

```
element author {
  common-content,
  element died { text }?
}
```

Grammar and start elements

In our first example, the Russian doll style that we had adopted was such that the definition of the root element (i.e. in our case the library element) could be used as a container for the whole schema. When we define named patterns, Relax NG requires that we define them globally. Thus we will need a container to contain at least the definition of the root element of the instance document and the definitions of the named patterns. This container is what Relax NG calls a `grammar` and it uses, of course, a `grammar` element. When we use a grammar element, Relax NG requires that we declare

explicitly which will be the root element (or elements since we will see later on how to define choices between patterns) and this is done using a `start` element. The top level structure of the schema defining a pattern `name-element` would thus be:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
 <start>
  <element name="library">
   .../...
  </element>
 </start>
 <define name="name-element">
  .../...
 </define>
</grammar>
```

Or, using the compact syntax:

```
grammar {
 name-element = .../...
 start =
  element library {
  .../...
 }
}
```

All together

We have seen the different bits and pieces needed to define and reference patterns and it's time to put them all together and see a complete schema using them. The first exercise we can do is to define a "DTD like" Relax NG schema which defines each of the elements in its own named pattern.

The full schema could be:

<?xml version="1.0" encoding="UTF-8"?>

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">


 <start>
  <ref name="element-library"/>
 </start>


 <define name="element-library">
  <element name="library">
   <oneOrMore>
    <ref name="element-book"/>
   </oneOrMore>
  </element>
 </define>


 <define name="element-book">
  <element name="book">
   <attribute name="id"/>
```

```
  <attribute name="available"/>
  <ref name="element-isbn"/>
  <ref name="element-title"/>
  <oneOrMore>
   <ref name="element-author"/>
  </oneOrMore>
  <zeroOrMore>
   <ref name="element-character"/>
  </zeroOrMore>
 </element>
</define>


<define name="element-isbn">
 <element name="isbn">
  <text/>
 </element>
</define>


<define name="element-title">
 <element name="title">
  <attribute name="xml:lang"/>
  <text/>
 </element>
</define>


<define name="element-author">
 <element name="author">
  <attribute name="id"/>
  <ref name="element-name"/>
  <optional>
   <ref name="element-born"/>
  </optional>
  <optional>
   <ref name="element-died"/>
  </optional>
 </element>
</define>


<define name="element-name">
 <element name="name">
  <text/>
 </element>
</define>


<define name="element-born">
 <element name="born">
  <text/>
 </element>
</define>


<define name="element-died">
 <element name="died">
  <text/>
```

```
  </element>
 </define>


 <define name="element-character">
  <element name="character">
   <attribute name="id"/>
   <ref name="element-name"/>
   <optional>
    <ref name="element-born"/>
   </optional>
   <ref name="element-qualification"/>
  </element>
 </define>


 <define name="element-qualification">
  <element name="qualification">
   <text/>
  </element>
 </define>


</grammar>
```

Or:

```
grammar{


start = element-library


element-library = element library {element-book +}


element-book = element book {
    attribute id { text },
    attribute available { text },
    element-isbn,
    element-title,
    element-author+,
    element-character*
   }


element-isbn = element isbn { text }


element-title = element title {
    attribute xml:lang { text },
    text
   }


element-author = element author {
    attribute id { text },
    element-name,
    element-born?,
    element-died?
```

```
   }

element-name = element name { text }

element-born = element born { text }

element-died = element died { text }

element-character = element character {
    attribute id { text },
    element-name,
    element-born?,
    element-qualification
   }

element-qualification = element qualification { text }

}
```

This "DTD style" is pretty common and has the advantage to facilitate finding the definition of each element in the schema. Another popular style is to define the content of each element as a pattern:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">

 <start>
  <element name="library">
   <ref name="library-content"/>
  </element>
 </start>

 <define name="library-content">
  <oneOrMore>
   <element name="book">
    <ref name="book-content"/>
   </element>
  </oneOrMore>
 </define>

 <define name="book-content">
  <attribute name="id"/>
  <attribute name="available"/>
  <element name="isbn">
   <ref name="isbn-content"/>
  </element>
  <element name="title">
   <ref name="title-content"/>
  </element>
  <oneOrMore>
   <element name="author">
    <ref name="author-content"/>
```

```
  </element>
 </oneOrMore>
 <zeroOrMore>
  <element name="character">
   <ref name="character-content"/>
  </element>
 </zeroOrMore>
</define>


<define name="isbn-content">
 <text/>
</define>


<define name="name-content">
 <text/>
</define>


<define name="born-content">
 <text/>
</define>


<define name="died-content">
 <text/>
</define>


<define name="qualification-content">
 <text/>
</define>


<define name="title-content">
 <attribute name="xml:lang"/>
 <text/>
</define>


<define name="author-content">
 <attribute name="id"/>
 <element name="name">
  <ref name="name-content"/>
 </element>
 <optional>
  <element name="born">
   <ref name="born-content"/>
  </element>
 </optional>
 <optional>
  <element name="died">
   <ref name="died-content"/>
  </element>
 </optional>
</define>


<define name="character-content">
```

```
  <attribute name="id"/>
  <element name="name">
   <ref name="name-content"/>
  </element>
  <optional>
   <element name="born">
    <ref name="born-content"/>
   </element>
  </optional>
  <element name="qualification">
   <ref name="qualification"/>
  </element>
 </define>


</grammar>
```

Or:

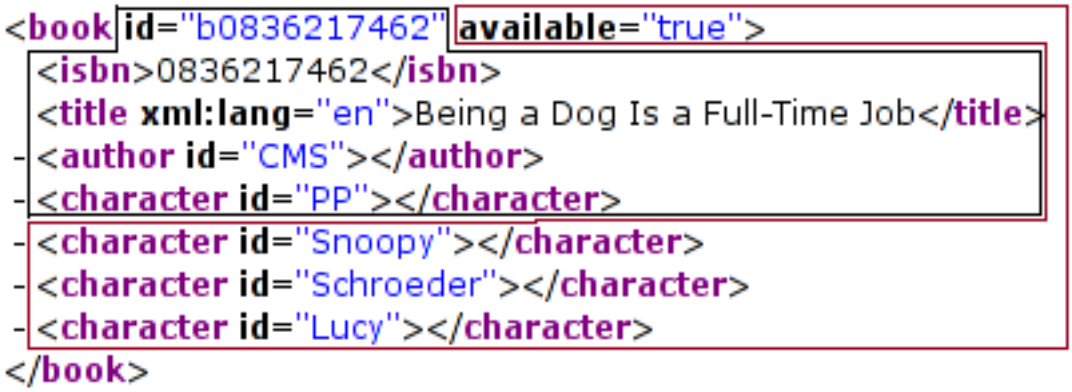
```
grammar {


start = element library {library-content}


library-content =
  element book { book-content } +


book-content =
    attribute id { text },
    attribute available { text },
    element isbn { isbn-content },
    element title { title-content },
    element author { author-content }+,
    element character { character-content }*


isbn-content = text


name-content = text


born-content = text


died-content = text


qualification-content = text


title-content =
  attribute xml:lang { text },
  text


author-content =
  attribute id { text },
```

```
  element name { name-content },
  element born { born-content }?,
  element died { died-content }?


character-content =
  attribute id { text },
  element name { name-content },
  element born { born-content }?,
  element qualification { qualification }


}
```

Note that we will see in "Chapter 12: Writing Extensible Schemas" that the style (Russian doll, DTD like or content oriented like this schema) has an impact on the extensibility of your schemas and the last option we've seen is the most extensible.

We could also revisit the "bizarre patterns" mentioned "Chapter 2: Simple Is Beautiful":

Figure 6.3. rng-full-pattern



When we think about it, this case is not so uncommon. Let's say for instance that we have a first pattern named "book-basic" with the `id` attribute and the `isbn`, `title` one or more `author` and an optional `character` element and a second pattern to extend the first one named "book-extended" with the `available` attribute and zero or more `character` elements. Well, yes that may happen if the the team in charge of defining the "book-basic" pattern has been short visioned and has thought that one character was enough for a book!

Updating the "DTD like" flavor of our schema is just a matter of splitting the definition of the `book` element:

```
<define name="element-book">
 <element name="book">
  <ref name="book-basic"/>
  <ref name="book-extended"/>
 </element>
</define>


<define name="book-basic">
 <attribute name="id"/>
 <ref name="element-isbn"/>
 <ref name="element-title"/>
 <oneOrMore>
  <ref name="element-author"/>
```

```
  </oneOrMore>
  <optional>
   <ref name="element-character"/>
  </optional>
 </define>


 <define name="book-extended">
  <attribute name="available"/>
  <zeroOrMore>
   <ref name="element-character"/>
  </zeroOrMore>
 </define>
```

Or:

```
element-book = element book {
    book-basic,
    book-extended
   }


book-basic =
    attribute id { text },
    element-isbn,
    element-title,
    element-author+,
    element-character?


book-extended =
    attribute available { text },
    element-character*
```

Non restrictions

One of the nice features of Relax NG is that some of the restrictions which add a lot of complexity in other schema languages are non-restrictions for Relax NG; we've seen at least two of them in this chapter. The first one is the ability to define attributes wherever you want in your patterns. This doesn't make a big difference when you define the content model of each elements straightforwardly like in our first schema, but this makes a huge difference when we start to combine patterns as we've done with our bizarre model. Without this non-restriction, it would have been impossible to define one attribute in the pattern "book-start" and a second one in the pattern "book-end".

The other non-restriction found in the chapter is the fact that Relax NG pays no attention to the pattern used to match a node of the instance document when there is several possibilities. Again, in our bizarre pattern, if we have a document with a book having only one author, there is no way to tell if this author matches the optional `author` element of the pattern "book-start" or the zero or more `author` elements of the pattern "book-author". This would be considered as an ambiguity intolerable for other schema languages. In this case, Relax NG considers that even though there is an ambiguity, since there is at least one interpretation of the schema for which the document is valid then the document should be considered as valid. We will learn more about these ambiguities and their consequences on the uses which can be done of the schemas in "Chapter 16: Determinism and Datatype Assignment".

Recursive models

As mentioned in the introduction of this chapter, named patterns are the only way to represent recursive models. We haven't seen yet all the building blocks needed to define a XHTML `div` element, but we

can take a simpler example. If our library is divided into categories, each of them having a title, zero or more embedded categories and zero or more books, we could write (assuming that named patterns have been defined for the `book` element:

```
<define name="category">
 <element name="category">
  <element name="title">
   <text/>
  </element>
  <zeroOrMore>
   <ref name="category"/>
  </zeroOrMore>
  <zeroOrMore>
   <ref name="book"/>
  </zeroOrMore>
 </element>
</define>
```

Or:

```
category = element category{
 element title{text},
 category *,
 book*
}
```

Note that in this case, the recursive reference to the "category" named pattern must be optional since otherwise the document would have to have an infinite depth!

Escaping named patterns identifiers in the compact syntax

In the last chapter "Chapter 4: Non XML Syntax" we have introduced the compact syntax and noticed that any reserved word could be used as element and attribute names. That's no longer the case for the identifiers of named patterns since they appear at the same position than the keywords.

If we had the funny idea to define a named pattern named `text`, `start` or `element` for instance, the identifier of this named pattern could be confused with this keyword and in this case we need to escape the identifier by a leading "/". For instance to define (and by extension to make a reference) to a named pattern named `start`, we would write:

```
grammar{



start = \start



\start = element start { text }
```

```
}
```

And in the XML syntax, this would translate as:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="start"/>
  </start>
  <define name="start">
    <element name="start">
      <text/>
    </element>
  </define>
</grammar>
```

Chapter 7. Chapter 6: More Patterns

So far, we have only described ordered groups of elements and text nodes and we will now see other patterns describing choices and unordered sequences. Although this class of patterns have no special name in the Relax NG specification, we will call them "compositors" in this book by analogy with the compositors defined by W3C XML Schema and because this name is accurate to describe these patterns which are used to compose complex patterns out of the basic patterns matching actual nodes in the document such as `element`, `attribute` and `text`. One of the key differentiators between these compositors and patterns matching nodes is that compositors are syntactical elements which have no existence out of a schema and we will insist on this aspect since it is easy to forget it when we focus on a schema and kind of forget the instance document and this may lead to unexpected errors.

The group pattern

When we have written, to define our `character` element:

```
<element name="character">
 <attribute name="id"/>
 <element name="name">
  <text/>
 </element>
 <element name="born">
  <text/>
 </element>
 <element name="qualification">
  <text/>
 </element>
</element>
```

we have not specified how the different nodes composing the `character` element needed to be composed and Relax NG has implied that we have been using a `group` compositor. This `group` compositor which is implied in the XML syntax is visible in the compact syntax where it is marked by the commas (",") used between the patterns:

```
element character {
 attribute id {text},
 element name {text},
 element born {text},
 element qualification {text}}
```

When using the XML syntax, the `group` compositor may be explicitly specified and the above definition is strictly equivalent to:

```
<element name="character">
 <group>
  <attribute name="id"/>
  <element name="name">
   <text/>
  </element>
  <element name="born">
   <text/>
  </element>
```

```
    <element name="qualification">
     <text/>
    </element>
   </group>
  </element>
```

Because the order of attributes is considered as not significant per XML 1.0, the semantic of the `group` compositor is slightly less straightforward than it appears and this compositor means something such as: "check that the patterns included in this compositor appear in the specified order except for attributes which may appear in any order in the start tag independently of the location where they are found in the patterns."

A last thing to keep in mind about `group` compositors is that as we have said about compositors in general there is no such thing as a "group of nodes" in an instance document and that the notion of `group` is a pattern which belongs only to our schema. One of the consequences of this statement is that there is no hard border to isolate nodes within a group from nodes out of the group and we will see in the section named "Why is it called "interleave" instead of "unorderedGroup? that in certain conditions nodes matching patterns defined outside of a group can be "inserted" in the group.

The `interleave` pattern

The second compositor, named `interleave`, describes a set of unordered patterns, i.e. a set of patterns which will be validated if they match the content of the instance documents in any order. As far as validation is concerned, this behavior is similar to the validation of attributes in a `group` compositor up to the point that the algorithms to validate attributes within groups is the same than the algorithm to validate any node in `interleave` compositors. This is, of course, for validation purposes only and using `interleave` patterns do not imply that the order of elements and text nodes in the instance document will not be reported to the application but only that they are allowed to appear in any order.

To specify that `character` elements may accept children elements in any order, we just need to replace our explicit or implicit group pattern by an `interleave` pattern:

```
<element name="character">
 <interleave>
  <attribute name="id"/>
  <element name="name">
   <text/>
  </element>
  <element name="born">
   <text/>
  </element>
  <element name="qualification">
   <text/>
  </element>
 </interleave>
</element>
```

In the compact syntax, `interleave` patterns are marked using a ampersand ("&") character as a separator instead of the comma (",") which is the mark of ordered groups:

```
element character {
 attribute id {text}&
 element name {text}&
```

```
element born {text}&
element qualification {text}}
```

These two equivalent schemas will validate `character` elements with children elements in any order, such as:

```
<character id="PP">
 <name>Peppermint Patty</name>
 <born>1966-08-22</born>
 <qualification>bold, brash and tomboyish</qualification>
 </character>
<character id="Snoopy">
 <born>1950-10-04</born>
 <qualification>extroverted beagle</qualification>
 <name>Snoopy</name>
</character>
<character id="Schroeder">
 <qualification>brought classical music to the Peanuts strip</qualification>
 <name>Schroeder</name>
 <born>1951-05-30</born>
</character>
```

Although `interleave` looks straightforward at this level of description, we will see more of its complete behavior and restrictions in the last sections of this chapter. You can skip them if they look too complex for now but you should remember to come back and revisit them if your `interleave` patterns show unexpected results or error messages!

The `choice` pattern

Let's say that we want to add some flexibility to the `name` element and still accept:

```
<name>Lucy</name>
```

but also:

```
<name>
 <first>Charles</first>
 <middle>M</middle>
 <last>Schulz</last>
</name>
```

and:

```
<name>
 <first>Peppermint</first>
 <last>Patty</last>
</name>
```

To express this, we will use a `choice` pattern which will accept either a text node or a group of three elements (one of them being optional):

```
<element name="name">
 <choice>
  <text/>
  <group>
   <element name="first"><text/></element>
   <optional>
     <element name="middle"><text/></element>
   </optional>
   <element name="last"><text/></element>
  </group>
 </choice>
</element>
```

The compact syntax uses a logical or character ("|") to mark choices:

```
element name {
 text|(
  element first{text},
  element middle{text}?,
  element last{text}
 )}
```

Note that we had to use parenthesis "(..)" to mark the boundary of the `group` pattern which is one of the two alternatives of our `choice` pattern.

Pattern compositions

In the preceding example we have combined a choice with a group. This process can be general generalized and there is virtually no restriction nor limit in the way compositors can be combined. As an example, let's say we want now in our character element to allow either a `name` element or the three elements "first-name", "middle-name" (optional) and "last-name" in any relative order but before the `born` and `qualification` elements. That's no problem with Relax NG and we can just write:

```
<element name="character">
 <attribute name="id"/>
 <choice>
  <element name="name"><text/></element>
  <interleave>
   <element name="first-name"><text/></element>
   <optional>
    <element name="middle-name"><text/></element>
   </optional>
   <element name="last-name"><text/></element>
  </interleave>
 </choice>
 <element name="born"><text/></element>
 <element name="qualification"><text/></element>
</element>
```

or, with the compact syntax:

```
element character {
 attribute id {text},
 (
  element name {text}|
  (
   element first-name {text}&
   element middle-name {text} ? &
   element last-name {text}
  )
 )
```

Note that we have added two levels of parenthesis here. This was needed because for the compact syntax, operators are used to determine the nature of compositors (`group`, `interleave` or `choice`). They cannot be mixed at the same level and we need to use these parenthesis to explicitly mark where each compositor starts and ends.

These schemas will validate character elements such as:

```
<character id="PP">
 <first-name>Peppermint</first-name>
 <last-name>Patty</last-name>
 <born>1966-08-22</born>
 <qualification>bold, brash and tomboyish</qualification>
</character>
```

```
<character id="PP2">
 <last-name>Patty</last-name>
 <first-name>Peppermint</first-name>
 <born>1966-08-22</born>
 <qualification>bold, brash and tomboyish</qualification>
</character>
```

```
<character id="Snoopy">
 <name>Snoopy</name>
 <born>1950-10-04</born>
 <qualification>extroverted beagle</qualification>
</character>
```

```
<character id="Snoopy2">
 <first-name>Snoopy</first-name>
 <middle-name>the</middle-name>
 <last-name>Dog</last-name>
 <born>1950-10-04</born>
 <qualification>extroverted beagle</qualification>
</character>
```

or

```
<character id="Snoopy3">
 <middle-name>the</middle-name>
 <last-name>Dog</last-name>
 <first-name>Snoopy</first-name>
 <born>1950-10-04</born>
 <qualification>extroverted beagle</qualification>
</character>
```

The flexibility and freedom with which patterns can be combined and the lack of restrictions associated with these combinations is one of the main assets of Relax NG and a major differentiator with other XML schema languages.

The lack of order in a schema may be a source of information in instances

Before we move on to text patterns and mixed contents, let's think again on the `interleave` pattern. As we have already noted, the fact that these content models are often called "unordered" is misleading since although no ordered is required by the schema, the nodes will be ordered in the instance documents and the order in which they will appear in the document may be considered as significant by the applications. Going back to our example with the names, any application managing names probably needs to know what's my first name and what's my last name and with little additional effort they can get the information whether the first name comes before or after the last name in a XML document. The most user friendly of these applications may also want to know whether I prefer to be called with my first or last name first. Should we add an information item to carry this information when we can just rely on the order of these elements in the instance document?

In other words, defining a content using `interleave` patterns shouldn't be seen as a "degradation" of a schema to make the life of document authors easier. Incidentally it does make their life easier and there is no reason to add arbitrary restrictions when there are not needed by the application but that's not the point I want to make here. The point I want to make here is that a content model defined with `interleave` patterns allow more combinations than those using `group` patterns and that they can be used to bring more information.

The downside with `interleave` patterns is that the freedom with which they can be used is unfortunately specific to Relax NG and if you need to insure that it will also be possible to model your vocabulary with a more rigid schema language such as W3C XML Schema, you will often have to restrict the usage of `interleave` patterns in your Relax NG schemas.

Text and empty patterns, whitespaces and mixed contents

So far, we have only used `text` patterns within `group` patterns and we've missed an important "detail" about this pattern which doesn't mean as one could have expected "a text node" but rather "zero or more text nodes" and this deserves some explanations. First, the reason why `text` patterns accept zero text nodes is linked to the policy adopted by Relax NG regarding whitespaces. Generally speaking, whitespace processing is somewhat fuzzy in XML and Relax NG has attempted to find a the "least surprising" policy to meet most common usages. We will see more whitespace processing when we will cover datatypes where they become significant, but right now, let's say that Relax NG doesn't make any difference between empty strings, no string at all, a string containing only whitespaces before or after an element node and in a lesser extend a single text child element containing only whitespaces.

For instance, in the following snippet:

```
<foo at1="" at2=" ">
 <bar/>
 <bar></bar>
 <bar>
  <baz/>
  <baz/>
 </bar>
 <bar>
 </bar>
</foo>
```

Relax NG considers that the values of at1 and at2, the content of the first and second `bar` elements, the text between the third `bar` start tag and the first "baz" element, the text between the two "baz" element and even the text within the last `bar` element is not significant and that they should be matching both `text` and `empty` patterns. Even though this is not obvious to explain and understand, that rule has been created to meet the most common practices in XML and you would probably never have noticed it if I had not insisted on it. Its two visible consequences for the patterns which we've seen so far are:

- Since `text` patterns match any text node they must match strings which are either empty or containing only whitespaces and since there is no difference between empty strings and no string, `text` patterns match "zero strings" i.e. they are always optional.
- Since `empty` patterns match "zero strings" and since there is no difference between no string and empty strings or strings containing only whitespaces, `empty` patterns match also strings either empty or containing only whitespaces.

In other words, the snippet shown above would match both content models where all the occurrences mentioned are described as `text` or `empty` patterns. And if we add the rule -already used a lot but not yet explicated- that says that you don't need to explicitly express `empty` patterns between elements, these two schemas would both validate this instance document:

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="foo">
 <attribute name="at1"><text/></attribute>
 <attribute name="at2"><text/></attribute>
 <oneOrMore>
  <element name="bar">
   <choice>
    <text/>
    <oneOrMore>
     <element name="baz"><text/></element>
    </oneOrMore>
   </choice>
  </element>
 </oneOrMore>
</element>
```

or

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="foo">
 <attribute name="at1"><empty/></attribute>
 <attribute name="at2"><empty/></attribute>
 <oneOrMore>
  <element name="bar">
   <choice>
    <empty/>
```

```
     <oneOrMore>
      <element name="baz"><empty/></element>
     </oneOrMore>
    </choice>
   </element>
  </oneOrMore>
 </element>
```

After having seen why text patterns had to be optional, we need to see why it's useful for them to match multiple instances. The first point to note is that when a `text` pattern is used with a `group` or `choice` pattern this doesn't make any difference since text nodes are merged when they are contiguous or separated by infoset items not checked by Relax NG such as comments or Processing Instructions (PIs). Within a `group` or a `choice`, there is thus no difference between a pattern which would match one or one or more text nodes. The only place where it can make a difference is thus within `interleave` compositors and that's the reason why this specificity has been introduced. Document oriented applications including XHTML provide numerous examples of so called mixed content elements which accept text end embedded elements in any order and in this case it would have no sense to limit the number of text nodes.

To introduce such a content model, let's say we want to extend the `title` element to include zero or more links using "a" elements with `href` attributes, such as:

```
<title xml:lang="en">Being a
 <a href="http://dmoz.org/Recreation/Pets/Dogs/">Dog</a>
 Is a Full-Time
 <a href="http://dmoz.org/Business/Employment/Job_Search/">Job</a>
</title>
```

The content of the new `title` element can be described as an `interleave` pattern allowing zero or more "a" elements and zero or more text nodes and the fact that the `text` pattern matches zero or more text nodes will avoid us to specify its cardinality and we will just write:

```
 <element name="title">
  <interleave>
   <attribute name="xml:lang"/>
   <zeroOrMore>
    <element name="a">
     <attribute name="href"/>
     <text/>
    </element>
   </zeroOrMore>
   <text/>
  </interleave>
 </element>
```

or, using the compact syntax:

```
element title {
 attribute xml:lang {text}&
 element a {attribute href {text}, text}*&
 text
}
```

Considering that this was still too verbose for something quite common, Relax NG has introduced a specific compositor named `mixed` which has the same meaning than "interleave including a text pattern" and these schemas are strictly equivalent to:

```
<element name="title">
 <mixed>
  <attribute name="xml:lang"/>
  <zeroOrMore>
   <element name="a">
    <attribute name="href"/>
    <text/>
   </element>
  </zeroOrMore>
 </mixed>
</element>
```

The `mixed` compositor is marked using a `mixed` keyword in the compact syntax and would be written as:

```
element title {
 mixed {
  attribute xml:lang {text}&
  element a {attribute href {text}, text} *
 }
}
```

Why is it called `interleave` instead of "unorderedGroup"?

If `interleave` was only about defining unordered groups why would it be called `interleave` and not "unorderedGroup" or something similar? As you'll have guessed, there is something else hidden behind this name and interleave is not only a definition of unordered groups, but a definition of unordered groups which let their child nodes intermix within subgroups even when these groups are ordered groups. Don't worry, this concept is simpler than it looks when we try to give a semi-formal definition and an example will make it easy to grasp!

This behavior of ordered groups immersed in an unordered may be surprising and we can try a real world metaphor to illustrate it. If we imagine that the leaf nodes of a XML document are like a bunch of tourists visiting a museum, we can define the unordered sets of all the tourists visiting and ordered groups of tourists following their guides. There are different ways to immerse ordered groups within the unordered set of the museum visitors and to mix ordered groups together and the interleave pattern describes one specific way to do this immersion: when the museum is an interleave pattern, the ordered groups only preserve the relative order of their members and not only allow individual tourists to insert themselves within a group but also two groups to `interleave` their members.

To come back to XML and Relax NG, let's take the following schema:

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="museum">
 <interleave>
  <element name="individual"><empty/></element>
  <group>
   <element name="group-member1"><empty/></element>
```

```
    <element name="group-member2"><empty/></element>
   </group>
  </interleave>
 </element>
```

or, using the compact syntax:

```
element museum {
 element individual {empty} &
 (
  element group-member1 {empty},
  element group-member2 {empty}
 )
}
```

Element "individual" is an individual visiting the museum and elements "group-member1" and "group-member2" are in a group. Because `interleave` patterns are non ordered groups, the following instance documents are valid:

```
<museum>
 <individual/>
 <group-member1/>
 <group-member2/>
</museum>
```

and

```
<museum>
 <group-member1/>
 <group-member2/>
 <individual/>
</museum>
```

These documents are instances where the element "individual" which matches the first pattern in the interleave pattern (i.e. the element pattern) is either before or after the elements "group-member1" and "group-member2" which match the `group` pattern which is the second sub-pattern of the `interleave` pattern. And, because the `interleave` pattern allows that the nodes matching its sub-pattern are mixed, the schema also validate this third combination:

```
<museum>
 <group-member1/>
 <individual/>
 <group-member2/>
</museum>
```

On the other hand, all the combinations where the relative order between group members would not be respected would be invalid. A example of such invalid combinations is:

```
<museum>
 <group-member2/>
```

```
 <individual/>
 <group-member1/>
</museum>
```

Interleave can also be used to "mix" two groups of patterns and in this case the relative order of the element of each group is maintained but the groups may elements of different groups may appear in any order and the groups may be "interleaved". For instance, the following schema:

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="museum">
 <interleave>
  <group>
   <element name="group1.member1"><empty/></element>
   <element name="group1.member2"><empty/></element>
  </group>
  <group>
   <element name="group2.member1"><empty/></element>
   <element name="group2.member2"><empty/></element>
  </group>
 </interleave>
</element>
```

or, using the compact syntax:

```
element museum{
 (
  element group1.member1 {empty},
  element group1.member2 {empty}
 ) & (
  element group2.member1 {empty},
  element group2.member2 {empty}
 )
}
```

will validate documents such as:

```
<museum>
 <group1.member1/>
 <group1.member2/>
 <group2.member1/>
 <group2.member2/>
</museum>
```

and

```
<museum>
 <group2.member1/>
 <group2.member2/>
 <group1.member1/>
 <group1.member2/>
</museum>
```

where the groups are kept separated but also:

```
<museum>
 <group1.member1/>
 <group2.member1/>
 <group2.member2/>
 <group1.member2/>
</museum>
```

or

```
<museum>
 <group1.member1/>
 <group2.member1/>
 <group1.member2/>
 <group2.member2/>
</museum>
```

where the groups are interleaved.

Ordered mixed content models

We have seen that an pattern interleaved with a group is allowed to appear anywhere between the patterns of the group and this feature may be used with a text pattern to define ordered mixed content models where the text nodes may appear anywhere but the order of the elements is fixed. These content models are quite uncommon in XML and a use case could be a data oriented vocabulary such as our library in which optional text could be inserted to provide more user friendly document such as:

```
<character id="Lucy">
 <name>Lucy</name> made her first apparition in a Peanuts strip on
 <born>1952-03-03</born>, and the least we can say about her is that she is
 <qualification>bossy, crabby and selfish</qualification>.
</character>
```

If we want to fix the order of the children elements, we can just embed a `group` pattern inside a `mixed` pattern:

```
<element name="character">
 <mixed>
  <attribute name="id"/>
  <group>
   <element name="name">
     <text/>
   </element>
   <element name="born">
     <text/>
   </element>
   <element name="qualification">
    <text/>
   </element>
  </group>
```

```
     </mixed>
    </element>
```

Per the definition of the "mixed pattern", this is equivalent to:

```
    <element name="character">
     <interleave>
      <attribute name="id"/>
      <text/>
      <group>
       <element name="name">
         <text/>
       </element>
       <element name="born">
         <text/>
       </element>
       <element name="qualification">
        <text/>
       </element>
      </group>
     </interleave>
    </element>
```

The `text` pattern matches text nodes before, after or between the elements of the group but as we've seen with our museum example in the previous section, the order of the elements in the group will be enforced. The compact syntax will use `mixed` keyword with commas (",") between sub-patterns to express this:

```
  element character {
   mixed {
    attribute id {text},
    element name {text},
    element born {text},
    element qualification {text}
   }
  }
```

We have already seen that the compact syntax `mixed` keyword can be used using ampersands ("&") and commas (",") to define unordered and ordered mixed patterns and to be exhaustive, we must mention that or ("|") can also be used to interleave text nodes in `choice` patterns:

```
element foo{
 mixed {
  (
   element in1.1 {empty},
   element in1.2 {empty}
  ) | (
   element in2.1 {empty}&
   element in2.2 {empty}
  )
 }
}
```

This pattern is interleaving text nodes into either a group of in1.1 and in1.2 elements or an interleave pattern of elements in2.1 and in2.2. In the first case because of the semantic of `group` patterns the order between elements is fixed while in the second one the order doesn't matter. Mixed choice contents do not constitute new content models and are equivalent to choices of mixed content models: we could rewrite this schema as:

```
element foo{
 (
  mixed{
   element in1.1 {empty},
   element in1.2 {empty}
   }
 ) | (
  mixed{
   element in2.1 {empty}&
   element in2.2 {empty}
  }
 )
}
}
```

Principal restriction related to `interleave`

We will see the different restrictions of Relax NG in "Chapter 15: Simplification And Restrictions", but we need to mention the principal restriction related to the `interleave` compositor which will probably bite you at some point if you combine mixed content models.

Let's say we want to extend our `title` element to allow not only links ("a" elements) but also bold characters marked by a `b` element:

```
<title xml:lang="en">Being a
 <a href="http://dmoz.org/Recreation/Pets/Dogs/">Dog</a>
 Is a <b>Full-Time</b>
 <a href="http://dmoz.org/Business/Employment/Job_Search/">Job</a>
</title>
```

Because text may appear before the "a" elements, between "a" and `b` and after the `b` element, we might be tempted to write the following schema:

```
<element name="title">
  <interleave>
    <attribute name="xml:lang"/>
    <text/>
    <zeroOrMore>
      <element name="a">
        <attribute name="href"/>
        <text/>
      </element>
    </zeroOrMore>
    <text/>
    <zeroOrMore>
      <element name="b">
        <text/>
```

```
      </element>
    </zeroOrMore>
    <text/>
  </interleave>
</element>
```

or:

```
element title {
  attribute xml:lang {text}
  & text
  & element a {attribute href {text}, text} *
  & text
  & element b {text} *
  & text
}
```

Running jing against this schema will raise the following error:

```
Error at URL "file:/home/vdv/xmlschemata-cvs/books/relaxng/examples/RngMorePa
line number 1, column number 2: both operands of "interleave" contain "text"
```

This is because there can be only one `text` pattern in each `interleave` pattern. We have seen that `text` patterns match zero or more text nodes, and in this case, the remedy is simple enough; the schema must be rewritten to:

```
<element name="title">
  <interleave>
    <attribute name="xml:lang"/>
    <text/>
    <zeroOrMore>
      <element name="a">
        <attribute name="href"/>
        <text/>
      </element>
    </zeroOrMore>
    <zeroOrMore>
      <element name="b">
        <text/>
      </element>
    </zeroOrMore>
  </interleave>
</element>
```

Or:

```
element title {
  attribute xml:lang {text}
  & text
  & element a {attribute href {text}, text} *
  & element b {text} *
```

```
}
```

This new schema is perfectly valid and it does what we meant to do with our invalid schema.

On this simple example, the diagnostic has been very simple but in practice the situation is usually more complex with conflicting `text` patterns belonging to different sub patterns of `interleave` or `mixed`. When using pattern libraries (as shown in "Chapter 10: Creating Building Blocks"), the conflicting `text` patterns will often belong to different Relax NG grammars, making it still tougher to pinpoint where the problem is. To make it still worse the error messages from the Relax NG processors are often quite cryptic, telling you that you have conflicting `text` patterns in `interleave` without saying where they come from and, unfortunately, you'll have to figure out by yourself.

The reason behind this restriction is to optimize Relax NG implementations using the derivative method described by James Clark. Instead of processing each text node when processing mixed content models these implementations can just memorize the fact that this is mixed content and ignore each text node. To do so, the implementation needs to be able to quickly find if a content model is mixed or not and that's where the restriction does make a difference in term of programming complexity and execution speed.

Missing pattern

We have seen that the `interleave` pattern associates two different features and is both an unordered group and something which alters the way sub-groups can be combined together. These two features are not totally independent since mixing child nodes only have a meaning when the order of the sub-groups is not maintained but they are not totally dependent either and in theory it would be possible to define a pattern with a meaning of "unordered group" which would not have the effect of interleaving child nodes and would keep groups unaltered.

If this pattern doesn't exist in Relax NG, this is not only to keep the language as simple as possible but also because although it is built on top of an abstract mathematical model, Relax NG is also built on top of the experience of its authors who have wanted to focus on general usages and best practices amongst the XML community and we must say that the lack of a "unordered group with no interleaving" hasn't been reported as a limitation with real world applications so far.

Chapter 8. Chapter 7: Constraining Text Values

A good deal of the simplicity of Relax NG comes from a careful definition of its scope and a focus on the validation of the structure of XML documents instead of the values placed within its nodes, however, even without using external libraries which will the subject of our next chapter ("Chapter 8: Datatype libraries"), Relax NG includes a simple and efficient support for values, enumerations, lists and whitespace processing which we will study in this chapter.

Values

The elementary pattern on which enumerations are built is `value`. The syntax and semantics of the `value` pattern is straightforward: the pattern will only be matched if the value found in the instance document matches the value specified in the `value` pattern. Out of the scope of an enumeration which we will see in the next section, `value` patterns can be used to check fixed values such as version identifiers of XML vocabularies. If we wanted a highly specialized vocabulary to describe the book with the ISBN number "0836217462" and only this one, we could replace the `text` pattern by a `value` pattern and write:

```
<element name="isbn">
 <value>0836217462</value>
</element>
```

or, with the compact syntax:

```
element isbn {"0836217462"}
```

and the schema will validate a book with a ISBN number equal to "0836217462" and refuse any other ISBN number.

Co-occurrence constraints

A more common usage of `value` patterns, still out of the scope of an enumeration, is to define so-called "co-occurrence constraints" where the value of a node (often an attribute) changes the content model of another node (often an element). In our library, we have both the `author` and `character` elements have a close semantic. We may want to group them as two variations over a "person" element differentiated by a `type` attribute and write:

```
<person id="CMS" type="author">
 <name>Charles M Schulz</name>
 <born>1922-11-26</born>
 <died>2000-02-12</died>
</person>
```

and

```
<person id="PP" type="character">
 <name>Peppermint Patty</name>
 <born>1966-08-22</born>
 <qualification>bold, brash and tomboyish</qualification>
```

```
</person>
```

However, in this kind of schemas, it is often required to validate that the content models are different according to the value of the `type` attribute and this can be done using `value` patterns. If we still want that all the authors precede the characters, we can just update the definitions of the elements describing authors and characters and keep them in sequence in the definition of the book element:

```
<element name="book">
 <attribute name="id"/>
 <attribute name="available"/>
 <element name="isbn">
  <text/>
 </element>
 <element name="title">
  <attribute name="xml:lang"/>
  <text/>
 </element>
 <zeroOrMore>
  <element name="person">
   <attribute name="type">
    <value>author</value>
   </attribute>
   <attribute name="id"/>
   <element name="name">
    <text/>
   </element>
   <element name="born">
    <text/>
   </element>
   <optional>
    <element name="died">
     <text/>
    </element>
   </optional>
  </element>
 </zeroOrMore>
 <zeroOrMore>
  <element name="person">
   <attribute name="type">
    <value>character</value>
   </attribute>
   <attribute name="id"/>
   <element name="name">
    <text/>
   </element>
   <element name="born">
    <text/>
   </element>
   <element name="qualification">
    <text/>
   </element>
  </element>
 </zeroOrMore>
</element>
```

Or, using the compact syntax:

```
element book {
 attribute id {text},
 attribute available {text},
 element isbn {text},
 element title {attribute xml:lang {text}, text},
 element person {
  attribute id {text},
  attribute type {"author"},
  element name {text},
  element born {text},
  element died {text}?}*,
 element person {
  attribute id {text},
  attribute type {"character"},
  element name {text},
  element born {text},
  element qualification {text}}*
}
```

We will see the main restrictions of W3C XML Schema which are non restrictions for Relax NG in "Chapter 16: Determinism and Datatype Assignment", but we need to mention here that this schema and schemas expressing co-occurrence constraints in general cannot be expressed with W3C XML Schema since they are not "deterministic". To be totally accurate, I have shown in my book "XML Schema" that some of them can be expressed in W3C XML Schema using "xs:key" as a tricky hack, but this doesn't work for the general case and this isn't something easy to implement in a schema translator.

However, if we are now using a single "person" element, this is probably to develop the interoperability between these elements and we may prefer allow to mix the definitions of characters and authors and express this part of the schema as zero or more "person" elements having two possible definitions such as:

```
<element name="book">
 <attribute name="id"/>
 <attribute name="available"/>
 <element name="isbn">
  <text/>
 </element>
 <element name="title">
  <attribute name="xml:lang"/>
  <text/>
 </element>
 <zeroOrMore>
  <element name="person">
   <choice>
    <group>
     <attribute name="type">
      <value>author</value>
     </attribute>
     <attribute name="id"/>
     <element name="name">
      <text/>
     </element>
     <element name="born">
      <text/>
     </element>
     <optional>
```

```
      <element name="died">
       <text/>
      </element>
     </optional>
    </group>
    <group>
     <attribute name="type">
      <value>character</value>
     </attribute>
     <attribute name="id"/>
     <element name="name">
      <text/>
     </element>
     <element name="born">
      <text/>
     </element>
     <element name="qualification">
      <text/>
     </element>
    </group>
   </choice>
  </element>
 </zeroOrMore>
</element>
```

Or:

```
element person {
 (
  attribute id {text},
  attribute type {"author"},
  element name {text},
  element born {text},
  element died {text}?
 ) | (
  attribute id {text},
  attribute type {"character"},
  element name {text},
  element born {text},
  element qualification {text}
 )}
```

Now that we have brought closer the definitions of the two contents for the "person" element, we see that an attribute and the two first sub-elements are common and can be "factorized". The definition of the "person" element can thus be simplified as:

```
<element name="person">
 <attribute name="id"/>
 <element name="name">
  <text/>
 </element>
 <element name="born">
  <text/>
 </element>
 <choice>
```

```
      <group>
       <attribute name="type">
        <value>author</value>
       </attribute>
       <optional>
        <element name="died">
         <text/>
        </element>
       </optional>
      </group>
      <group>
       <attribute name="type">
        <value>character</value>
       </attribute>
       <element name="qualification">
        <text/>
       </element>
      </group>
     </choice>
    </element>
```

or:

```
element person {
 attribute id {text},
 element name {text},
 element born {text},
 ((
  attribute type {"author"},
  element died {text}?
 ) | (
  attribute type {"character"},
  element qualification {text}
 ))
}
```

In the compact syntax first, note that we have had to use a double parenthesis to express our choice. This is because the operators used at any level must be homogeneous (you can't mix ",", "|" and "&" at a same level since it would be ambiguous). The other thing to note is that if we have been able to "factorize" the `id` attribute and the name and born elements and keep the `type` attribute and its two possible values in the choice, this is because we have been able to group the elements which are different with the attribute used to do the distinction between content models and this has been possible not only because the example had been carefully prepared for this but also because of the semantic of implicit `interleave` given to the `attribute` patterns which lets us locate the attribute either in or out of the choice. Finally, we should note that this factorization is just a syntactical variation and that even when such simplification is impossible the co-occurrence constraint may still be expressed even though it would have been more verbose.

Enumerations

We have in hand all the elements needed to define enumerations: what is an enumeration if not a choice between several values? Enumerations are this written in Relax NG through choices of values. A good candidate for an enumeration in our library is the `available` attribute which could be defined as:

```
<attribute name="available">
 <choice>
  <value>available</value>
  <value>checked out</value>
  <value>on hold</value>
 </choice>
</attribute>
```

or:

```
attribute available {"available"|"checked out"|"on hold"}
```

As expected, this definition will validate values such as `available`, "checked out" and "on hold", but also and this might be more unexpected , values such as " available ", "checked out " or even " on hold " with multiple spaces, tabs or CRs between "on" and "hold" and we will see the reason of this behavior -and how to change it if needed- in the next section.

Whitespaces and native datatypes

We use to say that Relax NG doesn't natively support datatypes and, strictly speaking, this assertion is not totally accurate: Relax NG does include a native type system but this type library is extremely weak and consists of only two datatypes (token and string) which only differ by the whitespace processing applied before the validation. The whole Relax NG datatype system can be seen as the ability to add validating transformations to text nodes. These transformations transform text nodes into canonical formats (ie a formats where all the different formats for a same value are converted into a single normalized or "canonical" format) and eventually detect format errors. The two native datatypes do not detect format errors (their formats are broad enough to allow any value) but still transform text nodes in their canonical forms and this makes a difference for enumerations.

Enumerations are thus the first place, before next chapter where we will see more complete datatype libraries, where we will see datatypes at work and this is simply done by adding a `type` attribute in `value` patterns.

Up to now, we haven't specified any datatype when we've written `value` elements and by default, Relax NG consider that they have the default type `token` of the built-in library. The transformation performed on text values per this datatype is a full whitespaces normalization similar to the one which is performed by the XPath "normalize-space()" function: all the sequences of one or more whitespaces -ie characters #x20 (space) #x9 (tab), #xA (linefeed) and #xD (carriage return)- are replaced by a single space and the leading and trailing space is then trimmed.

If we come back on previous examples, writing:

```
<attribute name="available">
 <choice>
  <value>available</value>
  <value>checked out</value>
  <value>on hold</value>
 </choice>
</attribute>
```

or:

```
attribute available {"available"|"checked out"|"on hold"}
```

has been using the default type value (token) and was equivalent to the following:

```
<attribute name="available">
 <choice>
  <value type="token">available</value>
  <value type="token">checked out</value>
  <value type="token">on hold</value>
 </choice>
</attribute>
```

or:

```
attribute available {token "available"|token "checked out"|token "on hold"}
```

When the `token` datatype is used, the string normalization is applied on the value defined in the schema and to the value found in the instance document and the comparison is done on the result of the normalization, which explains why we have seen that "on hold" was matching " on hold " with spaces or tabulations added before, between and after the words.

To suppress this normalization, we can use the second builtin datatype ("string") which doesn't perform any transformation on the values and write:

```
<attribute name="available">
 <choice>
  <value type="string">available</value>
  <value type="string">checked out</value>
  <value type="string">on hold</value>
 </choice>
</attribute>
```

or:

```
attribute available {string "available"|string "checked out"|string "on hold"}
```

With this new definition, the value of our attribute must match exactly the value specified in the schema, ie the strings `available`, "checked out" and "on hold".

Note that the native `token` and "string" datatypes have the same definition than the W3C XML Schema `token` and "string" datatypes with the difference than, as we will see in the next chapter "Chapter 8: Datatype Libraries", the additional restrictions which can be applied through `param` attributes to W3C XML Schema datatypes are not available with the native datatypes.

Also not that the name of the `token` datatype, borrowed to W3C XML Schema is highly confusing. In the general meaning in our IT jargon, a token is a piece of string between two delimiters, i.e. what we would call a "word" in plain English. The `token` datatype is not a word, otherwise "on" and "hold" would be valid tokens but "on hold" wouldn't be one. The `token` datatype is more a "tokenized" datatype, in the sense that it's a string made ready to be easily cut into token because non significant whitespaces have been removed. This confusion is dangerous since it's leading many people into using the "string" datatype when what they really need is `token` (we will see in a next section that using the "string" datatype should be reserved to very specific cases).

Beware of string datatypes in attributes

The fact that no whitespace normalization is performed when the builtin "string" datatype is used may lead to some surprises when we define attributes since the XML parsers must remove the line feeds and carriage returns which they find there and since the value is defined as a text node in the Relax NG which will not be submitted to the same treatment.

This behavior may be confusing in "both ways" and our previous schema defining that the attribute must match "on hold" will always match an attribute where the space between "on" and "hold" will have been replaced by a line feed such as in:

```
<book id="b0836217462" available="on
hold">
```

This is rather normal since any XML parser must send an attribute's value "on hold" in this case and no schema language will change this. The reverse is also possible and if we had written our schema as:

```
<attribute name="available">
 <choice>
  <value type="string">available</value>
  <value type="string">checked out</value>
  <value type="string">on
hold</value>
 </choice>
</attribute>
```

The compact syntax doesn't allow new lines within quotes and to translate this into the compact syntax, we need to introduce a couple of new features to show two different possibilities to include new lines in values.

The first one is borrowed from Python and if instead of using simple (') or double (") quotes, you use three simple (''') or three double (""") quotes, you can include pretty much everything in your values including new lines:

```
attribute available {string "available"|string "checked out"|string """on
hold"""}
```

or:

```
attribute available {string "available"|string "checked out"|string '''on
hold'''}
```

The second way is through escaping the new line character using the syntax "\x{A}" (where "A" is the Unicode value of new line in hexadecimal):

```
attribute available {string "available"|string "on hold"|string "on\x{A}hold"}
```

This pattern defines that the attribute could contain a value with a line feed, ie something which can only happen in XML if the newline in the attribute is explicitly specified through is numeric value, such as in:

```
<book id="b0836217462" available="who&#x0A;knows?">
```

Rule of thumb about string datatypes

That's amazing when you think about all the complex applications that have been made possible by SGML and XML, but something apparently as simple as whitespace processing has consistently been a nightmare for users and programmers.

The "string" datatype will expose you to all the issues related to whitespace handling: a wide class of users and applications will make their best to either remove or add whitespaces in your documents and your document will become invalid.

The `token` datatype will keep you much less exposed to this nightmare and that's the reason why Relax NG has chosen it as its default datatype but that's true for W3C XML Schema datatype too: you shouldn't use the "string" datatype unless you have a good reason to do so.

Examples of good reasons to use the "string" datatype include all the variety of program listings in which you want to preserve whitespaces.

Using different types in each value

In our previous schema, we have have been obliged to define the type for each `value` pattern:

```
attribute available {string "available"|string "checked out"|string "on hold"}
```

This is both verbose and very powerful and there is no restriction forbidding to use different datatypes in the different alternatives of an enumeration. While this would be confusing with our two builtin types to accept `available` as a `token` and "checked out" as a "string", this will become something more useful in the next chapter when we will have more datatypes at hand and could specify for instance that we want to accept either `true` as a boolean or "on hold" as a token.

Exclusions

What if, instead of giving a list of values which are allowed, I want to give a list of values which are forbidden? The `except` pattern has been designed for this purpose, and to exclude the value "0836217462" from the possible ISBN numbers, we would write:

```
<element name="isbn">
 <data type="token">
  <except>
   <value>0836217462</value>
  </except>
 </data>
</element>
```

Or, using the compact syntax:

```
element isbn {token - "0836217462"}
```

Although this looks simple, we must note that the type can be defined at two different levels here: it must be defined in the `data` pattern and may be defined in the `value` pattern and these two definitions have a different meaning. The type attached to the `data` pattern defines a validation performed on the text node and the type attached to the `value` pattern defines how the value should be interpreted and which whitespace processing should be performed.

In our example, both types are `token` and values such as " 0836217462 " would be excluded as well as "0836217462". The two datatypes could also be different such as in:

```
<attribute name="available">
 <data type="token">
  <except>
   <choice>
    <value type="string">available</value>
    <value type="string">checked out</value>
    <value type="string">on hold</value>
   </choice>
  </except>
 </data>
</attribute>
```

Or, using the compact syntax:

```
attribute available {token -(string "available"|string "checked out"|string "o
```

And in this case, the first control would be done on the datatype `token` and the comparison would use the datatype "string".

Lists

Relax NG supports the description of text nodes which are lists of whitespace separated values through the `list` pattern which is pretty non typical in that it can be seen as the only pattern which transforms the structure of the document at validation time by splitting text values into lists of tokens (`token` being used here in the meaning of words, not in the meaning of the `token` datatype). The benefit of doing so is that within a `list` pattern, all the pattern constraining data values may be used combined with the compositors which will let us constrain the combination of these tokens.

Funny enough, if we use a `list` pattern without defining a cardinality, we may not get what we expect. An attribute defined as:

```
<attribute name="see-also">
 <list>
  <data type="token"/>
 </list>
</attribute>
```

or, using the compact syntax:

```
attribute see-also {list {token}}
```

Would not match a list of tokens (such as see-also="0345442695 0449220230 0449214044 0061075647 0061075612") but a list of exactly one token (such as see-also="0345442695"). This is

because the `list` pattern splits the text value into a list of token and this list is then evaluated against the patterns which are included within the `list` pattern. If we want a list of any number of "tokens", we must thus use a `zeroOrMore` pattern to express it:

```
<attribute name="see-also">
 <list>
  <zeroOrMore>
   <data type="token"/>
  </zeroOrMore>
 </list>
</attribute>
```

Or, using the compact syntax:

```
attribute see-also {list {token*}}
```

This would consider the "see-also" attribute as a list of tokens and wouldn't add any constraint (this will of course be different when we will have more datatypes) but we could have used other compositors in the `list` pattern exactly as we have used them in other contexts. To express that we want a list with between 1 to 4 tokens, we could say:

```
<attribute name="see-also">
 <list>
   <data type="token"/>
   <optional>
    <data type="token"/>
   </optional>
   <optional>
    <data type="token"/>
   </optional>
   <optional>
    <data type="token"/>
   </optional>
 </list>
</attribute>
```

Or, using the compact syntax:

```
attribute see-also {list {token, token?, token?, token?}}
```

That's verbose but we have already seen that we had no other possibilities to define number of occurrences with RelaxNG...

We could also constraint the values of these tokens, for instance through an enumeration:

```
<attribute name="see-also">
 <list>
  <oneOrMore>
   <choice>
    <value>0836217462</value>
    <value>0345442695</value>
```

```
        <value>0449220230</value>
        <value>0449214044</value>
        <value>0061075647</value>
       </choice>
      </oneOrMore>
     </list>
    </attribute>
```

Or:

```
attribute see-also {list {("0836217462"|"0345442695"|"0449220230"|"044921404
```

A last point to note is that this mechanism gives us the ability to define different constraints for the different members of a list. To illustrate this feature, we need to take another example. Let's say that we want to give the dimension of a book by giving its three dimensions and a unit, such as:

```
<book id="b0836217462" available="true" dimensions="0.38 8.99 8.50 inches">
```

In this case, we can define the "dimensions" attribute as:

```
<attribute name="dimensions">
 <list>
  <data type="token"/>
  <data type="token"/>
  <data type="token"/>
  <choice>
   <value>inches</value>
   <value>cm</value>
   <value>mm</value>
  </choice>
 </list>
</attribute>
```

Or:

```
attribute dimensions {list {token, token, token, ("inches"|"cm"|"mm")}}
```

Data versus text

In the previous chapter (Chapter 6: More patterns) we have spend a lot of time to give a detailed description of the `text` pattern and of its behavior within `interleave` patterns. There is another pattern to describe and attach datatypes to text nodes and even though this pattern will become useful with the introduction of the datatype libraries in next chapter, we can describe its core features right now to be exhaustive on the subject of text nodes.

This pattern is the `data` pattern. The data pattern accepts a `type` attribute (like we have seen for the `value` pattern) and checks that the value is valid per this type. Since our two builtin types accept any value, the `data` pattern with builtin types is almost equivalent to a `text` pattern... Almost only because the `data` pattern does not mean like the `text` pattern "zero or more text nodes" but "one text node" and also because the `data` pattern has been designed to represent data and that it is forbidden

in mixed content models since the authors of the Relax NG specification have considered that this is a bad practice.

This restriction apply to all the patterns matching a single text node (ie, `data`, `value` and `list`) which can never be associated with patterns sibling matching elements (ie elements which could have add the same parent element in the same instance document). In practice, this means that we won't be able to use a `data` pattern describe content models such as:

```
<price><currency>USD</currency>20</price>
```

or

```
<price>20<currency>Euro</currency></price>
```

These content models have been considered bad practice by the authors of the Relax NG specification who advise to reformulate them as:

```
<price>
 <amount>20</amount>
 <currency>USD</currency>
</price>
```

or

```
<price currency="USD">20</price>
```

This is the second time (the first one being when we've seen that there is no "unordered non interleaved" pattern in the previous chapter) that we see Relax NG giving a priority to good practices over the ability to describe all the combinations possible per the XML recommendation. This second case is actually increasing the complexity of the implementations of Relax NG processors which must check that `data` patterns are not included directly or not within mixed content models while the support of data in mixed content models would have been implied by the general algorithms without any additional complexity. The only benefit for Relax NG processors is that they can skip whitespaces occurring between two elements since they cannot match a `data` element which is forbidden between two elements but this benefit seems really minimal compared to the possibilities which are lost by this restriction.

This restriction appears to come from a strict distinction between data oriented and document oriented applications of XML, mixed contents having been considered to belong to document oriented applications which shouldn't need datatypes and datatypes to belong to data oriented applications which shouldn't need mixed contents!

Chapter 9. Chapter 8: Datatype Libraries

In the previous chapter, we have seen the basics of the `data` pattern used with the highly restricted built-in datatype library.

The extreme simplicity of this built-in type library -limited to the two datatypes "string" and `token`- should not be seen as a limitation of Relax NG but rather as a fundamental design decision that validating the structure and the content of XML documents are different issues that are better solved by different tools working in close cooperation.

The Relax NG strategy is thus to rely on external pluggable libraries for the validation of the content of the text nodes and attributes.

There is no limit to the potential variety of external type libraries which could be implemented and used by a Relax NG schema and the designers of Relax NG think that there is probably room for both generic type libraries and application specific types libraries meeting the needs of a specific domain such as mathematics, physics or business.

It is also possible to implement language specific type libraries and my Python implementation of Relax NG supports a native Python types library which maps the built in types and allow to define restrictions using the Python syntax.

That being said, the it is expected that most of the users will use generic XML type libraries ranging from a library emulating the datatypes from the DTDs to an ISO/DSDL type library not yet defined through the W3C XML Schema datatype library and in this chapter we'll introduce the two of them which are already available, i.e. the W3C XML Schema and DTD compatibility type libraries.

W3C XML Schema type library

W3C XML Schema so called simple types are a part that's taking several chapters in my book about W3C XML Schema, but I'll try to give a brief overview here so that you can use their most basic features within Relax NG schemas. You will find their definition in "Chapter 19: W3C XML Schema Datatypes" and you are of course welcome to read the chapters 4, 5, 6 and 16 of my W3C XML Schema book to get a deeper understanding of their behavior!

The W3C XML Schema datatypes which can be used in a Relax NG schema are the so-called "predefined" W3C XML Schema types, i.e. those which are defined in the W3C XML Schema recommendation as opposed to "user defined types" which are derived from the predefined types using the W3C XML Schema language and can't be used from a Relax NG schema. We will see that restrictions (called "facets" in the terminology of W3C XML Schema) can be applied to these datatypes using the Relax NG `param` pattern.

Since we are able to defined named patterns in Relax NG, it means that even though there is no access to "user defined W3C XML Schema simple types", we will have a possibility to define "user defined Relax NG patterns consisting of a predefined W3C XML Schema type and a set of facets". This might be a bit confusing right now but it will become clearer with examples and I just wanted to draw your attention to the fact that Relax NG is just borrowing the most basic part of W3C XML Schema datatypes without borrowing its syntax and derivation methods.

The datatypes

The W3C XML Schema predefined datatypes are divided into "primitive" and "derived" types. Primitive types are basic types which do not share a common semantic and behave differently while each of the derived type could have been derived from a primitive type using the W3C XML Schema

derivation features, shares the semantic of this primitive type and are provided for the convenience of the users since it is expected that it will be commonly used.

The other notion which needs to be introduced before we start is the notion of lexical and value spaces: the lexical space is the string as it appears in the XML document after an eventual whitespace normalization while the value space is the matching value interpreted by the datatype library. The distinction is important since all the facets save one (the `pattern` facet which will be covered in depth in next chapter: "Chapter 9: W3C XML Schema Regular Expressions") are acting on the value space. For instance, the two text nodes `1` and "01" will be considered as different if the datatype is a token and identical if the datatype is an integer.

In this section, we will give a brief presentation of the datatypes classified by their primary types.

String datatypes

The string datatypes are:

- "string" : This is the only datatype for which no whitespace normalization is done. There is no restriction on the lexical or value spaces of this datatype which is identical to the "string" Relax NG built-in type with the exception that restriction can be applied through `param` patterns on the W3C XML Schema string type.
- "normalizedString" : An intermediate whitescape processing is done to this datatypes: the occurrences of whitespaces (is #x9 (tabs), #xA (linefeed) and #x20 (space) are replaced by the same number of spaces (#x20) but no space collapsing or trimming is performed. Like for the "string" datatype, there is no restriction on the lexical or value spaces of this datatype.
- `token` : This datatype is similar to the built-in token datatype: whitespaces are normalized, i.e. all the sequences of whitespaces are replaced by a single space and the leading and trailing spaces are removed. This is -with the two previous one- the third and last datatype which has no constraint on its value and lexical spaces. We must also note that all the datatypes except "string" and "normalizedString" follow the same normalization rules as the `token` datatype.
- "language" : This was created to accept all the language codes standardized by RFC 1766. Some valid values for this datatype are en, en-US, fr, or fr-FR.
- "NMTOKEN" : This corresponds to the XML 1.0 "Nmtoken" (Name token) production, which is a single token (a set of characters without spaces) composed of characters allowed in XML name. Some valid values for this datatype are "Snoopy", "CMS", "1950-10-04", or "0836217462". Invalid values include "brought classical music to the Peanuts strip" (spaces are forbidden) or "bold,brash" (commas are forbidden).
- "NMTOKENS" : The lexical and value spaces of "NMTOKENS" is the whitespace separated lists of "NMTOKEN".
- "Name" : This is similar to "NMTOKEN" with the additional restriction that the values must start with a letter or the characters ":" or `-`. This datatype conforms to the XML 1.0 definition of a "Name." Some valid values for this datatype are "Snoopy", "CMS", or "-1950-10-04-10:00". Invalid values include "0836217462" ("Name" cannot start with a number) or "bold,brash" (commas are forbidden). This datatype should not be used for names that may be "qualified" by a namespace prefix, since we will see another datatype ("QName") that has a specific semantic for these values.
- "NCName" : This is the "noncolonized name" defined by Namespaces in XML1.0, i.e., a "Name" without any colons (":"). As such, this datatype is probably the predefined datatype that is closest to the notion of a `name` in most of the programming languages, even though some characters such as `-` or "." may still be a problem in many cases. Some valid values for this datatype are "Snoopy", "CMS", "-1950-10-04-10-00", or "1950-10-04". Invalid values include "-1950-10-04:10-00" or "bold:brash" (colons are forbidden).
- `ID` : The lexical space of `ID` is the same than the lexical space of "NCName". As defined by the W3C XML Schema recommendation, there is one constraint added to its value space which is that

there must not be any duplicate values in a document. Relax NG doesn't allow datatype libraries to perform this type of checks. This is a job for the "DTD compatibility feature" as we will see at the end of this chapter and its specification asks to Relax NG processors supporting this feature to enforce ID uniqueness for W3C XML Schema ID datatypes. Other implementations will just check its lexical space as a "NCName".

- "IDREF" : The lexical space of "IDREF" is the same than the lexical space of "NCName". As for `ID`, W3C XML Schema adds the constraint that it must match an ID defined in the same document, and Relax NG makes this behavior optional for Relax NG processors supporting the W3C XML Schema type library without supporting the DTD compatibility feature.
- "IDREFS" : The lexical space of "IDREFS" is the whitespace separated lists of "NCName". As for `ID` and "IDREF", W3C XML Schema adds the constraint that each of the values must match an ID defined in the same document, and Relax NG makes this behavior optional for Relax NG processors supporting the W3C XML Schema type library without supporting the DTD compatibility feature.
- "ENTITY" : The lexical space of "ENTITY" is the same than the lexical space of "NCName". Also provided for compatibility with XML 1.0 DTDs, an "ENTITY" value and must match an unparsed entity defined in a DTD.
- "ENTITIES" : The lexical and value spaces of "ENTITIES" is the whitespace separated lists of "ENTITY".

URIs

Strictly speaking, "anyURI", the only representant of this family isn't considered as a string since its value can be different from its lexical representation to compensate the differences of format between XML and URIs as specified in the RFCs 2396 and 2732. These RFCs are not very friendly toward non-ASCII characters and require many character escaping that are not necessary in XML.

As an example of this transformation, the href attribute of an XHTML link written as:

```
<a href="http://dmoz.org/World/Fran&#65533;ais/">
  World/Fran&#65533;ais
</a>
```

would be converted to the value:

```
http://dmoz.org/World/Fran%C3%A7ais/
```

in the value space.

Also note that the "anyURI" datatype doesn't pay any attention to xml:base attributes that may have been defined in the document.

Qualified names

Up to know, we have only briefly mentioned XML namespaces and we will introduce them in "Chapter 11: Namespaces" but we need to use some of their concepts right now. If you're not familiar with namespaces, you should probably be safe to skip this section: you can be quite sure that you don't need qualified names and even if you are a XML namespace guru, I wouldn't recommend you to use them which I consider a bad practice!

What we're talking about here is different to using qualified names for element and attribute names. Using qualified names for element and attribute names is required by the recommendation

"Namespaces in XML 1.0" and there isn't much debate left on the subject. Here, we are speaking of using qualified names in element or attribute values which is much more controversial since it's creating a dependency between the markup and its content.

Because of this dependency, you cannot consider a qualified name as string datatypes since its prefix is only a shortcut to the associated namespace URI. The value space of a qualified named is thus not what we see but a tuple composed of the associated namespace URI (replacing the prefix) and its local part (i.e. what is after the prefix and the colon).

For instance, if the "xs" prefix has been associated with the namespace URI "http://www.w3.org/2001/XMLSchema", a qualified name (QName) "xsd:language" would thus have a value which is the tuple {"http://www.w3.org/2001/XMLSchema", "language"} and can be considered equal to a QName "foo:language" if the prefix "foo" has been associated with "http://www.w3.org/2001/XMLSchema" or even "language" if "http://www.w3.org/2001/XMLSchema" has been defined as the default namespace.

There are two QName datatypes considered as equivalent for Relax NG:

- "QName" : this is the "usual" QName datatype where the lexical space is the set of "colonized" names consisting of a prefix and a local names separated by a colon (":") and the value space is the set of tuples {namespace URI, local name} as explained above. Note that a prefix must be defined through a namespace declaration in the scope of the location where it is used to be considered as valid.

- "NOTATION" : for W3C XML Schema, a "NOTATION" is a QName declared as a notation in a schema W3C XML Schema. Since Relax NG has no equivalent syntax to declare notations, a Relax NG processor treats the "NOTATION" as a synonym to "QName".

Binary string-encoded datatypes

XML 1.0 is unable to hold binary content, which must be string-encoded before it can be included in a XML document. W3C XML Schema has defined two primary datatypes to support two encodings, one that are commonly used (base64) and one which is newer (hexBinary). These encodings may be used to include any binary content, including text formats whose content may be incompatible with the XML markup. Other binary text encodings may also be used (such as uuXXcode, Quote Printable, BinHex, aencode, or base85, to name a few), but their value would not be recognized by W3C XML Schema.

- "hexBinary": This defines a simple way to code binary content as a character string by translating the value of each binary octet into two hexadecimal digits. This encoding is different from the encoding method called BinHex (introduced by Apple, described by RFC 1741, and includes a mechanism to compress repetitive characters). A UTF-8 XML header such as: <?xml version="1.0" encoding="UTF-8"?> that is encoded as hexBinary would be: "3f3c6d78206c657673726f693d6e3122302e20226e656f636964676e223d54552d4622383e3f".

- "base64Binary": This matches the encoding known as "base64" and is described in RFC 2045. It maps groups of 6 bits into an array of 64 printable characters. The same header encoded as base64Binary would be: "PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4NCg==". The W3C XML Schema Recommendation missed the fact that RFC 2045 requests a line break every 76 characters. This should be clarified in an errata. The consequence of these line breaks being thought of as optional by W3C XML Schema, is that the lexical and value spaces of "base64Binary" cannot be considered identical.

Numeric datatypes

The numeric datatypes are built on top of four primitive datatypes: "decimal" for all the decimal types (including the integer datatypes, considered decimals without a fractional part), "double" and "float" for single and double precision floats, and `boolean` for Booleans.

The first family of numeric datatypes is derived from the primitive type "decimal":

- "decimal": This datatype represents the decimal numbers. The number of digits can be arbitrarily long (the datatype doesn't impose any restriction), but obviously, since a XML document has an arbitrary but finite length, the number of digits of the lexical representation of a "decimal" value needs to be finite. Although the number of digits is not limited, we will see in the next section (facets) how the author of a schema can derive user-defined datatypes with a limited number of digits if needed. Leading and trailing zeros are not significant and may be trimmed. The decimal separator is always a dot ("."); a leading sign ("+" or `-`) may be specified and any characters other than the 10 digits (including whitespaces) are forbidden. Allowed values for decimal include "123.456", "+1234.456", "-.456" or "-456".

- `integer`:This integer datatype is a subset of "decimal", representing numbers which don't have any fractional digits in its lexical or value spaces. The characters that are accepted are reduced to 10 digits and an optional leading sign. Like its base datatype, `integer` doesn't impose any limitation on the number of digits, and leading zeros are not significant. Note that the decimal separator is forbidden even if the decimal numbers are omitted or zeros.

- "nonPositiveInteger": The W3C has thought that negative statements would be clearer for developers here and "nonPositiveInteger" are the `integer` which are negative or null (because zero is neither positive for negative).

- "negativeInteger": `integer` which are strictly negative.

- "nonNegativeInteger": positive or null `integer`.

- "positiveInteger": strictly positive `integer`.

- "long": integer between -9223372036854775808 and 9223372036854775807, i.e., the values that can be stored in a 64-bit word.

- "int": integer between -2147483648 and 2147483647 (32 bits).

- "short": integer between -32768 and 32767 (16 bits).

- "byte": integer between -128 and 127 (8 bits).

- "unsignedLong": unsigned integers between 0 and 18446744073709551615, i.e., the values that can be stored in a 64-bit word.

- "unsignedInt": unsigned integers between 0 and 4294967295 (32 bits).

- "unsignedShort": unsigned integers between 0 and 65535 (16 bits).

- "unsignedByte": unsigned integers between 0 and 255 (8 bits).

The second family is made of the "float" and "double" datatypes which represent IEEE simple (32 bits) and double (64 bits) precision floating-point types. These store the values in the form of mantissa and an exponent of a power of 2 (m x 2^e), allowing a large scale of numbers in a storage that has a fixed length. Fortunately, the lexical space doesn't require that we use powers of 2 (in fact, it doesn't accept powers of 2), but instead lets us use a traditional scientific notation with integer powers of 10. Since the value spaces (powers of 2) don't exactly match the values from the lexical space (powers of 10), the recommendation specifies that the closest value is taken. The consequence of this approximate matching is that float datatypes are the domain of approximation; most of the float values can't be considered exact, and are approximate.

These datatypes accept several "special" values: positive zero (0), negative zero (-0) (which is less than positive 0 but greater than any negative value), infinity (INF) (which is greater than any value), negative infinity (-INF) (which is less than any float, and "not a number" (NaN).

The last member is `boolean`, a primitive datatype that can take the values `true` and `false` (or `1` and `0` considered as equivalent).

Date and time formats

This is probably the most controversial piece of W3C XML Schema datatypes. In order to meet the requirements of the "dates on the web", the W3C XML Schema Working Group has attempted to define a value space for a subset of the ISO 8601 date formats which is a syntactical specification of how dates should be exchanged on the web.

The result is overly complex and yet fails to satisfy the experts of date and time representations, doesn't support any other calendar system than Gregorian and has no support for localization.

One of the most fuzzy aspects of these datatypes is that many of them (such as "dateTime" which we'll introduce in a moment) accept both values with and without timezones introducing for the same datatypes two classes of values which can be compared only partially.

Let's take a closer look to this important distinction before we present the detail of these datatypes... Two "dateTime" with a timezone can be compared without any hesitation. W3C XML Schema states that a "dateTime" without a timezone has an undetermined timezone but that you can still compare two such "dateTime". Things get fuzzy when you want to compare a "dateTime" with a timezone and a "dateTime" without: all you know about the "dateTime" without having an undetermined timezone is that in can be in an interval from 14 hours before UTC to 14 hours after UTC and you can never conclude that the two "dateTime" are equal and can only say that one is before the other when they are different enough.

Why 14 hours? No, that's not a typo! National regulations have some level of flexibility with the timezones used in their countries and can vary from their geographical timezone. This variation does even often change with the date in the year with many countries having winter and summer times. As a result of that, the worse case when the W3C has published the W3C XML Schema recommendation was not between -12 and +12 hours from UTC but between -13 and +12 hours. And since the W3C doesn't expect that national authorities would ask them the permission if they wanted to enlarge this interval, they have taken a security margin and written this -14/+14 hours interval in their recommendation.

Since fuzziness isn't what computers like best, it's probably a very good practice to use exclusively "dateTime" with timezones!

All that being said, the date, time and related datatypes defined by W3C XML Schema are:

- "dateTime": This datatype is defined as representing a "specific instant of time." This is a subset of what ISO 8601 calls a "moment of time." Its lexical value follows the format "CCYY-MM-DDThh:mm:ss," in which all the fields must be present and may optionally be preceded by a sign and leading figures, if needed, and followed by fractional digits for the seconds and a time zone. The time zone may be specified using the letter "Z," which identifies UTC, or by the difference of time with UTC. As we've seen, a value such as "2001-10-26T21:32:52" which are defined without a timezone can't be compared to "2001-10-26T21:32:52+02:00" or "2001-10-26T19:32:52Z" which have a timezone and the two latest values are considered as equal since they identify the same moment.

- "date": This datatype has the same lexical space than the date part of "dateTime" with an optional timezone and is representing a period one day in its time zone, "independent of how many hours this day has." The consequence of this definition is that two dates defined in a different time zone cannot be equal except if they designate the same interval (2001-10-26+12:00 and 2001-10-25-12:00, for instance). Another consequence is that, like with "dateTime", the order relation between a date with a time zone and a date without a time zone is partial.

- "gYearMonth": ("g" for Gregorian) is a Gregorian calendar month ie a period of one calendar month in its timezone and its format is the format of "date" without the day part: "2001-10", "2001-10+02:00" or "2001-10Z" for instance.

- "gYear" is a Gregorian calendar year, ie a period of one calendar year in its timezone and its format is the format of "gYearMonth" without its month part: "2001", "2001+02:00" or "2001Z" for instance (note that these three values identify three different periods and are not considered equal).

- "time": The lexical space of "time" is identical to the time part of "dateTime". The semantic of "time" represents a point in time that recurs every day; the meaning of "01:20:15" is "the point in time recurring each day at 01:20:15 am." Like "date" and "dateTime", "time" accepts an optional time zone definition. The same issue arises when comparing times with and without time zones such as "21:32:52", "21:32:52+02:00" and "19:32:52Z".

- "gDay": The lexical space of "gDay" is "---DD" with an optional time zone specification and it represents a recurring period of one day in the specified time zone occurring each Gregorian calendar month. "---01" represents for instance the first day of each month with an undetermined timezone. Dates are pinned down depending of the number of days of each month and in February for instance, "--31Z" would occur on February 28th (or 29th for leap years).

- "gMonthDay" : The lexical space of "gMonthDay" is "--MM-DD" with an optional time zone specification and it represents a recurring period of one day in the specified time zone occurring each Gregorian calendar year. The Christmas day in UK would, for instance, be "--12-25Z".

- "gMonth": The lexical space of "gMonth" should have been "--MM" with an optional timezone, but a typo in the W3C XML Schema recommendation as specified it as "--MM--" which you can still find in some tools even though an erratum has fixed it back to "--MM" and it represents a recurring period of a calendar month in its timezone. The months of January in Paris would for instance be represented as "--01+01:00".

- "duration": The lexical space of "duration" is "PnYnMnDTnHnMnS", each part (except the leading "P") being optional and a significant amount of complexity comes from the fact that you can mix quantities expressed as months (which have a variable number of days) with quantities expressed as days such as for instance "P1Y2M8DT123S" which means a duration of 1 year, 2 months, 8 days and 123 seconds. We will not enter into the detail of the algorithms here, but this leads to a partial order relation between durations which do not facilitate the facets and processing of these datatypes when they use all the parts together.

Examples

After that long and dense enumeration of types, let's see how we could add W3C XML Schema datatypes in our first schema... The most natural choices seem to be:

- id attributes: the semantic of the `ID` datatype isn't captured when it is used with Relax NG, we won't use it in our schema since it would be misleading and we will use "NMTOKEN" for the id attributes.

- xml:lang: the natural candidate for xml:lang is "language".

- available: we can use a `boolean` for this attribute.

- born and died: "date" seem the right choice since we have been lucky enough to have ISO 8601 dates in our instance documents.

- other text content elements: we have no reason here to preserve whitespaces in these elements and will use `token` datatypes for all of them.

Our first schema could thus be rewritten (note the declaration of the datatype library) as:

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
 <oneOrMore>
  <element name="book">
   <attribute name="id">
```

```
     <data type="NMTOKEN"/>
    </attribute>
    <attribute name="available">
     <data type="boolean"/>
    </attribute>
    <element name="isbn">
     <data type="NMTOKEN"/>
    </element>
    <element name="title">
     <attribute name="xml:lang">
      <data type="language"/>
     </attribute>
     <data type="token"/>
    </element>
    <zeroOrMore>
     <element name="author">
      <attribute name="id">
       <data type="NMTOKEN"/>
      </attribute>
      <element name="name">
       <data type="token"/>
      </element>
      <element name="born">
       <data type="date"/>
      </element>
      <optional>
       <element name="died">
        <data type="date"/>
       </element>
      </optional>
     </element>
    </zeroOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id">
       <data type="NMTOKEN"/>
      </attribute>
      <element name="name">
       <data type="token"/>
      </element>
      <element name="born">
       <data type="date"/>
      </element>
      <element name="qualification">
       <data type="token"/>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

or:

```
 element library {
  element book {
```

```
   attribute id {xsd:NMTOKEN},
   attribute available {xsd:boolean},
   element isbn {xsd:NMTOKEN},
   element title {attribute xml:lang {xsd:language}, xsd:token},
   element author {
    attribute id {xsd:NMTOKEN},
    element name {xsd:token},
    element born {xsd:date},
    element died {xsd:date}?}*,
   element character {
    attribute id {xsd:NMTOKEN},
    element name {xsd:token},
    element born {xsd:date},
    element qualification {xsd:token}}*
  } +
 }
```

Note that the W3C XML Schema datatype library has a special privilege to have its prefix built in to the compact syntax: I have used the "xsd" prefix without needing to declare any datatype library! We will see later on that this isn't the case for the DTD compatibility type library.

We have noticed in the previous chapter that the data types declarations are kind of transient to a `data` pattern and are not inherited by its child patterns. Let's illustrate this now that we have a richer set of datatypes at hand.

In the schema which we've just written, we have defined the `available` attribute as a `boolean` but in our instance documents, we have only used one of the two syntaxes for a boolean (`true` or `false`) and not used the other equivalent one (0 or 1). We may want to exclude this second syntax for boolean (for instance if our applications haven't been designed to support it). In this case, we can just exclude these two values:

```
<attribute name="available">
 <data type="boolean">
  <except>
   <value>0</value>
   <value>1</value>
  </except>
 </data>
</attribute>
```

or:

```
attribute available {xsd:boolean - ("0"|"1")}
```

Seems rather natural, but why is this working? When you think about it, it's working because Relax NG forgets that the type of the attribute is `boolean` as soon as we've left the `data` pattern and does use the default type (Relax NG built in `token` type) to test that the value is neither `0` nor `1`. If Relax NG did not forget the type of the attribute, the schema would have removed the entire lexical space of `boolean` and would have been impossible to meet since `0` and `false` are equivalent (and `1` and `true` too).

We have seen a situation where we rely on the fact that the types used in the `data` and `value` patterns are different. There are also situations where we would like them to be the same and, then, we need to repeat the type attribute. If our applications are designed to accept both formats for the available

attributes and if we need to test that the books are available, we would prefer to use the same type for both patterns and in this case we can write:

```
<attribute name="available">
 <data type="boolean">
  <except>
   <value type="boolean">false</value>
  </except>
 </data>
</attribute>
```

or

```
attribute available {xsd:boolean - (xsd:boolean "false")},
```

We now rely on the datatype `boolean` to exclude both `0` and `false` which are equivalent. Of course, in the case of booleans, the number of possible values is limited and we could have simplified our schema to:

```
<attribute name="available">
 <value type="boolean">true</value>
</attribute>
```

or

```
attribute available {xsd:boolean "true"}
```

but this wouldn't have made the point I wanted to make which is also valid for other datatypes!

The facets

The restrictions that a user can apply on a predefined W3C XML Schema datatypes, known as "facets" in the W3C XML Schema recommendation can be applied in a Relax NG schema through a pattern named `param` directly included within `data` patterns before the optional `except` pattern which we already know. These `param` patterns have a name attribute which is the name of the facet and their text content is the value of the facet. When several `param` patterns are included, all the constraints must be matched (in other words, the result is a logical "and" of all the conditions) and a same facet can't be repeated twice except for the facet named `pattern`.

Yes I know, this is confusing but the vocabularies used by Relax NG and W3C XML Schema are different. What Relax NG calls `param` is called "facet" by W3C XML Schema and what's called a `pattern` by Relax NG should not be confused with the facet named `pattern` by W3C XML Schema... Also note that we have seen previously that what Relax NG calls whitespace normalization is not the same than whitespace processing applied on the W3C XML Schema "normalizedSpace" datatype.

The different facets defined by W3C XML Schema are:

- "whiteSpace": this somewhat controversial facet cannot be used in Relax NG.
- "enumeration": this facet cannot be used in Relax NG since equivalent to Relax NG own enumerations which should be used instead.

- `pattern`: this is the only facet working in the lexical space, all the other facets working in the value space only. This facet checks if the data matches a regular expression. This facet is covered in the next chapter "Chapter 9: W3C XML Schema Regular Expressions". For the moment, let's just say that it is a superset of Perl regular expressions (anchored to the beginning and the end of the values to match) and that it does not support the POSIX style character classes defined in Perl, includes a few XML goodies, supports all the Unicode classes and blocks and defines a special construct to define "differences" between character classes.
- "length": this facet is available only for string, binary and list datatypes. For string (and string like) type, this defines the number of Unicode characters, for binary (i.e. "hexBinary" and "base64Binary") datatypes it defines a number of bytes and for list datatypes ("entities", "idrefs" and "NMTOKENS") it defines the number of tokens in the list.
- "maxLength": same meaning and restrictions than "length" but defines a maximum length.
- "minLength": same meaning and restrictions than "length" but defines a minimum length.
- "maxExclusive": applies only to decimal, integer (and derived), float and double and all the date time and duration datatypes and defines a maximum value that cannot be reached. Note that, for date times and duration datatypes, the relation of order between two values is partial and that the result cannot always be determined.
- "minExclusive": same restriction than "maxExclusive" and defines a minimum value that cannot be reached.
- "maxInclusive": same restriction than "maxExclusive" and defines a maximum value that can be reached.
- "minInclusive": same restriction than "maxExclusive" and defines a minimum value that can be reached.
- "totalDigits": applies to decimal, integer and derived types to define the maximum number of digits (after and before the decimal point). As all the facets (except `pattern`) this facet works on the value space, and "000001.10000000" for instance would be considered as only having 2 digits.
- "fractionDigits": applies to decimal to define the maximum number of fractional digits (i.e. after and the decimal point). As all the facets (except `pattern`) this facet works on the value space, and "000001.10000000" for instance would be considered as only having 1 fractional digit.

Again, after this enumeration of facets, let's see how we could use some of our new knowledge to improve the schema of our library:

- xml:lang: we might want to ignore the regional differences and accept only two character codes using the "length" facet.
- isbn: there would be much more to check on isbn number but we may want to use a `pattern` to check that it's composed of 9 digits terminated by a character which is either a digit or the character `X`.
- born and died: assuming that our library is only interested in recent books we could check that they belong to the twentieth or twenty-first centuries (in other words between 1900 and 2099). We might also want to check that our dates do not specify a timezone since we've seen that comparing dates with and without timezone is fuzzy and that the instance documents which we've seen up to now have no timezones.
- and the maximum length of the other text data could be constrained using a "maxLength" facet.

The corresponding schema would be:

```
<element xmlns="http://relaxng.org/ns/structure/1.0"
 name="library" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
 <oneOrMore>
```

```
<element name="book">
 <attribute name="id">
  <data type="NMTOKEN">
    <param name="maxLength">16</param>
  </data>
 </attribute>
 <attribute name="available">
  <data type="boolean"/>
 </attribute>
 <element name="isbn">
  <data type="NMTOKEN">
    <param name="pattern">[0-9]{9}[0-9x]</param>
  </data>
 </element>
 <element name="title">
  <attribute name="xml:lang">
   <data type="language">
    <param name="length">2</param>
   </data>
  </attribute>
  <data type="token">
    <param name="maxLength">255</param>
  </data>
 </element>
 <zeroOrMore>
  <element name="author">
   <attribute name="id">
    <data type="NMTOKEN">
     <param name="maxLength">16</param>
    </data>
   </attribute>
   <element name="name">
    <data type="token">
     <param name="maxLength">255</param>
    </data>
   </element>
   <element name="born">
    <data type="date">
     <param name="minInclusive">1900-01-01</param>
     <param name="maxInclusive">2099-12-31</param>
     <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
    </data>
   </element>
   <optional>
    <element name="died">
     <data type="date">
      <param name="minInclusive">1900-01-01</param>
      <param name="maxInclusive">2099-12-31</param>
      <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
     </data>
    </element>
   </optional>
  </element>
 </zeroOrMore>
 <zeroOrMore>
  <element name="character">
   <attribute name="id">
    <data type="NMTOKEN">
```

```
        <param name="maxLength">16</param>
       </data>
      </attribute>
      <element name="name">
       <data type="token">
        <param name="maxLength">255</param>
       </data>
      </element>
      <element name="born">
       <data type="date">
        <param name="minInclusive">1900-01-01</param>
        <param name="maxInclusive">2099-12-31</param>
        <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
       </data>
      </element>
      <element name="qualification">
       <data type="token">
        <param name="maxLength">255</param>
       </data>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

or:

```
 element library {
  element book {
   attribute id {xsd:NMTOKEN {maxLength = "16"}},
   attribute available {xsd:boolean "true"},
   element isbn {xsd:NMTOKEN {pattern = "[0-9]{9}[0-9x]"}},
   element title {
     attribute xml:lang {xsd:language {length="2"}},
     xsd:token {maxLength="255"}
   },
   element author {
    attribute id {xsd:NMTOKEN {maxLength = "16"}},
    element name {xsd:token {maxLength = "255"}},
    element born {xsd:date {
      minInclusive = "1900-01-01"
      maxInclusive = "2099-12-31"
      pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
    }},
    element died {xsd:date {
      minInclusive = "1900-01-01"
      maxInclusive = "2099-12-31"
      pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
    }}?}*,
   element character {
    attribute id {xsd:NMTOKEN {maxLength = "16"}},
    element name {xsd:token {maxLength = "255"}},
    element born {xsd:date {
      minInclusive = "1900-01-01"
      maxInclusive = "2099-12-31"
```

```
      pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
    }},
    element qualification {xsd:token {maxLength = "255"}}}*
  } +
 }
```

Note the usage of the regular expressions in the `pattern` facets. The set of facets of W3C XML Schema isn't extremely rich and the `pattern` facet acts as a Swiss army knife helping you to do all the tricky tasks that other facets can't do!

Also note that facets are restrictions which are added to the restrictions of the lexical space and that you cannot extend the lexical space of a datatype.

DTD Compatibility

DTD Compatibility is both a library which checks the lexical spaces of its `ID`, "IDREF" and "IRDEFS" datatypes and a feature, i.e. a restriction added to the normal Relax NG processing, which enforces DTDlike rules on the schema and on the instance document. This package is designed to facilitate the transition from DTDs to Relax NG by emulating the attribute types `ID`, "IDREF" and "IDREFS". The DTD compatibility feature checks that `ID` values are unique within a document and that "IDREF" and "IDREFS" are references or whitespace separated lists of references to `ID` values actually defined in the document. It also checks rules on the schema itself such as the fact that these datatypes are used only in attributes. Unlike their W3C XML Schema counterpart, these datatypes have no facets.

That's pretty much all we have to know about this library and we can use it straight away to define the `id` attributes in our library:

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library" datatypeLi
 <oneOrMore>
  <element name="book">
   <attribute name="id">
    <data datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0"
   </attribute>
   <attribute name="available">
    <data type="boolean"/>
   </attribute>
   <element name="isbn">
    <data type="NMTOKEN">
      <param name="pattern">[0-9]{9}[0-9x]</param>
    </data>
   </element>
   <element name="title">
    <attribute name="xml:lang">
     <data type="language">
      <param name="length">2</param>
     </data>
    </attribute>
    <data type="token">
      <param name="maxLength">255</param>
    </data>
   </element>
   <zeroOrMore>
    <element name="author">
     <attribute name="id">
      <data datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0
     </attribute>
```

```
      <element name="name">
       <data type="token">
        <param name="maxLength">255</param>
       </data>
      </element>
      <element name="born">
       <data type="date">
        <param name="minInclusive">1900-01-01</param>
        <param name="maxInclusive">2099-12-31</param>
        <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
       </data>
      </element>
      <optional>
       <element name="died">
        <data type="date">
         <param name="minInclusive">1900-01-01</param>
         <param name="maxInclusive">2099-12-31</param>
         <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
        </data>
       </element>
      </optional>
     </element>
    </zeroOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id">
       <data datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0
      </attribute>
      <element name="name">
       <data type="token">
        <param name="maxLength">255</param>
       </data>
      </element>
      <element name="born">
       <data type="date">
        <param name="minInclusive">1900-01-01</param>
        <param name="maxInclusive">2099-12-31</param>
        <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
       </data>
      </element>
      <element name="qualification">
       <data type="token">
        <param name="maxLength">255</param>
       </data>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

or:

```
 datatypes dtd="http://relaxng.org/ns/compatibility/datatypes/1.0"
 element library {
  element book {
```

```
  attribute id {dtd:ID},
  attribute available {xsd:boolean "true"},
  element isbn {xsd:NMTOKEN {pattern = "[0-9]{9}[0-9x]"}},
  element title {
    attribute xml:lang {xsd:language {length="2"}},
    xsd:token {maxLength="255"}
  },
  element author {
   attribute id {dtd:ID},
   element name {xsd:token {maxLength = "255"}},
   element born {xsd:date {
     minInclusive = "1900-01-01"
     maxInclusive = "2099-12-31"
     pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
   }},
   element died {xsd:date {
     minInclusive = "1900-01-01"
     maxInclusive = "2099-12-31"
     pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
   }}?}*,
  element character {
   attribute id {dtd:ID},
   element name {xsd:token {maxLength = "255"}},
   element born {xsd:date {
     minInclusive = "1900-01-01"
     maxInclusive = "2099-12-31"
     pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
   }},
   element qualification {xsd:token {maxLength = "255"}}}*
 } +
}
```

As already mentioned, the DTD compatibility feature has been designed to provide compatibility with the features of the DTD and that includes emulating some of their restrictions. We have already mentioned the fact that these datatypes can only be used in attributes, not in elements and we need to mention another limitation which can be more insidious and have bitten renowned experts trying to do things such as write Relax NG schemas for XHTML.

This rule might be called the "consistent attribute definition rule": since a DTD won't allow you to give two different definition of the content of an element, Relax NG does enforce the fact that if an attribute `id` is defined as `ID`, "IDREF" or "IDREFS" in an element `bar` somewhere in a Relax NG schema, all the definitions of the same attribute under the same element must use the same type.

The simplest schemas which don't meet that and thus are not correct with respect to the DTD compatibility feature are schemas containing multiple declarations of the same element and attribute with different types, such as in:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="foo" xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0">
  <element name="bar">
    <attribute name="id">
      <data type="ID"/>
    </attribute>
  </element>
  <zeroOrMore>
    <element name="bar">
```

```
      <attribute name="id">
        <data type="token" datatypeLibrary=""/>
      </attribute>
    </element>
  </zeroOrMore>
</element>
```

or:

```
datatypes dtd="http://relaxng.org/ns/compatibility/datatypes/1.0"



element foo {
   element bar {
     attribute id { dtd:ID }
   },
   element bar {
     attribute id { token }
   } *
 }
```

Here, we have two definitions of `bar` with `id` attributes having competing types and, since one of these types is a dtd:ID type, this is forbidden.

A tougher to detect and tougher to fix situation is when one of these competing definitions involves patterns allowing name classes to allow the inclusion of any elements such as we will see in "Chapter 12: Writing Extensible Schemas". The restriction applies even in this case and the situation can become really nasty.

Which library should we use?

All the Relax NG implementations must support the native datatype library and many of them also support the DTD compatibility datatypes library and the W3C XML Schema datatypes library. That means that if we want to define a `token` or "string" datatype we will often have the choice between the native library and W3C XML Schema datatypes and if we are defining `ID`, "IDREF" or "IDREFS" we will often have the choice between the DTD compatibility library and W3C XML Schema datatypes.

That makes a lot of choices to do and in this section we'll try to give some general rules to do your choice.

Native types versus W3C XML Schema datatypes

The criteria to choose between native or W3C XML Schema datatypes to define "string" and `token` types is simple: if you need facets then use W3C XML Schema datatypes. If not use native datatypes: your schema will be more portable since the Relax NG processors are not obliged to support the W3C XML Schema type library.

DTD versus W3C XML Schema datatypes

When you need to define a datatype covered by both DTD and W3C XML Schema, i.e. `ID`, "IDREF" or "IDREFS", the same rule of thumb can be followed: if you are using the DTD compatibility library your schema should be slightly more portable but you will loose the facets.

The other factor to take into account is that the rules applied if you are using the DTD compatibility feature are strict and consistent over different implementations while if you are using the W3C XML Schema type library, a processor should apply these same rules if and only if it also supports the DTD datatype library: processors which only support W3C XML Schema datatypes are only supposed to check the lexical space of these datatypes.

In practice, that means that you can use `ID`, "IDREF" or "IDREFS" datatypes from the W3C XML Schema library but then it is safer to debug your schema using and implementation supporting both the DTD and the W3C XML Schema type libraries.

If you design a Relax NG schema using W3C XML Schema's `ID`, "IDREF" and "IDREFS" and test it with an implementation which supports only W3C XML Schema datatypes you will have a laxed control over both the instance documents and the schema --the rules of the DTD compatibility will not be enforced. When you will use the same schema and instance documents with a Relax NG processor supporting both the DTD and W3C XML Schema datatypes you will then get a stricter control; the instance documents and even the schema which were previously valid may suddenly become invalid or incorrect because of this stricter control.

A simple example of schema which is correct for Relax NG implementations supporting W3C XML Schema datatypes without supporting the DTD compatibility layer but doesn't meet the DTD compatibility feature for Relax NG implementations supporting both is a schema defining ID elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="foo" xmlns="http://relaxng.org/ns/structure/1.0"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <zeroOrMore>
    <element name="bar">
      <element name="id">
        <data type="ID"/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

or:

```
element foo {
  element bar {
    element id { xsd:ID }
  } *
}
```

Other examples include schemas which are not respecting the rule by which the definitions of attributes holding these datatypes must be consistent over the schema.

The reason for this behavior is that although I have often been speaking of "DTD compatibility datatype library" for clarity all over this chapter, DTD compatibility is more than a datatype library. Per the Relax NG formal specification, a datatype library must be decoupled from the validation of the structure of the document and the context passed to the datatype library is restricted to the namespace declarations available under the node being validated. This context itself is an exception required to process qualified names. The datatype library has thus not enough information to do the tests requires to support the DTD compatibility: it doesn't even know if the data to validate has been found in an element or an attribute. This part of the DTD compatibility is thus a feature and not a datatype library as defined per Relax NG.

When we use a datatype from the datatype library "http://relaxng.org/ns/compatibility/datatypes/1.0" we are then doing two different things:

- use a datatype library which will restrict the lexical space of our `data` and `value` patterns
- trigger a feature requesting to validate that the `ID` are unique, and that the "IDREF" and "IDREFS" are referring to ids and lists of ids.

Applied to the W3C XML Schema datatype library, this translates as: trigger the ID DTD compatibility feature when available if these datatypes are used.

Chapter 10. Using Regular Expressions to Specify Simple Datatypes

Among the different facets available to restrict the lexical space of simple datatypes, the most flexible (and also the one that we will often use as a last resort when all the other facets are unable to express the restriction on a user-defined datatype) is based on regular expressions.

Note

There is a terminology clash between Relax NG's patterns and the "pattern" facet of W3C XML Schema. To limit the risk of confusion we will refer to the facet as the "pattern facet" or "regular expression".

The Swiss Army Knife

The pattern facet (and regular expressions in general) is like a Swiss army knife when constraining simple datatypes. It is highly flexible, can compensate for many of the limitations of the other facets, and are often used to define user datatypes on various formats such as ISBN numbers, telephone numbers, or custom date formats. However, like a Swiss army knife, it has its own limitation.

Cutting a tree with a Swiss army knife is long, tiring, and dangerous. Writing regular expressions may also become long, tiring, and dangerous when the number of combinations grows. One should try to keep them as simple as possible.

A Swiss army knife cannot change lead into gold, and no facet can change the primary type of a simple datatype. A string datatype restricted to match a custom date format will still retain the properties of a string and will never acquire the facets of a datetime datatype. This means that there is no effective way to express localized date formats.

The Simplest Possible Pattern facets

In their simplest form, pattern facets may be used as enumerations applied to the lexical space rather than on the value space.

If, for instance, we have a byte value that can only take the values "1," "5," or "15," the classical way to define such a datatype is to use the Relax NG's `choice` pattern:

```
<choice>
  <value type="byte">1</value>
  <value type="byte">5</value>
  <value type="byte">15</value>
</choice>
```

or:

```
element foo {
xsd:byte "1"
| xsd:byte "5"
| xsd:byte "15"
```

```
}
```

This is the "normal" way of defining this datatype if it matches the lexical space and the value space of an `xsd:byte`. It gives the flexibility to accept the instance documents with values such as "1," "5," and "15," but also "01" or "0000005."

As far as validation only is concerned, if we wanted to remove the variations with leading zeros, we could just use another datatype such as `token` instead of `xsd:byte` in our `choice` pattern:

```
<choice>
  <value type="token">1</value>
  <value type="token">5</value>
  <value type="token">15</value>
</choice>
```

or:

```
xsd:token "1"
| xsd:token "5"
| xsd:token "15"
```

However, we might have good reasons to use a `xsd:byte`, for instance if we are interested in type annotation and want that a Relax NG processor supporting type annotation reports the datatype as `xsd:byte` and not `xsd:token`.

One of the particularities of the pattern facet is it must be the only facet constraining the lexical space. If we have an application that is disturbed by leading zeros, we can use pattern facets instead of enumerations to define our datatype:

```
<data type="byte">
  <param name="pattern">1|5|15</param>
</data>
```

or:

```
xsd:byte {pattern = "1|5|15"}
```

Here, we are still using the `xsd:byte` datatype with its semantic, but its lexical space is now constrained to accept only "1," "5," and "15," leaving out any variation that has the same value but a different lexical representation.

Tip

This is an important difference from Perl regular expressions, on which W3C XML Schema pattern facets are built. A Perl expression such as `/15/` matches any string containing "15," while the W3C XML Schema pattern facet matches only the string equal to "15." The Perl expression equivalent to this pattern facet is thus `/^15$/`.

This example has been carefully chosen to avoid using any of the meta characters used within pattern facets, which are: ".", "\", "?", "*", "+", "{", "}", "(", ")", "[", and "]". We will see the meaning of these characters later in this chapter; for the moment, we just need to know that each of these characters needs to be "escaped" by a leading "\" to be used as a literal. For instance, to define a similar datatype for a decimal when lexical space is limited to "1" and "1.5," we write:

```
<data type="decimal">
  <param name="pattern">1|1\.5</param>
</data>
```

Or:

```
xsd:decimal {pattern = "1|1\.5"}
```

A common source of errors is that "normal" characters should not be escaped: we will see later that a leading "\" changes their meaning (for instance, "\P" matches all the Unicode punctuation characters and not the character "P").

Quantifying

Despite an apparent similarity, the pattern facet interprets its value in a very different way than `value` does. `value` reads the value as a lexical representation, and converts it to the corresponding value for its base datatype, while pattern facet reads the value as a set of conditions to apply on lexical values. When we write:

```
pattern="15"
```

we specify three conditions (first character equals "1," second character equals "5," and the string must finish after this). Each of the matching conditions (such as first character equals "1" and second character equals "5") is called a piece. This is just the simplest form to specify piece.

Each piece in a pattern facet is composed of an atom identifying a character, or a set of characters, and an optional quantifier. Characters (except special characters that must be escaped) are the simplest form of atoms. In our example, we have omitted the quantifiers. Quantifiers may be defined using two different syntaxes: either a special character (`*` for 0 or more, `+` for one or more, and `?` for 0 or 1) or a numeric range within curly braces (`{n}` for exactly n times, `{n,m}` for between n and m times, or `{n,}` for n or more times).

Using these quantifiers, we can merge our three pattern facets into one:

```
<data type="byte">
  <param name="pattern">1?5?</param>
</data>
```

Or:

```
xsd:byte {pattern = "1?5?"}
```

This new pattern facet means there must be zero or one character ("1") followed by zero or one character ("5"). This is not exactly the same meaning as our three previous pattern facets since the empty string "" is now accepted by the pattern facet. However, since the empty string doesn't belong to the lexical space of our base type (`xsd:byte`), the new datatype has the same lexical space as the previous one.

We could also use quantifiers to limit the number of leading zeros--for instance, the following pattern facet limits the number of leading zeros to up to 2:

```
<data type="byte">
  <param name="pattern">0{0,2}1?5?</param>
</data>
```

Or:

```
xsd:byte {pattern = "0{0,2}1?5?"}
```

More Atoms

By this point, we have seen the simplest atoms that can be used in a pattern facet: "1," "5," and "\." are atoms that exactly match a character. The other atoms that can be used in pattern facets are special characters, a wildcard that matches any character, or predefined and user-defined character classes.

Special Characters

Table 6-1 shows the list of atoms that match a single character, exactly like the characters we have already seen, but also correspond to characters that must be escaped or (for the first three characters on the list) that are just provided for convenience.

Table 10.1. Special characters

\n	New line (can also be written as "
-- since we are in a XML document).
\r	Carriage return (can also be written as "
--).
\t	Tabulation (can also be written as " --)
\\	Character "\"
|	Character "|"
\.	Character "."
\-	Character "-"
\^	Character "^"
\?	Character "?"
*	Character "*"
\+	Character "+"
\{	Character "{"
\}	Character "}"
\(	Character "("
\)	Character ")"
\[	Character "["
\]	Character "]"

Wildcard

The character "." has a special meaning: it's a wildcard atom that matches any XML valid character except newlines and carriage returns. As with any atom, "." may be followed by an optional quantifier and ".*" is a common construct to match zero or more occurrences of any character. To illustrate the usage of ".*" (and the fact that pattern facet is a Swiss army knife), a pattern facet may be used to define the integers that are multiples of 10:

```
<define name="multipleOfTen">
  <data type="integer">
```

```
      <param name="pattern">.*0</param>
    </data>
  </define>
```

Or:

```
multipleOfTen = xsd:integer {pattern = ".*0"}
```

Character Classes

W3C XML Schema has adopted the "classical" Perl and Unicode character classes (but not the POSIX-style character classes also available in Perl).

Classical Perl character classes

W3C XML Schema supports the classical Perl character classes plus a couple of additions to match XML-specific productions. Each of these classes are designated by a single letter; the classes designated by the upper- and lowercase versions of the same letter are complementary:

`\s` Spaces. Matches the XML whitespaces (space #x20, tabulation #x09, line feed #x0A, and carriage return #x0D).

`\S` Characters that are not spaces.

`\d` Digits ("0" to "9" but also digits in other alphabets).

`\D` Characters that are not digits.

`\w` Extended "word" characters (any Unicode character not defined as "punctuation", "separator," and "other"). This conforms to the Perl definition, assuming UTF8 support has been switched on.

`\W` Nonword characters.

`\i` XML 1.0 initial name characters (i.e., all the "letters" plus "-"). This is a W3C XML Schema extension over Perl regular expressions.

`\I` Characters that may not be used as a XML initial name character.

`\c` XML 1.0 name characters (initial name characters, digits, ".", ":", "-", and the characters defined by Unicode as "combining" or "extender"). This is a W3C XML Schema extension to Perl regular expressions.

`\C` Characters that may not be used in a XML 1.0 name.

These character classes may be used with an optional quantifier like any other atom. The last pattern facet that we saw:

```
multipleOfTen = xsd:integer {pattern = ".*0"}
```

constrains the lexical space to be a string of characters ending with a zero. Knowing that the base type is a `xsd:integer`, this is good enough for our purposes, but if the base type had been a `xsd:decimal` (or `xsd:string`), we could be more restrictive and write:

```
multipleOfTen = xsd:integer {pattern = "-?\d*0"}
```

This checks that the characters before the trailing zero are digits with an optional leading `-` (we will see later on in Section 6.5.2.2 how to specify an optional leading `-` or `+`).

Unicode character classes

Patterns support character classes matching both Unicode categories and blocks. Categories and blocks are two complementary classification systems: categories classify the characters by their usage independently to their localization (letters, uppercase, digit, punctuation, etc.), while blocks classify characters by their localization independently of their usage (Latin, Arabic, Hebrew, Tibetan, and even Gothic or musical symbols).

The syntax `\p{Name}` is similar for blocks and categories; the prefix `Is` is added to the name of categories to make the distinction. The syntax `\P{Name}` is also available to select the characters that do not match a block or category. A list of Unicode blocks and categories is given in the specification. Table 6-2 shows the Unicode character classes and Table 6-3 shows the Unicode character blocks.

Table 10.2. Unicode character classes

Unicode Character Class	Includes
C	Other characters (non-letters, non symbols, non-numbers, non-separators)
Cc	Control characters
Cf	Format characters
Cn	Unassigned code points
Co	Private use characters
L	Letters
Ll	Lowercase letters
Lm	Modifier letters
Lo	Other letters
Lt	Titlecase letters
Lu	Uppercase letters
M	All Marks
Mc	Spacing combining marks
Me	Enclosing marks
Mn	Non-spacing marks
N	Numbers
Nd	Decimal digits
Nl	Number letters
No	Other numbers
P	Punctuation
Pc	Connector punctuation
Pd	Dashes
Pe	Closing punctuation
Pf	Final quotes (may behave like Ps or Pe)
Pi	Initial quotes (may behave like Ps or Pe)
Po	Other forms of punctuation
Ps	Opening punctuation
S	Symbols
Sc	Currency symbols
Sk	Modifier symbols

Unicode Character Class	Includes
Sm	Mathematical symbols
So	Other symbols
Z	Separators
Zl	Line breaks
Zp	Paragraph breaks
Zs	Spaces

Table 10.3. Unicode character blocks

AlphabeticPresentationForms	Arabic	ArabicPresentationForms-A
ArabicPresentationForms-B	Armenian	Arrows
BasicLatin	Bengali	BlockElements
Bopomofo	BopomofoExtended	BoxDrawing
BraillePatterns	ByzantineMusicalSymbols	Cherokee
CJKCompatibility	CJKCompatibilityForms	CJKCompatibilityIdeographs
CJKCompatibilityIdeographsSupplement	CJKRadicalsSupplement	CJKSymbolsandPunctuation
CJKUnifiedIdeographs	CJKUnifiedIdeographsExtensionA	CJKUnifiedIdeographsExtensionB
CombiningDiacriticalMarks	CombiningHalfMarks	CombiningMarksforSymbols
ControlPictures	CurrencySymbols	Cyrillic
Deseret	Devanagari	Dingbats
EnclosedAlphanumerics	EnclosedCJKLettersandMonths	Ethiopic
GeneralPunctuation	GeometricShapes	Georgian
Gothic	Greek	GreekExtended
Gujarati	Gurmukhi	HalfwidthandFullwidthForms
HangulCompatibilityJamo	HangulJamo	HangulSyllables
Hebrew	HighPrivateUseSurrogates	HighSurrogates
Hiragana	IdeographicDescriptionCharacters	IPAExtensions
Kanbun	KangxiRadicals	Kannada
Katakana	Khmer	Lao
Latin-1Supplement	LatinExtended-A	LatinExtendedAdditional
LatinExtended-B	LetterlikeSymbols	LowSurrogates
Malayalam	MathematicalAlphanumericSymbols	MathematicalOperators
MiscellaneousSymbols	MiscellaneousTechnical	Mongolian
MusicalSymbols	Myanmar	NumberForms
Ogham	OldItalic	OpticalCharacterRecognition
Oriya	PrivateUse	PrivateUse
PrivateUse	Runic	Sinhala
SmallFormVariants	SpacingModifierLetters	Specials
Specials	SuperscriptsandSubscripts	Syriac
Tags	Tamil	Telugu
Thaana	Thai	Tibetan
UnifiedCanadianAboriginalSyllabics	YiRadicals	YiSyllables

We will see in the next section that W3C XML Schema has introduced an extension to Regular Expressions to specify intersections and that this extension can be used to define the intersection between a block and a category in a single pattern facet.

Note

Although Unicode blocks seem to be a great idea to restrict text to use a set of characters which you know that you'll be able to print, display, read or store in a database, they have not been designed for this purpose and one must be careful when using them. Here is what John Cowan who is has always fascinated me by his knowledge of Unicode and its more obscure alphabets writes about this topic:

The five Latin blocks mentioned by John are BasicLatin, Latin-1Supplement, LatinExtended-A, LatinExtendedAdditional and LatinExtended-B.

User-defined character classes

These classes are lists of characters between square brackets that accept `-` signs to define ranges and a leading `^` to negate the whole list--for instance:

```
[azertyuiop]
```

to define the list of letters on the first row of a French keyboard,

```
[a-z]
```

to specify all the characters between "a" and "z",

```
[^a-z]
```

for all the characters that are not between "a" and "z," but also

```
[-^\\]
```

to define the characters "-," "^," and "\," or

```
[-+]
```

to specify a decimal sign.

These examples are enough to see that what's between these square brackets follows a specific syntax and semantic. Like the regular expression's main syntax, we have a list of atoms, but instead of matching each atom against a character of the instance string, we define a logical space. Between the atoms and the character class is the set of characters matching any of the atoms found between the brackets.

We see also two special characters that have a different meaning depending on their location! The character `-`, which is a range delimiter when it is between `a` and `z`, is a normal character when it is just after the opening bracket or just before the closing bracket (`[+-]` and `[-+]` are, therefore, both legal). On the contrary, `^`, which is a negator when it appears at the beginning of a class, loses this special meaning to become a normal character later in the class definition.

Note

Even though this is specified as valid by the W3C XML Schema recommendation, it is not supported by all the regular expression engines used by Relax NG processors. The current version of Jing (as I write these lines) do not support `[+-]` nor `[-+]` and it is wiser to escape the character `-` and write either `[+\-]` or `[\-+]`.

> Another frequent confusion is about the support of the escape format `#xXX` (such as in `#x2D`. Because this format is used in the W3C XML Schema recommendation to describe characters by their Unicode value, some people have thought that it could be used in regular expressions which is not meant to be the case. If you want to define characters by their Unicode values, you should instead use numeric entities (such as `&#x2D;` if you are using the XML syntax or the syntax for escaping characters in the compact syntax `\x{2D}`. Note that in both cases, this will be replaced by the corresponding character at parse time and that the regular expression engine will see the actual character instead of the escape sequence.

We also notice that characters may or must be escaped: "\\" is used to match the character "\". In fact, in a class definition, all the escape sequences that we have seen as atoms can be used. Even though some of the special characters lose their special meaning inside square brackets, they can always be escaped. So, the following:

```
[-^\\]
```

can also be written as:

```
[\-\^\\]
```

or as:

```
[\^\\-]
```

since the location of the characters doesn't matter any longer when they are escaped.

Within square brackets, the character "\" also keeps its meaning of a reference to a Perl or Unicode class. The following:

```
[\d\p{Lu}]
```

is a set of decimal digits (Perl class `\d`) and uppercase letters (Unicode category "Lu").

Mathematicians have found that three basic operations are needed to manipulate sets and that these operations can be chosen from a larger set of operations. In our square brackets, we already saw two of these operations: union (the square bracket is an implicit union of its atoms) and complement (a leading `^` realizes the complement of the set defined in the square bracket). W3C XML Schema extended the syntax of the Perl regular expressions to introduce a third operation: the difference between sets. The syntax follows:

```
[set1-[set2]]
```

Its meaning is all the characters in `set1` that do not belong to `set2`, where `set1` and `set2` can use all the syntactic tricks that we have seen up to now.

This operator can be used to perform intersections of character classes (the intersection between two sets A and B is the difference between A and the complement of B), and we can now define a class for the `BasicLatin Letters` as:

```
[\p{IsBasicLatin}-[^\p{L}]]
```

Or, using the `\P` construct, which is also a complement, we can define the class as:

```
[\p{IsBasicLatin}-[\P{L}]]
```

The corresponding definition would be:

```
  <define name="BasicLatinLetters">
```

```
    <data type="token">
      <param name="pattern">[\p{IsBasicLatin}-[\P{L}]]*</param>
    </data>
  </define>
```

Or:

```
BasicLatinLetters = xsd:token {pattern = "[\p{IsBasicLatin}-[\P{L}]]*"}
```

Oring and Grouping

We have already used a "or" in our first example pattern facet when we have written "1|5|15" to say that we wanted to allow either "1", "5" or "15".

These "ors" are especially interesting when used in conjunction with groups. Groups are complete regular expressions, which are, themselves, considered atoms and can be used with an optional quantifier to form more complete (and complex) regular expressions. Groups are enclosed by brackets ("(" and ")"). To define a comma-separated list of "1," "5," or "15," ignoring whitespaces between values and commas, the following pattern facet could be used:

```
<define name="myListOfBytes">
  <data type="token">
    <param name="pattern">(1|5|15)( *, *(1|5|15))*</param>
  </data>
</define>
```

Or:

```
myListOfBytes = xsd:token {pattern = "(1|5|15)( *, *(1|5|15))*"}
```

Note how we have relied on the whitespace processing of the base datatype (`xsd:token` collapses the whitespaces). We have not tested leading and trailing whitespaces that are trimmed and we have only tested single occurrences of spaces with the following atom:

```
run back " * " run back
```

before and after the comma.

Common Patterns

After this overview of the syntax used by pattern facets, let's see some common pattern facets that you may have to use (or adapt) in your schemas or just consider as examples.

String Datatypes

Regular expressions treat information in its textual form. This makes them an excellent mechanism for constraining strings.

Unicode blocks

Unicode is a great asset of XML; however, there are few applications able to process and display all the characters of the Unicode set correctly and still fewer users able to read them! If you need to check that your string datatypes belong to one (or more) Unicode blocks, you can use these pattern facets:

```
<define name="BasicLatinToken">
  <data type="token">
    <param name="pattern">\p{IsBasicLatin}*</param>
  </data>
</define>

<define name="Latin-1Token">
  <data type="token">
    <param name="pattern">[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*</param>
  </data>
</define>
```

Or:

```
BasicLatinToken = xsd:token {pattern = "\p{IsBasicLatin}*"}

Latin-1Token = xsd:token {pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}
```

Note that such pattern facets do not impose a character encoding on the document itself and that, for instance, the `Latin-1Token` datatype would validate instance documents using UTF-8, UTF-16, ISO-8869-1 or other encoding. (This assumes the characters used in this string belong to the two Unicode blocks `BasicLatin` and `Latin-1Supplement`.) In other words, working on the lexical space, i.e., after the transformations have been done by the parser, these pattern facets do not control the physical format of the instance documents.

Counting words

The pattern facet can be used to limit the number of words in a text. To do so, we will define an atom, which is a sequence of one or more "word" characters (`\w+`) followed by one or more nonword characters (`\W+`), and control the number of occurrences of this atom. If we are not very strict on the punctuation, we also need to allow an arbitrary number of nonword characters at the beginning of our value and to deal with the possibility of a value ending with a word (without further separation). One of the ways to avoid any ambiguity at the end of the string is to dissociate the last occurrence of a word to make the trailing separator optional:

```
<define name="story100-200words">
  <data type="token">
    <param name="pattern">\W*(\w+\W+){99,199}\w+\W*</param>
  </data>
</define>
```

Or:

```
story100-200words= xsd:token {pattern = "\W*(\w+\W+){99,199}\w+\W*"}
```

URIs

The `xsd:anyURI` datatype doesn't care about "absolutizing" relative URIs and in some cases it is wise to impose the usage of absolute URIs, which are easier to process. Furthermore, it can also be interesting for some applications to limit the accepted URI schemes. This can easily be done by a set of pattern facets such as:

```
<define name="httpURI">
  <data type="anyURI">
    <param name="pattern">http://.*</param>
```

```
    </data>
  </define>
```

Or:

```
  httpURI= xsd:anyURI {pattern = "http://.*"}
```

Numeric and Float Types

While numeric types aren't strictly text, pattern facets can still be used appropriately to constrain their lexical form.

Leading zeros

Getting rid of leading zeros is quite simple but requires some precautions if we want to keep the optional sign and the number "0" itself. This can be done using pattern facets such as:

```
  <define name="noLeadingZeros">
    <data type="integer">
      <param name="pattern">[+-]?([^0][0-9]*|0)</param>
    </data>
  </define>
```

Or:

```
  noLeadingZeros= xsd:integer {pattern = "[+-]?([^0][0-9]*|0)"}
```

Note that in this pattern facet, we chose to redefine all the lexical rules that apply to an integer. This pattern facet would give the same lexical space applied to a `xsd:token` datatype as on a `xsd:integer`. We could also have relied on the knowledge of the base datatype and written:

```
  <define name="noLeadingZeros">
    <data type="integer">
      <param name="pattern">[+-]?([^0].*|0)</param>
    </data>
  </define>
```

Or:

```
  noLeadingZeros= xsd:integer {pattern = "[+-]?([^0].*|0)"}
```

Relying on the base datatype in this manner can produce simpler pattern facets, but can also be more difficult to interpret since we would have to combine the lexical rules of the base datatype to the rules expressed by the pattern facet to understand the result.

Fixed format

The maximum number of digits can be fixed using `xsd:totalDigits` and `xsd:fractionDigits`. However, these facets are only maximum numbers and work on the value space. If we want to fix the format of the lexical space to be, for instance, "DDDD.DD", we can write a pattern facet such as:

```
  <define name="fixedDigits">
```

```
    <data type="decimal">
      <param name="pattern">[+-]?\.{4}\..{2}</param>
    </data>
  </define>
```

Or:

```
  fixedDigits= xsd:decimal {pattern = "[+-]?\.{4}\..{2}"}
```

Datetimes

Dates and time have complex lexical representations. Patterns can give developers extra control over how they are used.

Time zones

The time zone support of W3C XML Schema is quite controversial and needs some additional constraints to avoid comparison problems. These pattern facets can be kept relatively simple since the syntax of the datetime is already checked by the schema validator and only simple additional checks need to be added. Applications which require that their datetimes specify a time zone may use the following template, which checks that the time part ends with a "Z" or contains a sign:

```
  <define name="dateTimeWithTimezone">
    <data type="dateTime">
      <param name="pattern">.+T[^Z+-]+</param>
    </data>
  </define>
```

or:

```
 dateTimeWithTimezone= xsd:dateTime {pattern = ".+T[^Z+-]+"}
```

Simpler, applications that want to make sure that none of their datetimes specify a time zone may just check that the time part doesn't contain the characters "+", "-", or "Z":

```
  <define name="dateTimeWithoutTimezone">
    <data type="dateTime">
      <param name="pattern">.+T[^Z+-]+</param>
    </data>
  </define>
```

or:

```
 dateTimeWithoutTimezone= xsd:dateTime {pattern = ".+T[^Z+-]+"}
```

In these two datatypes, we used the separator "T". This is convenient, since no occurrences of the signs can occur after this delimiter except in the time zone definition. This delimiter would be missing if we wanted to constrain dates instead of datetimes, but, in this case, we can detect the time zones on their ":" instead:

```
  <define name="dateWithTimezone">
    <data type="date">
      <param name="pattern">.+[:Z].*</param>
```

```
    </data>
  </define>
  <define name="dateWithoutTimezone">
    <data type="date">
      <param name="pattern">[^:Z]</param>
    </data>
  </define>
```

or:

```
dateWithTimezone= xsd:date {pattern = ".+[:Z].*"}
dateWithoutTimezone= xsd:date {pattern = "[^:Z]"}
```

Applications may also simply impose a set of time zones to use:

```
  <define name="dateTimeInMyTimezones">
    <data type="dateTime">
      <param name="pattern">.+(\+02:00|\+01:00|\+00:00|Z|-04:00)</param>
    </data>
  </define>
```

or:

```
 dateTimeInMyTimezones= xsd:dateTime {
  pattern = ".+(\+02:00|\+01:00|\+00:00|Z|-04:00)"
 }
```

We can also constrain `xsd:duration` to a couple of subsets that have a complete sort order. The first datatype will consist of durations expressed only in months and years, and the second will consist of durations expressed only in days, hours, minutes, and seconds. The criteria used for the test can be the presence of a "D" (for day) or a "T" (the time delimiter). If neither of those characters are detected, then the datatype uses only year and month parts. The test for the other type cannot be based on the absence of "Y" and "M", since there is also an "M" in the time part. We can test that, after an optional sign, the first field is either the day part or the "T" delimiter:

```
  <define name="YMduration">
    <data type="duration">
      <param name="pattern">[^TD]+</param>
    </data>
  </define>
  <define name="DHMSduration">
    <data type="duration">
      <param name="pattern">-?P((\d+D)|T).*</param>
    </data>
  </define>
```

or:

```
YMduration= xsd:duration {pattern = "[^TD]+"}
DHMSduration= xsd:duration {pattern = "-?P((\d+D)|T).*"}
```

Chapter 11. Chapter 10: Creating Building Blocks

Up to now, we have seen how named patterns could be used to define flat schemas which are more modular, easier to ready when the number of elements and attributes grows and can be used define recursive content models. In this chapter we will see how they can be used as building blocks to build libraries of content models that can be assembled to create complete schemas. To do so, we will introduce examples which can be seen as basic use cases for these features.

External references

With Russian doll schemas

The first use case is when we want to reuse existing schemas as a whole, without modifying their definitions. Imagine, for instance that we have defined two grammars in two schemas to describe our `author` and `character` elements. We have a first Relax NG schema to describe our authors:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="author" xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <attribute name="id">
    <data type="ID"/>
  </attribute>
  <element name="name">
    <data type="token" datatypeLibrary=""/>
  </element>
  <optional>
    <element name="born">
      <data type="date"/>
    </element>
  </optional>
  <optional>
    <element name="died">
      <data type="date"/>
    </element>
  </optional>
</element>
```

(author.rng)

or:

```
element author {
  attribute id { xsd:ID },
  element name { token },
  element born { xsd:date }?,
  element died { xsd:date }?
}
```

(author.rnc)

And a second one to describe our characters:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="character" xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <attribute name="id">
    <data type="ID"/>
  </attribute>
  <element name="name">
    <data type="token" datatypeLibrary=""/>
  </element>
  <optional>
    <element name="born">
      <data type="date"/>
    </element>
  </optional>
  <element name="qualification">
    <data type="token" datatypeLibrary=""/>
  </element>
</element>
```

(character.rng)

or:

```
element character { element library {
  element book {
    attribute id { xsd:ID },
    attribute available { xsd:boolean },
    element isbn { token },
    element title {
      attribute xml:lang { xsd:language },
      token
    },
    external "author.rnc" +,
    external "character.rnc" *
  }+
}
  attribute id { xsd:ID },
  element name { token },
  element born { xsd:date }?,
  element qualification { token }
}
```

(character.rnc)

If we want to use them in a schema describing our library, we will use `externalRef` patterns:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="library" xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <oneOrMore>
    <element name="book">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
```

```
      <attribute name="available">
        <data type="boolean"/>
      </attribute>
      <element name="isbn">
        <data type="token" datatypeLibrary=""/>
      </element>
      <element name="title">
        <attribute name="xml:lang">
          <data type="language"/>
        </attribute>
        <data type="token" datatypeLibrary=""/>
      </element>
      <oneOrMore>
        <externalRef href="author.rng"/>
      </oneOrMore>
      <zeroOrMore>
        <externalRef href="character.rng"/>
      </zeroOrMore>
    </element>
  </oneOrMore>
</element>
```

In the compact syntax, `externalRef` patterns are translated into keyword `external`:

```
element library {
  element book {
    attribute id { xsd:ID },
    attribute available { xsd:boolean },
    element isbn { token },
    element title {
      attribute xml:lang { xsd:language },
      token
    },
    external "author.rnc" +,
    external "character.rnc" *
  }+
}
```

These patterns have a semantic of straight inclusion: when a Relax NG processor reads a schema it just replaces `externalRef` by the content of the referred document.

With flat schemas

In this first example, we've been using `externalRef` with "Russian doll" schemas, but this would work fine too with flat schemas. For instance, if we change our schemas to:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <start>
    <ref name="element-author"/>
  </start>
```

```
  <define name="element-author">
    <element name="author">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <ref name="element-name"/>
      <optional>
        <ref name="element-born"/>
      </optional>
      <optional>
        <ref name="element-died"/>
      </optional>
    </element>
  </define>

  <define name="element-name">
    <element name="name">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>

  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>

  <define name="element-died">
    <element name="died">
      <data type="date"/>
    </element>
  </define>

</grammar>
```

or:

```
start = element-author
element-author =
  element author {
    attribute id { xsd:ID },
    element-name,
    element-born?,
    element-died?
  }
element-name = element name { token }
element-born = element born { xsd:date }
element-died = element died { xsd:date }
```

And:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w

  <start>
    <ref name="element-character"/>
  </start>

  <define name="element-character">
    <element name="character">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <ref name="element-name"/>
      <optional>
        <ref name="element-born"/>
      </optional>
      <ref name="element-qualification"/>
    </element>
  </define>

  <define name="element-name">
    <element name="name">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>

  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>

  <define name="element-qualification">
    <element name="qualification">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>




</grammar>
```

or:

```
start = element-character
element-character =
  element character {
    attribute id { xsd:ID },
    element-name,
    element-born?,
    element-qualification
  }
element-name = element name { token }
element-born = element born { xsd:date }
element-qualification = element qualification { token }
```

Embedded grammars

This seems straightforward and logical, but why does that work? How come that there is no collision between the named patterns `element-name` and `element-born` defined in both "author.rng" and "character.rng"? How come that `start` patterns defined in "author.rng" and "character.rng" do not apply to the schema for our library?

This is working because we are using a feature called "embedded grammars". As I have already mentioned, the semantic of `externalRef` patterns is a strict inclusion of the referred schema and in our last example, this means that our resulting schema is:

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="library" xmlns="http://relaxng.org/ns/structure/1.0" datatypeLi
  <oneOrMore>
    <element name="book">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <attribute name="available">
        <data type="boolean"/>
      </attribute>
      <element name="isbn">
        <data type="token" datatypeLibrary=""/>
      </element>
      <element name="title">
        <attribute name="xml:lang">
          <data type="language"/>
        </attribute>
        <data type="token" datatypeLibrary=""/>
      </element>
      <oneOrMore>
        <grammar>
          <start>
            <ref name="element-author"/>
          </start>
          <define name="element-author">
            <element name="author">
              <attribute name="id">
                <data type="ID"/>
              </attribute>
              <ref name="element-name"/>
              <optional>
                <ref name="element-born"/>
              </optional>
              <optional>
                <ref name="element-died"/>
              </optional>
            </element>
          </define>
          <define name="element-name">
            <element name="name">
              <data type="token" datatypeLibrary=""/>
            </element>
          </define>
          <define name="element-born">
            <element name="born">
              <data type="date"/>
```

```
            </element>external "character.rnc"
          </define>
          <define name="element-died">
            <element name="died">
              <data type="date"/>
            </element>
          </define>
        </grammar>
      </oneOrMore>
      <zeroOrMore>
        <grammar>
          <start>
            <ref name="element-character"/>
          </start>
          <define name="element-character">
            <element name="character">
              <attribute name="id">
                <data type="ID"/>
              </attribute>
              <ref name="element-name"/>
              <optional>
                <ref name="element-born"/>
              </optional>
              <ref name="element-qualification"/>
            </element>
          </define>
          <define name="element-name">
            <element name="name">
              <data type="token" datatypeLibrary=""/>
            </element>
          </define>
          <define name="element-born">
            <element name="born">
              <data type="date"/>
            </element>
          </define>
          <define name="element-qualification">
            <element name="qualification">
              <data type="token" datatypeLibrary=""/>
            </element>
          </define>
        </grammar>
      </zeroOrMore>
    </element>
  </oneOrMore>
</element>
```

or:

```
element library {
  element book {
    attribute id { xsd:ID },
    attribute available { xsd:boolean },
    element isbn { token },
    element title {
      attribute xml:lang { xsd:language },
```

```
      token
    },
    grammar {
      start = element-author
      element-author =
        element author {
          attribute id { xsd:ID },
          element-name,
          element-born?,
          element-died?
        }
      element-name = element name { token }
      element-born = element born { xsd:date }
      element-died = element died { xsd:date }
    }+,
    grammar {
      start = element-character
      element-character =
        element character {
          attribute id { xsd:ID },
          element-name,
          element-born?,
          element-qualification
        }
      element-name = element name { token }
      element-born = element born { xsd:date }
      element-qualification = element qualification { token }
    }*
  }+
}
```

Here we are thus embedding grammars within our schema and they behave as patterns. In fact that's even more than that: for Relax NG, grammars are patterns! The semantic of these patterns is twofold:

- As far as validation is concerned, embedded grammars are equivalent to their start patterns: the grammar describing the `character` element for instance will instance nodes corresponding to its start pattern, i.e. instance nodes matching the pattern `element-character` which is what we were expecting.

- Grammars are also setting the scope of their definitions: `start` and named patterns defined in a grammar are visible only in this grammar. Their scope (i.e. the location where they can be referred to) is strictly limited to the grammar in which they are defined.

Applied to our example, the strict scoping of `start` and named patterns means that:

- The `born` pattern of the grammar describing the `character` element cannot be seen from its parent grammar, i.e. the grammar describing the `library` and `book` elements nor from its sibling grammar, i.e. the grammar describing the `author` element. The same applies to `start` patterns.

- Unlike common usage among programming languages, the scope of `start` and named patterns do not include embedded grammars and `start` and named patterns defined in the grammar describing `library` and `book` elements would not be visible in the embedded grammars.

Reference to a pattern in the parent grammar

This strict isolation of `start` and named patterns in their grammars is usually what we need in references to external grammars. It means that these external grammars can be written independently

without any risk of collision or incompatibility. Or in other words that you can take any Relax NG schema and drop it into a new schema to see it as a single pattern without any risk of collision.

On the other hand, that doesn't let you modify what you are including (we will see how to do so in the next section) nor even to leverage on a set of common named patterns. In our example, since we already had two definitions of `element-name` and `element-born`, that was a good thing that they have been isolated in their grammars. Now, if we were designing the same building blocks from scratch, we would probably want to have only one definition of these two elements which are common to the `author` and `character` elements. In fact if we were following the principle "if it's written more than once make it common" we would also want to share the definition of the `id` attribute.

We will see another way to do so, but it is also possible to do so through making an explicit reference to a pattern from the parent grammar, i.e. the grammar embedding the current one. In this case, we need to add the definitions which we want to share in the top level schema even if we do not use all of theme in this schema:

<?xml version="1.0" encoding="UTF-8"?>

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
```

```
  <start>
    <element name="library">
      <oneOrMore>
        <element name="book">
          <ref name="attribute-id"/>
          <attribute name="available">
            <data type="boolean"/>
          </attribute>
          <element name="isbn">
            <data type="token" datatypeLibrary=""/>
          </element>
          <element name="title">
            <attribute name="xml:lang">
              <data type="language"/>
            </attribute>
            <data type="token" datatypeLibrary=""/>
          </element>
          <oneOrMore>
            <externalRef href="author.rng"/>
          </oneOrMore>
          <zeroOrMore>
            <externalRef href="character.rng"/>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </start>
```

```
  <define name="element-name">
    <element name="name">
      <data type="token" datatypeLibrary=""/>
    </element>
```

```
  </define>

  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>

  <define name="attribute-id">
    <attribute name="id">
      <data type="ID"/>
    </attribute>
  </define>

</grammar>
```

or:

```
start =
  element library {
    element book {
      attribute-id,
      attribute available { xsd:boolean },
      element isbn { token },
      element title {
        attribute xml:lang { xsd:language },
        token
      },
      external "author.rnc"+,
      external "character.rnc"*
    }+
  }
element-name = element name { token }
element-born = element born { xsd:date }
attribute-id = attribute id { xsd:ID }
```

Now, to make a reference to the named patterns `element-name`, `element-born` and `attribute-id` in the embedded grammars, we will use a pattern called `parentRef`:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
  <start>
    <ref name="element-author"/>
  </start>
  <define name="element-author">
    <element name="author">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <parentRef name="element-name"/>
      <optional>
        <parentRef name="element-born"/>
      </optional>
      <optional>
        <ref name="element-died"/>
```

```
      </optional>
    </element>
  </define>
  <define name="element-died">
    <element name="died">
      <data type="date"/>
    </element>
  </define>
</grammar>
```

(author.rng)

and:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
  <start>
    <ref name="element-character"/>
  </start>
  <define name="element-character">
    <element name="character">
      <attribute name="id">
        <data type="ID"/>
      </attribute>
      <parentRef name="element-name"/>
      <optional>
        <parentRef name="element-born"/>
      </optional>
      <ref name="element-qualification"/>
    </element>
  </define>
  <define name="element-qualification">
    <element name="qualification">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>
</grammar>
```

(character.rng)

The `parentRef` pattern is translated to a `parent` keyword in the compact syntax:

```
start = element-author
element-author =
  element author {
    attribute id { xsd:ID },
    parent element-name,
    parent element-born?,
    element-died?
  }
element-died = element died { xsd:date }
```

(author.rnc)

and:

```
start = element-character
element-character =
  element character {
    attribute id { xsd:ID },
    parent element-name,
    parent element-born?,
    element-qualification
  }
element-qualification = element qualification { token }
```

(character.rnc)

Here again, we are using these features in the context of multiple schema documents, but the semantic of the `externalRef` pattern is unchanged from the previous section and this schema is equivalent to the same schema with the `externalRef` patterns expended in a single monolithic schema with two embedded grammars:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
  <start>
    <element name="library">
      <oneOrMore>
        <element name="book">
          <ref name="attribute-id"/>
          <attribute name="available">
            <data type="boolean"/>
          </attribute>
          <element name="isbn">
            <data type="token" datatypeLibrary=""/>
          </element>
          <element name="title">
            <attribute name="xml:lang">
              <data type="language"/>
            </attribute>
            <data type="token" datatypeLibrary=""/>
          </element>
          <oneOrMore>
            <grammar>
              <start>
                <ref name="element-author"/>
              </start>
              <define name="element-author">
                <element name="author">
                  <attribute name="id">
                    <data type="ID"/>
                  </attribute>
                  <parentRef name="element-name"/>
                  <optional>
                    <parentRef name="element-born"/>
                  </optional>
                  <optional>
                    <ref name="element-died"/>
                  </optional>
                </element>
              </define>
              <define name="element-died">
```

```
                <element name="died">
                  <data type="date"/>
                </element>
              </define>
            </grammar>
          </oneOrMore>
          <zeroOrMore>
            <grammar>
              <start>
                <ref name="element-character"/>
              </start>
              <define name="element-character">
                <element name="character">
                  <attribute name="id">
                    <data type="ID"/>
                  </attribute>
                  <parentRef name="element-name"/>
                  <optional>
                    <parentRef name="element-born"/>
                  </optional>
                  <ref name="element-qualification"/>
                </element>
              </define>
              <define name="element-qualification">
                <element name="qualification">
                  <data type="token" datatypeLibrary=""/>
                </element>
              </define>
            </grammar>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </start>
  <define name="element-name">
    <element name="name">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>
  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>
  <define name="attribute-id">
    <attribute name="id">
      <data type="ID"/>
    </attribute>
  </define>
</grammar>
```

or:

```
start =
  element library {
    element book {
```

```
      attribute-id,
      attribute available { xsd:boolean },
      element isbn { token },
      element title {
        attribute xml:lang { xsd:language },
        token
      },
      grammar {
        start = element-author
        element-author =
          element author {
            attribute id { xsd:ID },
            parent element-name,
            parent element-born?,
            element-died?
          }
        element-died = element died { xsd:date }
      }+,
      grammar {
        start = element-character
        element-character =
          element character {
            attribute id { xsd:ID },
            parent element-name,
            parent element-born?,
            element-qualification
          }
        element-qualification = element qualification { token }
      }*
    }+
  }
element-name = element name { token }
element-born = element born { xsd:date }
attribute-id = attribute id { xsd:ID }
```

See how `start` and named patterns have been defined in each of the three grammars composing this schema:

- `element-died` is defined in the grammar defining the `author` element and can only be used in this grammar.
- similarly, `element-qualification` is defined in the grammar defining the `character` element and can only be used there.
- `element-name`, `element-born` and `attribute-id` are defined in the top level grammar. They can be used in this grammar through normal references (i.e. `ref` patterns) and can also be used in its children grammars, i.e. the grammars which are directly embedded into this one, using a `parentRef` pattern.

There is a couple of more things to note about the `parentRef` pattern:

- If the depth of imbrication of grammar is higher than two, you may run into troubles since you can only make a reference to your immediate parent grammar, not to the other grammar ancestors. The Relax NG working group has considered this issue but hasn't found any real world use case for generalizing `parentRef` patterns to higher imbrication depths. If you find one they will probably welcome a mail on the subject! In practice, if we needed to do so, we would have as a workaround to define named patterns in the intermediary grammars that would act as "proxies".

- Now that we have added the `parentRef` patterns our two schemas "author.rng" and "character.rng" cannot be used as standalone schemas to validate documents which root elements are `author` or `character` elements: they need to be embedded into grammars providing the definitions for the named patterns they are using to be complete and operational.

Merging grammars

In the preceding sections we have seen how we could use an external grammar as a single pattern. This is useful in cases like those we've seen where we want to include a content model described by an external schema at a single point, not unlike when you mount a UNIX file system: the description contained in the external grammar is "mounted" at the point where you make your reference.

The main drawback is that you cannot individually reuse the definitions contained in the external schema. To do so, we need to introduce a new pattern, with a different semantic which will merge two grammars into a single one.

Merging without redefinition

In the simplest case, we will want to reuse patterns defined in common libraries of patterns without modifying them. Let's say we have defined a grammar with some common patterns which can be reused in many different schemas, such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w

  <define name="element-name">
    <element name="name">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>

  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>

  <define name="attribute-id">
    <attribute name="id">
      <data type="ID"/>
    </attribute>
  </define>

  <define name="content-person">
    <ref name="attribute-id"/>
    <ref name="element-name"/>
    <optional>
      <ref name="element-born"/>
    </optional>
  </define>

</grammar>
```

(common.rng)

Or:

```
element-name = element name { token }
element-born = element born { xsd:date }
attribute-id = attribute id { xsd:ID }
content-person = attribute-id, element-name, element-born?
```

(common.rnc)

These schemas are obviously not meant to be used as standalone schemas: they have no `start` patterns and would be considered incorrect. However, they contain definitions which can be used to write the schema of our library. To use these definitions, we need to use `include` patterns:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w

  <include href="common.rng"/>



  <start>
    <element name="library">
      <oneOrMore>
        <element name="book">
          <ref name="attribute-id"/>
          <attribute name="available">
            <data type="boolean"/>
          </attribute>
          <element name="isbn">
            <data type="token" datatypeLibrary=""/>
          </element>
          <element name="title">
            <attribute name="xml:lang">
              <data type="language"/>
            </attribute>
            <data type="token" datatypeLibrary=""/>
          </element>
          <oneOrMore>
            <element name="author">
              <ref name="content-person"/>
              <optional>
                <ref name="element-died"/>
              </optional>
            </element>
          </oneOrMore>
          <zeroOrMore>
            <element name="character">
              <ref name="content-person"/>
              <ref name="element-qualification"/>
            </element>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </start>
```

```
  <define name="element-died">
    <element name="died">
      <data type="date"/>
    </element>
  </define>

  <define name="element-qualification">
    <element name="qualification">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>

</grammar>
```

The `include` pattern is translated as and `include` keyword in the compact syntax:

```
include "common.rnc"
start =
  element library {
    element book {
      attribute-id,
      attribute available { xsd:boolean },
      element isbn { token },
      element title {
        attribute xml:lang { xsd:language },
        token
      },
      element author {
        content-person,
        element-died?
      }+,
      element character {
        content-person,
        element-qualification
      }*
    }+
  }
element-died = element died { xsd:date }
element-qualification = element qualification { token }
```

Note that the name of the `include` pattern is slightly misleading. The `include` pattern here doesn't include the external grammar as a pattern (we have seen that this was the job of the `externalRef` pattern) but it includes the content of the external grammar, performing a merge of both grammars. This is exactly what we needed, though, and this is the reason why we have been able to do references to the named patterns defined in the "common.rng" grammar.

The result of this inclusion is thus equivalent to the following monolithic schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
<!-- Content of the included grammar -->
  <define name="element-name">
    <element name="name">
```

```
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>
  <define name="element-born">
    <element name="born">
      <data type="date"/>
    </element>
  </define>
  <define name="attribute-id">
    <attribute name="id">
      <data type="ID"/>
    </attribute>
  </define>
  <define name="content-person">
    <ref name="attribute-id"/>
    <ref name="element-name"/>
    <optional>
      <ref name="element-born"/>
    </optional>
  </define>
<!-- End of the included grammar -->
  <start>
    <element name="library">
      <oneOrMore>
        <element name="book">
          <ref name="attribute-id"/>
          <attribute name="available">
            <data type="boolean"/>
          </attribute>
          <element name="isbn">
            <data type="token" datatypeLibrary=""/>
          </element>
          <element name="title">
            <attribute name="xml:lang">
              <data type="language"/>
            </attribute>
            <data type="token" datatypeLibrary=""/>
          </element>
          <oneOrMore>
            <element name="author">
              <ref name="content-person"/>
              <optional>
                <ref name="element-died"/>
              </optional>
            </element>
          </oneOrMore>
          <zeroOrMore>
            <element name="character">
              <ref name="content-person"/>
              <ref name="element-qualification"/>
            </element>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </start>
  <define name="element-died">
    <element name="died">
```

```
      <data type="date"/>
    </element>
  </define>
  <define name="element-qualification">
    <element name="qualification">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>
</grammar>
```

or:

```
element-name = element name { token }
element-born = element born { xsd:date }
attribute-id = attribute id { xsd:ID }
content-person = attribute-id, element-name, element-born?
```

```
start =
  element library {
    element book {
      attribute-id,
      attribute available { xsd:boolean },
      element isbn { token },
      element title {
        attribute xml:lang { xsd:language },
        token
      },
      element author {
        content-person,
        element-died?
      }+,
      element character {
        content-person,
        element-qualification
      }*
    }+
  }
element-died = element died { xsd:date }
element-qualification = element qualification { token }
```

Merging and replacing definitions

In the previous example, we have been lucky and the definitions of the common patterns which we've included were exactly matching what we needed. In the real world, this isn't always the case and it is quite handy to be able to replace the definitions found in the grammar that we're including.

Let's say that we have already written this very flat version of our schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
```

```
<start>
  <ref name="element-library"/>
</start>

<define name="element-library">
  <element name="library">
    <zeroOrMore>
      <ref name="element-book"/>
    </zeroOrMore>
  </element>
</define>

<define name="element-book">
  <element name="book">
    <ref name="attribute-id"/>
    <ref name="attribute-available"/>
    <ref name="element-isbn"/>
    <ref name="element-title"/>
    <oneOrMore>
      <ref name="element-author"/>
    </oneOrMore>
    <zeroOrMore>
      <ref name="element-character"/>
    </zeroOrMore>
  </element>
</define>

<define name="element-author">
  <element name="author">
    <ref name="content-person"/>
    <optional>
      <ref name="element-died"/>
    </optional>
  </element>
</define>

<define name="element-character">
  <element name="character">
    <ref name="content-person"/>
    <ref name="element-qualification"/>
  </element>
</define>




<define name="element-isbn">
  <element name="isbn">
    <data type="token" datatypeLibrary=""/>
  </element>
</define>




<define name="element-title">
```

```
  <element name="title">
    <ref name="attribute-xml-lang"/>
    <data type="token" datatypeLibrary=""/>
  </element>
</define>



<define name="attribute-xml-lang">
  <attribute name="xml:lang">
    <data type="language"/>
  </attribute>
</define>



<define name="attribute-available">
  <attribute name="available">
    <data type="boolean"/>
  </attribute>
</define>



<define name="element-name">
  <element name="name">
    <data type="token" datatypeLibrary=""/>
  </element>
</define>



<define name="element-born">
  <element name="born">
    <data type="date"/>
  </element>
</define>



<define name="element-died">
  <element name="died">
    <data type="date"/>
  </element>
</define>



<define name="attribute-id">
  <attribute name="id">
    <data type="ID"/>
  </attribute>
</define>
```

```
  <define name="content-person">
    <ref name="attribute-id"/>
    <ref name="element-name"/>
    <optional>
      <ref name="element-born"/>
    </optional>
  </define>



  <define name="element-qualification">
    <element name="qualification">
      <data type="token" datatypeLibrary=""/>
    </element>
  </define>



</grammar>
```

(library.rng)

or:

```
start = element-library
element-library = element library { element-book* }
element-book =
  element book {
    attribute-id,
    attribute-available,
    element-isbn,
    element-title,
    element-author+,
    element-character*
  }
element-author = element author { content-person, element-died? }
element-character =
  element character { content-person, element-qualification }
element-isbn = element isbn { token }
element-title = element title { attribute-xml-lang, token }
attribute-xml-lang = attribute xml:lang { xsd:language }
attribute-available = attribute available { xsd:boolean }
element-name = element name { token }
element-born = element born { xsd:date }
element-died = element died { xsd:date }
attribute-id = attribute id { xsd:ID }
content-person = attribute-id, element-name, element-born?
element-qualification = element qualification
```

(library.rnc)

If this is a good schema used in production to validate incoming documents from a variety of patterns and we wouldn't want to modify it. However, we might have a new application that doesn't work at

the level of a library but only at the level of a book. This application would need to validate instance documents with `book` root elements. Of course we wouldn't want to copy and paste the definition of our existing schema into another one since that would mean maintaining do different branches.

This is a case were we would want to redefine the `start` element of our schema. To do so, we would use an `include` pattern and embed the definitions which must be substitute to the one from the included grammar in the `include` pattern itself:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="library.rng">
    <start>
      <ref name="element-book"/>
    </start>
  </include>
</grammar>
```

Or:

```
include "library.rnc" {
  start = element-book
}
```

Note how the new definitions are embedded in the `include` pattern: the content of the `include` pattern is where all the redefinitions must be written. This short schema is including all the definitions from "library.rng" and redefining the `start` pattern. It validates instance documents with a `book` root element and since we are performing an inclusion instead of a copy, we will inherit any modification done on "library.rng".

We have been able to redefine the `start` pattern, but each named pattern can be redefined using the same syntax. Let's say for instance that I am not happy with the definition of the `element-name` pattern and want to check that the name is shorter than 80 characters. If I don't want (or can't) modify the original schema, I can include it and redefine this pattern:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <include href="library.rng">
    <define name="element-name">
      <element name="name">
        <data type="token">
          <param name="maxLength">80</param>
        </data>
      </element>
    </define>
  </include>
</grammar>
```

Or:

```
include "library.rnc" {
  element-name = element name { xsd:token{maxLength = "80"} }
}
```

Here again, the grammar of "library.rnc" is merged with the grammar of the new schema (which happens to be empty) but before the merge, the definitions which are embedded in the `include` pattern are substituted to the original definitions.

The new definition can be as different from the original one as I want. Without wanting to argue that it would be a good practice, I could for instance redefine `attribute-available` and replace the attribute by an element:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://w
  <include href="library.rng">
    <define name="attribute-available">
      <element name="available">
        <data type="boolean"/>
      </element>
    </define>
  </include>
</grammar>
```

Or:

```
include "library.rnc" {
  attribute-available = element available { xsd:boolean }
}
```

That would be rather confusing (the named pattern is called `attribute-available` and it's now describing an element) but the schema is perfectly valid and describes instance documents where the `available` attribute is replaced by an `available` element. This could also be used to remove this attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="library.rng">
    <define name="attribute-available">
      <empty/>
    </define>
  </include>
</grammar>
```

Or:

```
include "library.rnc" {
  attribute-available = empty
}
```

Note how we are using here a new pattern named `empty`. This pattern will match only text nodes made of non significant white spaces) and it will have the same effect than if the named pattern had been removed from the schema.

I have said that `include` patterns have the effect to merge the content of their grammar, after replacement of the patterns to redefine, with the content of the current grammar. This means that these redefinition can make references to any definition from either the including or the included grammars.

If we want to add zero or more email addresses to out `author` element while keeping a flat structure, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <include href="library.rng">

    <define name="element-author">
      <element name="author">
        <ref name="content-person"/>
        <optional>
          <ref name="element-died"/>
        </optional>
        <zeroOrMore>
          <ref name="element-email"/>
        </zeroOrMore>
      </element>
    </define>

  </include>

  <define name="element-email">
    <element name="email">
      <data type="anyURI">
        <param name="pattern">mailto:.*</param>
      </data>
    </element>
  </define>
</grammar>
```

Or:

```
include "library.rnc" {
  element-author =
    element author { content-person, element-died?, element-email* }
}
element-email =
  element email {
    xsd:anyURI { pattern = "mailto:.*" }
  }
```

Here, in the redefinition of the `element-author` pattern, we are making three references to three named patterns: `content-person` and `element-died` are defined in "library.rng", i.e. the grammar which is included and the third one, `element-email` is defined in the top level grammar i.e. the including grammar.

Combining definitions

When we've replaced the definitions in our previous examples, the original definition was completely replaced by the new one and this can make the maintenance of these schemas more complicated than it should be. In the last example, if the included schema (library.rng) is updated and the definition of `element-author` changed to add a new element to include a telephone number, this addition is lost if we do not add it explicitly in the including schema. As far as the `element-author` pattern

is concerned, this redefinition is no better than a copy paste and we'd benefit using a mechanism more similar to inheritance.

If we want to keep the definition from the included grammar, we can combine a new definition with the existing one instead of replacing it. Unlike redefinition, combination of `start` and named pattern do not take place in the `include` pattern and is done at the level of the including grammar. Actually, it isn't even necessary to include a grammar to combine definitions, but the main interest of combining definitions is to combine new definitions with existing ones from included grammars.

There are two options to combine definitions: by `choice` and by `interleave`.

Combining by choice

When definitions are combined by choice, the result is similar to using a `choice` pattern between the content of the definitions.

A use case for this would be to define a schema accepting either a `library` or a `book` element from the schema used in the previous section. In the XML syntax, combining by choice is done through a `combine` attribute:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="library.rng"/>
  <start combine="choice">
    <ref name="element-book"/>
  </start>
</grammar>
```

In the compact syntax, combining by choice is done through using the `|=` operator (instead of `=`) in the definition:

```
include "library.rnc"
start |= element-book
```

Note that in both cases, the combination is done outside of the inclusion. It's effect is to add a choice between the content of the `start` pattern which definition is now equivalent to :

```
  <start>
    <choice>
      <ref name="element-library"/>
      <ref name="element-book"/>
    </choice>
  </start>
```

Or:

```
start = element-library | element-book
```

The logic behind this combination is to allow the content model corresponding to the original pattern and in addition to allow a different content. This is different for the logic behind pattern redefinitions where the original pattern was replaced by a new one.

Named patterns can be combined too and if we wanted to accept either an `available` attribute or element, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <include href="library.rng"/>

  <define name="attribute-available" combine="choice">
    <element name="available">
      <data type="boolean"/>
    </element>
  </define>

</grammar>
```

Or:

```
include "library.rnc"
attribute-available |= element available { xsd:boolean }
```

Another interesting and common case is if we want to make this attribute optional and this can be achieved by combining this pattern by choice with an `empty` pattern:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">

  <include href="library.rng"/>

  <define name="attribute-available" combine="choice">
    <empty/>
  </define>

</grammar>
```

Or:

```
include "library.rnc"
attribute-available |= empty
```

Combining by interleave

We have seen how the "old" pattern could be replaced by a new one using pattern redefinition and how we could give the choice between an "old" definition and a new one using a combination by choice. The last option is to combine by interleave and the logic here is to allow to add pieces to the original content model and to let these pieces been interleaved, i.e. added anywhere before, after and between the sub patterns of the original pattern.

Do you remember the `email` element that we had added to the content of the `author` element using a redefinition? We can also use a combination by interleave to add this email pattern to the `content-person` pattern:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <include href="library.rng"/>

  <define name="content-person" combine="interleave">
    <zeroOrMore>
      <ref name="element-email"/>
    </zeroOrMore>
  </define>

  <define name="element-email">
    <element name="email">
      <data type="anyURI">
        <param name="pattern">mailto:.*</param>
      </data>
    </element>
  </define>

</grammar>
```

Or:

```
include "library.rnc"
content-person &= element-email *
element-email =
  element email {
    xsd:anyURI { pattern = "mailto:.*" }
  }
```

The effect of this combination is that the `content-model` pattern is now equivalent to an `interleave` pattern embedding both the original and the new definition, i.e.:

```
  <define name="content-person">
    <interleave>
      <group>
        <ref name="attribute-id"/>
        <ref name="element-name"/>
        <optional>
          <ref name="element-born"/>
        </optional>
      </group>
      <zeroOrMore>
        <ref name="element-email"/>
      </zeroOrMore>
    </interleave>
  </define>
```

Or:

```
content-person =
  (attribute-id, element-name, element-born?) & element-email *
```

The effect of this definition is thus to allow any number of `email` elements before the `name` element, between the `name` element and the `born` element and after the "born element".

The logic here is to allow extension by adding new content anywhere in the original definition. This is neat and safe if the applications which read the documents are coded to ignore what they don't know. In our example, if I have designed an application to read the original content model, this application will be just fine with the new content model if it ignores the `email` elements which have been added.

We have seen how a combination by choice can be used to turn a pattern into being optional. Combination by interleave cannot reverse the process, but it can turn a pattern into being forbidden! If we don't want to end up with a schema which won't validate any instance document, we must take care to do so on a pattern which reference is made optional, such as the `element-died` pattern:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="library.rng"/>
  <define name="element-died" combine="interleave">
    <notAllowed/>
  </define>
</grammar>
```

Or:

```
include "library.rnc"
element-died &= notAllowed
```

Here, we are interleaving a new pattern, `notAllowed` with the content of the named pattern `element-died`. The effect of this operation is that this pattern will not match any content model any longer. This is OK since the reference to the `element-died` in the definition of the `author` element is optional and the effect is that a document can be valid per the resulting schema only if there is no `died` element.

What about combining `start` patterns by interleave? This may seem weird and even illegal since we have seen `start` patterns in a context where they are using to define the root element of XML documents and that a well formed XML document can only have one root element.

Another use case where combining by interleave is handy and very widely used is to add attributes to a named pattern. In this case, the fact that interleave is unordered doesn't make any difference since attributes are always unordered.

Why can't we combine definitions by group?

We have seen how to combine definitions by `interleave` and `choice` and since `group` is the third compositor, we might be tempted to combine definitions by `group`. Unfortunately, definitions of named patterns are declarations and since the relative order of these declarations is considered as not significant, combining definitions by group wouldn't give reliable results and has thus been forbidden. This issue doesn't happen with `choice` and `interleave` compositors since the relative order of their children elements is not significant for a schema.

A real world example: XHTML 2.0

Let's leave our library for a while to look at XHTML. The modularization of XHTML 1.1, i.e. the fact to split XHTML 1.0 which was described as a monolithic DTD into a set of independent modules described as in independent DTDs that can be combined together to create as many flavor as people may want is one of the most challenging exercise for schema languages. In their Working Drafts, the

W3C HTML Working Group -now in charge of XHTML- has published a set of Relax NG schemas to describe XHTML 2.0 and its many modules which illustrates the flexibility of Relax NG to perform this type of exercises.

The solution chosen by XHTML 2.0 (see http://www.w3.org/TR/xhtml2/xhtml20_relax.html#a_xhtml20_relaxng for more detail) is to define each module by its own schema and include all these modules in a top level schema (called the "RELAX NG XHTML 2.0 Driver") :

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar ns="http://www.w3.org/2002/06/xhtml2"
        xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:x="http://www.w3.org/1999/xhtml">



 <x:h1>RELAX NG schema for XHTML 2.0</x:h1>



 <x:pre>
   Copyright &#xA9;2003 W3C&#xAE; (MIT, ERCIM, Keio), All Rights Reserved.



     Editor:   Masayasu Ishikawa &lt;mimasa@w3.org&gt;
     Revision: $Id: ch10.xml 95 2003-09-09 17:55:14Z vdv $



   Permission to use, copy, modify and distribute this RELAX NG schema
   for XHTML 2.0 and its accompanying documentation for any purpose and
   without fee is hereby granted in perpetuity, provided that the above
   copyright notice and this paragraph appear in all copies. The copyright
   holders make no representation about the suitability of this RELAX NG
   schema for any purpose.



   It is provided "as is" without expressed or implied warranty.
   For details, please refer to the W3C software license at:



     <x:a href="http://www.w3.org/Consortium/Legal/copyright-software"
     >http://www.w3.org/Consortium/Legal/copyright-software</x:a>
 </x:pre>



 <div>
   <x:h2>XHTML 2.0 modules</x:h2>
```

```
<x:h3>Attribute Collections Module</x:h3>
<include href="xhtml-attribs-2.rng"/>



<x:h3>Structure Module</x:h3>
<include href="xhtml-struct-2.rng"/>



<x:h3>Block Text Module</x:h3>
<include href="xhtml-blktext-2.rng"/>



<x:h3>Inline Text Module</x:h3>
<include href="xhtml-inltext-2.rng"/>



<x:h3>Hypertext Module</x:h3>
<include href="xhtml-hypertext-2.rng"/>



<x:h3>List Module</x:h3>
<include href="xhtml-list-2.rng"/>



<x:h3>Linking Module</x:h3>
<include href="xhtml-link-2.rng"/>



<x:h3>Metainformation Module</x:h3>
<include href="xhtml-meta-2.rng"/>



<x:h3>Object Module</x:h3>
<include href="xhtml-object-2.rng"/>



<x:h3>Scripting Module</x:h3>
<include href="xhtml-script-2.rng"/>
```

```
  <x:h3>Style Attribute Module</x:h3>
  <include href="xhtml-inlstyle-2.rng"/>



  <x:h3>Style Sheet Module</x:h3>
  <include href="xhtml-style-2.rng"/>



  <x:h3>Tables Module</x:h3>
  <include href="xhtml-table-2.rng"/>



  <x:h3>Support Modules</x:h3>



  <x:h4>Datatypes Module</x:h4>
  <include href="xhtml-datatypes-2.rng"/>



  <x:h4>Events Module</x:h4>
  <include href="xhtml-events-2.rng"/>



  <x:h4>Param Module</x:h4>
  <include href="xhtml-param-2.rng"/>



  <x:h4>Caption Module</x:h4>
  <include href="xhtml-caption-2.rng"/>
</div>



<div>
  <x:h2>XML Events module</x:h2>
  <include href="xml-events-1.rng"/>
</div>



<div>
  <x:h2>Ruby module</x:h2>
  <include href="full-ruby-1.rng"/>
```

```
  </div>



  <div>
    <x:h2>XForms module</x:h2>
    <x:p>To-Do: work out integration of XForms</x:p>
    <!--include href="xforms-1.rng"/-->
  </div>



 </grammar>
```

Don't worry for the moment about the `ns` attribute which we'll see in "Chapter 11: Namespaces" nor about the foreign (non Relax NG) namespaces and the `div` elements which we'll see in "Chapter 13: Annotating Schemas". One of these modules, the "Structure Module" defines the basic structure of a XHTML 2.0 document. For instance, the `head` element is defined as:

```
   <define name="head">
      <element name="head">
        <ref name="head.attlist"/>
        <ref name="head.content"/>
      </element>
    </define>



    <define name="head.attlist">
      <ref name="Common.attrib"/>
    </define>



    <define name="head.content">
      <ref name="title"/>
    </define>
```

Or:

```
  head = element head { head.attlist, head.content }
   head.attlist = Common.attrib
   head.content = title
```

This shows another design decision which is, for each element, to define a named pattern with the same name than the element (here `head`) and two separated named patterns to define the list of its attributes (here `head.attlist`) and its content (here `head.content`). This design decision makes it easy for other modules to add new elements and attributes just by combining these named patterns by interleave. For instance, the "Metainformation Module" adds a `meta` element to the content of the `head` element by combining by interleave the `head.content` pattern with zero or more `meta` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:x="http://www.w3.org/1999/xhtml">



 <x:h1>Metainformation Module</x:h1>



 <div>
   <x:h2>The meta element</x:h2>



   <define name="meta">
     <element name="meta">
       <ref name="meta.attlist"/>
       <choice>
         <ref name="Inline.model"/>
         <oneOrMore>
           <ref name="meta"/>
         </oneOrMore>
       </choice>
     </element>
   </define>



   <define name="meta.attlist">
     <ref name="Common.attrib"/>
     <optional>
       <attribute name="name">
         <ref name="NMTOKEN.datatype"/>
       </attribute>
     </optional>
   </define>
 </div>



 <define name="head.content" combine="interleave">
   <zeroOrMore>
     <ref name="meta"/>
   </zeroOrMore>
 </define>



</grammar>
```

Or:

```
namespace x = "http://www.w3.org/1999/xhtml"



meta = element meta { meta.attlist, (Inline.model | meta+) }
meta.attlist =
  Common.attrib,
  attribute name { NMTOKEN.datatype }?
head.content &= meta*
```

The fact that the content models are combined by interleave guarantees the independence between modules: we can add or remove modules independently of each other and also the independence of the resulting schema over the order in which the different modules are included in the top level schema: we can switch the "Metainformation Module" and the "Scripting Module" which both add content into the `head` element without any impact on the set of valid documents.

This modularity fully relies on combinations by interleave and Relax NG would have no easy solution if we wanted to add stuff to what has already be defined in the `head` element. Of course, if we are interested only by the "Structure Module" and want to add a `foo` element after the `title` element, we can redefine `head.content`:

```
<include href="xhtml-struct-2.rng">
  <define name="head.content">
    <ref name="title"/>
    <element name="foo">
      <empty/>
    </element>
  </define>
</include>
```

But this won't take into account all the content added by the other modules into the `head` element.

Other options

What if we really needed a feature which is really missing in Relax NG to create our building blocks? What if, for instance, we needed to reuse a name class or a datatype parameter defined once and only once in multiple locations of a schema?

If this was an absolute requirement, which is not often the case, we would have to use non Relax NG tools or features and what's unique with Relax NG compared to DTDs or W3C XML Schema is that we have two possible syntaxes, leaving the option to use either XML mechanisms with the XML syntax or plain text tools with the compact syntax.

There is no limit to the tools we may want to use to produce our result, but let's set up a possible use case and some examples of implementations.

A possible use case

Let's just say we want to set the set of possible characters in our documents and that we want to implement this rule in our Relax NG schemas. The pattern we might have in mind to perform this restriction could be the one we've seen as an example in "Chapter 9: W3C XML Schema Regular Expressions". It's not very complex but not very simple either:

```
pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
```

And of course, we would like to be able to easily update it if we had to and wouldn't want to have to copy it in each data type definition and we would like to be able to use this pattern in different contexts over different data types and eventually combined to other parameters.

XML tools

All the flavors of XML parsed entities (internal or external and in the internal DTD or in an external DTD) may be used in this case. Using internal entities in an internal DTD, we could for instance write:

```
<?xml version = '1.0' encoding = 'utf-8' ?>
<!DOCTYPE element [
<!ENTITY validChars "<param name='pattern'>[\p{IsBasicLatin}\p{IsLatin-1Supple
]>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
 <oneOrMore>
  <element name="book">
   <attribute name="id">
    <data type="NMTOKEN">&validChars;</data>
   </attribute>
   <attribute name="available">
    <data type="boolean"/>
   </attribute>
   <element name="isbn">
    <data type="NMTOKEN">&validChars;</data>
   </element>
   <element name="title">
    <attribute name="xml:lang">
     <data type="language"/>
    </attribute>
    <data type="token">&validChars;</data>
   </element>
   <zeroOrMore>
    <element name="author">
     <attribute name="id">
      <data type="NMTOKEN">&validChars;</data>
     </attribute>
     <element name="name">
      <data type="token">&validChars;</data>
     </element>
     <element name="born">
      <data type="date"/>
     </element>
     <optional>
      <element name="died">
       <data type="date"/>
      </element>
     </optional>
    </element>
   </zeroOrMore>
   <zeroOrMore>
    <element name="character">
     <attribute name="id">
      <data type="NMTOKEN">&validChars;</data>
     </attribute>
```

```
      <element name="name">
       <data type="token">&validChars;</data>
      </element>
      <element name="born">
       <data type="date"/>
      </element>
      <element name="qualification">
       <data type="token">&validChars;</data>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

The trickery here is to define an entity for the parameter:

```
<!ENTITY validChars "<param name='pattern'>[\p{IsBasicLatin}\p{IsLatin-1Supple
```

And to use this entity where we need it, for instance:

```
       <data type="token">&validChars;</data>
```

What about the compact syntax? The compact syntax doesn't support entities but if I convert this schema into the compact syntax I just get:

```
element library {
  element book {
    attribute id {
      xsd:NMTOKEN {
        pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
      }
    },
    attribute available { xsd:boolean },
    element isbn {
      xsd:NMTOKEN {
        pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
      }
    },
    element title {
      attribute xml:lang { xsd:language },
      xsd:token {
        pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
      }
    },
    element author {
      attribute id {
        xsd:NMTOKEN {
          pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
        }
      },
      element name {
        xsd:token {
```

```
            pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
          }
        },
        element born { xsd:date },
        element died { xsd:date }?
      }*,
      element character {
        attribute id {
          xsd:NMTOKEN {
            pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
          }
        },
        element name {
          xsd:token {
            pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
          }
        },
        element born { xsd:date },
        element qualification {
          xsd:token {
            pattern = "[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*"
          }
        }
      }*
    }+
}
```

This means that as long as I keep the XML version as my reference for this schema, I can easily get the compact syntax but can't go the other way round (compact to XML) without loosing my entity definition: the fact that I am using a XML mechanism has broken the round tripping between the two syntaxes.

Other XML tools (such as XInclude or even just writing the schema as a XSLT transformation) could be used with pretty much the same effect. Depending on the case, these solutions will be supported straight away by the parser which will parse the Relax NG schema (like this is the case with out internal entity) or will require a first phase during which your schema is compiled into a fully compatible Relax NG schema.

As an example, let's use XSLT. When you need to do simple stuff, XSLT has a simplified syntax where the `xsl:stylesheet` and `xsl:template` elements may be omitted (exactly like the Relax NG `grammar` and `start` elements may be omitted in a simple Relax NG schema). That means that if we just want to use XSLT for its simplest features (here only to expend the values of variables), we can write our schema as:

```
<?xml version = '1.0' encoding = 'utf-8' ?>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xsl:version="1.0">
  <xsl:variable name="validChars">
    <param name='pattern'>[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*</param>
  </xsl:variable>
  <oneOrMore>
   <element name="book">
    <attribute name="id">
     <data type="NMTOKEN"><xsl:copy-of select="$validChars"/></data>
    </attribute>
```

```
    <attribute name="available">
     <data type="boolean"/>
    </attribute>
    <element name="isbn">
     <data type="NMTOKEN"><xsl:copy-of select="$validChars"/></data>
    </element>
    <element name="title">
     <attribute name="xml:lang">
      <data type="language"/>
     </attribute>
     <data type="token"><xsl:copy-of select="$validChars"/></data>
    </element>
    <zeroOrMore>
     <element name="author">
      <attribute name="id">
       <data type="NMTOKEN"><xsl:copy-of select="$validChars"/></data>
      </attribute>
      <element name="name">
       <data type="token"><xsl:copy-of select="$validChars"/></data>
      </element>
      <element name="born">
       <data type="date"/>
      </element>
      <optional>
       <element name="died">
        <data type="date"/>
       </element>
      </optional>
     </element>
    </zeroOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id">
       <data type="NMTOKEN"><xsl:copy-of select="$validChars"/></data>
      </attribute>
      <element name="name">
       <data type="token"><xsl:copy-of select="$validChars"/></data>
      </element>
      <element name="born">
       <data type="date"/>
      </element>
      <element name="qualification">
       <data type="token"><xsl:copy-of select="$validChars"/></data>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

Applied to any XML document, this transformation will produce a Relax NG schema where the XSLT instruction:

```
<xsl:copy-of select="$validChars"/>
```

will have been replaced by the content of the variable $validChars, i.e.:

```
<param name='pattern'>[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*</param>
```

Text tools

Here, the situation is different and we can only use tools which, like the XSLT example just shown above, will require a first phase to produce a schema. One of the first tools which will come to mind to people familiar with C programming is the C pre processor (CPP). The syntax for defining a text replacement with CPP is `#define` and references are just done using the name of the definition. Something equivalent to our two previous examples could thus be:

```
#define VALIDCHARS  pattern = '[\p{IsBasicLatin}\p{IsLatin-1Supplement}]*'
element library {
  element book {
    attribute id {
      xsd:NMTOKEN {
        VALIDCHARS
      }
    },
    attribute available { xsd:boolean },
    element isbn {
      xsd:NMTOKEN {
        VALIDCHARS
      }
    },
    element title {
      attribute xml:lang { xsd:language },
      xsd:token {
        VALIDCHARS
      }
    },
    element author {
      attribute id {
        xsd:NMTOKEN {
          VALIDCHARS
        }
      },
      element name {
        xsd:token {
          VALIDCHARS
        }
      },
      element born { xsd:date },
      element died { xsd:date }?
    }*,
    element character {
      attribute id {
        xsd:NMTOKEN {
          VALIDCHARS
        }
      },
      element name {
        xsd:token {
          VALIDCHARS
        }
      },
```

```
        element born { xsd:date },
        element qualification {
          xsd:token {
            VALIDCHARS
          }
        }
      }*
    }+
  }
```

And, when compiled through CPP, this gives a fully valid Relax NG schema (compact syntax) where the occurrences of `VALIDCHARS` have been replaced by the parameter.

Chapter 12. Chapter 11: Namespaces

At this point, you may be wondering why we need a chapter about namespaces; after all since the very first example our schemas include an attribute from the xml:lang namespace and that doesn't seem like a big deal.

If you think about it more carefully, you'll see that namespaces are presenting two different challenges to schema languages. The first is syntactical: schema languages need to provide a way to define to which namespaces belong the elements and attributes which are described; the second is how schema languages can cope with the extensibility which is one of the objectives of XML namespaces.

In this chapter, we'll have a closer look at these two challenges before seeing how Relax NG addresses them.

A ten minutes guide to XML namespaces

Let's go back to the motivations beyond XML Namespaces. The first and basic one is to be a kind of replacement for the XML (SGML inherited) doctype public identifier and provide an way to identify which vocabulary, i.e. which set of names is being used in a document. The XML/SGML way of identifying the vocabulary used in our library would be to add a public identifier in the document type declaration such as:

```
<?xml version="1.0"?>
<!DOCTYPE library PUBLIC "-//ERICVANDERVLIST//DTD for library//EN" "library.dt
<library>
.../...
</library>
```

One of the downsides of this method is that it doesn't totally decouple the identification of the vocabulary ("-//ERICVANDERVLIST//DTD for library//EN") from the location of the DTD describing the vocabulary ("library.dtd"): I can't identify without giving the location! This is normal since in fact, the identification is the identification of the DTD rather than the identification of the abstract set of names.

The first goal of XML namespaces is to provide identifiers for this abstract notion of "vocabulary", "set of names" or simply "namespaces" without linking these identifiers to any technical implementation (DTD, schemas or whatever) defining or enforcing what they are. These identifiers are no longer public identifiers like those used in doctype declarations but URIs (or rather to be picky "URI references") and to assign a namespace to all the elements from our example, we could write:

```
<?xml version="1.0"?>
<library xmlns="http://eric.van-der-vlist.com/ns/library">
.../...
</library>
```

The identifier for my namespace is the string "http://eric.van-der-vlist.com/ns/library" and this address doesn't have to host any document. At the time I am writing these lines, this address doesn't lead to any web page at all and if you copy it into your favorite web browser, you'd just get a "404 Not Found" error. The assumption is just that I use it only if I own the domain and that I won't use it for several different namespaces. Later on, I could publish something at that location, either a documentation (formal or not) or any kind of schema; that's not forbidden by the namespaces recommendation, this can be useful but the whole subject is highly controversial. Also note that XML namespaces per see do not define any way to associate resources such as schemas or documentations with a namespace URI.

The namespace declaration (xmlns="http://eric.van-der-vlist.com/ns/library") has been done for the document element ("library") and is inherited by all its children elements.

The second goal of XML namespaces is to provide a way to mix elements and attributes from different namespaces in a single document. In our library for instance, the library and book elements use a vocabulary specific to libraries while the author element could use a vocabulary for human resources and the character element be a mixed of both: the character element itself and the qualification element would be from the library namespace while the `name` and `born` elements would be from the HR vocabulary.

Leveraging on declaration inheritance, this could be achieved using the `xmlns` declaration has we have already seen:

```
<?xml version="1.0"?>
<library xmlns="http://eric.van-der-vlist.com/ns/library">
 <book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
  <author id="CMS" xmlns="http://eric.van-der-vlist.com/ns/person">
   <name>Charles M Schulz</name>
   <born>1922-11-26</born>
   <died>2000-02-12</died>
  </author>
  <character id="PP">
   <name xmlns="http://eric.van-der-vlist.com/ns/person">Peppermint Patty</nam
   <born xmlns="http://eric.van-der-vlist.com/ns/person">1966-08-22</born>
   <qualification>bold, brash and tomboyish</qualification>
   </character>
  <character id="Snoopy">
   <name xmlns="http://eric.van-der-vlist.com/ns/person">Snoopy</name>
   <born xmlns="http://eric.van-der-vlist.com/ns/person">1950-10-04</born>
   <qualification>extroverted beagle</qualification>
  </character>
  <character id="Schroeder">
   <name xmlns="http://eric.van-der-vlist.com/ns/person">Schroeder</name>
   <born xmlns="http://eric.van-der-vlist.com/ns/person">1951-05-30</born>
   <qualification>brought classical music to the Peanuts strip</qualification>
  </character>
  <character id="Lucy">
   <name xmlns="http://eric.van-der-vlist.com/ns/person">Lucy</name>
   <born xmlns="http://eric.van-der-vlist.com/ns/person">1952-03-03</born>
   <qualification>bossy, crabby and selfish</qualification>
  </character>
 </book>
</library>
```

But we see that it would rapidly become very verbose and XML namespaces provide a way to assign prefixes to namespaces which can then be used to prefix the names of the elements (and attributes) to declare their namespaces. The namespace declared using the `xmlns` attribute is called the "default namespace" since it is assigned to elements which have no prefix. The document above could be rewritten using the default namespace for the library and assigning a prefix to the other namespace:

```
<?xml version="1.0"?>
<library
  xmlns="http://eric.van-der-vlist.com/ns/library"
  xmlns:hr="http://eric.van-der-vlist.com/ns/person">
```

```
 <book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
  <hr:author id="CMS">
   <hr:name>Charles M Schulz</hr:name>
   <hr:born>1922-11-26</hr:born>
   <hr:died>2000-02-12</hr:died>
  </hr:author>
  <character id="PP">
   <hr:name>Peppermint Patty</hr:name>
   <hr:born>1966-08-22</hr:born>
   <qualification>bold, brash and tomboyish</qualification>
   </character>
  <character id="Snoopy">
   <hr:name>Snoopy</hr:name>
   <hr:born>1950-10-04</hr:born>
   <qualification>extroverted beagle</qualification>
  </character>
  <character id="Schroeder">
   <hr:name>Schroeder</hr:name>
   <hr:born>1951-05-30</hr:born>
   <qualification>brought classical music to the Peanuts strip</qualification>
  </character>
  <character id="Lucy">
   <hr:name>Lucy</hr:name>
   <hr:born>1952-03-03</hr:born>
   <qualification>bossy, crabby and selfish</qualification>
  </character>
 </book>
</library>
```

The value of the prefix (`hr` in this example) is chosen arbitrarily by the author of the XML document and should be considered as not significant by applications conform the the namespaces specification. Here, I have chosen `hr` standing for "Human Resources" to make it easier to remember for human readers, but I could have chosen any other value (such as `foo`) for this prefix as long as, of course, the prefix matches its definition.

We could also prefer, for symmetry, use a prefix for both namespaces:

```
<?xml version="1.0"?>
<lib:library
  xmlns:lib="http://eric.van-der-vlist.com/ns/library"
  xmlns:hr="http://eric.van-der-vlist.com/ns/person">
 <lib:book id="b0836217462" available="true">
  <lib:isbn>0836217462</lib:isbn>
  <lib:title xml:lang="en">Being a Dog Is a Full-Time Job</lib:title>
  <hr:author id="CMS">
   <hr:name>Charles M Schulz</hr:name>
   <hr:born>1922-11-26</hr:born>
   <hr:died>2000-02-12</hr:died>
  </hr:author>
  <lib:character id="PP">
   <hr:name>Peppermint Patty</hr:name>
   <hr:born>1966-08-22</hr:born>
   <lib:qualification>bold, brash and tomboyish</lib:qualification>
   </lib:character>
  <lib:character id="Snoopy">
```

```
    <hr:name>Snoopy</hr:name>
    <hr:born>1950-10-04</hr:born>
    <lib:qualification>extroverted beagle</lib:qualification>
   </lib:character>
   <lib:character id="Schroeder">
    <hr:name>Schroeder</hr:name>
    <hr:born>1951-05-30</hr:born>
    <lib:qualification>brought classical music to the Peanuts strip</lib:qualif
   </lib:character>
   <lib:character id="Lucy">
    <hr:name>Lucy</hr:name>
    <hr:born>1952-03-03</hr:born>
    <lib:qualification>bossy, crabby and selfish</lib:qualification>
   </lib:character>
  </lib:book>
 </lib:library>
```

Note that for a namespace aware application, these three documents are considered equivalent: the prefixes are only shortcuts to associate a namespace URI and a "local name" (the part of the name which is after the colon) to disambiguate this name from eventual synonyms defined in other namespaces.

Up to now, we've spoken of elements and attributes are given a similar yet special treatment. Similar in that attribute names can be prefixed to show that they belong to a namespace but special since the default namespace doesn't apply to them and that the attributes which have no prefix are considered to have no namespace URI but still "belong" to the namespace of their parent element. The reason for this is that attributes are supposed to be used to provide meta-data qualifying their parent element rather than to contain actual information and being qualifiers, it has been considered that by default they "belong" to the same vocabulary than their parent elements. This is the reason why I have kept the `id` and `available` attributes without prefix in my there examples.

The last goal of XML namespaces (and the motivation for taking that much pain to allow several namespaces in a single document) is to facilitate the development of independent (or semi-independent) vocabularies which can be used as building blocks. One of the ideas is that if applications are cleanly developed and just drop elements and attributes which they don't understand documents can be extended without breaking existing applications.

For instance, in our library we've not defined the publisher of the book. We can add a `publisher` element in our namespace, but instead we might want to use the definition given by the Dublin Core Metadata Initiative (DCMI) and use their namespaces to write:

```
<?xml version="1.0"?>
<library
  xmlns="http://eric.van-der-vlist.com/ns/library"
  xmlns:hr="http://eric.van-der-vlist.com/ns/person"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
 <book id="b0836217462" available="true">
  <isbn>0836217462</isbn>
  <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
  <dc:publisher>Andrews Mc Meel Publishing</dc:publisher>
  .../...
 </book>
</library>
```

A double benefit is expected from doing so. Everyone understands that the `publisher` element is corresponding to the definition given by the DCMI:

```
URI:            http://purl.org/dc/elements/1.1/publisher
Namespace:      http://purl.org/dc/elements/1.1/
Name:           publisher
Label:          Publisher
Definition:     An entity responsible for making the resource available
Comment:        Examples of a Publisher include a person, an organization,
                or a service.
                Typically, the name of a Publisher should be used to
                indicate the entity.
Type of term:   http://dublincore.org/usage/documents/principles/#element
Status:         http://dublincore.org/usage/documents/process/#recommended
Date issued:    1998-08-06
Date modified:  2002-10-04
Decision:       http://dublincore.org/usage/decisions/#Decision-2002-03
This version:   http://dublincore.org/usage/terms/dc/#publisher-004
Replaces:       http://dublincore.org/usage/terms/dc/#publisher-003
```

Again, note that the mechanism to retrieve this definition is not specified by the XML namespace recommendation. The second benefit is that if my application has been implemented to skip elements and attributes from unsupported namespaces, the addition of this `dc:publisher` element won't break anything.

The two challenges of namespaces

The progression followed in our "10 minutes guide to namespaces" has carefully be designed to let you guess what these two challenges are and you'll probably have already seen where I wanted to go!

The first issue is to provide the ad-hoc mechanisms to associate namespace URIs to patterns describing elements and attributes and this will be described in the next section.

The second issue is to provide mechanisms to write extensible schemas for those applications skipping unknown namespaces. Of course, writing extensible schemas is not limited to multi namespaces documents and we will see more of it in our next chapter, but we will start introducing the mechanism called "name classes" which is the key to extensibility with Relax NG in the last section of this chapter to fully cover the case of namespaces.

Namespace declarations

Namespace declarations in a Relax NG schema are following the same principles as namespace declarations in an instance document with some differences in the syntax and we will also find both the possibility to use default namespaces and the possibility to use prefixes.

Using default namespaces

The namespace expected in the instance document can be defined through the `ns` attribute which is an "inherited" attribute like the `datatypeLibrary` attribute seen before. Being inherited means that we can define it in the document element of the schema if it remains the same all over the schema. For instance, to write a schema for the example where all the library is using the same namespace, we could write:

```
<?xml version="1.0" encoding="utf-8"?>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library"
         ns="http://eric.van-der-vlist.com/ns/library">
 <oneOrMore>
```

```
  <element name="book">
   <attribute name="id"/>
   .../...
  </element>
 </oneOrMore>
</element>
```

Or, using the compact syntax:

```
default namespace = "http://eric.van-der-vlist.com/ns/library"



element library
{
   element book
   {
      attribute id { text },
      .../...
      }*
   }+
}
```

Note that the definition of the default namespace in a Relax NG schema do not apply to attributes (exactly as the default namespace doesn't apply to attributes in instance documents).

Exactly as default namespaces could be used and changed all over a multi namespace document, when we are using the XML syntax the `ns` attribute can be changed in a schema and to validate the documents with two namespaces show in our "10 minutes guide to namespaces", we could write:

```
<?xml version="1.0" encoding="utf-8"?>
<element xmlns="http://relaxng.org/ns/structure/1.0"
 name="library"
 ns="http://eric.van-der-vlist.com/ns/library">
 <oneOrMore>
  <element name="book">
   <attribute name="id"/>
   <attribute name="available"/>
   <element name="isbn">
    <text/>
   </element>
   <element name="title">
    <attribute name="xml:lang"/>
    <text/>
   </element>
   <zeroOrMore>
    <element name="author" ns="http://eric.van-der-vlist.com/ns/person">
     <attribute name="id"/>
     <element name="name">
      <text/>
     </element>
     <element name="born">
      <text/>
     </element>
```

```
      <optional>
       <element name="died">
        <text/>
       </element>
      </optional>
     </element>
    </zeroOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id"/>
      <element name="name" ns="http://eric.van-der-vlist.com/ns/person">
       <text/>
      </element>
      <element name="born" ns="http://eric.van-der-vlist.com/ns/person">
       <text/>
      </element>
      <element name="qualification">
       <text/>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

The compact syntax does not provide such a way to redefine the default namespace and defining prefixes will be the preferred way to define schemas with multiple namespaces when using the compact syntax.

Since the three variations used to write the document with the two namespaces in our "10 minutes guide" are considered equivalent for namespaces aware applications, the schema which we've just written will validate them indifferently. There is thus a complete independence between the prefixes and default namespaces used to write the instance document and those used in the schema and the match between namespaces is only done through matching the namespace URIs of each element and attribute.

Using prefixes

Although the definition of the default target namespace in a Relax NG is done through a `ns` attribute and thus do not rely on the declaration of the default namespace of the Relax NG document itself (in our examples, the default namespace of the Relax NG document is the Relax NG namespace), the declaration of the prefixes used as shortcut to the non default target namespaces is done through namespaces declarations. In other words, to define a `hr` prefix which will be used as a prefix for the namespaces in names or attributes of the instance, I declare this prefix through a `xmlns:hr` declaration as if I wanted to use it as a prefix for an element or attribute of the Relax NG document.

We can mix both default and non default namespaces in our schemas and write:

```
<?xml version="1.0" encoding="utf-8"?>
<element xmlns="http://relaxng.org/ns/structure/1.0"
 name="library"
 ns="http://eric.van-der-vlist.com/ns/library"
 xmlns:hr="http://eric.van-der-vlist.com/ns/person">
 <!-- The default target namespace is "http://eric.van-der-vlist.com/ns/library
 <oneOrMore>
  <element name="book">
```

```
   <attribute name="id"/>
   <attribute name="available"/>
   <element name="isbn">
    <text/>
   </element>
   <element name="title">
    <attribute name="xml:lang"/>
    <text/>
   </element>
   <zeroOrMore>
    <element name="hr:author">
    <!-- Here we are using a "hr" prefix to match "http://eric.van-der-vlist.c
     <attribute name="id"/>
     <element name="hr:name">
      <text/>
     </element>
     <element name="hr:born">
      <text/>
     </element>
     <optional>
      <element name="hr:died">
       <text/>
      </element>
     </optional>
    </element>
   </zeroOrMore>
   <zeroOrMore>
    <element name="character">
     <attribute name="id"/>
     <element name="hr:name">
      <text/>
     </element>
     <element name="hr:born">
      <text/>
     </element>
     <element name="qualification">
      <text/>
     </element>
    </element>
   </zeroOrMore>
  </element>
 </oneOrMore>
</element>
```

A namespace declaration is provided in the compact syntax to define namespace prefixes:

```
default namespace = "http://eric.van-der-vlist.com/ns/library"
namespace hr = "http://eric.van-der-vlist.com/ns/person"



element library
{
   element book
   {
      attribute id { text },
```

```
      attribute available { text },
      element isbn { text },
      element title { attribute xml:lang { text }, text },
      element hr:author
      {
         attribute id { text },
         element hr:name { text },
         element hr:born { text },
         element hr:died { text }?
      }*,
      element character
      {
         attribute id { text },
         element hr:name { text },
         element hr:born { text },
         element qualification { text }
      }*
   }+
}
```

Again, this schema will validate the three variations seen in the "10 minutes guide". In fact this schema validates exactly the same set of document than the schema using only default namespaces. The third variation would be to use prefixes for both namespaces:

```
<?xml version="1.0" encoding="utf-8"?>
<element xmlns="http://relaxng.org/ns/structure/1.0"
 name="lib:library"
 xmlns:lib="http://eric.van-der-vlist.com/ns/library"
 xmlns:hr="http://eric.van-der-vlist.com/ns/person">
 <oneOrMore>
  <element name="lib:book">
   <attribute name="id"/>
   <attribute name="available"/>
   <element name="lib:isbn">
    <text/>
   </element>
   <element name="lib:title">
    <attribute name="xml:lang"/>
    <text/>
   </element>
   <zeroOrMore>
    <element name="hr:author">
     <attribute name="id"/>
     <element name="hr:name">
      <text/>
     </element>
     <element name="hr:born">
      <text/>
     </element>
     <optional>
      <element name="hr:died">
       <text/>
      </element>
     </optional>
    </element>
   </zeroOrMore>
```

```
    <zeroOrMore>
     <element name="lib:character">
      <attribute name="id"/>
      <element name="hr:name">
       <text/>
      </element>
      <element name="hr:born">
       <text/>
      </element>
      <element name="lib:qualification">
       <text/>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

or:

```
namespace lib = "http://eric.van-der-vlist.com/ns/library"
namespace hr = "http://eric.van-der-vlist.com/ns/person"



element lib:library
{
   element lib:book
   {
      attribute id { text },
      attribute available { text },
      element lib:isbn { text },
      element lib:title { attribute xml:lang { text }, text },
      element hr:author
      {
         attribute id { text },
         element hr:name { text },
         element hr:born { text },
         element hr:died { text }?
      }*,
      element lib:character
      {
         attribute id { text },
         element hr:name { text },
         element hr:born { text },
         element lib:qualification { text }
      }*
   }+
}
```

And again, this schema is equivalent to the previous ones since it's validating all the variations of namespaces declarations in the instance documents.

Accepting "foreign namespaces"

The last schemas which we have seen will validate instance documents independently of the prefixes being used and they meet the first goals of the namespaces which are to disambiguate elements in multi-namespaces documents. However, they will fail to validate the instance document where we've added the `dc:publisher` element. We could easily update our schema to explicitly add this element to the content model of our `book` element, but that wouldn't make it an open schema accepting addition of elements from any other namespace.

Instead of some "magic feature" which would probably have been quite rigid, Relax NG has introduced a flexible and clever feature to let you define your own level of "openness". The idea to do so is to let you define your own "wildcard" and once you get it, you can include it wherever you want in your content model.

Constructing our wildcard

Before we start, let's define what we are trying to achieve! We want a named pattern allowing any element or attribute which do not belong to our `lib` and `hr` namespaces. We probably want to exclude elements and attributes with no namespaces: attributes because our own attributes have no namespace and we might want to differentiate them and elements because allowing elements without namespaces in a document using namespaces is kind of messy.

For the content model of these foreign elements, we have two main options:

- We can be liberal and accept anything including elements and attributes from the namespaces described in the schema.
- We can be more conservative and accept only foreign elements.

If we want to be liberal, we must define the inner content of the wildcard to formalize what `anything` is. In this case, this can be expressed as any number of elements (themselves containing `anything`), attributes and text in any order and this is a good candidate for a recursive named pattern:

```
<define name="anything">
  <zeroOrMore>
    <choice>
      <element>
        <anyName/>
        <ref name="anything"/>
      </element>
      <attribute>
        <anyName/>
      </attribute>
      <text/>
    </choice>
  </zeroOrMore>
</define>
```

or:

```
anything = ( element * { anything } | attribute * { text } | text )*
```

The only thing new here is the `anyName` element (XML syntax) or `*` operator (compact syntax) to replace the name of an element or attribute. This is the first example of a "name class" (i.e. a class of names) and we'll see that there are many ways to restrict this name class. Now that we have a named pattern to express what `anything` is, we can use it to define what "foreign" elements mean:

```
<define name="foreign-elements">
  <zeroOrMore>
    <element>
      <anyName>
        <except>
          <nsName ns=""/>
          <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
          <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
        </except>
      </anyName>
      <ref name="anything"/>
    </element>
  </zeroOrMore>
</define>
```

or:

```
default namespace lib = "http://eric.van-der-vlist.com/ns/library"
namespace local = ""
namespace hr = "http://eric.van-der-vlist.com/ns/person"



.../...



foreign-elements = element * - (local:* | lib:* | hr:*) { anything }*
```

To achieve our purpose, we have introduced two new elements embedded in the `anyName` name class: `except` (or `-` in the compact syntax) which has here the same meaning than with enumerations and `nsName` (xxx:* in the compact syntax) which means "any name from a namespace". When using the XML syntax, `nsName` takes a `ns` attribute while prefixes are used when using the compact syntax. This usage of prefixes in the compact syntax implies that declarations are added to define prefixes for the `lib` (which is also the default namespace) and `hr` namespaces but also for "no namespace" (here we have used the prefix `local`).

Note that name classes are not considered as patterns but as a specific set of elements with a specific purpose. A consequence of this statement is that name classes definitions cannot be placed within named patterns to be reused and that we had to repeat the same name class for both elements and attributes.

The same can be done to define foreign attributes:

```
<define name="foreign-attributes">
  <zeroOrMore>
    <attribute>
      <anyName>
        <except>
          <nsName ns=""/>
          <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
          <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
        </except>
```

```
        </anyName>
      </attribute>
    </zeroOrMore>
  </define>
```

or:

```
foreign-attributes = attribute * - (local:* | lib:* | hr:*) { text }*
```

And for our convenience, we can also define foreign nodes by combining foreign elements and attributes:

```
  <define name="foreign-nodes">
    <zeroOrMore>
      <choice>
        <ref name="foreign-attributes"/>
        <ref name="foreign-elements"/>
      </choice>
    </zeroOrMore>
  </define>
```

or:

```
foreign-nodes = ( foreign-attributes | foreign-elements )*
```

Using our wildcard

So far so good and now that we have defined what `foreign-nodes` are, we can use it to give more extensibility to our schema. To enable foreign-nodes where we have added the `dc:publisher` element, i.e. between the `title` and `author` elements, we could write (switching to a "flatter" style to make it more readable):

```
    <element name="book">
     <attribute name="id"/>
     <attribute name="available"/>
     <ref name="isbn-element"/>
     <ref name="title-element"/>
     <ref name="foreign-nodes"/>
     <zeroOrMore>
       <ref name="author-element"/>
     </zeroOrMore>
     <zeroOrMore>
       <ref name="character-element"/>
     </zeroOrMore>
   </element>
```

or:

```
book-element =
```

```
element book
{
   attribute id { text },
   attribute available { text },
   isbn-element,
   title-element,
   foreign-nodes,
   author-element*,
   character-element*
}
```

This would do the trick for the instance document shown above, but wouldn't validate a document where foreign nodes would be added at any other place, for instance between the `isbn` and `title` elements. We could insert a reference to the `foreign-nodes` pattern between all the elements, but this would be very verbose and if we think about it, we want here to `interleave` these foreign nodes between the content defined for the `book` element and that's a good opportunity to use the `interleave` pattern:

```
<element name="book">
  <interleave>
    <group>
      <attribute name="id"/>
      <attribute name="available"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
    </group>
    <ref name="foreign-nodes"/>
  </interleave>
</element>
```

or:

```
element book
{
   (
      attribute id { text },
      attribute available { text },
      isbn-element,
      title-element,
      author-element*,
      character-element*
   )
 & foreign-nodes
}
```

As seen in "Chapter 6: More Patterns", foreign nodes will be interleaved in the group which was previously defining the content of the book element: the order of the group is still enforced but foreign nodes may appear in any order and everywhere.

Where should we allow foreign nodes?

We may be tempted to allow these foreign nodes everywhere in our document, however if the extensibility which would be given is usually fine in elements such as book which already have children element, it's often considered a bad practice to do the same in elements which content is text only, such as the `isbn` element where this would transform a text content models into a mixed content model.

This is due to the weak support for mixed content models which we've already mentioned in "Chapter 6: More patterns" when we've discussed the limitations of the `mixed` pattern. A consequence of allowing foreign elements in `isbn` elements would be that the content of this element could not be considered a `data` any longer and that no datatypes nor restrictions could be applied any longer.

Beyond the limitation of Relax NG, applications would have to concatenate text nodes spread over the foreign elements and this is may be pretty verbose with tools such as XPath and XSLT.

A compromise which is often taken is to allow only foreign attributes in text content models, but that's not an issue for us since our `foreign-attributes` is ready for this purpose:

```
<element name="isbn">
  <ref name="foreign-attributes"/>
  <text/>
</element>
```

or:

```
element isbn { foreign-attributes, text }
```

A couple of traps to avoid

If most of the time, using our wildcards is straightforward there are some situations where this may lead to unexpected schema errors, especially with attributes which usage is subject to restrictions.

The first of the traps which I'd like to mention here is related to the fact that the definition of attributes cannot be duplicated in a schema and the following definition would be invalid:

```
element title { attribute xml:space, attribute xml:space, text } # this is inva
```

This seems to be pretty sensible since duplicate attributes are forbidden in the instance document. Unfortunately, the attribute `xml:space` is allowed by our `foreign-attributes` named template and we will get an error as well if we extend the definition of our title element without taking care and write:

```
element title { foreign-attributes, attribute xml:space, text } # this is also
```

To fix this error, we will need either to remove the xml:space attribute from the name class of our foreign attributes or to remove the implicit mention of xml:space in our definition and just write:

```
element title { foreign-attributes, text }
```

Of course, this doesn't remove the possibility to include a xml:space attribute in the `title` element since this attribute is a "foreign attribute" as defined in our named pattern.

The second trap is a level higher on the same line and is specific to the DTD compatibility `ID` datatype. In "Chapter 8: Datatype libraries", when we have seen this datatype, we have used it to define the book element:

```
<element name="book">
 <attribute name="id">
  <data datatypeLibrary="http://relaxng.org/ns/compatibility/datatypes/1.0"
 </attribute>
 .../...
</element>
```

or:

```
element book {
 attribute id {dtd:ID},
 .../...
}
```

Here again, we will generate an error if we add our foreign nodes and the reason for this error is that this datatype is emulating the DTD in all its aspects including the fact that if an element `book` is defined with an `id` attribute having a type `ID`, all the other definitions of an attribute `id` hosted by an element `book` must have the same type `ID`. The problem here is that, hidden in the definition of `anything`, there can be elements `book` having an attribute `id` of type `text` and this is considered as an error.

The only workaround if we want to use the DTD type `ID` is to remove this possibility from the named pattern `anything`. A fast solution would be to exclude our namespaces from the class names in `anything` but we can find more elaborated constructions with the elements which will be introduced in the next chapter.

Adding foreign nodes through combination

To add our foreign nodes, we have transformed:

```
<element name="book">
  <attribute name="id"/>
  <attribute name="available"/>
  <ref name="isbn-element"/>
  <ref name="title-element"/>
  <zeroOrMore>
    <ref name="author-element"/>
  </zeroOrMore>
  <zeroOrMore>
    <ref name="character-element"/>
  </zeroOrMore>
</element>
```

or:

```
element book
{
```

```
      attribute id { text },
      attribute available { text }
      isbn-element,
      title-element,
      author-element*,
      character-element*
    }
```

into:

```
    <element name="book">
      <interleave>
        <group>
          <attribute name="id"/>
          <attribute name="available"/>
          <ref name="isbn-element"/>
          <ref name="title-element"/>
          <zeroOrMore>
            <ref name="author-element"/>
          </zeroOrMore>
          <zeroOrMore>
            <ref name="character-element"/>
          </zeroOrMore>
        </group>
        <ref name="foreign-nodes"/>
      </interleave>
    </element>
```

or:

```
    element book
    {
       (
          attribute id { text },
          attribute available { text },
          isbn-element,
          title-element,
          author-element*,
          character-element*
       )
     & foreign-nodes
    }
```

and this operation could be done as a pattern combination by interleave if the content of the element `book` is described as a named pattern:

```
  <define name="book-content">
    <attribute name="id"/>
    <attribute name="available"/>
    <ref name="isbn-element"/>
    <ref name="title-element"/>
    <zeroOrMore>
      <ref name="author-element"/>
    </zeroOrMore>
```

```
    <zeroOrMore>
      <ref name="character-element"/>
    </zeroOrMore>
  </define>
```

or:

```
book-content =
   attribute id { text },
   attribute available { text },
   isbn-element,
   title-element,
   author-element*,
   character-element*
```

This pattern can then easily be extended as:

```
  <define name="book-content" combine="interleave">
    <ref name="foreign-nodes"/>
  </define>
```

or

```
book-content &= foreign-nodes
```

and used to define the `book` element:

```
    <element name="book">
      <ref name="book-content"/>
    </element>
```

or:

```
element book { book-content }
```

This combination can be done in a single document but this mechanism can also be used to extend a vocabulary through merging a grammar containing only these combinations.

Note that the exact same combination does also work for appending foreign attributes to the elements which have a text only content model.

Namespaces and building blocks, chameleon design

Back to XHTML 2.0

In "Chapter 10: Creating Building Blocks", we've seen the example of the schemas for XHTML 2.0 and I have urged you not to worry about the namespace declarations which hadn't been introduced yet. Let's have a closer look now that we know how to declare namespaces.

If we look for namespace declarations in the top level schema (the "driver"), we will find them only in the grammar element:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar ns="http://www.w3.org/2002/06/xhtml2"
        xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:x="http://www.w3.org/1999/xhtml">



 <x:h1>RELAX NG schema for XHTML 2.0</x:h1>
 .../...
   <x:h3>Structure Module</x:h3>
   <include href="xhtml-struct-2.rng"/>
  .../...
</grammar>
```

What do we see in this snippet?

- xmlns="http://relaxng.org/ns/structure/1.0" means that the default namespace of the schema as a XML document is "http://relaxng.org/ns/structure/1.0". That just means that elements without prefix in the schema as a XML document are Relax NG patterns.
- ns="http://www.w3.org/2002/06/xhtml2" defines the default namespace for the schema itself: the schema describes elements from the "http://www.w3.org/2002/06/xhtml2" namespace unless some other namespace is explicitly defined. Let's call it the "target namespace" to avoid any confusion with the default namespace of the schema considered as a XML document.
- xmlns:x="http://www.w3.org/1999/xhtml" defines that the prefix "x" is assigned to "http://www.w3.org/1999/xhtml". This declaration is used here to include XHTML documentation in the schema and we will see this in more detail in "Chapter 13: Annotating Schemas".

This would translate in the compact syntax as:

```
default namespace = "http://www.w3.org/2002/06/xhtml2"
namespace x = "http://www.w3.org/1999/xhtml"



.../...



include "xhtml-struct-2.rnc"
```

Let's now have a look at the module describing the structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:x="http://www.w3.org/1999/xhtml">
```

```
  <x:h1>Structure Module</x:h1>
  .../...



</grammar>
```

Or:

```
namespace x = "http://www.w3.org/1999/xhtml"



x:h1 [ "Structure Module" ]
start = html
...
```

The big difference with the top level schema is that the target namespace isn't defined in the schema defining the module.

How can that work? It's a feature common to the `include` and `externalRef` pattern than when no target namespace is defined in the imported schema, the target namespace from the schema performing the inclusion or external reference is used. In our case, that means that the target namespace from the driver ("http://www.w3.org/2002/06/xhtml2") is used by any module which do no specify a target namespace.

For this reason, schemas without target namespace are often called "chameleon schemas" since they take the target namespace of any context in which they are included or referenced.

In the compact syntax, an `inherit` qualifier has been added to specify a namespace must be inherited at inclusion or external reference time:

```
namespace xhtml2 = "http://www.w3.org/2002/06/xhtml2"
namespace x = "http://www.w3.org/1999/xhtml"



.../...



include "xhtml-struct-2.rnc" inherit = xhtml2
```

This `inherit` qualifier has the same role as a `ns` attribute in an `include` or externalRef" of the XML syntax.

Applicability to our library

Back to XHTML 2.0 and our library, we may want to include XHTML elements into the description of our library, for instance to allow the same content for the definition of our titles and qualification than

in the XHTML `p` element, i.e. what's described in the "Inline text" module as the `Inline.model` named pattern. The idea beside this mechanism of modules is that we can select the modules we'll want to include and we can follow this principle to pick just what we need and that's just what we'll do by including the common modules ("xhtml-attribs-2.rng" and "xhtml-datatypes-2.rng") and the "Inline Text" module (xhtml-inltext-2.rng"):

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="library"/>
  </start>
  <include href="xhtml-attribs-2.rng"/>
  <include href="xhtml-inltext-2.rng"/>
  <include href="xhtml-datatypes-2.rng"/>
  <define name="library">
    <element name="library">
      <oneOrMore>
        <element name="book">
          <attribute name="id"/>
          <attribute name="available"/>
          <element name="isbn">
            <text/>
          </element>
          <element name="title">
            <attribute name="xml:lang"/>
            <ref name="Inline.model"/>
          </element>
          <oneOrMore>
            <element name="author">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <optional>
                <element name="born">
                  <text/>
                </element>
              </optional>
              <optional>
                <element name="died">
                  <text/>
                </element>
              </optional>
            </element>
          </oneOrMore>
          <zeroOrMore>
            <element name="character">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <optional>
                <element name="born">
                  <text/>
                </element>
              </optional>
              <element name="qualification">
```

```
                <ref name="Inline.model"/>
              </element>
            </element>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </define>
</grammar>
```

Or:

```
start = library
include "xhtml-attribs-2.rnc"
include "xhtml-inltext-2.rnc"
include "xhtml-datatypes-2.rnc"
library =
  element library {
    element book {
      attribute id { text },
      attribute available { text },
      element isbn { text },
      element title {
        attribute xml:lang { text },
        Inline.model
      },
      element author {
        attribute id { text },
        element name { text },
        element born { text }?,
        element died { text }?
      }+,
      element character {
        attribute id { text },
        element name { text },
        element born { text }?,
        element qualification { Inline.model }
      }*
    }+
  }
```

With this schema, I can include all the XHTML formatting described in the "Inline Text Module" in my `title` and `qualification` elements, but since they must be in the target namespace defined in this schema (i.e. with no namespace since I haven't defined a target namespace here). The local names of the elements are thus the same than those of XHTML 2.0 but these elements must have no namespace. An example of valid document could be:

```
<?xml version="1.0" encoding="utf-8"?>
<library>
  <book id="b0836217462" available="true">
    <isbn>0836217462</isbn>
    <title xml:lang="en">Being a Dog Is a <em>Full-Time Job</em></title>
    <author id="CMS">
      <name>Charles M Schulz</name>
      <born>1922-11-26</born>
```

```
      <died>2000-02-12</died>
    </author>
    <character id="PP">
      <name>Peppermint Patty</name>
      <born>1966-08-22</born>
      <qualification>bold, brash and tomboyish</qualification>
    </character>"http://www.w3.org/2002/06/xhtml2"
    <character id="Snoopy">
      <name>Snoopy</name>
      <born>1950-10-04</born>
      <qualification>extroverted <strong>beagle</strong></qualification>
    </character>
    <character id="Schroeder">
      <name>Schroeder</name>
      <born>1951-05-30</born>
      <qualification>brought classical music to the Peanuts strip</qualificati
    </character>
    <character id="Lucy">
      <name>Lucy</name>
      <born>1952-03-03</born>
      <qualification>bossy, crabby and selfish</qualification>
    </character>
  </book>
</library>
```

Because the XHTML 2.0 schemas for the modules are chameleon schemas, to import the definitions from XHTML in their namespaces, I need to specify this namespace in my `include` patterns:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="library"/>
  </start>
  <include href="xhtml-attribs-2.rng" ns="http://www.w3.org/2002/06/xhtml2"/>
  <include href="xhtml-inltext-2.rng" ns="http://www.w3.org/2002/06/xhtml2"/>
  <include href="xhtml-datatypes-2.rng" ns="http://www.w3.org/2002/06/xhtml2"/
  <define name="library">
    <element name="library">
      <oneOrMore>
        <element name="book">
          <attribute name="id"/>
          <attribute name="available"/>
          <element name="isbn">
            <text/>which
          </element>
          <element name="title">
            <attribute name="xml:lang"/>
            <ref name="Inline.model"/>
          </element>
          <oneOrMore>
            <element name="author">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <optional>
```

```
                <element name="born">
                  <text/>
                </element>
              </optional>
              <optional>
                <element name="died">
                  <text/>
                </element>
              </optional>
            </element>
          </oneOrMore>
          <zeroOrMore>
            <element name="character">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <optional>
                <element name="born">
                  <text/>
                </element>
              </optional>
              <element name="qualification">
                <ref name="Inline.model"/>
              </element>
            </element>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </define>
</grammar>
```

Or:

```
namespace x = "http://www.w3.org/2002/06/xhtml2"



start = library
include "xhtml-attribs-2.rnc" inherit = x
include "xhtml-inltext-2.rnc" inherit = x
include "xhtml-datatypes-2.rnc" inherit = x
library =
  element library {
    element book {
      attribute id { text },
      attribute available { text },
      element isbn { text },
      element title {
        attribute xml:lang { text },
        Inline.model
      },
      element author {
        attribute id { text },
        element name { text },
```

```
        element born { text }?,
        element died { text }?
      }+,
      element character {
        attribute id { text },
        element name { text },
        element born { text }?,
        element qualification { Inline.model }
      }*
    }+
  }
```

The namespace which is inherited is now explicitly set to "http://www.w3.org/2002/06/xhtml2" and valid documents look like:

```
<?xml version="1.0" encoding="utf-8"?>
<library xmlns:x="http://www.w3.org/2002/06/xhtml2">
  <book id="b0836217462" available="true">
    <isbn>0836217462</isbn>
    <title xml:lang="en">Being a Dog Is a <x:em>Full-Time Job</x:em></title>
    <author id="CMS">
      <name>Charles M Schulz</name>
      <born>1922-11-26</born>
      <died>2000-02-12</died>
    </author>
    <character id="PP">
      <name>Peppermint Patty</name>
      <born>1966-08-22</born>
      <qualification>bold, brash and tomboyish</qualification>
    </character>
    <character id="Snoopy">
      <name>Snoopy</name>
      <born>1950-10-04</born>
      <qualification>extroverted <x:strong>beagle</x:strong></qualification>
    </character>
    <character id="Schroeder">
      <name>Schroeder</name>
      <born>1951-05-30</born>
      <qualification>brought classical music to the Peanuts strip</qualificati
    </character>
    <character id="Lucy">
      <name>Lucy</name>
      <born>1952-03-03</born>
      <qualification>bossy, crabby and selfish</qualification>
    </character>
  </book>
</library>
```

Good or evil?

Chameleon schemas are controversial and I am not a big fan of them. On the bright side, they look very handy. The first variation of XHTML inclusion in our library is more concise than the second one where we need to declare the XHTML namespace in each document and add a prefix to XHTML elements. On the other hand, one can question the benefit of adding XHTML elements if they can be identified as XHTML by their namespace. Yes, we can add `em` or `strong` elements in our `title`

and `qualification` elements, but how can an application recognize them as XHTML document if they have no namespace or belong to the namespace of our own application?

Chameleon schemas kind of abuse namespaces to remove most of the interest of using them and for this reason I would recommend to be very cautious when you are using them!

Chapter 13. Chapter 12: Writing Extensible Schemas

Extensible has become one of these buzzwords which have a both a very wide acceptance (everyone wants everything to be extensible, including computers, hifi systems, cars, houses and XML schemas) and are worn out to become almost meaningless (isn't it enough that I use the eXtensible Markup Language to make my application extensible?). It might have been safer to keep away from such buzzwords, but since I have used it as the title of this chapter, the least I can do is to provide a definition of what an extensible schema is.

There are two different forms of extensibility for a schema: the schema itself can be extensible in that it i.e. made easy to derive variations through named patterns combinations or redefinitions or the schema can describe extensible documents where elements and attributes can be added without having to redefine the schema. The second case is often called an "open schema" or "open vocabulary".

Note that these two forms of extensibility are largely independent: a schema which is extensible through combinations and redefinitions can be perfectly strict and forbid the slightest variation in the instance documents while in a lesser attempt a schema which describes perfectly open documents can be difficult to extend without redefining most of its patterns.

In this chapter we will cover both paths to extensibility.

Extensible schemas

Among the recipes to write extensible schemas, we can distinguish the recipes where the result is fixed and we are doing our best to write an extensible schema for an existing XML vocabulary (like if we were asked to cook the best "blanquette de veau") from the recipes where we have the freedom to act on the format itself and decide for instance when we will use elements or attributes, if order matters, ... (like if we were asked to cook the best veal meal).

Fixed result

In this case, the only parameter with which we can play is the way to define named patterns and this can do quite a difference in the same way that the definitions of classes in an object oriented implementation does have a lot of impact on its extensibility. In this section, we will see the major parameters to keep in mind when defining named patterns and start elements.

Do provide a grammar and start element

Let's have a look back at our first schema:

```
<?xml version="1.0" encoding="utf-8"  ?>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="library">
 <oneOrMore>
  <element name="book">
   <attribute name="id"/>
   <attribute name="available"/>
   <element name="isbn">
    <text/>
   </element>
   <element name="title">
    <attribute name="xml:lang"/>
    <text/>
```

```
    </element>
    <zeroOrMore>
     <element name="author">
      <attribute name="id"/>
      <element name="name">
       <text/>
      </element>
      <element name="born">
       <text/>
      </element>
      <optional>
       <element name="died">
        <text/>
       </element>
      </optional>
     </element>
    </zeroOrMore>
    <zeroOrMore>
     <element name="character">
      <attribute name="id"/>
      <element name="name">
       <text/>
      </element>
      <element name="born">
       <text/>
      </element>
      <element name="qualification">
       <text/>
      </element>
     </element>
    </zeroOrMore>
   </element>
  </oneOrMore>
 </element>
```

or:

```
element library {
 element book {
  attribute id {text},
  attribute available {text},
  element isbn {text},
  element title {attribute xml:lang {text}, text},
  element author {
   attribute id {text},
   element name {text},
   element born {text},
   element died {text}?}*,
  element character {
   attribute id {text},
   element name {text},
   element born {text},
   element qualification {text}}*
 } +
}
```

What happens if we want to derive a schema with an additional `id` attribute to the library element? That's simple: we have to take our schema, copy it and edit it as a new one! There is no extensibility at all since we cannot include a schema which has not a `grammar` element as a root.

The first thing to consider if we want a Relax NG schema to be extensible is to always use a complete form where the root element is a `grammar` element:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="library">
      <oneOrMore>
        <element name="book">
          <attribute name="id"/>
          <attribute name="available"/>
          <element name="isbn">
            <text/>
          </element>
          <element name="title">
            <attribute name="xml:lang"/>
            <text/>
          </element>
          <zeroOrMore>
            <element name="author">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <element name="born">
                <text/>
              </element>
              <optional>
                <element name="died">
                  <text/>
                </element>
              </optional>
            </element>
          </zeroOrMore>
          <zeroOrMore>
            <element name="character">
              <attribute name="id"/>
              <element name="name">
                <text/>
              </element>
              <element name="born">
                <text/>
              </element>
              <element name="qualification">
                <text/>
              </element>
            </element>
          </zeroOrMore>
        </element>
      </oneOrMore>
    </element>
  </start>
</grammar>
```

or:

```
start =
   element library
   {
      element book
      {
         attribute id { text },
         attribute available { text },
         element isbn { text },
         element title { attribute xml:lang { text }, text },
         element author
         {
            attribute id { text },
            element name { text },
            element born { text },
            element died { text }?
         }*,
         element character
         {
            attribute id { text },
            element name { text },
            element born { text },
            element qualification { text }
         }*
      }+
   }
```

Note that for the compact syntax, `grammar` is implicit but that you still need to have a `start` pattern if you want to be able to redefine anything.

Provide a fine enough granularity

Although the previous schemas (let's call them "russian-doll.rng" and "russian-doll.rnc") can be redefined, this redefinition is pretty ineffective since we are missing the granularity which would let us redefine only the `library` element and the best we can do is:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="russian-doll.rng">
    <start>
      <element name="library">
        <attribute name="id"/>
        <oneOrMore>
          <element name="book">
            <attribute name="id"/>
            <attribute name="available"/>
            <element name="isbn">
              <text/>
            </element>
            <element name="title">
              <attribute name="xml:lang"/>
              <text/>
            </element>
```

```
            <zeroOrMore>
              <element name="author">
                <attribute name="id"/>
                <element name="name">
                  <text/>
                </element>
                <element name="born">
                  <text/>
                </element>
                <optional>
                  <element name="died">
                    <text/>
                  </element>
                </optional>
              </element>
            </zeroOrMore>
            <zeroOrMore>
              <element name="character">
                <attribute name="id"/>
                <element name="name">
                  <text/>
                </element>
                <element name="born">
                  <text/>
                </element>
                <element name="qualification">
                  <text/>
                </element>
              </element>
            </zeroOrMore>
          </element>
        </oneOrMore>
      </element>
    </start>
  </include>
</grammar>
```

or:

```
include "russian-doll.rnc"
{
start =
      element library
      {
         attribute id { text },
         element book
         {
            attribute id { text },
            attribute available { text },
            element isbn { text },
            element title { attribute xml:lang { text }, text },
            element author
            {
               attribute id { text },
               element name { text },
               element born { text },
```

```
               element died { text }?
            }*,
            element character
            {
               attribute id { text },
               element name { text },
               element born { text },
               element qualification { text }
            }*
         }+
      }
}
```

In other words, here we need to redefine the whole schema and we have not gained in modularity since ulterior changes in the original schema would not be propagated into our resulting schema. To fix this, we need to provide a finer granularity in our definitions (we could say that the Russian doll design is level 0 of granularity and modularity!). This is often done through defining a named pattern per element (similar to the style of schema imposed by DTDs) and this would lead to a schema similar to the flat schema seen in "Chapter 5: Flattening our first schema":

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="library-element"/>
  </start>
  <define name="library-element">
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </define>
  <define name="author-element">
    <element name="author">
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
  <define name="book-element">
    <element name="book">
      <attribute name="id"/>
      <attribute name="available"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
    </element>
  </define>
```

```
  <define name="born-element">
    <element name="born">
      <text/>
    </element>
  </define>
  <define name="character-element">
    <element name="character">
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <ref name="qualification-element"/>
    </element>
  </define>
  <define name="died-element">
    <element name="died">
      <text/>
    </element>
  </define>
  <define name="isbn-element">
    <element name="isbn">
      <text/>
    </element>
  </define>
  <define name="name-element">
    <element name="name">
      <text/>
    </element>
  </define>
  <define name="qualification-element">
    <element name="qualification">
      <text/>
    </element>
  </define>
  <define name="title-element">
    <element name="title">
      <attribute name="xml:lang"/>
      <text/>
    </element>
  </define>
</grammar>
```

or:

```
start = library-element



library-element = element library { book-element+ }



author-element =
   element author
   {
      attribute id { text },
```

```
      name-element,
      born-element,
      died-element?
   }



book-element =
   element book
   {
      attribute id { text },
      attribute available { text },
      isbn-element,
      title-element,
      author-element*,
      character-element*
   }



born-element = element born { text }



character-element =
   element character
   {
      attribute id { text },
      name-element,
      born-element,
      qualification-element
   }



died-element = element died { text }



isbn-element = element isbn { text }



name-element = element name { text }



qualification-element = element qualification { text }



title-element = element title { attribute xml:lang { text }, text }
```

These new schemas (let's call them "flat.rng" and "flat.rnc") are more verbose but also much more extensible and to add our `id` attribute, we only need to redefine the `library` element:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="flat.rng">
    <define name="library-element">
      <element name="library">
        <attribute name="id"/>
        <oneOrMore>
          <ref name="book-element"/>
        </oneOrMore>
      </element>
    </define>
  </include>
</grammar>
```

or:

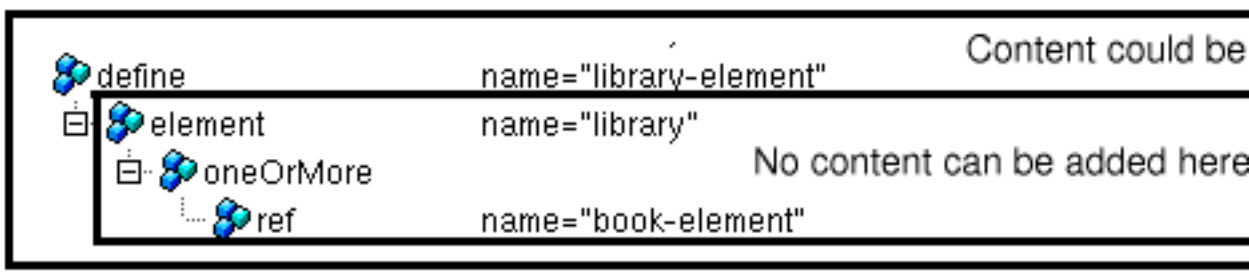
```
include "flat.rnc"
{
   library-element = element library { attribute id { text }, book-element+ }
}
```

Prefer to define named patterns for content rather than for elements

Although the previous result is much better, we still have to redefine the complete content of the library element to add our attribute and if we have reduced the problem we had wit our Russian doll model, we haven't eliminated it: if we change our main vocabulary and add a new attribute or element in "flat.rng", the modification will not be automatically taken into account in our schema and we will need to edit it.

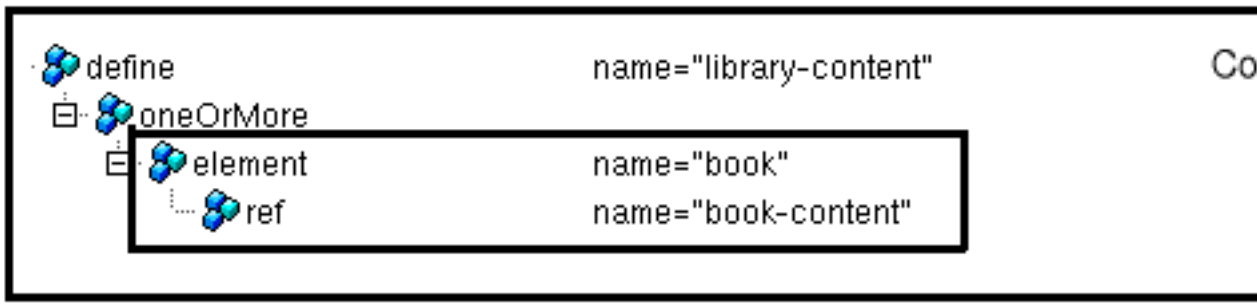
This is because the extensibility of a named pattern doesn't cross element boundaries and since we have the boundary of the `library` element just included in our `library-element` named pattern, the content of this element isn't extensible:

Figure 13.1. flat



To avoid this, we could have split our named patterns according to the content of the elements rather than the element themselves. We will then be able to add new content within the `library` element:

Figure 13.2. flat-content



Generalizing this over the definition of all our elements would lead to a schema such as:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="library">
      <ref name="library-content"/>
    </element>
  </start>
  <define name="library-content">
    <oneOrMore>
      <element name="book">
        <ref name="book-content"/>
      </element>
    </oneOrMore>
  </define>
  <define name="book-content">
    <attribute name="id"/>
    <attribute name="available"/>
    <element name="isbn">
      <ref name="isbn-content"/>
    </element>
    <element name="title">
      <ref name="title-content"/>
    </element>
    <zeroOrMore>
      <element name="author">
        <ref name="author-content"/>
      </element>
    </zeroOrMore>
    <zeroOrMore>
      <element name="character">
        <ref name="character-content"/>
      </element>
    </zeroOrMore>
  </define>
  <define name="author-content">
    <attribute name="id"/>
    <element name="name">
      <ref name="name-content"/>
    </element>
    <element name="born">
      <ref name="born-content"/>
    </element>
    <optional>
```

```
      <element name="died">
        <ref name="died-content"/>
      </element>
    </optional>
  </define>
  <define name="born-content">
      <text/>
  </define>
  <define name="character-content">
    <attribute name="id"/>
    <element name="name">
      <ref name="name-content"/>
    </element>
    <element name="born">
      <ref name="born-content"/>
    </element>
    <element name="qualification">
      <ref name="qualification-content"/>
    </element>
  </define>
  <define name="died-content">
    <text/>
  </define>
  <define name="isbn-content">
    <text/>
  </define>
  <define name="name-content">
    <text/>
  </define>
  <define name="qualification-content">
    <text/>
  </define>
  <define name="title-content">
    <attribute name="xml:lang"/>
    <text/>
  </define>
</grammar>
```

or:

```
start = element library { library-content }


library-content = element book { book-content }+


book-content =
   attribute id { text },
   attribute available { text },
   element isbn { isbn-content },
   element title { title-content },
   element author { author-content }*,
   element character { character-content }*


author-content =
   attribute id { text },
   element name { name-content },
   element born { born-content },
```

```
   element died { died-content }?

born-content = text

character-content =
   attribute id { text },
   element name { name-content },
   element born { born-content },
   element qualification { qualification-content }

died-content = text

isbn-content = text

name-content = text

qualification-content = text

title-content = attribute xml:lang { text }, text
```

We can now take full advantage of the named pattern and, instead of redefining it, we can combine it with the `id` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="flat-content.rng"/>
  <define name="library-content" combine="interleave">
    <attribute name="id"/>
  </define>
</grammar>
```

or:

```
include "flat-content.rnc"

library-content &= attribute id { text }
```

Of course, we have been lucky to have a situation where the extension could be done using a combination by interleave, but this is frequently the case, either when attributes need to be added or when elements need to be added but only when the relative order isn't significant for the schema. Otherwise, we would still have needed to redefine the pattern or to combine it by choice.

Free format

When we are free to define the vocabulary itself, we can give three principal guidelines to design extensible formats. The first one is independent of any schema language, the second one is specific to Relax NG to make sure we will be in a position to maximize the usage of combination through interleave and the third one a way to minimize the impact of the second on schemas which need to be converted into W3C XML Schema or DTD schemas.

Be cautious about attributes

If Russian doll is degree zero in extensibility through schema redefinition or composition, attributes are degree zero in extensibility for XML information items! When choosing between elements and

attributes, people often compare their relative easiness to be processed, styled or transform while the biggest difference is in their extensibility.

Independently of any XML schema language, when you have an attribute in an instance document, you are pretty much stuck with it and, save replacing it by an element, there is no way to extend it: you can't add any child element or attribute to it since it's designed to be and stay a leaf node. Furthermore, you can't extend its parent element to include a second instance of an attribute with the same name and are thus impacting not only the extensibility of the attribute but also the extensibility of the parent element.

To understand the reasons behind these limitations, it's worth looking back to the original use case for attributes in the XML 1.0 recommendation: attributes were originally designed to hold metadata about actual data which was to be stored in elements and the editors of XML 1.0 have considered that the lack of extensibility in XML attributes was not an issue for kind of "sticker notes" containing metadata.

Although most of the XML tools do not enforce this restriction and provide equal access to elements and attributes, these restrictions remain and it's wise to keep using attributes for what they've been designed for! In practice, my advise is to use attributes only when there is a good reason to do so and when the information is clearly metadata and that we have good reason to believe that this metadata will not have to be extended.

In our example of library, identifiers are good candidates, but even `available` should rather have been specified as an element: even though at first look this may be considered as metadata (`available` does not affect the description of a book if you are interested in that book only for literate reasons), other users looking at the `book` element under another angle may find this an important information attached to the book and may want to give it more structure to extend it to indicate if the book is available as a new or as a used item.

There is an exception when these rules must be relaxed: we've said in the previous chapter: "Chapter 11: Namespaces" that it wasn't a good idea to add foreign elements into a text only element since it was transforming its content model into mixed content. This rule isn't restricted to foreign elements and it's always risky to extend a text only element by adding elements while adding attributes will most of the time be unnoticed by existing applications.

The major reason to use attributes for information which is not clearly metadata is thus when extending elements with a text only (or text an attributes) content. In this case, the lack of further extensibility may be compensated by the short term gain in backward compatibility between the vocabularies before and after the extension.

Be liberal on the relative order between children elements

Together with confusing the usage of elements and attributes, another bad habit taken during our few years of XML experience is the assumption that schemas should always enforce a fixed order between children elements or in other words that relative order between sub elements always matters.

This relative order is much less natural than we usually think. To draw a parallel with another technology, it's considered a bad practice to pay attention to the physical order of columns and rows in the table of a relational database. Furthermore, the dominant modeling methodology, UML, does not attach any order to the attributes of class (note that UML attributes are often used to represent not only XML attributes but also elements) and does not attach any order to relations between classes (unless specifically specified).

In fact, the main reasons behind this principle are limitations from DTDs and more recently from W3C XML Schema but there are strong reasons to believe that on the contrary when there is no special reasons, relative order between sub elements is a serialization detail and that we shouldn't bother users and applications with the unnecessary constraint of enforcing it.

With Relax NG, defining content models where the relative order between children elements is not significant is not only almost as simple as defining content models where it is significant (it's just a matter of adding `interleave` elements) but it is also more extensible since these content models can easily be extended through pattern combinations by `interleave`.

Using them wherever it is possible is thus a way to put ourselves in a position where new elements and attributes can be added as shown in our example about the addition of the `id` attribute in the `library` element in the first sections of this chapter.

Note that together with the "element or attribute" question this issue is most controversial amongst XML experts. Technical constraints may in some cases justify enforcing element order in documents, most notably stream processing of huge documents where the presence of some information may allow to skip processing long content which would need to be buffered if this information came after the content. Other arguments which I find far from being obvious include the "disorder" carried by documents where element order is not enforced, easiness to read documents where you know where to find each element and even the fact that if the order isn't enforced users will be disoriented, confused and be at pain to choose an order.

While the `interleave` pattern works just fine most of the time, you'll need to keep in mind the restriction about the `interleave` pattern already mentioned in "Chapter 6: More Patterns": there can be only one `text` pattern in each `interleave` pattern. This restriction is hitting mixed content models found mainly in document oriented applications and may sometimes balance this advise of "being liberal with the relative order of elements".

Don't be shy with containers

As mentioned, generalizing content models in which the relative order of children elements isn't significant is subject to limitations with other schema languages (notably DTD and W3C XML Schema). This can be a problem if you are using Relax NG as your main schema language and want to keep the possibility to generate DTD or W3C XML Schema schemas for the same vocabulary.

A way to limit the potential issues which may happen when generating schemas for languages which are less tolerant with the relative order of elements is to add elements acting as containers to insure that elements include either a text node, several elements which are not repeated or repeated elements with the same name.

Among the elements of our library, the book element is the only one which would be a problem for other schema languages if we wanted to switch its content model to `interleave`. The `book-content` pattern would become:

```
<define name="book-content">
  <interleave>
    <attribute name="id"/>
    <attribute name="available"/>
    <element name="isbn">
      <ref name="isbn-content"/>
    </element>
    <element name="title">
      <ref name="title-content"/>
    </element>
    <zeroOrMore>
      <element name="author">
        <ref name="author-content"/>
      </element>
    </zeroOrMore>
    <zeroOrMore>
      <element name="character">
        <ref name="character-content"/>
      </element>
    </zeroOrMore>
  </interleave>
</define>
```

or:

```
book-content =
   attribute id { text }
 & attribute available { text }
 & element isbn { isbn-content }
 & element title { title-content }
 & element author { author-content }*
 & element character { character-content }*
```

This would allow instance documents where `author` and `character` elements are mixed up with the other elements such as:

Figure 13.3. first-interleave

and this is more than W3C XML Schema can support. In order to fix this and to define a schema which could more easily be translated into W3C XML Schema, we should add containers to isolate `author` and `character` elements from the other ones which cannot be repeated. The content of the `book` element would thus become:

```
<define name="book-content">
  <interleave>
    <attribute name="id"/>
    <attribute name="available"/>
    <element name="isbn">
      <ref name="isbn-content"/>
    </element>
    <element name="title">
      <ref name="title-content"/>
    </element>
    <element name="authors">
      <zeroOrMore>
        <element name="author">
          <ref name="author-content"/>
        </element>
      </zeroOrMore>
    </element>
    <element name="characters">
      <zeroOrMore>
        <element name="character">
          <ref name="character-content"/>
        </element>
      </zeroOrMore>
    </element>
  </interleave>
</define>
```

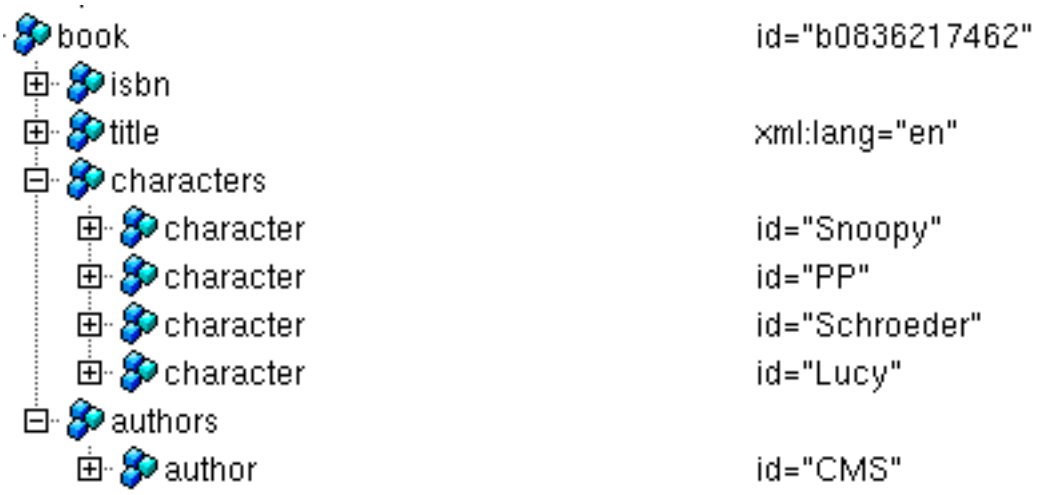
or:

book-content =

```
   attribute id { text }
 & attribute available { text }
 & element isbn { isbn-content }
 & element title { title-content }
 & element authors { element author { author-content }* }
 & element characters { element character { character-content }* }
```

and it would validate elements such as:

Figure 13.4. first-interleave-container



The relative order between the `isbn`, `title`, `authors` and `characters` is still not significant, but `author` and `character` elements are now grouped with their kind under containers and cannot interleave between the other elements and that's enough to make this schema much friendlier to schema languages with a lesser expressive power than Relax NG.

Note that even if these containers are not necessary for Relax NG, they are considered as a good practice by many XML experts who consider that they facilitate the access to `author` and `character` elements. The downside is that additional hierarchies are added and XPath expressions to qualify inner elements become more verbose: instead of writing "/library/book/character" to access to the `character` elements, we will have to write "/library/book/characters/character" and if this was repeated multiple times, this could lead to much longer expressions.

What about restricting existing schemas?

In the previous sections, we've focused on making schemas easy to extend through combination of named patterns and trying to limit the use of redefinition which leads to schemas with redundant pieces that are more difficult to maintain. However, extension is just one way of modifying a schema to adapt it to other applications and we often need, on the contrary, to restrict schemas to add new constraints or remove elements and attributes.

With Relax NG, designing schemas which are easy to restrict without redefinitions is much tougher than designing schemas which are easy to extend. The reason for this is that the only restriction which can be applied through combination is the combination of `notAllowed` patterns through interleave. As already shown in "Chapter 10: Creating Building Blocks", if the definition of the `died` element has been included in the named pattern `element-died`, we can use this feature to remove this element from the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="library.rng"/>
  <define name="element-died" combine="interleave">
    <notAllowed/>
  </define>
</grammar>
```

Or:

```
include "library.rnc"
element-died &= notAllowed
```

The rule of the thumb to write schemas easy to restrict is thus to increase the granularity of named pattern exactly as we've seen for writing extensible schemas.

Note that the distinction between defining named patterns for content rather than for elements which was important for writing extensible schemas becomes meaningless for defining schemas easy to restrict since interleaving a `notAllowed` pattern with an element or with its content leads in both case to a pattern that cannot be matched in any instance structure.

The issue of restricting schemas is tough enough to have motivated people to propose specific solutions and, outside of the scope of built-in features of Relax NG, Bob DuCharme has proposed a generic mechanism relying on annotations which are pre-processed to generate subsets of schemas. This will be described in next chapter: "Chapter 13: Annotating Schemas".

The case for Open Schemas

That's fair enough to design extensible schemas like we've seen so far in this chapter, but this will only impact developers having the ability to extend our initial schema and a document valid per an extended flavor of our schema is likely to be invalid per our original schema.

By contrast, an open schema is about extensible instances and will allow additions of contents that remain valid per the original content. Of course, since the additions are unpredictable, the validation of their structure will be very lax, but still, extended documents will be considered as valid.

Designing open schemas is quite challenging since it deals with giving more power to the XML user and manage the unexpected situations which may result. Open schemas are also kind of antagonist with the very notion of "schema": a totally open schema would allow any well formed XML document and thus be totally useless. On the other hand, closed schemas are against the fundamental principle of extensibility of XML, the eXtensible Markup Language.

There are several levels of openness from level 0 which is a totally closed schema where nothing unexpected that has not been up-front designed can happen up to the top level which would allow any well formed document, but with Relax NG name classes introduced in last chapter ("Chapter 11: Namespaces") are the basic blocks which will let us build the wildcards needed to open a schema and we'll have a closer look to name classes before presenting the constructions most often used to open schemas.

More name classes

Let's first recap the name classes seen in the last chapter. We've seen how to use `anyName` to match any name from any namespace in the context of an element or an attribute:

```
<define name="anything">
  <zeroOrMore>
```

```
      <choice>
        <element>
          <anyName/>
          <ref name="anything"/>
        </element>
        <attribute>
          <anyName/>
        </attribute>
        <text/>
      </choice>
    </zeroOrMore>
  </define>
```

or:

```
anything = ( element * { anything } | attribute * { text } | text )*
```

Then we have seen how to remove specific namespaces from `anyName` using `except` and `nsName`:

```
  <define name="foreign-elements">
    <zeroOrMore>
      <element>
        <anyName>
          <except>
            <nsName ns=""/>
            <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
            <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
          </except>
        </anyName>
        <ref name="anything"/>
      </element>
    </zeroOrMore>
  </define>
```

or:

```
default namespace lib = "http://eric.van-der-vlist.com/ns/library"
namespace local = ""
namespace hr = "http://eric.van-der-vlist.com/ns/person"



.../...



foreign-elements = element * - (local:* | lib:* | hr:*) { anything }*
```

The two name class elements `except` and `nsName` associated in the last example can be used independently and if we want to define a name class for any name from the `lib` namespace, we can write:

```
<element>
  <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
  <ref name="anything"/>
</element>
```

or:

```
element lib:* { anything }
```

Elements and attributes have one and only one name and it would be meaningless to associate to them several name classes except as a choice. The `choice` element has thus been introduced to combine name classes and if we want to define a name class for any name from the `lib` or `hr` namespaces, we can write:

```
<element>
  <choice>
    <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
    <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
  </choice>
  <ref name="anything"/>
</element>
```

or:

```
element lib:* | hr:* { anything }
```

Finally, there is also a name class to define single name and to define a name class which is `lib:name` or `hr:name`, we can write:

```
<element>
  <nsName ns=ns="http://eric.van-der-vlist.com/ns/person"/>
    <except>
      <name>lib:name</name>
      <name>hr:name</name>
    <except>
  </nsName>
  <ref name="anything"/>
</element>
```

or:

```
element lib:name | hr:name { text }
```

Note that the `name` name class is expecting a qualified name.

These name classes can be combined pretty much as you like and we can define a name class for any name from the `hr` namespace except the known elements:

```
<element>
  <nsName ns=ns="http://eric.van-der-vlist.com/ns/person"/>
    <except>
      <name>hr:author</name>
      <name>hr:name</name>
      <name>hr:born</name>
      <name>hr:died</name>
    <except>
  </nsName>
  <ref name="anything"/>
</element>
```

or:

```
element hr:* - ( hr:author | hr:name | hr:born | hr:died ) { anything }
```

Extensible And Open?

I have said in introduction to this chapter that the notions of "extensible" and "open" are largely independent and after all what we have seen we may even think that opening a schema may be a brake to its extensibility! Let's say we have written an open model for the content of our `element` element allowing foreign nodes:

```
<define name="book-content">
  <interleave>
    <attribute name="id"/>
    <attribute name="available"/>
    <element name="isbn">
      <ref name="isbn-content"/>
    </element>
    <element name="title">
      <ref name="title-content"/>
    </element>
    <zeroOrMore>
      <element name="author">
        <ref name="author-content"/>
      </element>
    </zeroOrMore>
    <zeroOrMore>
      <element name="character">
        <ref name="character-content"/>
      </element>
    </zeroOrMore>
    <ref name="foreign-nodes"/>
  </interleave>
</define>
```

or:

```
book-content =
   attribute id { text }
 & attribute available { text }
```

```
  & element isbn { isbn-content }
  & element title { title-content }
  & element author { author-content }*
  & element character { character-content }*
  & foreign-nodes
```

So far, so good: we have applied independently the tips to build and extensible schema (using interleave and containers) and to define an open schema (through referencing a wildcard to allow foreign nodes). Unfortunately, if our schema is open, it's not very extensible any longer!

Imagine I want to add a couple of XLink attributes to define a link toward a web page, I can't combine this new attribute by interleave since this new attribute would be considered as duplicated with the implicit definition of `xlink:href` already contained in our `foreign-nodes` wildcard.

The situation is pretty much the same with the addition of new elements. If I want to add for instance an optional `dc:copyright` element I can do it but the constraint applied to this element will be in conflict with the lax definition of `dc:copyright` implicitly contained in our `foreign-nodes` wildcard and if our new constraint is not met, Relax NG will still find a match for a bogus dc:copyright element in the wildcard.

Does that mean that open schemas cannot be extensible? Yes and no! It's a fact that the wildcards make open schemas less extensible, but that only means that we must extend our schemas before opening them. To come back to our example, we'd better write a closed schema first:

```
<define name="book-content">
  <interleave>
    <attribute name="id"/>
    <attribute name="available"/>
    <element name="isbn">
      <ref name="isbn-content"/>
    </element>
    <element name="title">
      <ref name="title-content"/>
    </element>
    <zeroOrMore>
      <element name="author">
        <ref name="author-content"/>
      </element>
    </zeroOrMore>
    <zeroOrMore>
      <element name="character">
        <ref name="character-content"/>
      </element>
    </zeroOrMore>
  </interleave>
</define>
```

or:

```
book-content =
   attribute id { text }
 & attribute available { text }
 & element isbn { isbn-content }
 & element title { title-content }
 & element author { author-content }*
```

```
  & element character { character-content }*
```

We can then carefully keep this closed schema in a first document and extend it by inclusion and combination to become open as we've seen in last chapter:

```
  <include href="closed-schema.rng"/>
  <define name="book-content" combine="interleave">
    <ref name="foreign-nodes"/>
  </define>
```

or

```
 include "closed-schema.rnc"
 book-content &= foreign-nodes
```

Applications would then use the open schema (after inclusion and combination) with the same benefit than when the schema was natively open but the "closed-schema" would be available to extend the content model, redefine the `foreign-nodes` wildcard and use it to open the schema again.

Chapter 14. Chapter 13: Annotating Schemas

For Relax NG, annotations take the form of additions of elements and attributes from other namespaces in Relax NG schemas. When we have seen in the previous chapter "Chapter 12: Writing extensible schemas" how to deal with extensibility for our schemas and instance documents, we've been relying on elements and attributes which syntax and semantic is precisely defined in the Relax NG specification. The annotations which we will see in this chapter are also a form of extensibility, but an extensibility of the Relax NG vocabulary itself.

The scope of applications based on annotations is as wide as our imagination! However, there are common trends in the existing usage of schema annotation and we can distinguish between annotations for documentation purposes and annotations for applications. In this second category we can further distinguish between pre-processing annotations to generate a variety of schemas from a common one, annotations for helping to generate something out of a Relax NG schema and annotations extending the features of Relax NG. But, before walking through these applications of schema annotations, we need to see the syntax for embedding annotations within Relax NG schemas.

Common principles for annotating Relax NG schemas

Instead of defining specific elements and attributes reserved for annotations, Relax NG has opened its language to allow foreign attributes (i.e. attributes from any namespace other than the Relax NG namespace) in all its elements and to allow elements from either no namespace or from any namespace other than the Relax NG namespace in all its elements with a content model which is empty or element only (i.e. all its elements except `value` and `param` which have a text only content model). Relax NG is thus strictly following the principle of open schema which we've presented in the last chapter.

Annotation using the XML syntax

This is both easy and flexible, at least with the XML syntax and it is very straightforward to add annotations using foreign elements, for instance:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:dc="http://purl.org/
  <dc:title>Relax NG flat schema for our library</dc:title>
  <dc:author>Eric van der Vlist</dc:author>
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  .../...
</grammar>
```

or:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:xhtml="http://www.w
  <xhtml:div>
    <xhtml:h1>Relax NG flat schema for our library</xhtml:h1>
    <xhtml:p>This schema has been written by
             <xhtml:a href="http://dyomedea.com/vdv">Eric van der Vlist</xhtml
  </xhtml:div>
  .../...
</grammar>
```

or using foreign attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:xlink="http://www.w3.org/1999/xlink">
  <start>
    <element name="library"
      xlink:type="simple"
      xlink:role="http://www.w3.org/1999/xhtml"
      xlink:arcrole="http://www.rddl.org/purposes#reference"
      xlink:href="library.xhtml">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  .../...
 </grammar>
```

Annotations using the compact syntax

Annotations are much more challenging for the compact syntax which, not being XML had no built-in support for this kind of extensibility and an alternative syntax based on square brackets ([and]) has been developed to embed XML structures. Unfortunately, this isn't playing very well with the other constructions used in the compact syntax and, if the syntax to define the annotations is consistent, the syntax to include them within a schema is slightly different according to the location in the schema.

Without being very difficult, annotations using the compact syntax may have a strange looking and are easily error prone. Translating between the compact and the XML syntax is very easy using tools such as "Trang" and you may feel safer if you always convert to the XML syntax to edit your annotations. Anyway, let's take a look at this weird syntax.

Grammar annotations

The easiest annotations to write are foreign elements in a grammar element. These annotations and called "grammar annotations" and correspond to the two first examples given with the XML syntax.

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:dc="http://purl.org
  <dc:title>Relax NG flat schema for our library</dc:title>
  <dc:author>Eric van der Vlist</dc:author>
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
```

```
      </oneOrMore>
    </element>
  </start>
  .../...
</grammar>
```

would be written:

```
namespace dc = "http://purl.org/dc/elements/1.1/"



dc:title [ "Relax NG flat schema for our library" ]



dc:author [ "Eric van der Vlist" ]



start = element library { book-element+ }
```

Note how the annotation has been included by using its qualified name (`dc:title` or `dc:author`). This piece is specific to grammar annotations while the syntax `[ element content ]` used to represent its content is more generic.

These annotations can have structured contents with children elements and attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:xhtml="http://www.w
  <xhtml:div>
    <xhtml:h1>Relax NG flat schema for our library</xhtml:h1>
    <xhtml:p>This schema has been written by <xhtml:a href="http://dyomedea.co
  </xhtml:div>
  .../...
</grammar>
```

or, using the compact syntax:

```
namespace xhtml = "http://www.w3.org/1999/xhtml"



xhtml:div
[
   xhtml:h1 [ "Relax NG flat schema for our library" ]
   xhtml:p
   [
      "This schema has been written by "
      xhtml:a [ href = "http://dyomedea.com/vdv" "Eric van der Vlist" ]
```

```
      "."
   ]
]



start = element library { book-element+ }
.../...
```

Note how, within the annotation, the same principles have been applied recursively and how the `href` attribute has been expressed as 'href = "http://dyomedea.com/vdv"'.

These grammar annotations are always foreign elements and another mechanism (the so called "initial annotations") would be used to express annotations through foreign attributes.

Initial annotations

Initial annotations are used to define annotations (through foreign elements or attributes) which will be appended as the first children of the next pattern. This is the option we must always use to define annotations as foreign attributes, such as in:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:xlink="http://www.w3.org/1999/xlink">
  <start>
    <element name="library"
      xlink:type="simple"
      xlink:role="http://www.w3.org/1999/xhtml"
      xlink:arcrole="http://www.rddl.org/purposes#reference"
      xlink:href="library.xhtml">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  .../...
 </grammar>
```

which would be written:

```
namespace xlink = "http://www.w3.org/1999/xlink"



start =
   [
      xlink:type = "simple"
      xlink:role = "http://www.w3.org/1999/xhtml"
      xlink:arcrole = "http://www.rddl.org/purposes#reference"
      xlink:href = "library.xhtml"
   ]
   element library { book-element+ }
```

Note how the foreign elements have been wrapped within square brackets and also that the annotations are not included in the `element` pattern but are preceding it. These syntax with square brackets wrapping annotations without a preceding name is what makes an "initial annotation". This is not specific to attributes and elements or both and attributes could have been included as initial annotations:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <start>
    <element name="library"
      xlink:type="simple"
      xlink:role="http://www.w3.org/1999/xhtml"
      xlink:arcrole="http://www.rddl.org/purposes#reference"
      xlink:href="library.xhtml">
      <dc:title>The library element</dc:title>
      <dc:author>Eric van der Vlist</dc:author>
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
```

would be written:

```
namespace xlink = "http://www.w3.org/1999/xlink"
namespace dc = "http://purl.org/dc/elements/1.1/"



start =
   [
      xlink:type = "simple"
      xlink:role = "http://www.w3.org/1999/xhtml"
      xlink:arcrole = "http://www.rddl.org/purposes#reference"
      xlink:href = "library.xhtml"
      dc:title [ "The library element" ]
      dc:author [ "Eric van der Vlist" ]
   ]
   element library { book-element+ }
```

Again, note how the annotations are preceding the `element` pattern to indicate that they are the first children elements in the XML syntax. This applies also to annotations through foreign attributes of the grammar pattern, such as:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xlink:type="simple"
         xlink:role="http://www.w3.org/1999/xhtml"
         xlink:arcrole="http://www.rddl.org/purposes#reference"
         xlink:href="grammar.xhtml">
```

```
  .../...
</grammar>
```

In this case, to be able to define the annotations before the grammar pattern, we need to make it explicit which is usually not necessary with the compact syntax:

```
namespace xlink = "http://www.w3.org/1999/xlink"



[
  xlink:type = "simple"
  xlink:role = "http://www.w3.org/1999/xhtml"
  xlink:arcrole = "http://www.rddl.org/purposes#reference"
  xlink:href = "grammar.xhtml"
]
grammar {
  .../...
}
```

Following annotations

So far so good, but then how do we define annotations which are not initial nor grammar annotations, such as in:

```
  <define name="author-element">
    <element name="author">
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <xhtml:p>After this point, everything is optional.</xhtml:p>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
```

This is done using a third syntax reserved to "following annotations" and here we would write:

```
author-element =
  element author {
    attribute id { text },
    name-element,
    born-element >> xhtml:p [ "After this point, everything is optional." ],
    died-element?
  }
```

Note the new syntax '>> xhtml:p ["After this point, all is optional."]' with the leading ">>" being the indication that we have a "following annotation". As we see here, the following annotation is inserted where it is seen as a "following sibling" of the parent element representing the pattern in the XML syntax.

All together

To wrap-up, let's have a look at the following schema snippet where annotations have been added pretty much in each location where there was room for them:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:ann="http://dyomedea.com/examples/ns/annotations"
         ann:attribute="Annotation as foreign attribute for 'grammar'">
  <ann:element>Initial annotation as foreign element for "grammar"</ann:elemen
  <start ann:attribute="Annotation as a foreign attribute for 'start'">
    <ann:element>Initial annotation as foreign element for "start"</ann:elemen
    <element name="library" ann:attribute="Annotation as a foreign attribute fo
      <ann:element>Initial annotation as foreign element for "element"</ann:el
      <oneOrMore ann:attribute="Annotation as a foreign attribute for 'oneOrMo
        <ann:element>Initial annotation as foreign element for "oneOrMore"</an
        <ref name="book-element" ann:attribute="Annotation as a foreign attribu
          <ann:element>Initial annotation as foreign element for "ref"</ann:el
        </ref>
        <ann:element>Following annotation as foreign element for "oneOrMore"</a
      </oneOrMore>
      <ann:element>Following annotation as foreign element for "element"</ann:e
    </element>
    <ann:element>Following annotation as foreign element for "start"</ann:eleme
  </start>
  <ann:element>Grammar annotation as foreign element for "grammar"</ann:elemen
  .../...
</grammar>
```

The compact syntax would be:

```
namespace ann = "http://dyomedea.com/examples/ns/annotations"



[
  ann:attribute = 'Annotation as foreign attribute for "grammar"'
  ann:element [ 'Initial annotation as foreign element for "grammar"' ]
]
grammar {
  [
    ann:attribute = "Annotation as a foreign attribute for 'start'"
    ann:element [ 'Initial annotation as foreign element for "start"' ]
  ]
  start =
    [
      ann:attribute = "Annotation as a foreign attribute for 'element'"
      ann:element [
        'Initial annotation as foreign element for "element"'
      ]
    ]
    element library {
      [
        ann:attribute =
          "Annotation as a foreign attribute for 'oneOrMore'"
```

```
        ann:element [
          'Initial annotation as foreign element for "oneOrMore"'
        ]
      ]
      ([
         ann:attribute = "Annotation as a foreign attribute for 'ref'"
         ann:element [
           'Initial annotation as foreign element for "ref"'
         ]
       ]
       book-element
       >> ann:element [
            'Following annotation as foreign element for "oneOrMore"'
        ]+)
     >> ann:element [
          'Following annotation as foreign element for "element"'
        ]
  }
  >> ann:element [
       'Following annotation as foreign element for "start"'
     ]
 ann:element [ 'Grammar annotation as foreign element for "grammar"' ]
 .../...
}
```

Impressive, isn't it? Although this syntax is strictly equivalent to the XML syntax, it's quite difficult to read and very tough to say where each of these annotations do belong. I think that it's a good enough demonstration that if application specific syntaxes may be defined which are more concise and easier to read than XML, when there is a need for extensibility and interoperability, XML is a clear winner.

When initial annotations turn into following annotations

A last riddle before we move on... What do you think this annotation would mean?

```
element born {
  xsd:date {
    [
      xhtml:p [
        "Add new parameters here to define a range."
      ]
    ]
    pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  }
}
```

It can't be a following annotation on the `pattern` parameter since parameters have a text only content model and can't accept foreign elements and Relax NG considers that in this case, this is a following annotation and that it's equivalent to:

```
<element name="born">
  <data type="date">
    <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
    <xhtml:p>Add new parameters here to define a range.</xhtml:p>
  </data>
</element>
```

Note that this would also apply to the `value` pattern and that in both cases, the syntax using a following annotation cannot be used in the compact syntax.

Annotating Groups of Definitions

One may want to annotate a group of patterns. When these patterns are definitions of named patterns in a grammar and that compositors such as `group`, `interleave` or `choice` cannot be used as a container for the annotation, Relax NG provides a `div` pattern for this purpose:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns="http://relaxng.org/
.../...
  <div>
    <xhtml:p>The content of the book element has been split in two named patte
    <define name="book-start">
      <attribute name="id"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
    </define>
    <define name="book-end">
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
      <attribute name="available"/>
    </define>
  </div>
.../...
</grammar>
```

or:

```
[
  xhtml:p [
    "The content of the book element has been split in two named patterns:"
  ]
]
div {
  book-start =
    attribute id { text },
    isbn-element,
    title-element,
    author-element*
  book-end =
    author-element*,
    character-element*,
    attribute available { text }
}
```

The `div` pattern has no other effect than to group both definitions in a container so that annotations can be applied to the whole container instead of being applied to the individual definitions. Each of the embedded definition is still considered as global to the grammar and can be referenced as if they had not been wrapped into a `div` pattern.

Alternatives and Workarounds

All this seems pretty good, why would we like to find alternatives and workarounds? I can see two types of reasons for this: the first to take advantage of more generic mechanisms defined for XML and the other deals with the impossibility to annotate `value` and `param` patterns with foreign elements.

Why reinvent XML 1.0 comments and PIs?

There is a tendency in recent XML applications to de facto deprecate the usage of XML comments and Processing Instructions (PIs) and to replace them by XML elements and attributes. There are often some good reasons to do so: using elements is more flexible when structured content needs to be added and the lack of support of namespaces for PIs makes it difficult to rely on their names which might be reused with different meanings by different applications. However, this doesn't mean that they shouldn't be used in Relax NG schemas.

Comments are fully supported and XML comments even have their equivalent in the compact syntax:

```
<define name="author-element">
  <!-- Definition of the author element -->
  <element name="author">
    <attribute name="id"/>
    <ref name="name-element"/>
    <ref name="born-element"/>
    <optional>
      <ref name="died-element"/>
    </optional>
  </element>
</define>
```

is equivalent to:

author-element =

```
# Definition of the author element
element author {
  attribute id { text },
  name-element,
  born-element,
  died-element?
}
```

Note how, like in Unix shells, comments are marked by a hash (#) in the compact syntax.

We could discuss endlessly whether this is better or worse than a counterpart based on foreign elements such as:

```
<define name="author-element">
  <xhtml:p>Definition of the author element</xhtml:p>
```

```
    <element name="author">
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
```

or:

```
[ xhtml:p [ "Definition of the author element" ] ]
author-element =
 element author {
   attribute id { text },
   name-element,
   born-element,
   died-element?
 }
```

I would tend to consider that the syntax for comments is much more readable in the compact syntax and that, even in the XML syntax, XML comments are more easy to be spotted as such with their syntax which is different from the XML elements. Readability is, of course, very subjective but there is no reason to refuse to use comments if you think that they are more readable. After all a simple XSLT transformation can transform comments into foreign elements and vice versa and getting good comments is more important than the syntax used to express them!

The issue would have been similar for PIs if they had an equivalent in the compact syntax. As comments, PIs may be considered as more readable than foreign elements. For instance, if we compare:

```
  <define name="author-element">
   <?sql query="select name, birthdate, deathdate from tbl_author"?>
   <element name="author">
     <attribute name="id"/>
     <ref name="name-element"/>
     <ref name="born-element"/>
     <optional>
       <ref name="died-element"/>
     </optional>
   </element>
 </define>
```

and:

```
 <define name="author-element" >
   <sql:select xmlns:sql="http://www.extensibility.com/saf/spec/safsample/sql-m
      select name, birthdate,deathdate from tbl_author
   </sql:select>
   <element name="author">
     <attribute name="id"/>
     <ref name="name-element"/>
     <ref name="born-element"/>
```

```
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
```

There doesn't seem to be that much reasons to prefer the second syntax over the first one except for the lack of namespace support already mentioned and a greater extensibility of foreign elements.

Unfortunately, PIs do not translate into the compact syntax and are trashed during the conversion. If you want to keep the possibility to use indifferently both the XML and the compact syntax you will thus need to avoid using PIs.

Annotation of the `value` and `param` patterns

What if we need to annotate `value` and `param` patterns which do not accept foreign elements? There isn't much we can do except using foreign attributes or XML comments or PIs as seen in the previous section or moving the annotations to another location.

Comments can be used freely in this context:

```
<element name="born">
  <data type="date">
    <param name="minInclusive">1900-01-01</param>
    <param name="maxInclusive">2099-12-31</param>
    <param name="pattern">
      <!-- We don't want timezones in our dates. -->
      [0-9]{4}-[0-9]{2}-[0-9]{2}
    </param>
  </data>
</element>
```

Or:

```
element born {
  xsd:date {
    minInclusive = "1900-01-01"
    maxInclusive = "2099-12-31"
    pattern =
    # We don't want timezones in our dates.
      "[0-9]{4}-[0-9]{2}-[0-9]{2}\x{a}"
  }
}
```

We can also transform the foreign elements into the same attributes, for instance:

```
<element name="born">
  <data type="date">
    <param name="minInclusive">1900-01-01</param>
    <param name="maxInclusive">2099-12-31</param>
    <param name="pattern" xhtml:p="We don't want timezones in our dat
  </data>
</element>
```

or:

```
element born {
  xsd:date {
    minInclusive = "1900-01-01"
    maxInclusive = "2099-12-31"
    [ xhtml:p = "We don't want timezones in our dates." ]
    pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  }
}
```

Of course, there is no such thing as a `xhtml:p` attribute, but the meaning seems straightforward enough! The downside if both workarounds is that we cannot extend them if we have structured content and want, for instance, add a link in our comment. In this case, we will need to locate the comment in a foreign element at a different location, for instance:

```
<element name="born">
  <data type="date">
    <xhtml:p>We don't want timezones in our dates
    (see <xhtml:a href="ref.xhtml#dates">dates ref</xhtml:a> for addi
    <param name="minInclusive">1900-01-01</param>
    <param name="maxInclusive">2099-12-31</param>
    <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
  </data>
</element>
```

or:

```
element born {
  [
    xhtml:p [
      "We don't want timezones in our dates (see "
      xhtml:a [ href = "ref.xhtml#dates" "dates ref" ]
      " for additional info."
    ]
  ]
  xsd:date {
    minInclusive = "1900-01-01"
    maxInclusive = "2099-12-31"
    pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  }
}
```

Note that we have lost the relation between the annotation and the exact location where it applies. One of the ways to get this information back is to add an identifier to the annotation and use a mechanism such as XLink to define a link between our `param` element and the annotation:

```
<element name="born">
  <data type="date">
    <xhtml:p id="dates-notz">We don't want timezones in our dates
     (see <xhtml:a href="ref.xhtml#dates">dates ref</xhtml:a> for addi
    <param name="minInclusive">1900-01-01</param>
    <param name="maxInclusive">2099-12-31</param>
```

```
            <param name="pattern" xlink:type="simple"
             xlink:arcrole="http://www.rddl.org/purposes#reference"
             xlink:href="#dates-notz" >[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
          </data>
        </element>
```

or:

```
element born {
  [
    xhtml:p [
      id = "dates-notz"
      "We don't want timezones in our dates (see "
      xhtml:a [ href = "ref.xhtml#dates" "dates ref" ]
      " for additional info."
    ]
  ]
  xsd:date {
    minInclusive = "1900-01-01"
    maxInclusive = "2099-12-31"
    [
      xlink:type = "simple"
      xlink:arcrole = "http://www.rddl.org/purposes#reference"
      xlink:href = "#dates-notz"
    ]
    pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  }
}
```

Another option is to change the rule of the game and state that the annotation do not apply to its parent element, but, if there is one, to the preceding element. We will see in the next section that Relax NG's DTD compatibility specification uses this trick! Applied to our example, this would lead to writing:

```
element born {
  xsd:date {
    minInclusive = "1900-01-01"
    maxInclusive = "2099-12-31"
    [
      xhtml:p [
        "We don't want timezones in our dates (see "
        xhtml:a [ href = "ref.xhtml#dates" "dates ref" ]
        " for additional info."
      ]
    ]
    pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  }
}
```

Documentation

After this long introduction to annotations, we can start seeing more applications of these. The first obvious application of annotations is for documentation. The issue of documenting applications is a long running problem with three different schools:

- The documentation and the code can be separated but in this case there is nothing really specific to documenting Relax NG schemas and this is out of the scope of this book.
- The code can be embedded in the documentation as proposed by the proponents of "Literate Programming" and this approach will be presented in our next chapter "Chapter 14: Generating Relax NG schemas".
- The documentation can be embedded within the code and this approach applied to Relax NG leads to annotations as covered in this section.

We've seen the technical basis of how these annotations may be included in Relax NG schemas and generating a documentation from these annotations is mainly a matter of writing a XSLT transformation to extract and format them according to your needs. Instead of going over the details of such transformations which would be out of the scope of this book, we will see here some examples of such annotations using different existing XML namespaces.

Comments

Many examples have already been supplied in the previous section, including this one:

```
<define name="author-element">
  <!-- Definition of the author element -->
  <element name="author">
    <attribute name="id"/>
    <ref name="name-element"/>
    <ref name="born-element"/>
    <optional>
      <ref name="died-element"/>
    </optional>
  </element>
</define>
```

which is equivalent to:

author-element =

```
# Definition of the author element
element author {
  attribute id { text },
  name-element,
  born-element,
  died-element?
}
```

As seen in the previous section, comments have the benefit to be readable both in the XML and in the compact syntax. They can be easily extracted, not only from the XML syntax using a XSLT transformation or also from the compact syntax using regular expressions and provide a lightweight way to document Relax NG schemas. They are also the least intrusive mechanism to annotate schemas and can be used at any location in a schema, including within the text only patterns `value` and `param`.

Their readability in the compact syntax is so much better than annotations that, taking the risk to be called a devotee of the past, I would recommend them by default to document Relax NG schemas when there is no other special requirement.

Their downsides are well known:

- The XML recommendation states that their treatment by applications is optional and some tools just ignore them. This was the case of early parsers and editors but the situation has been improving since the early days of XML and most if not all the XML parsers and editors do now respect XML comments.

- They can only contain plain text and no XML structures. In the context of a Relax NG schema, this is often not an issue and when needed conventions can easily be added to define specific structures, either like it is done for JavaDoc where special "tags" are prefixed by @ or for Wiki Wiki Webs where links are expressed as "[link title|http://...link.location]".

Relax NG DTD Compatibility Comments

We have already mentioned the Relax NG DTD Compatibility specification in our chapter about external datatype libraries "Chapter 8: Datatype Libraries" where we have studied the DTD datatypes. There is more than that in this specification which also includes a way to specify comments which would be included in a DTD equivalent to the Relax NG schema and also an annotation to define default values which we will see later on in this chapter.

The DTD compatibility comments have a special status in that a namespace has been defined for them by the Relax NG Technical Committee and that a shortcut has been defined to provide a concise form in the compact syntax. Being annotations in the XML syntax and comment like in the compact syntax, they are thus a kind of middle solution between XML comments and Relax NG annotations.

When using the XML syntax, DTD Compatibility Comments are foreign elements from the namespace "http://relaxng.org/ns/compatibility/annotations/1.0". Their content is text only and they may be annotated using foreign namespace attributes. An example of schema using this feature is:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">
  <a:documentation>Relax NG flat schema for our library</a:documentation>
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  <define name="author-element">
    <a:documentation>Definition of the author element</a:documentation>
    <element name="author">
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
  .../...
</grammar>
```

An equivalent schema using the compact syntax is:

```
## Relax NG flat schema for our library
```

```
grammar{



start = element library { book-element+ }



## Definition of the author element
author-element =
 element author {
   attribute id { text },
   name-element,
   born-element,
   died-element?
 }
 .../...
}
```

Note the syntax with the leading double sharps (##) analogous to the `/**` comments used in JavaDoc and also the fact that even though they look like comments, these are annotations which have the same meaning and rules than initial annotations and must precede the pattern to which they apply. In fact, this syntax is only a shortcut to the corresponding annotations and this is equivalent to:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"



[ a:documentation [ "Relax NG flat schema for our library" ] ]



grammar {
 start = element library { book-element+ }



 [ a:documentation [ "Definition of the author element" ] ]
 author-element =
  element author {
    attribute id { text },
    name-element,
    born-element,
    died-element?
  }
  ../..
}
```

Also note that this shortcut has the same restrictions than initial annotations and that they must precede all the initial annotations. It is possible to mix them with other types of annotations and write for instance:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"



a:documentation [ "Relax NG flat schema for our library" ]
start = element library { book-element+ }



## Definition of the author element
author-element =
  element author {
    attribute id { text },
    name-element,
    born-element,
    died-element?
  }
.../...
```

Up to now, we have seen examples of these compatibility comments which were the first child element in their parent and this has been hiding an important feature of these comments: they are using the last trick we've mentioned in the previous section about workarounds for annotating `param` and `value` patterns and apply to the preceding sibling from the Relax NG namespace when there is one. This means that if we want to annotate the reference to the `name-element` definition, we can either write:

```
<define name="author-element">
  <element name="author">
    <attribute name="id"/>
    <ref name="name-element">
      <a:documentation>Definition of the author element</a:documentation>
    </ref>
    <ref name="born-element"/>
    <optional>
      <ref name="died-element"/>
    </optional>
  </element>
</define>
```

or:

```
<define name="author-element">
  <element name="author">
    <attribute name="id"/>
    <ref name="name-element"/>
    <a:documentation>Definition of the author element</a:documentation>
    <ref name="born-element"/>
    <optional>
      <ref name="died-element"/>
    </optional>
  </element>
</define>
```

In the first case, the DTD compatibility annotation is the first child element of its parent element (`ref`) and applies to the `ref` pattern for this reason. In the second case, the annotation is not the first child element from the Relax NG namespace and applies to his preceding sibling, which is the `ref` pattern again.

The compact syntax has the same rules, and the following annotations are equivalent too:

author-element =

```
  element author {
    attribute id { text },




    ## Definition of the author element
    name-element,
    born-element,
    died-element?
  }
```

and:

author-element =

```
  element author {
    attribute id { text },
    name-element
    >> a:documentation [ "Definition of the author element" ],
    born-element,
    died-element?
  }
```

Here again, a following annotation is considered as an annotation of the `name-element` reference.

Of course, if we were annotating a `param` or `value` pattern we would have no other choice than to locate the annotation after the pattern and this is why this tricky mechanism has been introduced.

XHTML Annotations

XHTML seems like a natural choice for embedding documentation in Relax NG schemas and we have already seen several examples of such annotations. The main benefit of XHTML is to be so similar to HTML that it is known by pretty much anyone who has ever published a web page. A lot of documentation and books are available on XHTML and many editors can be used to edit XHTML documents. Furthermore, if you keep to a reasonable subset of XHTML (such as for instance XHTML Basic), you have a simple and generic language to write documentation. And of course, the work needed if you want to publish the result of the extraction of XHTML annotations as XHTML is minimal since your annotations are already XHTML.

You will find more information about XHTML on the W3C web site: http://www.w3.org/MarkUp/ as well as in specialized books such as "HTML & XHTML: The Definitive Guide" (O'Reilly) and "XHTML: Moving Toward XML" (M&T Books).

We have also seen many examples of XHTML annotations, such as:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:xhtml="http://www.w
  <xhtml:div>
    <xhtml:h1>Relax NG flat schema for our library</xhtml:h1>
    <xhtml:p>This schema has been written by <xhtml:a href="http://dyomedea.co
  </xhtml:div>
  .../...
</grammar>
```

or, using the compact syntax:

```
namespace xhtml = "http://www.w3.org/1999/xhtml"



xhtml:div
[
   xhtml:h1 [ "Relax NG flat schema for our library" ]
   xhtml:p
   [
      "This schema has been written by "
      xhtml:a [ href = "http://dyomedea.com/vdv" "Eric van der Vlist" ]
      "."
   ]
]



start = element library { book-element+ }
.../...
```

Beyond the syntax which has already been discussed in the first part of this chapter, note how we have embedded a title (`xhtml:h1`) and a paragraph (`xhtml:p`) within a divisions (`xhtml:div`). This is generally a good practice which makes it easier to associate the title with the rest of the content and to manipulate the annotation as a whole.

DocBook Annotations

First design as a SGML application, DocBook is now also a XML language very popular for writing technical documentations. Beyond features which can be compared to those of XHTML, DocBook offers many predefined bells and whistles to facilitate indexes and cross references, to say that a text is a snippet of source code, identify acronyms and many other things. These features can be emulated in XHTML using the `class` attribute, but in DocBook they are built-in from the beginning and people agree with their meaning.

You will find more information about DocBook on its web site: http://www.oasis-open.org/committees/docbook/ and in the book "DocBook: The Definitive Guide" (O'Reilly).

DocBook is defined as a DTD which doesn't use any namespace and that's not an issue since Relax NG allows annotations through elements without namespace. To give you an idea of what DocBook looks like as well as an example showing how to "undeclare" a namespace in XML, the following example would match more or less what we had written in XHTML in the previous one:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
```

```
  <sect1 xmlns="">
    <title>Relax NG flat schema for our library</title>
    <para>This schema has been written by <xref linkend="vdv"/>.</para>
  </sect1>
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  .../...
</grammar>
```

Or, with the compact syntax:

```
sect1 [
  title [ "Relax NG flat schema for our library" ]
  para [
    "This schema has been written by "
    xref [ linkend = "vdv" ]
    "."
  ]
]
start = element library { book-element+ }
.../...
```

Dublin Core Annotations

While XHTML and DocBook are great to include content as documentation, Dublin Core is widely used over the web to include metadata about all type of resources and includes a set of elements with a description of their semantics which provide interoperable information including notions very relevant in a schema such as the name of its authors, their organization, the date, the copyright associated with the schema or the subjects which are relevant. Dublin Core is very complementary to DocBook and XHTML and Dublin Core is often use in XHTML documents where it finds a natural fit in the `meta` elements.

You will find more information about Dublin Core on their site: http://dublincore.org/ .

In a Relax NG schema, Dublin Core elements may be included wherever it makes sense: under the `grammar` pattern, they will qualify the whole grammar while under an `element` pattern, they would be useful to qualify the specific element.

A more complete example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns="http://relaxng.org
  <dc:title>The library element</dc:title>
  <dc:creator>Eric van der Vlist</dc:creator>
  <dc:subject>library, book, relax ng</dc:subject>
  <dc:description>This Relax NG schema has been written as an example to show
  <dc:date>2003-01-30</dc:date>
  <dc:language>en</dc:language>
  <dc:rights>Copyright Eric van der Vlist, Dyomedea.
    During development, I give permission for non-commercial copying for
```

```
    educational and review purposes.
    After publication, all text will be released under the
    Free Software Foundation GFDL.</dc:rights>
 .../...
</grammar>
```

or:

namespace dc = "http://purl.org/dc/elements/1.1/"

```
dc:title [ "The library element" ]
dc:creator [ "Eric van der Vlist" ]
dc:subject [ "library, book, relax ng" ]
dc:description [
  "This Relax NG schema has been written as an example to show how Dublin Core
]
dc:date [ "2003-01-30" ]
dc:language [ "en" ]
dc:rights [
  "Copyright Eric van der Vlist, Dyomedea. \x{a}" ~
  "    During development, I give permission for non-commercial copying for \x
  "    educational and review purposes. \x{a}" ~
  "    After publication, all text will be released under the \x{a}" ~
  "    Free Software Foundation GFDL."
]
.../...
```

SVG Annotations

There is no reason to limit ourselves to text and metadata and graphics can be included, such as:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:svg="http://www.w3.
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  <define name="author-element">
    <element name="author">
      <svg:svg>
        <svg:title>A typical author</svg:title>
        <svg:ellipse style="stroke:#000000; fill:#e3e000; stroke-width:2pt;" i
        <svg:ellipse style="stroke:none; fill:#7f7f7f; " id="leftEye" cx="240"
        <svg:ellipse style="stroke:none; fill:#7f7f7f; " id="rightEye" cx="320
        <svg:path style="fill:none;stroke:#7F7F7F; stroke-width:5pt;" id="mout
      </svg:svg>
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
```

```
      </optional>
    </element>
  </define>
  .../...
</grammar>
```

Or, using the compact syntax:

```
namespace svg = "http://www.w3.org/2000/svg"



start = element library { book-element+ }
author-element =
  [
    svg:svg [
      svg:title [ "A typical author" ]
      svg:ellipse >[
        style = "stroke:#000000; fill:#e3e000; stroke-width:2pt;"
        id = "head"
        cx = "280"
        cy = "250"
        rx = "110"
        ry = "130"
      ]
      svg:ellipse [
        style = "stroke:none; fill:#7f7f7f; "
        id = "leftEye"
        cx = "240"
        cy = "225"
        rx = "18"
        ry = "18"
      ]
      svg:ellipse [
        style = "stroke:none; fill:#7f7f7f; "
        id = "rightEye"
        cx = "320"
        cy = "225"
        rx = "18"
        ry = "18"
      ]
      svg:path [
        style = "fill:none;stroke:#7F7F7F; stroke-width:5pt;"
        id = "mouth"
        d = "M 222 280 A 58 48 0 0 0 338 280"
      ]
    ]
  ]
  element author {
    attribute id { text },
    name-element,
    born-element,
    died-element?
  }
 .../...
```

I leave it to you as an additional exercise to visualize what a typical author looks like! Note we could have included a UML representation of the author element instead.

The Scalable Vector Graphics (SVG) is a XML vocabulary published by the W3C. You will find more information about SVG on its web site: http://www.w3.org/Graphics/SVG/ as well as in the book "SVG Essentials" (O'Reilly).

RDDL Annotations

The last type of annotation I'd like to mention here is a good transition between annotations for documentation purposes which we are seeing in this section and annotation for applications which we will see in the next section: RDDL has been designed as a XML vocabulary which can be used both by humans as documentation and by applications! Although RDDL has been invented to document namespaces, it can find a very good fit in a Relax NG schema from which it can be extracted to constitute RDDL documentations for the namespaces described in the schema. RDDL is based on XHTML and XLink and plays well with XHTML documentation.

You will find more information about RDDL on its web site: http://rddl.org .

RDDL main benefit is to provide a way to associate resources with a document and could be used to associate for instance a XSLT template and a CSS style to the definition of the author element:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:rddl="http://www.rddl.org/"
         xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <start>
    <element name="library">
      <oneOrMore>
        <ref name="book-element"/>
      </oneOrMore>
    </element>
  </start>
  <define name="author-element">
    <element name="author">
      <xhtml:div>
        <rddl:resource id="author-transform" xlink:arcrole="http://www.w3.org/1
          xlink:role="http://www.w3.org/1999/XSL/Transform" xlink:title="Autho
          xlink:href="library.xslt#author">
          <xhtml:div class="resource">
            <xhtml:h4>XSLT Transformation</xhtml:h4>
            <xhtml:p>This <xhtml:a href="library.xslt#author">XSLT template</xh
          </xhtml:div>
        </rddl:resource>
        <rddl:resource id="CSS" xlink:title="CSS Stylesheet"
          xlink:role="http://www.isi.edu/in-notes/iana/assignments/media-types/
          <xhtml:div class="resource">
            <xhtml:h4>CSS Stylesheet</xhtml:h4>
            <xhtml:p>A <xhtml:a href="author.css">CSS stylesheet</xhtml:a> defi
          </xhtml:div>
        </rddl:resource>
      </xhtml:div>
      <attribute name="id"/>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
```

```
      </optional>
    </element>
  </define>
  .../...
</grammar>
```

or:

```
namespace rddl = "http://www.rddl.org/"
namespace xhtml = "http://www.w3.org/1999/xhtml"
namespace xlink = "http://www.w3.org/1999/xlink"



start = element library { book-element+ }
author-element =
  [
    xhtml:div [
      rddl:resource [
        id = "author-transform"
        xlink:arcrole = "http://www.w3.org/1999/xhtml"
        xlink:role = "http://www.w3.org/1999/XSL/Transform"
        xlink:title = "Author XSLT template"
        xlink:href = "library.xslt#author"
        xhtml:div [
          class = "resource"
          xhtml:h4 [ "XSLT Transformation" ]
          xhtml:p [
            "This "
            xhtml:a [ href = "library.xslt#author" "XSLT template" ]
            " displays the description of an author as XHTML."
          ]
        ]
      ]
      rddl:resource [
        id = "CSS"
        xlink:title = "CSS Stylesheet"
        xlink:role =
          "http://www.isi.edu/in-notes/iana/assignments/media-types/text/css"
        xlink:href = "author.css"
        xhtml:div [
          class = "resource"
          xhtml:h4 [ "CSS Stylesheet" ]
          xhtml:p [
            xhtml:a [ href = "author.css" "CSS stylesheet" ]
            " defining some cool styles to display an author."
          ]
        ]
      ]
    ]
  ]
  element author {
    attribute id { text },
    name-element,
    born-element,
    died-element?
```

```
  }
 .../...
```

Annotation for applications

As mentioned in the introduction of this chapter, common uses of annotations by applications include using them as pre-processing instructions, as helpers for generating other schemas out of a Relax NG schema and as extensions to Relax NG itself.

Annotations for pre-processing

An interesting application of annotation for pre-processing has been proposed by Bob DuCharme and can be used to derive specific schemas by restriction out of a generic schema. The benefits of this approach are that it is extremely simple an provides a very straightforward workaround to the lack of derivation by restriction of Relax NG. It is also language neutral and can be applied to other schema languages such as W3C XML Schema where it is much simpler than the derivation by restriction feature built into the language.

You can find Bob DuCharme's proposal on the web: http://www.snee.com/xml/schemaStages.html and can download the XSLT transformation implementing it at http://www.snee.com/xml/schemaStages.zip.

The idea is to add annotations in elements which needs to be removed in a variant of the schema and to use these annotations to generate the different variants using a XSLT transformation. Each variant is called a stage. The list of the available stages is declared in a `sn:stages` element and for each element which is conditional, the list of the stages in which it needs to be kept is declared through a `sn:stages` attributes.

Since this technique is using annotations, the global schema can still be a valid schema which will validate a superset of the instance documents valid per each of the stages.

If we wanted to derive schemas requiring either a `book`, `author`, `library` or `character` element or both `book` or `author` as a document element from a generic schema allowing any of these, we could write:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:sn="http://www.snee
  <sn:stages>
    <sn:stage name="library"/>
    <sn:stage name="book"/>
    <sn:stage name="author"/>
    <sn:stage name="character"/>
    <sn:stage name="author-or-book"/>
  </sn:stages>
  <start>
    <choice>
      <ref name="library-element" sn:stages="library"/>
      <ref name="book-element" sn:stages="book author-or-book"/>
      <ref name="author-element" sn:stages="author author-or-book"/>
      <ref name="character-element" sn:stages="character"/>
    </choice>
  </start>
  .../...
</grammar>
```

or:

```
namespace sn = "http://www.snee.com/ns/stages"



sn:stages [
  sn:stage [ name = "library" ]
  sn:stage [ name = "book" ]
  sn:stage [ name = "author" ]
  sn:stage [ name = "character" ]
  sn:stage [ name = "author-or-book" ]
]
start =
  [ sn:stages = "library" ] library-element
  | [ sn:stages = "book author-or-book" ] book-element
  | [ sn:stages = "author author-or-book" ] author-element
  | [ sn:stages = "character" ] character-element
.../...
```

This schema is a valid Relax NG schema which would accept any of these elements as a root. A transformation of the XML syntax through the XSLT transformation "getStage.xsl" provided in the zip file mentioned above with a parameter `stageName` set to `author-or-book` would remove all the elements with a `sn:stage` attribute that do not have `author-or-book` in their list of values:

```
$ xsltproc --stringparam stageName author-or-book getStage.xsl doc-snee.rng
<?xml version="1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:sn="http://www.snee



 <start>
   <choice>



     <ref name="book-element"/>
     <ref name="author-element"/>



   </choice>
 </start>
.../...
</grammar>
```

This transformation has thus performed a restriction on the schema and as many schemas can be generated this way as stages have been declared in the `sn:stages` element.

Annotations for conversion

We will see in "Appendix B: Using Relax NG As a Pivot Format" that Relax NG is a good fit to be a "pivot" format, i.e. a reference format in which schemas are kept and transformed into other languages.

One of the limits of this approach is that features which are part of the target languages and not part of Relax NG seem to be out of reach and that would be true if we had no annotations. The two most notorious examples of such annotations are for generating DTDs and W3C XML Schema.

Annotations to generate DTDs

This is the third and last facet of the DTD Compatibility specifications and this deals with default values for attributes which can be declared using a `a:defaultValue` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<element xmlns="http://relaxng.org/ns/structure/1.0" xmlns:a="http://relaxng.o
  <oneOrMore>
    <element name="book">
      <attribute name="id"/>
       <optional>
        <attribute name="available" a:defaultValue="true">
          <choice>
            <value>true</value>
            <value>false</value>
          </choice>
        </attribute>
      </optional>
      .../...
    </element>
  </oneOrMore>
   .../...
</element>
```

or:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"



element library {
  element book {
    attribute id { text },
    [ a:defaultValue = "true" ]
    attribute available { "true" | "false" }?,
    element isbn { text },
    element title {
      attribute xml:lang { text },
      text
    },
    .../...
  }+
}
```

The attribute needs to be declared as optional to use this feature and that means that there is no impact on the validation by a Relax NG processor. However, converters such as Trang will use this annotation to generate a default value in a DTD:

```
<!ATTLIST book
```

```
id CDATA #REQUIRED
available (true|false) 'true'>
```

Annotations to generate W3C XML Schema schemas

There is no official specification about how to generate W3C XML Schema schemas from Relax NG and what we will say in this small section is derived from the documentation of Trang available on the web at http://www.thaiopensource.com/relaxng/trang-manual.html.

The first thing to note is that Trang does support the `a:defaultValue` attribute and that the schema presented above would be translated as:

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="isbn"/>
      <xs:element ref="title"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="author"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="character"/>
    </xs:sequence>
    <xs:attribute name="id" use="required"/>
    <xs:attribute name="available" default="true">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="true"/>
          <xs:enumeration value="false"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
```

Note the `default` attribute in the declaration of the `available` attribute.

In addition to this annotation, James Clark has created a specific namespace: http://www.thaiopensource.com/ns/relaxng/xsd to control the translation to W3C XML Schema. This translation is far from being obvious and a Relax NG schema can often be translated using different features of W3C XML Schema. James Clark has made a lot of choices in his implementation based on best practices, but there are still some options which are really context dependent and for which the users may be given a choice.

In the current version (as of 30 January 2003), there is only one annotation attribute available to perform such choices, the `tx:enableAbstractElements` attribute which may be included in `grammar`, `div` or `include`. This attribute can take the values `true` or `false` and controls whether abstract elements may be used in substitution groups. This is a fairly advanced feature of W3C XML Schema and we won't present it here nor give any example. You will find more information on this feature in my tutorial on XML.com: http://xml.com/pub/a/2000/11/29/schemas/part1.html or in my book "XML Schema: The W3C Object-Oriented Descriptions for XML" (O'Reilly).

The Trang manual indicates that more annotations might be added in the future.

Schema Adjunct Framework

The Schema Adjunct Framework (SAF) is more or less falling into this category as well. SAF is a generic framework to store processing information in relation with schemas and can work either as standalone or as "schema adornments", i.e. annotations embedded in schemas. Although it has been

developed to work with W3C XML Schema, there is no reason that it couldn't be used to adorn Relax NG schemas.

You can find more information about SAF on the web: http://www.tibco.com/solutions/products/extensibility/resources/saf.jsp

The momentum behind SAF seems to have decreased a lot since end of 2001, but this is definitely something to check if you need to add processing information in a schema. A simplified example for a SAF adornment in Relax NG could be:

```
<define name="author-element">
  <sql:select>select <sql:elem>name</sql:elem>, <sql:elem>birthdate</sql:elem>
    from tbl_author</sql:select>
  <element name="author">
    <attribute name="id"/>
    <ref name="name-element"/>
    <ref name="born-element"/>
    <optional>
      <ref name="died-element"/>
    </optional>
  </element>
</define>
```

or:

```
[
  sql:select [
    "select "
    sql:elem [ "name" ]
    ", "
    sql:elem [ "birthdate" ]
    ", "
    sql:elem [ "deathdate" ]
    " from tbl_author"
  ]
]
author-element =
  element author {
    attribute id { text },
    name-element,
    born-element,
    died-element?
  }
```

Annotations for extension

Annotations can also be used as extensions to influence the behavior of the Relax NG processors which support them which is more controversial but can also be very useful. The two applications which I am aware of in this category are for embedding Schematron rule and my own XVIF project which allows to define pipes of validations and transformations that act as Relax NG patterns.

Embedded Schematron rules

Schematron is a XML schema language rather untypical since instead of being grammar based like Relax NG and focus on describing documents, Schematron is rule based and consists in lists of rules

to check on documents. Giving the exhaustive list of all the rules needed to validate a document is a very verbose and error prone task but on the other hand, the ability to write your own rules gives a flexibility and a power which can't be matched by a grammar based schema language. The two types of languages appear thus to be more complementary than competitors and using both together allows to take the best of each of them.

You will find more information about Schematron on its web site: http://www.ascc.net/xml/resource/schematron/schematron.html.

Schematron is a good fit, for instance, if we want to check that the `id` attribute of our book element is composed of the ISBN number prefixed by the letter `b`. In this case we would write:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:s="http://www.ascc.
  <define name="book-element">
    <element name="book">
      <s:rule context="book">
        <s:assert test="@id = concat('b', isbn)"> The id needs to be the isbn
      </s:rule>
      <attribute name="id"/>
      <attribute name="available"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
    </element>
  </define>
  .../...
</grammar>
```

or:

```
namespace s = "http://www.ascc.net/xml/schematron"



book-element =
  [
    s:rule [
      context = "book"
      s:assert [
        test = "@id = concat('b', isbn)"
        ' The id needs to be the isbn number prefixed by "b" '
      ]
    ]
  ]
  element book {
    attribute id { text },
    attribute available { text },
    isbn-element,
    title-element,
```

```
    author-element*,
    character-element*
  }
.../...
```

The Schematron annotation is composed of a `rule` element which is setting the context and embedded `assert` elements defining assertions. Instead of `assert`, `report` elements can also be used which are the opposite of assertions and report errors when they are true. These checks are applied to all the elements meeting the XPath expression expressed in the `context` attribute of the `rule` elements and the `test` attribute of the `assert` or `report` elements are also XPath expressions.

At this point, we must note that there is a difference of appreciation between implementations on the scope in which the rules must be applied leading to potential issues of interoperability between implementations.

On one side, the Schematron specification states that when Schematron rules are embedded in another language, they must be collected and bundled into a Schematron schema independently of where they have been found in the original schema. In other words, this means that the rule which we has been defined above should be applied to all the `book` elements in the instance documents. This is the approach taken by the Topologi multi validator (see http://www.topologi.com/products/validator/index.html).

On the other side, when a schematron rule is embedded in a Relax NG `element` pattern like it is the case here, it is rather tempting to evaluate the rule in the context of the pattern. In that case, the rule will only apply to the `book` elements which are included in the context node and if the rule fails, the element pattern will fail and other alternatives will be checked. This is the approch taken by Sun's Multi Schema Validator (see http://wwws.sun.com/software/xml/developers/multischema/).

The difference can be seen in an example such as:

```
<define name="book-element">
  <choice>
    <element name="book">
      <s:rule context="book">
        <s:assert test="@id = concat('b', isbn)"> The id needs to be the isbn
      </s:rule>
      <attribute name="id"/>
      <attribute name="available"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
    </element>
    <element name="book">
      <attribute name="id">
        <value>ggjh0836217462</value>
      </attribute>
      <attribute name="available"/>
      <ref name="isbn-element"/>
      <ref name="title-element"/>
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
```

```
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
    </element>
  </choice>
</define>
```

In this case, the approach taken by the Schematron specification would lead to consider an instance document with a book id equal to "ggjh0836217462" as invalid since the evaluation of the Schematron rules is completely decoupled from the validation by the Relax NG schema and the approach taken by MSV would consider the same document as valid since it's meeting one of the alternative definitions for the book element.

XVIF

The interoperability issue mentioned above is a good illustration of the difficulty to mix elements from different languages which have been specified independently and the XML Validation Interoperability Framework (XVIF) is a proposal for a framework which would take care of this kind of issues. You will find more information on XVIF at its home page: http://downloads.xmlschemata.org/python/xvif/.

The principle of XVIF is to define "micro pipes" of transformations and validations which can be embeded in different transformation and validation languages. When the host language is Relax NG, these micro pipes behave as Relax NG patterns.

There are many use cases for such micro pipes and one of them is to include transformations to fit text nodes into existing datatypes. For instance, we have been lucky enough to have dates which are using the ISO 8601 format in our documents, but we could as well have had let's say French date formats. In this case, a set of regular expressions can be defined to do the transformation between these dates and the ISO 8601 format and XVIF gives a way to integrate these regular expressions in a Relax NG schema:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:if="http://namespac
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <define name="born-element">
    <element name="born">
      <if:pipe>
        <if:validate type="http://namespaces.xmlschemata.org/xvif/regexp"
                     apply="m/[0-9]+ .+ [0-9]+/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/^[ \t\n]*([0-9] .*)$/0\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) janvier ([0-9]+)/\2-01-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) fevrier ([0-9]+)/\2-02-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) mars ([0-9]+)/\2-03-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) avril ([0-9]+)/\2-04-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) mai ([0-9]+)/\2-05-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) juin ([0-9]+)/\2-06-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) juillet ([0-9]+)/\2-07-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) aout ([0-9]+)/\2-08-\1/"/>
```

```
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) septembre ([0-9]+)/\2-09-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) octobre ([0-9]+)/\2-10-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) novembre ([0-9]+)/\2-11-\1/"/>
        <if:transform type="http://namespaces.xmlschemata.org/xvif/regexp"
                      apply="s/([0-9]+) decembre ([0-9]+)/\2-12-\1/"/>
        <if:validate type="http://relaxng.org/ns/structure/1.0">
          <if:apply>
            <data type="date">
              <param name="minInclusive">1900-01-01</param>
              <param name="maxInclusive">2099-12-31</param>
            </data>
          </if:apply>
        </if:validate>
      </if:pipe>
      <text if:ignore="1"/>
    </element>
  </define>
  .../...
</grammar>
```

or:

```
namespace if = "http://namespaces.xmlschemata.org/xvif/iframe"
namespace rng = "http://relaxng.org/ns/structure/1.0"



datatypes d = "http://relaxng.org/ns/compatibility/datatypes/1.0"



born-element =
  [
    if:pipe [
      if:validate [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "m/[0-9]+ .+ [0-9]+/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/^[ \t\n]*([0-9] .*)$/0\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) janvier ([0-9]+)/\2-01-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) fevrier ([0-9]+)/\2-02-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
```

```
        apply = "s/([0-9]+) mars ([0-9]+)/\2-03-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) avril ([0-9]+)/\2-04-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) mai ([0-9]+)/\2-05-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) juin ([0-9]+)/\2-06-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) juillet ([0-9]+)/\2-07-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) aout ([0-9]+)/\2-08-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) septembre ([0-9]+)/\2-09-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) octobre ([0-9]+)/\2-10-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) novembre ([0-9]+)/\2-11-\1/"
      ]
      if:transform [
        type = "http://namespaces.xmlschemata.org/xvif/regexp"
        apply = "s/([0-9]+) decembre ([0-9]+)/\2-12-\1/"
      ]
      if:validate [
        type = "http://relaxng.org/ns/structure/1.0"
        if:apply [
          rng:data [
            type = "date"
            rng:param [ name = "minInclusive" "1900-01-01" ]
            rng:param [ name = "maxInclusive" "2099-12-31" ]
          ]
        ]
      ]
    ]
  ]
  element born { [ if:ignore = "1" ] text }
```

In this example, we have defined a pipe (`if:pipe`) of twelve transformations (`if:transform`) using regular expressions and each of them converting one of the twelve months and a final validation (`if:validate`) which itself is using Relax NG to check that the result is a ISO 8601 date between 1900 and 2099. The `text` pattern has a `if:ignore` attribute showing to XVIF compliant processors that it's a fallback pattern for the other Relax NG processors.

Chapter 15. Chapter 14: Generating Relax NG schemas

In the previous chapter (Chapter 12: Annotating Schemas) we have seen how information could be added to Relax NG schemas to make them more readable and also to improve the ability to extract information from the schemas and transform them into other useful documents such as documentation, diagrams or even applications. The underlying assumption up to now in this book has been that Relax NG was a natural level at which we would work and that we would be editing these schemas and this is certainly a valid point of view. However, a Relax NG schema (and any XML schema in general) is a physical model of a class of instance documents and we could also want to work at another level and generate our Relax NG schemas from this other level.

From this other point of view, Relax NG shines as an ideal choice for a target language because of its almost complete lack of restrictions. This lack of restrictions means that during the transformation of a model into a Relax NG schema, you won't have to worry with things such as "I must declare all my attributes after my elements", "I should take care to disallow unordered models in such and such circumstances", "if I have already declared this content here, I can't declare it again here", ... In other words, it will make your life much easier and let you concentrate on which document you want to get instead of having to worry about the constraints of the schema language.

What are these other levels on which we might want to work? In fact we could want to be either more concrete or more abstract than Relax NG and could either have a "bottom up" or "top down" approach. Proponents of a "bottom up" approach would enjoy working with instance documents rather than with schemas and Examplotron has been designed for them while the adepts of a "top down" approach will want to work at a higher level and use a methodology such as UML to model their documents. These two approaches may lead to many other variants and we will also see how literate programmers can include Relax NG patterns in their documentations and when you may want to replace your Relax NG schema by a simple spreadsheet.

Examplotron: the instance document is its own schema

I have created Examplotron in March 2001 from a very simple idea: when you want to describe the element `foo`, why would you have to learn yet another language and write "<element name='foo'><empty/></element>' or 'element foo {empty}'? Wouldn't it be so much simpler to just write the element in plain XML: "<foo/>"? Or, in other terms, instead of describing instance documents, why couldn't we show them?

The first implementation, published with the original description of Examplotron relied on a double XSLT transformation: the Examplotron "schema" was compiled by a XSLT transformation into another XSLT transformation which performed the validation of the instance documents. The concept has received many good comments when I have announced it, but nothing very significant did happen: moving along to add new features would have meant to redefine the full semantics of a new schema language and the implementation as a XSLT transformation was becoming very complex and the project was stalled until I realized the potential of using Relax NG as a target format.

Since its release 0.5, Examplotron is now implemented as a XSLT transformation transforming its schema into Relax NG and due to the potential of this approach, Examplotron has made more progress in two weeks than in two years under the previous architecture!

Ten minutes guide to Examplotron

Let's take a snippet of our example document:

```
<?xml version="1.0" encoding="utf-8"?>
<character id="Snoopy">
  <name>Snoopy</name>
  <born>1950-10-04</born>
  <qualification>extroverted beagle</qualification>
</character>
```

The good news is that this is an Examplotron schema and to get an idea of what this schema means, we can translate it into a Relax NG schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
 xmlns:ega="http://examplotron.org/annotations/"
 xmlns:sch="http://www.ascc.net/xml/schematron"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
   <start>
      <element name="character">
         <optional>
            <attribute name="id">
               <ega:example id="Snoopy"/>
            </attribute>
         </optional>
         <element name="name">
            <text>
               <ega:example>Snoopy</ega:example>
            </text>
         </element>
         <element name="born">
            <data type="date">
               <ega:example>1950-10-04</ega:example>
            </data>
         </element>
         <element name="qualification">
            <text>
               <ega:example>extroverted beagle</ega:example>
            </text>
         </element>
      </element>
   </start>
</grammar>
```

or:

```
namespace ega = "http://examplotron.org/annotations/"
namespace sch = "http://www.ascc.net/xml/schematron"



start =
  element character {
    [ ega:example [ id = "Snoopy" ] ] attribute id { text }?,
    element name { [ ega:example [ "Snoopy" ] ] text },
    element born { [ ega:example [ "1950-10-04" ] ] xsd:date },
```

```
    element qualification {
      [ ega:example [ "extroverted beagle" ] ] text
    }
  }
```

We see there that the Examplotron, schema has not only the same modeling power than its Relax NG counterpart, but that annotations need to be added to the Relax NG schema if we don't want to loose the "examples" provided in Examplotron which are useful for documentation purposes and also to allow the reverse transformation (Relax NG to Examplotron) if that was needed.

The other thing to note in this example is that Examplotron is making inferences from what is found in the schema: here, Examplotron has assumed that the order between `name`, `born` and `qualification` is significant, that these elements are mandatory, that the id attribute is optional, that the `born` element has a type `xsd:date` and that all the other elements and attribute are just text. These assumptions are designed to catch what is most likely to have been the intention of the designer of the document. The idea behind this is to bet that most of the time, people won't have to do anything to tweak their Examplotron schema.

What's happening when Examplotron got it wrong? There is no magic here: if you want to go against the inferences of Examplotron, you need to say it. And the way to say it is through annotating the Examplotron schema. To say that the `qualification` element is optional, you would for instance add an `eg:occurs` attribute with a value `?` and to say that the `id` attribute has a type `dtd:ID`, you would have to set its content to `{dtd:id}`:

```
<?xml version="1.0" encoding="utf-8"?>
<character id="{dtd:id}" xmlns:eg="http://examplotron.org/0/">
  <name>Snoopy</name>
  <born>1950-10-04</born>
  <qualification eg:occurs="?">extroverted beagle</qualification>
</character>
```

And this would be translated as:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
 xmlns:ega="http://examplotron.org/annotations/"
 xmlns:sch="http://www.ascc.net/xml/schematron"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
   <start>
      <element name="character">
         <optional>
            <attribute name="id">
               <data type="id" datatypeLibrary="http://relaxng.org/ns/compatib
            </attribute>
         </optional>
         <element name="name">
            <text>
               <ega:example>Snoopy</ega:example>
            </text>
         </element>
         <element name="born">
            <data type="date">
               <ega:example>1950-10-04</ega:example>
            </data>
         </element>
```

```
         <optional>
            <element name="qualification">
               <text>
                  <ega:example>extroverted beagle</ega:example>
               </text>
            </element>
         </optional>
      </element>
   </start>
</grammar>
```

or:

```
namespace ega = "http://examplotron.org/annotations/"
namespace sch = "http://www.ascc.net/xml/schematron"



datatypes d = "http://relaxng.org/ns/compatibility/datatypes/1.0"



start =
  element character {
    attribute id { d:id }?,
    element name { [ ega:example [ "Snoopy" ] ] text },
    element born { [ ega:example [ "1950-10-04" ] ] xsd:date },
    element qualification {
      [ ega:example [ "extroverted beagle" ] ] text
    }?
  }
```

If you compare the compact syntax and the Examplotron schema, you see that we have something of similar conciseness, the compact syntax looking more formal and Examplotron looking more "visual". However, given the rules described in the documentation of Examplotron, these two schemas are equivalent and a round trip is possible (Examplotron to Relax NG and back from Relax NG to Examplotron).

One can go pretty far with these annotations as shown in this more complete example which defines contents as `interleave`, mandatory attributes and defines the content of the complex elements as named patterns:

```
<?xml version="1.0" encoding="utf-8"?>
<library xmlns:eg="http://examplotron.org/0/" eg:content="eg:interleave" eg:de
  <book available="true" eg:occurs="*" eg:define="book-content">
    <eg:attribute name="id" eg:content="dtd:id">b0836217462</eg:attribute>
    <isbn>0836217462</isbn>
    <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
    <author eg:occurs="+" eg:define="author-content" eg:content="eg:interleave
      <eg:attribute name="id" eg:content="dtd:id">CMS</eg:attribute>
      <name>Charles M Schulz</name>
      <born>1922-11-26</born>
      <died>2000-02-12</died>
```

```
    </author>
    <character eg:define="character-content" eg:content="eg:interleave">
      <eg:attribute name="id" eg:content="dtd:id">PP</eg:attribute>
      <name>Peppermint Patty</name>
      <born>1966-08-22</born>
      <qualification>bold, brash and tomboyish</qualification>
    </character>
    <character id="Snoopy">
      <name>Snoopy</name>
      <born>1950-10-04</born>
      <qualification>extroverted beagle</qualification>
    </character>
    <character id="Schroeder">
      <name>Schroeder</name>
      <born>1951-05-30</born>
      <qualification>brought classical music to the Peanuts strip</qualificati
    </character>
    <character id="Lucy">
      <name>Lucy</name>
      <born>1952-03-03</born>
      <qualification>bossy, crabby and selfish</qualification>
    </character>
  </book>
</library>
```

The schema generated by this Examplotron is:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:ega="http://examplo
   <start>
      <element name="library">
         <ref name="library-content" ega:def="true"/>
      </element>
   </start>
   <define name="library-content">
      <interleave>
         <zeroOrMore>
            <element name="book">
               <ref name="book-content" ega:def="true"/>
            </element>
         </zeroOrMore>
      </interleave>
   </define>
   <define name="book-content">
      <optional>
         <attribute name="available">
            <data type="boolean">
               <ega:example available="true"/>
            </data>
         </attribute>
      </optional>
      <attribute name="id">
         <ega:skipped>b0836217462</ega:skipped>
         <data type="id" datatypeLibrary="http://relaxng.org/ns/compatibility/
      </attribute>
      <element name="isbn">
```

```
            <data type="integer">
               <ega:example>0836217462</ega:example>
            </data>
         </element>
         <element name="title">
            <optional>
               <attribute name="lang" ns="http://www.w3.org/XML/1998/namespace">
                  <ega:example xml:lang="en"/>
               </attribute>
            </optional>
            <text>
               <ega:example>Being a Dog Is a Full-Time Job</ega:example>
            </text>
         </element>
         <oneOrMore>
            <element name="author">
               <ref name="author-content" ega:def="true"/>
            </element>
         </oneOrMore>
         <oneOrMore>
            <element name="character">
               <ref name="character-content" ega:def="true"/>
            </element>
         </oneOrMore>
         <ega:skipped>
            <character xmlns="" xmlns:eg="http://examplotron.org/0/" id="Snoopy">
               <name>Snoopy</name>
               <born>1950-10-04</born>
               <qualification>extroverted beagle</qualification>
            </character>
         </ega:skipped>
         <ega:skipped>
            <character xmlns="" xmlns:eg="http://examplotron.org/0/" id="Schroede
               <name>Schroeder</name>
               <born>1951-05-30</born>
               <qualification>brought classical music to the Peanuts strip</quali
            </character>
         </ega:skipped>
         <ega:skipped>
            <character xmlns="" xmlns:eg="http://examplotron.org/0/" id="Lucy">
               <name>Lucy</name>
               <born>1952-03-03</born>
               <qualification>bossy, crabby and selfish</qualification>
            </character>
         </ega:skipped>
      </define>
      <define name="author-content">
         <interleave>
            <attribute name="id">
               <ega:skipped>CMS</ega:skipped>
               <data type="id" datatypeLibrary="http://relaxng.org/ns/compatibili
            </attribute>
            <element name="name">
               <text>
                  <ega:example>Charles M Schulz</ega:example>
               </text>
            </element>
            <element name="born">
```

```
            <data type="date">
               <ega:example>1922-11-26</ega:example>
            </data>
         </element>
         <element name="died">
            <data type="date">
               <ega:example>2000-02-12</ega:example>
            </data>
         </element>
      </interleave>
   </define>
   <define name="character-content">
      <interleave>
         <attribute name="id">
            <ega:skipped>PP</ega:skipped>
            <data type="id" datatypeLibrary="http://relaxng.org/ns/compatibili
         </attribute>
         <element name="name">
            <text>
               <ega:example>Peppermint Patty</ega:example>
            </text>
         </element>
         <element name="born">
            <data type="date">
               <ega:example>1966-08-22</ega:example>
            </data>
         </element>
         <element name="qualification">
            <text>
               <ega:example>bold, brash and tomboyish</ega:example>
            </text>
         </element>
      </interleave>
   </define>
</grammar>
```

Or (skipping some annotation for readability):

```
namespace eg = "http://examplotron.org/0/"
namespace ega = "http://examplotron.org/annotations/"
namespace sch = "http://www.ascc.net/xml/schematron"



datatypes d = "http://relaxng.org/ns/compatibility/datatypes/1.0"



start = element library { [ ega:def = "true" ] library-content }
library-content = element book { [ ega:def = "true" ] book-content }*
book-content =
  attribute available {
    [ ega:example [ available = "true" ] ] xsd:boolean
  }?,
  [ ega:skipped [ "b0836217462" ] ] attribute id { d:id },
```

```
  element isbn { [ ega:example [ "0836217462" ] ] xsd:integer },
  element title {
    [ ega:example [ xml:lang = "en" ] ] attribute lang { text }?,
    [ ega:example [ "Being a Dog Is a Full-Time Job" ] ] text
  },
  element author { [ ega:def = "true" ] author-content }+,
  (element character { [ ega:def = "true" ] character-content }+)
author-content =
  [ ega:skipped [ "CMS" ] ] attribute id { d:id }
  & element name { [ ega:example [ "Charles M Schulz" ] ] text }
  & element born { [ ega:example [ "1922-11-26" ] ] xsd:date }
  & element died { [ ega:example [ "2000-02-12" ] ] xsd:date }
character-content =
  [ ega:skipped [ "PP" ] ] attribute id { d:id }
  & element name { [ ega:example [ "Peppermint Patty" ] ] text }
  & element born { [ ega:example [ "1966-08-22" ] ] xsd:date }
  & element qualification {
      [ ega:example [ "bold, brash and tomboyish" ] ] text
    }
```

And for those of us who would like still more flexibility, the next versions should "import" all the Relax NG patterns in the Examplotron namespace so that we can use Relax NG compositors, patterns and name classes where needed.

Use scenarios

Why should we want to use Examplotron instead of Relax NG? We could reverse the question and ask why should we want to use Relax NG instead of Examplotron! At the end of the day, I think that it doesn't really matter. What's important is that the semantics of the validation engine is rock solid and has no limitations. As for the syntax which you'll be using to express your schemas, you can just use the one you like. And if you like the visual qualities of Examplotron, there is no reason to use any other: you will just be looking at a Relax NG schema under a different angle.

Literate Programming

A common approach to software documentation include extracting documentation from the source documents relying on the structure of the programs and their comments (a good example is JavaDoc, the documentation extracting tool shipped with Java and u universally used on Java projects). Other project keep code and documentation separate and for both approaches, it is often the case that documentation and comments evolve separately from the code and that the documentation becomes sooner or later out of date.

The reason for this is that the focus is on the code and that documentation is often considered as a side product, less important than the code. Donald Knuth, the inventor of the term "literate programming", claims on the contrary:

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names

of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other."

(Donald Knuth. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, pg. 99.)

Norm Walsh has adapted the concept to XML and tools for doing Literate Programming in XML are shipped under the name "litprog" by the DocBook project on SourceForge (http://sourceforge.net/projects/docbook/). The basic idea is to include snippet of code (or snippet of schemas in our case) within the documentation which can be written in any XML format including XHTML or DocBook. From this single document embedding code in documentation, a couple of XSLT transformation generate then a formated documentation and the source code.

Beyond the fact that you're working upside down compared to the common usage to add comments in the code, the other major practical difference is that you are now defining the relations between the snippets of code or schema using the mechanisms of "litprog" instead of using the mechanisms which are specific to each programming language. The granularity of the documentation becomes virtually independent of the granularity of your functions, methods, or in our case of the granularity of our named patterns. This has also the benefit, if needed, that we could group in a single literate documentation several languages and describe for instance the Relax NG schema of a document together with a XSLT transformation to manipulate the document and a DOM application to read it.

Out of the box

Literate Programming plays rather well with Relax NG as we will show on a small example. A literate programming document is a documentation embedding `src:fragment` elements to mark the fragments of schema which will assembled into complete schemas. The documentation can use any format such as DocBook, XHTML or even RDDL. Using XHTML, the description of the `name` element could be:

```
<div>
<h2>The <tt>name</tt> element</h2>
<p>This is the name of the character.</p>
<src:fragment id="name" xmlns="">
 <rng:element name="name">
   <rng:text/>
 </rng:element>
</src:fragment>
</div>
```

Or, with the compact syntax:

```
<div>
<h2>The <tt>name</tt> element</h2>
<p>This is the name of the character.</p>
<src:fragment id="name" xmlns="">
```

element name { text }

```
</src:fragment>
</div>
```

In this first snippet, the definition of our element doesn't use any existing pattern, but a definition can also make a reference to a `src:fragment` element using `src:fragref`, such as in:

```
<div>
<h1>The <tt>character</tt> element</h1>
<p>The <tt>character</tt> element is the container holding all the information
<src:fragment id="character" xmlns="">
```

```
 <rng:element name="character">
  <src:fragref linkend="id"/>
  <src:fragref linkend="name"/>
  <src:fragref linkend="born"/>
  <rng:optional>
    <src:fragref linkend="qualification"/>
  </rng:optional>
 </rng:element>
 </src:fragment>
 </div>
```

Or, using the compact syntax:

```
 <div>
 <h1>The <tt>character</tt> element</h1>
 <p>The <tt>character</tt> element is the container holding all the information
 <src:fragment id="character" xmlns="">
  element character {
    <src:fragref linkend="id"/>,
    <src:fragref linkend="name"/>,
    <src:fragref linkend="born"/>,
    <src:fragref linkend="qualification"/> ?

}

 </src:fragment>
 </div>
```

From this literate programming document, two different outputs may be generated through XSLT transformations. The first one is the schema itself. Assuming we've defined all the sub-elements and attribute of our character element, the generated schema will be:

```
 <?xml version="1.0" encoding="utf-8"?>
 <rng:grammar xmlns:rng="http://relaxng.org/ns/structure/1.0"
   xmlns:src="http://nwalsh.com/xmlns/litprog/fragment"
   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
   <rng:start>
     <rng:element name="character">
       <rng:attribute name="id">
         <rng:data type="id" datatypeLibrary="http://relaxng.org/ns/compatibili
       </rng:attribute>
       <rng:element name="name">
         <rng:text/>
       </rng:element>
       <rng:element name="born">
         <rng:data type="date"/>
       </rng:element>
       <rng:optional>
         <element name="qualification">
           <text/>
         </element>
       </rng:optional>
     </rng:element>
   </rng:start>
 </rng:grammar>
```

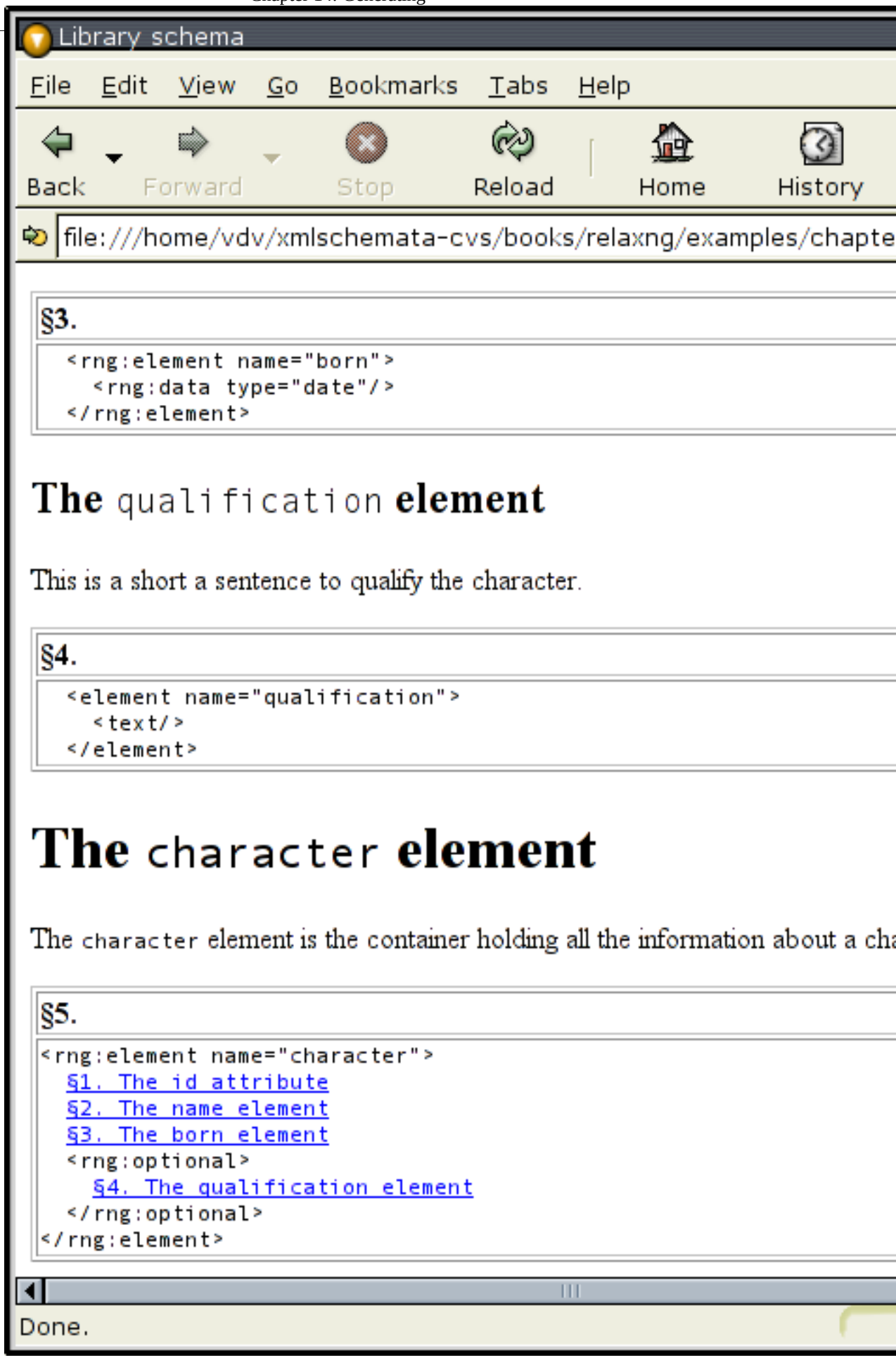
Or, using the compact syntax:

```
 datatypes d = "http://relaxng.org/ns/compatibility/datatypes/1.0"
```

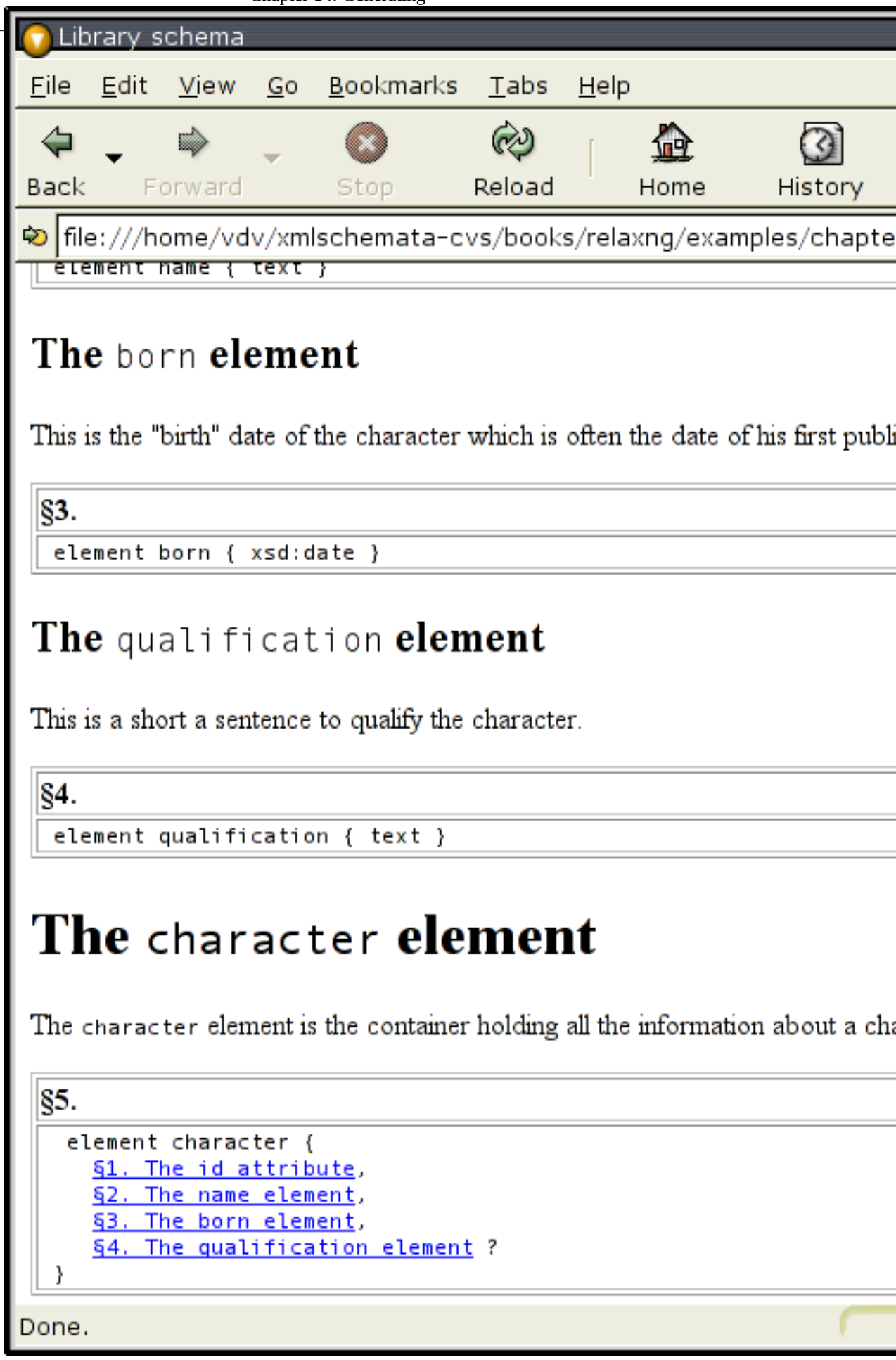
```
start=
      element character {
  attribute id { d:id },
     element name { text },
     element born { xsd:date },
     element qualification { text } ?
 }
```

This is a pretty normal schema and the thing we need to highlight is the way it has been modularized. Up to now, to split a schema into small and manageable pieces, we've been using named patterns, a mechanism provided by Relax NG itself. We could have done so in our last example, but we have now a second possibility which is to use the mechanisms provides by the literate programming framework and define fragments and combine them using `src:fragment` and `src:fragref` instead of the `define` and `ref` elements from Relax NG. By doing so, we have generated a monolithic Russian doll schema through a modular description of its elements and attributes.

The second output from this literate programming document is the XHTML documentation:


Library schema
File Edit View Go Bookmarks Tabs Help
Back Forward Stop Reload Home History
file:///home/vdv/xmlschemata-cvs/books/relaxng/examples/chapte
§3.
<rng:element name="born">
<rng:data type="date"/>
</rng:element>
The qualification element
This is a short a sentence to qualify the character.
§4.
<element name="qualification">
<text/>
</element>
The character element
The character element is the container holding all the information about a cha
§5.
<rng:element name="character">
§1. The id attribute
§2. The name element
§3. The born element
<rng:optional>
§4. The qualification element
</rng:optional>
</rng:element>
Done.

or, for the compact syntax:


Library schema
File Edit View Go Bookmarks Tabs Help
Back Forward Stop Reload Home History
file:///home/vdv/xmlschemata-cvs/books/relaxng/examples/chapte
element name { text }
The born element
This is the "birth" date of the character which is often the date of his first publi
§3.
element born { xsd:date }
The qualification element
This is a short a sentence to qualify the character.
§4.
element qualification { text }
The character element
The character element is the container holding all the information about a cha
§5.
element character {
§1. The id attribute,
§2. The name element,
§3. The born element,
§4. The qualification element ?
}
Done.

Adding bells and whistles for RDDL

We have already mentioned RDDL in "Chapter 13: Annotating Schemas". RDDL (see http://rddl.org for more details) can be seen as a compromise to standardize a usage of XHTML and XLink that can be read both as plain XHTML by human beings with a standard web browser and by applications which will use the semantic attributes of XLink to discover resources such as schemas and stylesheets.

RDDL documents can be generated from annotated Relax NG schema. When documenting XML vocabularies, the opposite works very well too and it is very tempting to use the literate programming framework to produce RDDL documents. There is no basic difference compared to what we've seen with XHTML and we could use the DocBook litprog stylesheets right away but we can also import them into stylesheets which will facilitate the authoring of RDDL documents.

RDDL is basically a compromise to share the same document between human readers through XHTML and applications through RDDL and XLink and the main burden when writing RDDL documents is that the information made available for the application needs to be repeated for the readers. For instance, to publish the snippet of schema describing the name element as a RDDL normative reference, we could write (note the "exclude-result-prefixes" and "mundane-result-prefixes" attributes which become needed to control various namespaces introduced for RDDL):

```
<rddl:resource id="name-elt" xlink:type="simple"
               xlink:arcrole="http://www.rddl.org/purposes#normative-reference
               xlink:role="http://www.w3.org/1999/xhtml"
               xlink:title="The name element"
               xlink:href="#name-elt">
<div class="resource">
<h2>The <tt>name</tt> element</h2>
<src:fragment id="name" xmlns=""
              exclude-result-prefixes="cr xlink rddl rng"
              mundane-result-prefixes="rng">
 <rng:element name="name">
   <rng:text/>
 </rng:element>
</src:fragment>
</div>
</rddl:resource>
```

That's not really tough, but there are some repetitions here: the content of the "h2" element is copied into "xlink:title" and xlink:href is constructed after the id attribute because the resource is local. For external resources, different but similar redundancies may be found too. When the RDDL document is generated by a XSLT transformation like it's the case with Literate Programming, it's tempting to define shortcuts that avoid these redundancies and write for instance:

```
<cr:resource id="name-elt"
             role="http://www.w3.org/1999/xhtml"
             arcrole="http://www.rddl.org/purposes#normative-reference">
<h2>The <tt>name</tt> element</h2>
<p>This is the name of the character.</p>
<src:fragment id="name" xmlns=""
              exclude-result-prefixes="cr xlink rddl rng"
              mundane-result-prefixes="rng">
 <rng:element name="name">
   <rng:text/>
 </rng:element>
</src:fragment>
</cr:resource>
```

Other features can easily be added, such as numbering the divisions, generating table of contents and indexes of resources and pretty printing code snippets and our document becomes:

1. A simple vocabulary for lib

This schema is an illustration of how to describe programming.

This schema defines the element `character`, which simple elements (`name`, `born` and `qualification`).

2. Table of content

3. The simple elements and a

3.1. The `id` attribute

This is the id of the `character` element.

§1.

```
<rng:attribute name="id">
  <rng:data type="id"
            datatypeLibrary="http://rel
</rng:attribute>
```

UML

UML (Unified Modeling Language) is an OMG standards which can be seen as the successor of the many Object Oriented methods invented in the 80s and 90s. The idea of using UML to model XML document is not new and good stuff has already been published on the subject (see for instance the book "Modeling XML applications with UML" by David Carlson published by Addison Wesley or his articles on XML.com).

There are two different levels at which UML and XML can be mapped.

- UML can be used to model the structure of XML documents directly. In this case, XML schemas can be generated for the purpose of validating the documents but they are provided as a convenience for the application more than a main delivery. Their style and modularity is not important as such and the algorithm to produce these schemas is focused on expressing validation rules as close as possible to the UML diagram.

- UML can be used to model a XML schema. In this case, the UML diagram is a higher level view of the schema and the schema by itself is the main delivery. The UML diagram needs to be able to control exactly how each schema structure is described and specific stereotypes and parameters are often added to provide the right level of control.

One of the points which appear quite clearly in all the work related to this topic is that it is quite easy to map UML objects into XML or, which is quite equivalent, to use UML to describe classes of instance documents. The most difficult issue when doing so is that UML is a graph while XML is a tree and some links need to be either removed or serialized using techniques such as XLink which are not built-in within XML 1.0. Except for this issue, the relationship between UML and XML is quite natural in both directions: UML provides a simple language to model XML documents and XML provides a natural serialization syntax for UML objects.

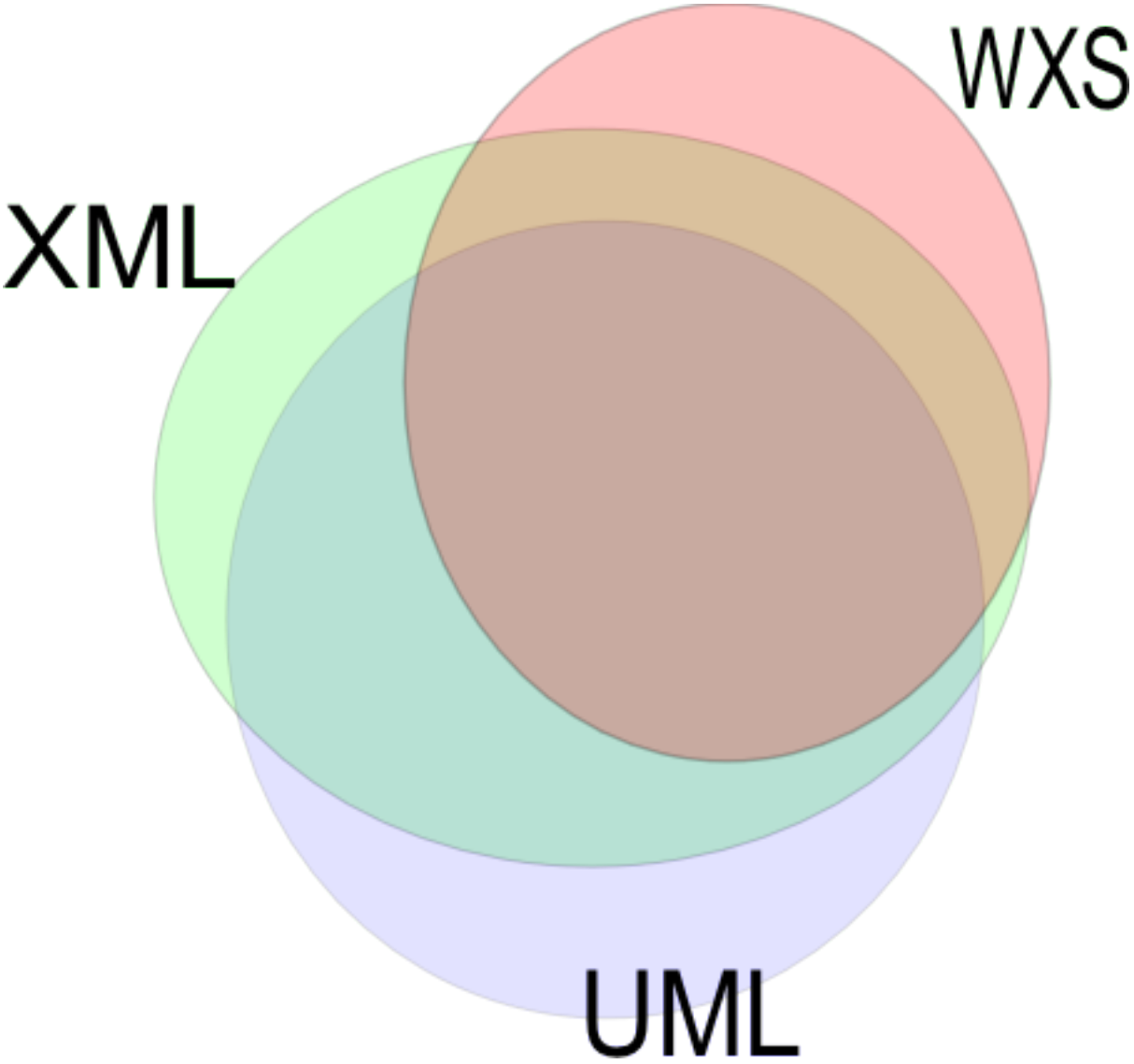
The other point which is also very clear is that it's not that simple to generate DTDs and W3C XML Schema schemas from UML.

The issue when generating DTDs or W3C XML Schemas from UML is to cope with the multiple restrictions of these languages, starting with those related to unordered content models: unordered content models are natural for UML for which the attributes of a class are unordered and the limitations of DTDs and W3C XML Schemas are a problem when UML attributes are serialized as XML elements.

The issue when modeling W3C XML Schema schemas is that the model does not only need to describe the XML instances but also the schema itself and all the complexity of W3C XML Schema kind of enters into UML world.

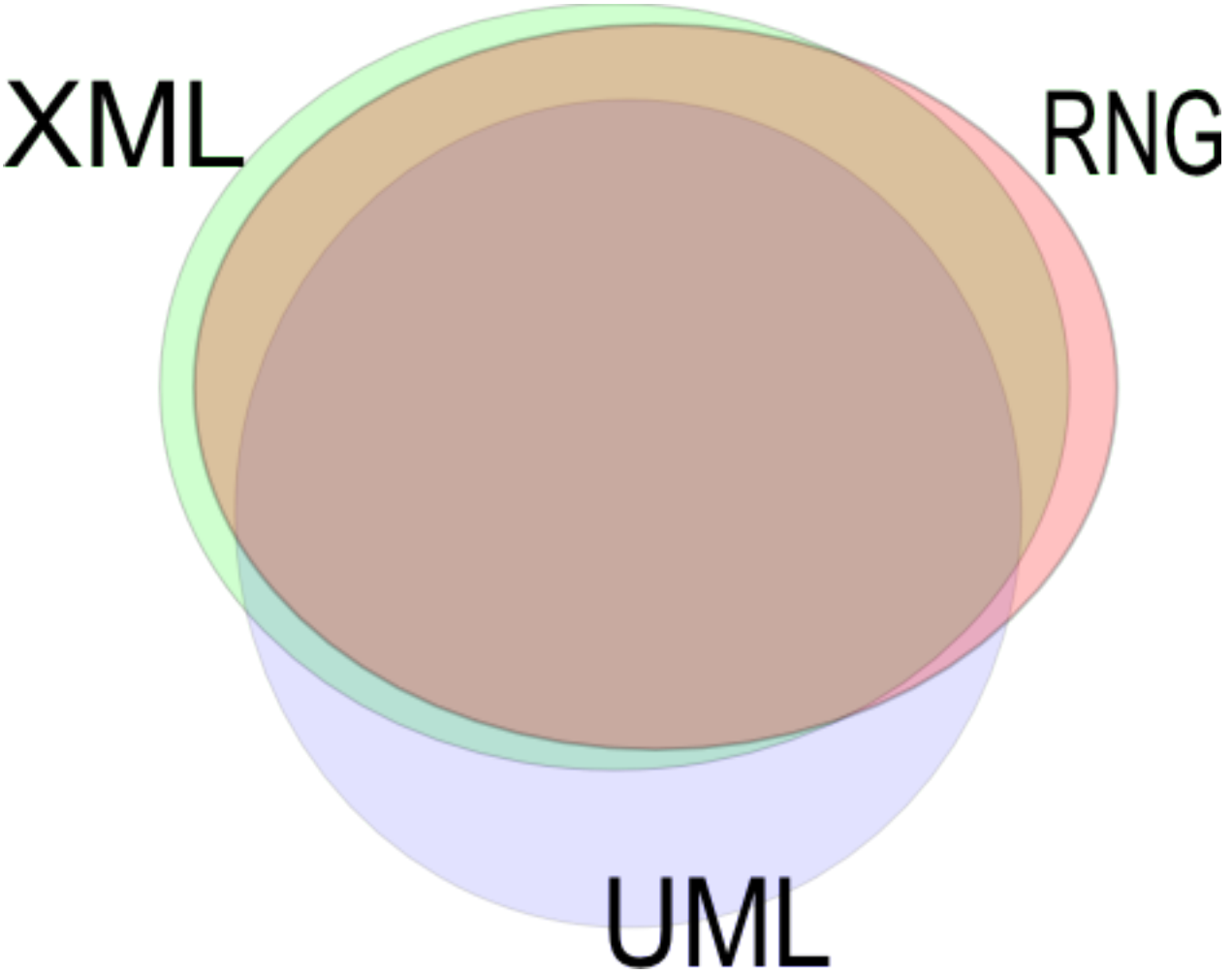
The situation with UML, XML and W3C XML Schema is that if there is a good overlap between UML and XML, the overlap is not so good between XML and W3C XML Schema and W3C XML Schema adds to XML its own concepts. To draw a rough picture, the situation is not unlike:

Figure 15.4. overlap



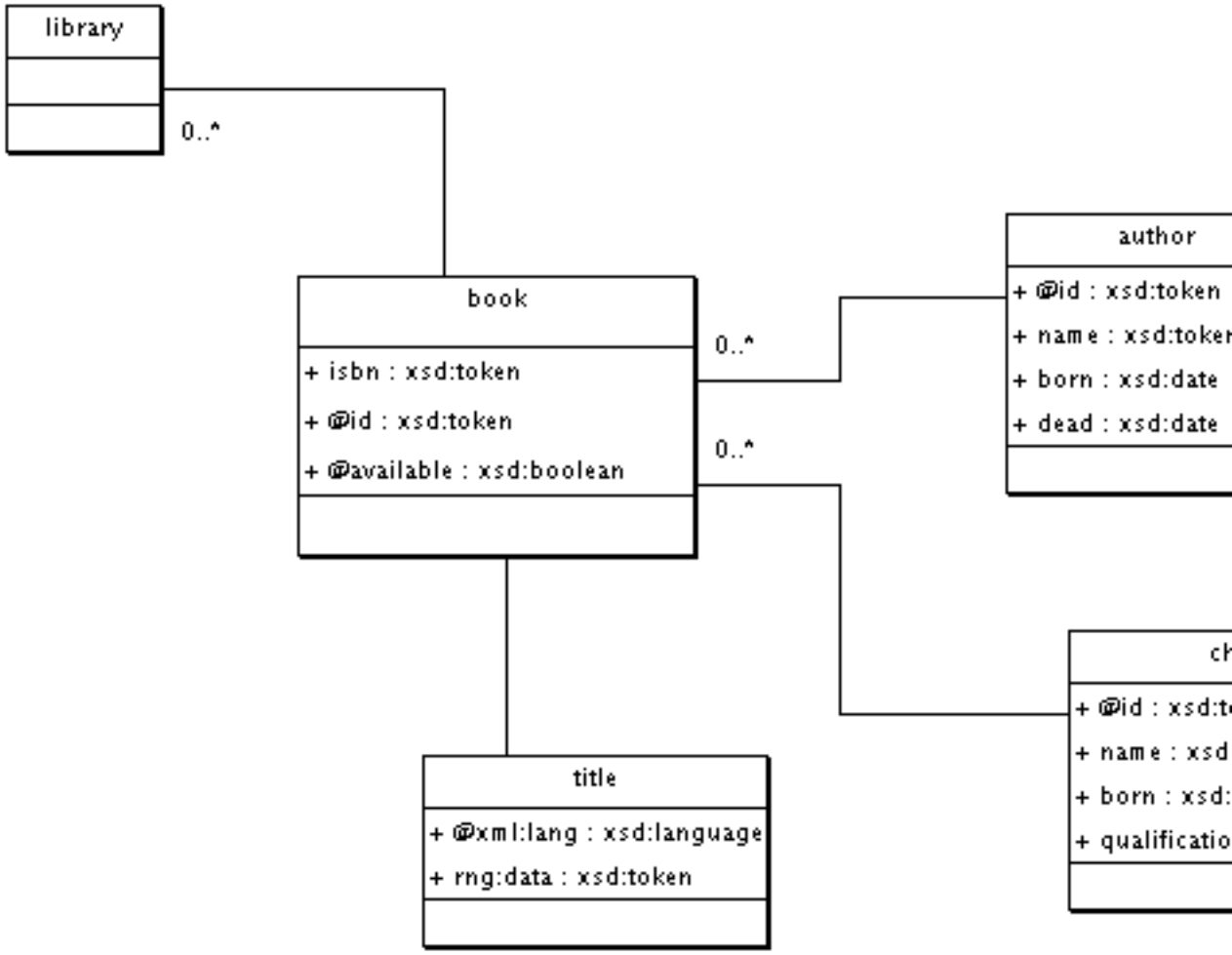
With Relax NG, on the contrary, the overlap between XML and the schema language is near to perfection: Relax NG can describe almost any XML structure and having no notion of Post Schema Validation Infoset (PSVI), Relax NG doesn't add anything to XML. The overlap between UML, XML and Relax NG is thus almost as big as the overlap between UML and XML:

Figure 15.5. overlap2



Designed with a UML editor such as ArgoUML, our library could be described as:

Figure 15.6. argouml



Note that I have been using conventions which look natural but are far from being official. For instance, I have prefixed the attribute names with @, an idea borrowed to Will Provost on XML.com. Also, to model the title element with its text node and attribute, I have used the name "rng:data" to name its text content as a UML attribute.

ArgoUML saves its documents using the XML Metadada Interchange (XMI) format defined by the Object Management Group (OMG). You'll find more information about XMI at http://www.omg.org/technology/xml/.

XMI is pretty verbose and the XMI document generated by ArgoUML for this diagram is more than 800 lines long. I won't include it here, but there is no major difficulty to generate from this document a schema with unordered content models, such as:

```
 <?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="library">
      <interleave>
        <zeroOrMore>
          <element name="book">
            <interleave>
              <element name="isbn">
                <data type="token"/>
```

```
                </element>
                <attribute name="id">
                  <data type="token"/>
                </attribute>
                <attribute name="available">
                  <data type="boolean"/>
                </attribute>
                <zeroOrMore>
                  <element name="author">
                    <interleave>
                      <attribute name="id">Foundation.Data_Types.Multiplicity
                        <data type="token"/>
                      </attribute>
                      <element name="name">
                        <data type="token"/>
                      </element>
                      <element name="born">
                        <data type="date"/>
                      </element>
                      <element name="died">
                        <data type="date"/>
                      </element>
                    </interleave>Foundation.Data_Types.Multiplicity
                  </element>
                </zeroOrMore>
                <zeroOrMore>Foundation.Data_Types.Multiplicity
                  <element name="character">
                    <interleave>
                      <attribute name="id">
                        <data type="token"/>
                      </attribute>
                      <element name="name">
                        <data type="token"/>
                      </element>
                      <element name="born">
                        <data type="date"/>
                      </element>
                      <element name="qualification">
                        <data type="token"/>
                      </element>
                    </interleave>
                  </element>
                </zeroOrMore>
                <element name="title">
                  <attribute name="xml:lang">
                    <data type="language"/>
                  </attribute>
                  <data type="token"/>
                </element>
              </interleave>
            </element>
          </zeroOrMore>
        </interleave>
      </element>
    </start>
  </grammar>
```

Or, after conversion by Trang:

```
start =
  element library {
    element book {
      element isbn { xsd:token }
      & attribute id { xsd:token }
      & attribute available { xsd:boolean }
      & element author {
          attribute id { xsd:token }
          & element name { xsd:token }
          & element born { xsd:date }
          & element died { xsd:date }
        }*
      & element character {
          attribute id { xsd:token }
          & element name { xsd:token }
          & element born { xsd:date }
          & element qualification { xsd:token }
        }*
      & element title {
          attribute xml:lang { xsd:language },
          xsd:token
        }
    }*
  }
```

In fact, the only trouble I have had with Relax NG itself comes out of one of the few restrictions of Relax NG which we've mentioned in "Chapter 7: Constraining Text Values": data pattern cannot be interleaved and when generating this schema one must be careful to treat complex type simple content models (i.e. elements such as the `title` element which accepts attributes and text nodes but no children elements) as an exception. This straight translation is, of course, impossible with W3C XML Schema because of the cardinality of the character and author sub elements and containers would need to be added to fit into the limitations of the language.

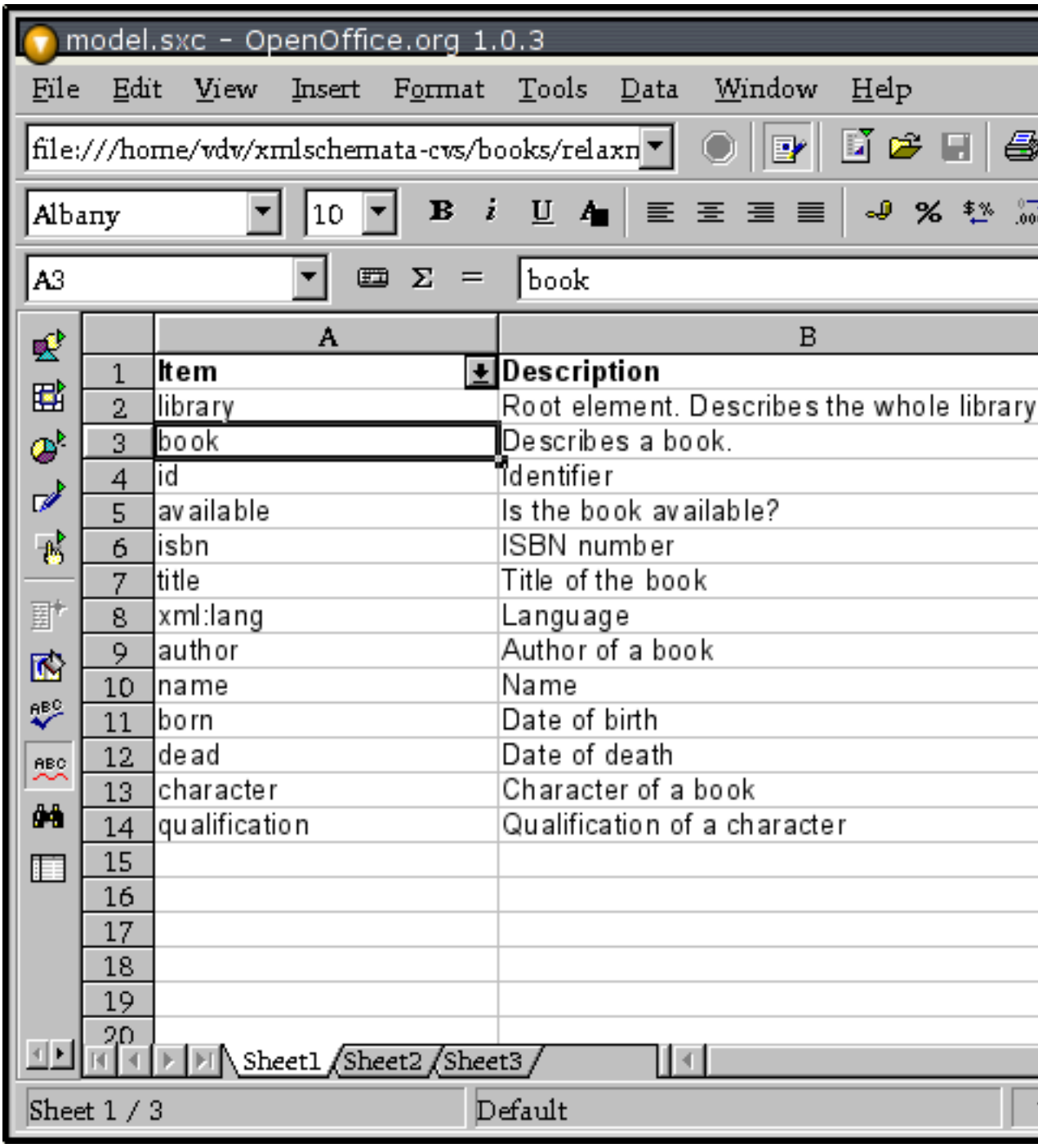
Note that here we have generated a Russian doll design but that depending on the strategy used in the translation, we could have generated other "flavors" as well.

Spreadsheets

The last transformation I'd like to show here is probably much widely used than one would think. Spreadsheets are very convenient to store and manipulate large lists of information items and have been used as a modeling tool for many years. This has been recently acknowledged by the UBL Oasis Technical committee (see http://www.oasis-open.org/committees/ubl/) which is in charge of a set of "core components" to be used by B2B applications and frameworks such as ebXML. Although this project is using a UML methodology, the release note of their 0.70 version states: "The current spreadsheet matrix used by UBL has proved the most versatile and manageable in developing a logical model of the UBL Library."

Now most if not all of the spreadsheet software are supporting XML format, generating Relax NG schemas from such a tool is really easy. There is no standard ways of representing XML documents in a spreadsheet. The benefit of spreadsheets is their flexibility and we can define specific layouts specific to each application. Coming back to our library, we could formalize it as:

Figure 15.7. oo



That's basically nothing more than than a catalog of each information item with just enough information to generate a schema. The benefit of getting it as a spreadsheet is that it's easy to read and, when the catalog gets bigger, features such as filters, sort and search become increasingly useful to navigate over the catalog.

Again, generating Relax NG schemas is really easy and it doesn't take long before you turn this spreadsheet into schemas such as:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
```

```
<start>
  <element name="library">
    <a:documentation>Root element. Describes the whole library.</a:documenta
    <zeroOrMore>
      <element name="book">
        <a:documentation>Describes a book.</a:documentation>
        <attribute name="id">
          <a:documentation>Identifier</a:documentation>
          <data type="token"/>
        </attribute>
        <attribute name="available">
          <a:documentation>Is the book available?</a:documentation>
          <data type="boolean"/>
        </attribute>
        <element name="isbn">
          <a:documentation>ISBN number</a:documentation>
          <data type="token"/>
        </element>
        <element name="title">
          <a:documentation>Title of the book</a:documentation>
          <data type="token"/>
          <attribute name="xml:lang">
            <a:documentation>Language</a:documentation>
            <data type="language"/>
          </attribute>
        </element>
        <zeroOrMore>
          <element name="author">
            <a:documentation>Author of a book</a:documentation>
            <attribute name="id">
              <a:documentation>Identifier</a:documentation>
              <data type="token"/>
            </attribute>
            <element name="name">
              <a:documentation>Name</a:documentation>
              <data type="token"/>
            </element>
            <element name="born">
              <a:documentation>Date of birth</a:documentation>
              <data type="date"/>
            </element>
            <element name="died">
              <a:documentation>Date of death</a:documentation>
              <data type="date"/>
            </element>
          </element>
        </zeroOrMore>
        <zeroOrMore>
          <element name="character">
            <a:documentation>Character of a book</a:documentation>
            <attribute name="id">
              <a:documentation>Identifier</a:documentation>
              <data type="token"/>
            </attribute>
            <element name="name">
              <a:documentation>Name</a:documentation>
              <data type="token"/>
            </element>
```

```
              <element name="born">
                <a:documentation>Date of birth</a:documentation>
                <data type="date"/>
              </element>
              <element name="qualification">
                <a:documentation>Qualification of a character</a:documentation>
                <data type="token"/>
              </element>
            </element>
          </zeroOrMore>
        </element>
      </zeroOrMore>
    </element>
  </start>
</grammar>
```

or, after a translation by Trang:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"

start =
  ## Root element. Describes the whole library.
  element library {
    ## Describes a book.
    element book {
      ## Identifier
      attribute id { xsd:token },
      ## Is the book available?
      attribute available { xsd:boolean },
      ## ISBN number
      element isbn { xsd:token },
      ## Title of the book
      element title {
        xsd:token,
        ## Language
        attribute xml:lang { xsd:language }
      },
      ## Author of a book
      element author {
        ## Identifier
        attribute id { xsd:token },
        ## Name
        element name { xsd:token },
        ## Date of birth
        element born { xsd:date },
        ## Date of death
        element died { xsd:date }
      }*,
      ## Character of a book
      element character {
        ## Identifier
        attribute id { xsd:token },
        ## Name
        element name { xsd:token },
        ## Date of birth
        element born { xsd:date },
        ## Qualification of a character
        element qualification { xsd:token }
```

```
      }*
   }*
}
```

And here again, instead of a Russian doll schema, we could have generated any other style of schema.

Chapter 16. Chapter 15: Simplification And Restrictions

Simplification and restrictions are two topics on which I have been very evasive all over this book. The reason for this is that they are pretty technical and have few direct concrete impact when you're writing a Relax NG schema. Still, this book wouldn't be complete without describing it.

Why should we care at all of the simplification if it's so technical and looks like an implementation algorithm? To be honest, most of the time we do not have to care about this stuff at all. The simplification can be seen as an intermediary step when a Relax NG processor reads a schema. During this step, all the syntactical sugar is removed and the processor can then work with a perfectly normalized schema. On the other hand, the few restrictions existing with Relax NG are formalized relatively to this normalized version of the schema. Because of the flexibility of Relax NG, formalizing them on schemas before simplification would be very complex and very difficult to read. The downside is that when you hit one of these restrictions you often need to understand the main principles of the simplification process to understand what's happening. The good news is that it doesn't happen so often!

Simplification

During its conception, Relax NG has always tried to keep a balance between simplicity of use, simplicity to implement and the simplicity of its data model. What's simple to implement is often simple to use, but there are many features which are very useful to the users, add complexity for the implementers and clutter the data model. This is the case, for instance, of all the features designed to create building blocks (named patterns, includes, embedded grammars): they are very useful for the users but the fact that you've used named patterns or a Russian doll style, has zero impact on the validation itself. This is also the case for shorthands, such as the mixed pattern which is just a more concise way of writing an interleave pattern with an embedded text pattern.

The quest for simplicity has had a huge influence over the design of Relax NG and here is the view of James Clark on the subject:

"Simplicity of specification often goes hand in hand with simplicity of use. But I find that these are often in conflict with simplicity of implementation. An example would be ambiguity restrictions as in XSD: these make implementation simpler (well, at least for people who don't want to learn a new algorithm) but make specification and use more complex. In general, RELAX NG aims for implementation to be practical and safe (i.e. implementations shouldn't use huge amounts of time/memory for particular schemas/instances), but apart from that favors simplicity of specification/use over simplicity of implementation."

To keep the description of the restrictions and validation algorithm simple while offering those useful features to the users, Relax NG has chosen to describe validation through a Relax NG as a two step process:

- First, the schema is read and simplified. The purpose of the simplification is to remove all the additional complexity of the "syntactic sugar" and to reduce the schema to its most simple equivalent form.

- Then, instance documents are validated against the simplified schema. Since all the syntactic sugar has been removed from the simplified schema, it does not need to be taken into account in the description of the validation, leading to much simpler algorithms.

The simplification is described for each Relax NG element in the reference manual and we won't do deep into its details here but just give the main points. If you don't want to go into too much detail, let's just say that the simplification removes all the syntactic sugar, consolidate all the external schemas, uses a subset of all the available Relax NG elements and transforms the resulting structure into a flat

schema where each element is embedded in a named pattern and all the named patterns contain the definition of a single element.

The Relax NG specification is very clear that this simplification is done by the Relax NG processors after on the data model resulting of the reading of the complete schema and that the results of this simplification doesn't have to be serialized as XML. However, I think that showing intermediary results presented as XML help to visualize the simplification process (note that even if these intermediary results are presented indented for readability even if we have seen that text nodes with only whitespaces have been removed in one of the first steps of the simplification).

The XML syntax is closer to the data model used to describe the simplification than the compact syntax and the details of the simplification will be shown below on XML snippets but for each sequence of steps I will also give the compact syntax for the whole schema to show a better overall view of the impact on the structure of the schema (note that some impact of the simplification are just lost on the compact syntax).

The schema which will be used in this chapter is a consolidation of features seen all over this book to cover most of the elements impacted by the simplification. It is composed of three documents:

- library.rnc (or .rng):

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
namespace hr = "http://eric.van-der-vlist.com/ns/person"
namespace local = ""
default namespace ns1 = "http://eric.van-der-vlist.com/ns/library"
namespace sn = "http://www.snee.com/ns/stages"



a:documentation [ "Relax NG schema for our library" ]
sn:stages [
  sn:stage [ name = "library" ]
  sn:stage [ name = "book" ]
  sn:stage [ name = "author" ]
  sn:stage [ name = "character" ]
  sn:stage [ name = "author-or-book" ]
]
start =
  [ sn:stages = "library" ] element library { book-element+ }
  | [ sn:stages = "book author-or-book" ] book-element
  | [ sn:stages = "author author-or-book" ] author-element
  | [ sn:stages = "character" ] character-element
include "foreign.rnc" {
  foreign-elements = element * - (local:* | ns1:* | hr:*) { anything }*
  foreign-attributes = attribute * - (local:* | ns1:* | hr:*) { text }*
}
author-element =
  element hr:author {
    attribute id {
      xsd:NMTOKEN { maxLength = " 16 " }
    },
    name-element,
    born-element,
    died-element?
  }
include "book-content.rnc"
book-content &= foreign-nodes
```

```
book-element = element book { book-content }
born-element = element hr:born { xsd:date }
character-element = external "character-element.rnc"
died-element = element hr:died { xsd:date }
isbn-element = element isbn { foreign-attributes, token }
name-element = element hr:name { xsd:token }
qualification-element = element qualification { text }
title-element = element title { foreign-attributes, text }
available-content = "true" | xsd:token " false " | " "
```

- book-content.rnc (or .rng)

```
book-content =
  attribute id { text },
  attribute available { available-content },
  isbn-element,
  title-element,
  author-element*,
  character-element*
```

- foreign.rnc (or .rng)

```
anything =
  (element * { anything }
   | attribute * { text }
   | text)*
foreign-elements = element * { anything }*
foreign-attributes = attribute * { text }*
foreign-nodes = (foreign-attributes | foreign-elements)*
```

- character-element.rnc (or .rng)

```
start =
  element character {
    attribute id { text },
    parent name-element,
    parent born-element,
    parent qualification-element
  }
```

Whitespace and attribute normalization and inheritance

The first sequence of simplification steps realizes various normalizations without changing the structure of the schema:

- Annotations (i.e. attributes and elements from foreign namespaces) are removed.
- Text nodes with only whitespaces are removed except when found in `value` and `param` elements and whitespaces are normalized in `name`, `type` and `combine` attributes and in `name` elements.
- The characters which are not allowed in the `datatypeLibrary` attributes are escaped and these attributes are transfered through inheritance to each `data` and `value` pattern.

- The `type` attributes of the `value` pattern are defaulted to the `token` datatype from the built in datatype library.

After this sequence of steps, the structure of the schema is still unchanged, but all the cosmetic features which have no impact on the schema have been removed. For instance, the following schema snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:hr="http://eric.van-der-vlist.com/ns/person"
         ns="http://eric.van-der-vlist.com/ns/library"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
         xmlns:sn="http://www.snee.com/ns/stages"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <a:documentation>Relax NG schema for our library</a:documentation>
   <sn:stages>
     <sn:stage name="library"/>
     <sn:stage name="book"/>
     <sn:stage name="author"/>
     <sn:stage name="character"/>
     <sn:stage name="author-or-book"/>
   </sn:stages>
  <start>
    <choice>
      <element name=" library "  sn:stages="library">
        <oneOrMore>
          <ref name="book-element"/>
        </oneOrMore>
      </element>
      <ref name="book-element" sn:stages="book author-or-book"/>
      <ref name="author-element" sn:stages="author author-or-book"/>
      <ref name="character-element" sn:stages="character"/>
    </choice>
  </start>
.../...
  <define name=" author-element ">
    <element name="hr:author" datatypeLibrary="">
      <attribute name="id" datatypeLibrary="http://www.w3.org/2001/XMLSchema-da
        <data type="NMTOKEN">
         <param name="maxLength"> 16 </param>
       </data>
      </attribute>
      <ref name=" name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
```

.../...

```
  <define name="available-content">
    <choice>
      <value>true</value>
      <value type="token"> false </value>
      <value> </value>
```

```
    </choice>
  </define>
</grammar>
```

will be transformed during this sequence of steps into this (note that I am still showing whitespace for readability even though they should have been removed):

```
<?xml version="1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:hr="http://eric.van-der-vlist.com/ns/person"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
         xmlns:sn="http://www.snee.com/ns/stages"
         ns="http://eric.van-der-vlist.com/ns/library">
  <start>
    <choice>
      <element name="library">
        <oneOrMore>
          <ref name="book-element"/>
        </oneOrMore>
      </element>
      <ref name="book-element"/>
      <ref name="author-element"/>
      <ref name="character-element"/>
    </choice>
  </start>
```

.../...

```
  <define name="author-element">
    <element name="hr:author">
      <attribute name="id">
        <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" typ
          <param name="maxLength"> 16 </param>
        </data>
      </attribute>
      <ref name="name-element"/>
      <ref name="born-element"/>
      <optional>
        <ref name="died-element"/>
      </optional>
    </element>
  </define>
```

.../...

```
  <define name="available-content">
    <choice>
      <value type="token" datatypeLibrary="">true</value>
      <value datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" type
      <value type="token" datatypeLibrary=""> </value>
    </choice>
  </define>
</grammar>
```

After this sequence of steps, our schema is:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
namespace hr = "http://eric.van-der-vlist.com/ns/person"
namespace local = ""
default namespace ns1 = "http://eric.van-der-vlist.com/ns/library"
namespace sn = "http://www.snee.com/ns/stages"



start =
  element library { book-element+ }
  | book-element
  | author-element
  | character-element
include "foreign.rnc" {
  foreign-elements = element * - (local:* | ns1:* | hr:*) { anything }*
  foreign-attributes = attribute * - (local:* | ns1:* | hr:*) { text }*
}
author-element =
  element hr:author {
    attribute id {
      xsd:NMTOKEN { maxLength = " 16 " }
    },
    name-element,
    born-element,
    died-element?
  }
include "book-content.rnc"
book-content &= foreign-nodes
book-element = element book { book-content }
born-element = element hr:born { xsd:date }
character-element = external "character-element.rnc"
died-element = element hr:died { xsd:date }
isbn-element = element isbn { foreign-attributes, token }
name-element = element hr:name { xsd:token }
qualification-element = element qualification { text }
title-element = element title { foreign-attributes, text }
available-content = "true" | xsd:token " false " | " "
```

Retrieval of external schemas

This second sequence of steps reads and processes `externalRef` and `include` patterns:

- `externalRef` patterns are replaced by the content of the resource referenced by their `href` attributes and all the simplification steps up to this one must be recursively applied during this replacement to make sure that we merge schemas at the same level of simplification.
- The schemas referenced by `include` patterns are read and all the simplification steps up to this are recursively are applied on these schemas. Their definitions are overridden by those found in the `include` pattern itself when it is the case and the content of their grammar is added in a new `div` pattern to the current schema. This `div` pattern is needed to carry namespace information to the next sequence of steps and will be remove in the following one.

After this sequence of steps, we obtain a standalone schema without any reference to external documents.

The following snippet:

```
<define name="character-element">
  <externalRef href="character-element.rng" ns="http://eric.van-der-vlist.co
</define>
```

will be transformed into:

```
<define name="character-element">
  <grammar ns="http://eric.van-der-vlist.com/ns/library">
    <start>
      <element name="character">
        <attribute name="id"/>
        <parentRef name="name-element"/>
        <parentRef name="born-element"/>
        <parentRef name="qualification-element"/>
      </element>
    </start>
  </grammar>
</define>
```

And the snippet:

```
<include href="foreign.rng">
  <define name="foreign-elements">
    <zeroOrMore>
      <element>
        <anyName>
          <except>
            <nsName ns=""/>
            <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
            <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
          </except>
        </anyName>
        <ref name="anything"/>
      </element>
    </zeroOrMore>
  </define>
  <define name="foreign-attributes">
    <zeroOrMore>
      <attribute>
        <anyName>
          <except>
            <nsName ns=""/>
            <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
            <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
          </except>
        </anyName>
      </attribute>
    </zeroOrMore>
  </define>
</include>
```

is replaced by:

```
<div>
  <define name="foreign-elements">
    <zeroOrMore>
      <element>
        <anyName>
          <except>
            <nsName ns=""/>
            <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
            <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
          </except>
        </anyName>
        <ref name="anything"/>
      </element>
    </zeroOrMore>
  </define>
  <define name="foreign-attributes">
    <zeroOrMore>
      <attribute>
        <anyName>
          <except>
            <nsName ns=""/>
            <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
            <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
          </except>
        </anyName>
      </attribute>
    </zeroOrMore>
  </define>
  <define name="anything">
    <zeroOrMore>
      <choice>
        <element>
          <anyName/>
          <ref name="anything"/>
        </element>
        <attribute>
          <anyName/>
        </attribute>
        <text/>
      </choice>
    </zeroOrMore>
  </define>
  <define name="foreign-nodes">
    <zeroOrMore>
      <choice>
        <ref name="foreign-attributes"/>
        <ref name="foreign-elements"/>
      </choice>
    </zeroOrMore>
  </define>
</div>
```

The schema after this sequence of steps is:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
```

```
namespace hr = "http://eric.van-der-vlist.com/ns/person"
namespace local = ""
default namespace ns1 = "http://eric.van-der-vlist.com/ns/library"
namespace sn = "http://www.snee.com/ns/stages"



start =
  element library { book-element+ }
  | book-element
  | author-element
  | character-element
div {
  foreign-elements = element * - (local:* | ns1:* | hr:*) { anything }*
  foreign-attributes = attribute * - (local:* | ns1:* | hr:*) { text }*
  anything =
    (element * { anything }
     | attribute * { text }
     | text)*
  foreign-nodes = (foreign-attributes | foreign-elements)*
}
author-element =
  element hr:author {
    attribute id {
      xsd:NMTOKEN { maxLength = " 16 " }
    },
    name-element,
    born-element,
    died-element?
  }
div {
  book-content =
    attribute id { text },
    attribute available { available-content },
    isbn-element,
    title-element,
    author-element*,
    character-element*
}
book-content &= foreign-nodes
book-element = element book { book-content }
born-element = element hr:born { xsd:date }
character-element =
  grammar {
    start =
      element character {
        attribute id { text },
        parent name-element,
        parent born-element,
        parent qualification-element
      }
  }
died-element = element hr:died { xsd:date }
isbn-element = element isbn { foreign-attributes, token }
name-element = element hr:name { xsd:token }
qualification-element = element qualification { text }
title-element = element title { foreign-attributes, text }
```

```
available-content = "true" | xsd:token " false " | " "
```

Name classes normalization

This third sequence of steps performs the normalization of name classes:

- The `name` attribute of the `element` and `attribute` patterns are replaced by `name` element, i.e. a name class matching only this single name.
- The `ns` attributes are transfered through inheritance to the elements which need them, i.e. `name`, `nsName` and `value` (`value` patterns need this attribute to support QName datatypes). Note that the `ns` attribute behaves like the default namespace in XML and isn't passed to attributes which, by default, are considered as having no namespace URI.
- The QName (qualified name) used in `name` elements are replaced by their local part and the `ns` attribute of these elements is replaced by the namespace URI defined for their prefix.

After this sequence of steps, name classes are almost normalized (the except and choice name class will be normalized in the next sequence of steps).

During this sequence of steps, the snippet:

```
<element name="hr:author">
  <attribute name="id">
    <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" typ
      <param name="maxLength"> 16 </param>
    </data>
  </attribute>
  <ref name="name-element"/>
  <ref name="born-element"/>
  <optional>
    <ref name="died-element"/>
  </optional>
</element>
```

is transformed into:

```
<element>
  <name ns="http://eric.van-der-vlist.com/ns/person">author</name>
  <attribute>
    <name ns="">id</name>
    <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" typ
      <param name="maxLength"> 16 </param>
    </data>
  </attribute>
  <ref name="name-element"/>
  <ref name="born-element"/>
  <optional>
    <ref name="died-element"/>
  </optional>
</element>
```

Note that none of these modifications are visible on the compact syntax which requires that all the namespace declarations are done in the declaration section of the schema and for which there is no difference between name elements and attributes.

Pattern normalization

In this fourth sequence of steps, pattern are normalized:

- `div` elements are replaced by their children.
- `define`, `oneOrMore`, `zeroOrMore`, `optional`, `list` and `mixed` patterns are transformed to have exactly one child pattern: if they had more than one pattern, these patterns are wrapped into a `group` pattern.
- `element` patterns follow a similar rule and are transformed to have exactly a name class and a single child pattern.
- `except` patterns and name classes are also transformed to have exactly one child pattern, but since they have a different semantic, their children elements are wrapped in a `choice` element.
- If an `attribute` pattern has no child pattern, a `text` pattern is added.
- The `group` and `interleave` patterns and the `choice` pattern and name class are recursively transformed to have exactly two sub elements: if it has only one child, it is replaced by this child and if it has more than two children, the first two child elements are combined into a new element until there are exactly two child elements.
- `mixed` patterns are transformed into `interleave` patterns between their unique child pattern and a `text` pattern.
- `optional` patterns are transformed into `choice` patterns between their unique child pattern and an `empty` pattern.
- `zeroOrMore` patterns are transformed into `choice` patterns between a `oneOrMore` pattern including their unique child pattern and an `empty` pattern.

After this sequence of steps, the number of different type of patterns has been reduced to a set of "primitive" patterns and all the patterns which are left have a fixed number of children elements.

During this sequence of steps, the snippet:

```
<define name="foreign-elements">
  <zeroOrMore>
    <element>
      <anyName>
        <except>
          <nsName ns=""/>
          <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
          <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
        </except>
      </anyName>
      <ref name="anything"/>
    </element>
  </zeroOrMore>
</define>
```

is transformed into:

```
<define name="foreign-elements">
  <choice>
```

```
      <oneOrMore>
        <element>
          <anyName>
            <except>
              <choice>
                <choice>
                  <nsName ns=""/>
                  <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
                </choice>
                <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
              </choice>
            </except>
          </anyName>
          <ref name="anything"/>
        </element>
      </oneOrMore>
      <empty/>
    </choice>
  </define>
```

After this sequence of steps, our schema is:

```
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
namespace hr = "http://eric.van-der-vlist.com/ns/person"
namespace local = ""
default namespace ns1 = "http://eric.van-der-vlist.com/ns/library"
namespace sn = "http://www.snee.com/ns/stages"



start =
  ((element library { book-element+ }
    | book-element)
   | author-element)
  | character-element
foreign-elements =
  element * - ((local:* | ns1:*) | hr:*) { anything }+
  | empty
foreign-attributes =
  attribute * - ((local:* | ns1:*) | hr:*) { text }+
  | empty
anything =
  ((element * { anything }
    | attribute * { text })
   | text)+
  | empty
foreign-nodes = (foreign-attributes | foreign-elements)+ | empty
author-element =
  element hr:author {
    ((attribute id {
        xsd:NMTOKEN { maxLength = " 16 " }
      },
      name-element),
     born-element),
    (died-element | empty)
  }
```

```
book-content =
  ((((attribute id { text },
      attribute available { available-content }),
     isbn-element),
    title-element),
   (author-element+ | empty)),
  (character-element+ | empty)
book-content &= foreign-nodes
book-element = element book { book-content }
born-element = element hr:born { xsd:date }
character-element =
  grammar {
    start =
      element character {
        ((attribute id { text },
          parent name-element),
         parent born-element),
        parent qualification-element
      }
  }
died-element = element hr:died { xsd:date }
isbn-element = element isbn { foreign-attributes, token }
name-element = element hr:name { xsd:token }
qualification-element = element qualification { text }
title-element = element title { foreign-attributes, text }
available-content = ("true" | xsd:token " false ") | " "
```

First set of constraints

A first set of constraints is defined at this point. They are mainly sanity checks conform to the common XML sense but are easier and safer to check at that stage than on the complete schema:

- It's not possible to define name classes -or `except`- which would contain no name at all by including `anyName` in an `except` name class or `nsName` in an `except` name class included in another `nsName`.
- It's not possible to define attributes having the name `xmlns` or a namespace URI equal to the namespace URI "http://www.w3.org/2000/xmlns" (corresponding to the "xmlns" prefix).
- Datatype libraries are used correctly: each type exists in their datatype library and its `param` elements are valid for it).

Grammar merge

During this sequence of steps, `define` and `start` elements are combined when needed in each grammar and the grammar are merged into a single top level grammar:

- In each grammar, multiple `start` elements and multiple `define` element with the same name are combined as defined in their `combine` attribute.
- The names of the named patterns are then changed to be unique to the whole schema and the references to these named patterns changed accordingly.
- A top level `grammar` and its `start` element are created if not already present, all the named patterns are moved to be children if this top level grammar, `parentRef` elements are replaced by `ref` elements and and all the other `grammar`, `start` elements are replaced by their children elements.

During this sequence of steps,

```
<define name="born-element">
  <element>
    <name ns="http://eric.van-der-vlist.com/ns/person">born</name>
    <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" type="
  </element>
</define>
<define name="character-element">
  <grammar>
    <start>
      <element>
        <name ns="http://eric.van-der-vlist.com/ns/library">character</name>
        <group>
          <group>
            <group>
              <attribute>
                <name ns="">id</name>
                <text/>
              </attribute>
              <parentRef name="name-element"/>
            </group>
            <parentRef name="born-element"/>
          </group>
          <parentRef name="qualification-element"/>
        </group>
      </element>
    </start>
  </grammar>
</define>
```

is replaced by:

```
<define name="born-element-id2613943">
  <element>
    <name ns="http://eric.van-der-vlist.com/ns/person">born</name>
    <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" type="
  </element>
</define>
<define name="character-element-id2613924">
  <element>
    <name ns="http://eric.van-der-vlist.com/ns/library">character</name>
    <group>
      <group>
        <group>
          <attribute>
            <name ns="">id</name>
            <text/>
          </attribute>
          <ref name="name-element-id2613832"/>
        </group>
        <ref name="born-element-id2613943"/>
      </group>
      <ref name="qualification-element-id2613840"/>
    </group>
  </element>
```

```
 </define>
```

No specific algorithm to create unique names for named pattern is described in the specification and these names will vary between implementations.

The complete schema after this sequence of steps is:

```
namespace local = ""
namespace ns1 = "http://eric.van-der-vlist.com/ns/person"
default namespace ns2 = "http://eric.van-der-vlist.com/ns/library"



start =
  ((element library { book-element-id2613963+ }
    | book-element-id2613963)
   | author-element-id2614058)
  | character-element-id2613924
foreign-elements-id2614183 =
  element * - ((local:* | ns2:*) | ns1:*) { anything-id2614112 }+
  | empty
foreign-attributes-id2614152 =
  attribute * - ((local:* | ns2:*) | ns1:*) { text }+
  | empty
anything-id2614112 =
  ((element * { anything-id2614112 }
    | attribute * { text })
   | text)+
  | empty
foreign-nodes-id2614043 =
  (foreign-attributes-id2614152 | foreign-elements-id2614183)+ | empty
author-element-id2614058 =
  element ns1:author {
    ((attribute id {
        xsd:NMTOKEN { maxLength = " 16 " }
      },
      name-element-id2613832),
     born-element-id2613943),
    (died-element-id2613856 | empty)
  }
book-content-id2614016 =
  (((((attribute id { text },
       attribute available { available-content-id2613805 }),
      isbn-element-id2613872),
     title-element-id2613819),
    (author-element-id2614058+ | empty)),
   (character-element-id2613924+ | empty))
  & foreign-nodes-id2614043
book-element-id2613963 = element book { book-content-id2614016 }
born-element-id2613943 = element ns1:born { xsd:date }
character-element-id2613924 =
  element character {
    ((attribute id { text },
      name-element-id2613832),
     born-element-id2613943),
    qualification-element-id2613840
```

```
  }
died-element-id2613856 = element ns1:died { xsd:date }
isbn-element-id2613872 =
  element isbn { foreign-attributes-id2614152, token }
name-element-id2613832 = element ns1:name { xsd:token }
qualification-element-id2613840 = element qualification { text }
title-element-id2613819 =
  element title { foreign-attributes-id2614152, text }
available-content-id2613805 = ("true" | xsd:token " false ") | " "
```

Schema flattening

The basic style of the schema (Russian doll or named templates) has been preserved by the previous steps. The goal of this new step is on the contrary to normalize the use of named templates. The target is to make the schema similar in structure to a DTD with each element cleanly embedded in its own named pattern and no other use of named pattern than to embed a single element:

- For each element which is not the unique child of a `define` element, a named pattern is created to embed its definition.
- Each named pattern which does not embed a single element pattern is suppressed and the references to this named pattern replaced by its definition.

During this step,

```
<start>
  <choice>
    <choice>
      <choice>
        <element>
          <name ns="http://eric.van-der-vlist.com/ns/library">library</name>
          <oneOrMore>
            <ref name="book-element-id2613963"/>
          </oneOrMore>
        </element>
        <ref name="book-element-id2613963"/>
      </choice>
      <ref name="author-element-id2614058"/>
    </choice>
    <ref name="character-element-id2613924"/>
  </choice>
</start>
```

is replaced by:

```
<start>
  <choice>
    <choice>
      <choice>
        <ref name="__library-elt-id2615152"/>
        <ref name="book-element-id2613963"/>
      </choice>
      <ref name="author-element-id2614058"/>
    </choice>
    <ref name="character-element-id2613924"/>
  </choice>
```

```
  </start>
```

.../...

```
  <define name="__library-elt-id2615152">
    <element>
      <name ns="http://eric.van-der-vlist.com/ns/library">library</name>
      <oneOrMore>
        <ref name="book-element-id2613963"/>
      </oneOrMore>
    </element>
  </define>
```

The full schema after this step is:

```
namespace local = ""
namespace ns1 = "http://eric.van-der-vlist.com/ns/person"
default namespace ns2 = "http://eric.van-der-vlist.com/ns/library"



start =
  ((__library-elt-id2615152 | book-element-id2613963)
   | author-element-id2614058)
  | character-element-id2613924
author-element-id2614058 =
  element ns1:author {
    ((attribute id {
        xsd:NMTOKEN { maxLength = " 16 " }
      },
      name-element-id2613832),
     born-element-id2613943),
    (died-element-id2613856 | empty)
  }
book-element-id2613963 =
  element book {
    (((((attribute id { text },
         attribute available { ("true" | xsd:token " false ") | " " }),
        isbn-element-id2613872),
       title-element-id2613819),
      (author-element-id2614058+ | empty)),
     (character-element-id2613924+ | empty))
    & (((attribute * - ((local:* | ns2:*) | ns1:*) { text }+
         | empty)
        | (__-elt-id2615098+ | empty))+
       | empty)
  }
born-element-id2613943 = element ns1:born { xsd:date }
character-element-id2613924 =
  element character {
    ((attribute id { text },
      name-element-id2613832),
     born-element-id2613943),
    qualification-element-id2613840
```

```
  }
died-element-id2613856 = element ns1:died { xsd:date }
isbn-element-id2613872 =
  element isbn {
    (attribute * - ((local:* | ns2:*) | ns1:*) { text }+
     | empty),
    token
  }
name-element-id2613832 = element ns1:name { xsd:token }
qualification-element-id2613840 = element qualification { text }
title-element-id2613819 =
  element title {
    (attribute * - ((local:* | ns2:*) | ns1:*) { text }+
     | empty),
    text
  }
__-elt-id2615020 =
  element * {
    ((__-elt-id2615020
      | attribute * { text })
     | text)+
    | empty
  }
__library-elt-id2615152 = element library { book-element-id2613963+ }
__-elt-id2615098 =
  element * - ((local:* | ns2:*) | ns1:*) {
    ((__-elt-id2615020
      | attribute * { text })
     | text)+
    | empty
  }
```

Final cleanup

Now, we're almost done and just need to do a bit of final cleanup:

- Recursively escalate `notAllowed` patterns when they are at a location where their effect is that their parent pattern itself is `notAllowed` and remove choices which are `notAllowed`. Note that this simplification doesn't pass through elements boundaries and that "element foo { notAllowed }" is not transformed into `notAllowed`.
- Remove `empty` elements which have no effect.
- Move `empty` elements to be the first child in `choice` elements.

After this cleanup, our schema is:

```
namespace local = ""
namespace ns1 = "http://eric.van-der-vlist.com/ns/person"
default namespace ns2 = "http://eric.van-der-vlist.com/ns/library"



start =
  ((__library-elt-id2615152 | book-element-id2613963)
   | author-element-id2614058)
  | character-element-id2613924
```

```
author-element-id2614058 =
  element ns1:author {
    ((attribute id {
        xsd:NMTOKEN { maxLength = " 16 " }
      },
      name-element-id2613832),
     born-element-id2613943),
    (empty | died-element-id2613856)
  }
book-element-id2613963 =
  element book {
    (((((attribute id { text },
         attribute available { ("true" | xsd:token " false ") | " " }),
        isbn-element-id2613872),
       title-element-id2613819),
      (empty | author-element-id2614058+)),
     (empty | character-element-id2613924+))
    & (empty
       | ((empty
           | attribute * - ((local:* | ns2:*) | ns1:*) { text }+)
          | (empty | __-elt-id2615098+))+)
  }
born-element-id2613943 = element ns1:born { xsd:date }
character-element-id2613924 =
  element character {
    ((attribute id { text },
      name-element-id2613832),
     born-element-id2613943),
    qualification-element-id2613840
  }
died-element-id2613856 = element ns1:died { xsd:date }
isbn-element-id2613872 =
  element isbn {
    (empty
     | attribute * - ((local:* | ns2:*) | ns1:*) { text }+),
    token
  }
name-element-id2613832 = element ns1:name { xsd:token }
qualification-element-id2613840 = element qualification { text }
title-element-id2613819 =
  element title {
    (empty
     | attribute * - ((local:* | ns2:*) | ns1:*) { text }+),
    text
  }
__-elt-id2615020 =
  element * {
    empty
    | ((__-elt-id2615020
        | attribute * { text })
       | text)+
  }
__library-elt-id2615152 = element library { book-element-id2613963+ }
__-elt-id2615098 =
  element * - ((local:* | ns2:*) | ns1:*) {
    empty
    | ((__-elt-id2615020
        | attribute * { text })
```

```
      | text)+
  }
```

Restrictions

With the exception of the constraints expressed by the Relax NG schema for Relax NG and those which are part of the simplification itself (see above), all the restrictions of Relax NG are expressed on the simplified schema. Most of them are obvious and easy to understand.

Constraints on the attributes

These constraints match the definition of the attributes given by the XML 1.0 recommendation, namely:

- Attributes can't contain other attributes: `attribute` patterns cannot have another `attribute` pattern in their descendants.
- Attributes can't contain elements: `attribute` patterns cannot have a `ref` pattern in their descendants.
- Attributes can't be duplicated: an attribute may not be found in a `oneOrMore` pattern with a combination by `group` or `interleave`. Furthermore, if "attributes" patterns are combined in a group or interleave pattern, their name classes must not overlap, i.e. they cannot be any name which belongs to both name classes.
- Attributes which have an infinite name class (`anyName` or `nsName`) must be enclosed in a `oneOrMore` pattern. In other words, we can't specify that we want to allow only one or a certain number of occurrences of these attributes. Furthermore, they can only have `text` as their model (in other words, `data` patterns are forbidden here).

Let's give some examples of schemas which may look valid through a quick glance but are hit by these restrictions (please note that all these schemas are invalid).

Example: content model of attributes

This schema is defining that any content model can be accepted in the `bar` attribute:

```
anything =
  (element * { anything }
   | attribute * { text }
   | text)*
start =
  element foo {
    attribute bar { anything },
    text
  }
```

Unfortunately, it's translated into:

```
start = __foo-elt-id2602800
__-elt-id2602788 =
  element * {
    empty
    | ((__-elt-id2602788
```

```
        | attribute * { text })
      | text)+
  }
__foo-elt-id2602800 =
  element foo {
    attribute bar {
      empty
      | ((__-elt-id2602788
          | attribute * { text })
        | text)+
    },
    text
  }
```

And this is allowing a reference to a named pattern (which means an element in the simplified syntax) and an attribute. Both are forbidden.

To fix this, we must make sure that the `anything` defined for the content of the attribute is compatible with the content of attributes as defined by the XML specification, for instance:

```
anything =
  (text)
start =
  element foo {
    attribute bar { anything },
    text
  }
```

which will be simplified into:

```
start = __foo-elt-id2602296
__foo-elt-id2602296 =
  element foo {
    attribute bar { text },
    text
  }
```

This schema is expressing what we wanted to express and it is valid.

Example: duplication of attributes

Let's say we want to extend the definition of our `title` element to have the same attributes and content model than the XHTML 2.0 `span` element. If we look into the Relax NG module implementing the `span` element, we can see that its definition is:

```
span = element span { span.attlist, Inline.model }
```

and ma want to just include it in the definition of the `title` element which already includes an xml:lang" attribute:

```
namespace x = "http://www.w3.org/2002/06/xhtml2"
```

```
start = book
include "xhtml-attribs-2.rnc" inherit = x
include "xhtml-inltext-2.rnc" inherit = x
include "xhtml-datatypes-2.rnc" inherit = x
book =
  element book {
    attribute id { text },
    attribute available { text },
    element isbn { text },
    element title {
      attribute xml:lang { xsd:language },
      span.attlist,
      Inline.model
    }
  }
```

Unfortunately, this is invalid because the `xml:lang` attribute is already included somewhere into the " span.attlist" pattern and gets pulled during the simplification which defines the `title` element as:

```
__title-elt-id2641936 =
 element title {
   (attribute xml:lang { xsd:language },
    (((((((((empty
             | attribute id { xsd:ID }),
            (empty
             | attribute class { xsd:NMTOKENS })),
           (empty
            | attribute title { text })),
          (empty
           | attribute xml:lang { xsd:language })),
         (empty
          | attribute dir {
              (("ltr" | "rtl") | "lro")
              | "rlo"
            })),
        ((empty
          | attribute edit {
              (("inserted" | "deleted") | "changed")
              | "moved"
            }),
         (empty default namespace lib = "http://eric.van-der-vlist.com/ns/libra
namespace local = ""

start = book



book =
  element book {
    attribute id { text },
    attribute available { text },
    foreign-attributes,
    element isbn { text },
    element title {
      attribute xml:lang { xsd:language },
      text
```

```
  }
}



foreign-attributes = attribute * - (local:* | lib:* ) { text }*
        | attribute datetime { xsd:dateTime }))),
     (((((((empty
             | attribute href { xsd:anyURI }),
            (empty
             | attribute cite { xsd:anyURI })),
           (empty
            | attribute target { xsd:NMTOKEN })),
          (empty
           | attribute rel { xsd:NMTOKENS })),
         (empty
          | attribute rev { xsd:NMTOKENS })),
        (empty
         | attribute accesskey {
             xsd:string { length = "1" }
           })),
       (empty
        | attribute navindex {
            xsd:nonNegativeInteger {
              pattern = "0-9+"
              minInclusive = "0"
              maxInclusive = "32767"
            }
          })),
      (empty
       | attribute base { xsd:anyURI }))),
    ((empty
      | attribute src { xsd:anyURI }),
     (empty
      | attribute type { text }))),
   ((((empty
       | attribute usemap { xsd:anyURI }),
      (empty
       | attribute ismap { "ismap" })),
     (empty
      | attribute shape {
          (("rect" | "circle") | "poly")
          | "default"
        })),
    (empty
     | attribute coords { text }))))),
 (empty
  | (empty
     | (text
        | ((((((((((((abbr-id2635861 | cite-id2635889)
                      | code-id2635918)
                     | dfn-id2635947)
                    | em-id2635975)
                   | kbd-id2636004)
                  | l-id2636032)
                 | quote-id2636061)
                | samp-id2636090)
```

```
                | span-id2636118)
               | strong-id2636147)
              | sub-id2636176)
             | sup-id2636204)
            | var-id2636233)))+)
 }
```

To fix this, we just need to remove the `xml:lang` from our definition:

```
namespace x = "http://www.w3.org/2002/06/xhtml2"

start = book
include "xhtml-attribs-2.rnc" inherit = x
include "xhtml-inltext-2.rnc" inherit = x
include "xhtml-datatypes-2.rnc" inherit = x
book =
  element book {
    attribute id { text },
    attribute available { text },
    element isbn { text },
    element title {
      span.attlist,
      Inline.model
    }
  }
```

Example: name class overlap

Let's say we have the following schema:

```
default namespace lib = "http://eric.van-der-vlist.com/ns/library"
namespace local = ""

start = book




book =
  element book {
    attribute id { text },
    attribute available { text },
    foreign-attributes,
    element isbn { text },
    element title {
      attribute xml:lang { xsd:language },
      text
    }
  }



  foreign-attributes = attribute * - (local:* | lib:* ) { text }*
```

(book.rnc)

Although we have accepted foreign attributes, we may want to be more precise on the definition of some Dublin Core elements and extend our schema as:

```
namespace dc="http://purl.org/dc/elements/1.1/"



include "book.rnc"

book.content &= attribute dc:rights { text } ?
```

Unfortunately, this is invalid because it gets simplified as:

```
book-id2604347 =
  element book {
    ((((attribute id { text },
        attribute available { text }),
       (empty
        | attribute * - (lib:* | local:*) { text }+)),
      __isbn-elt-id2604556),
     __title-elt-id2604551)
    & attribute ns1:rights { text }
  }
```

Where the attribute `dc:Rights` is included in the name class "* - (lib:* | local:*)". To fix this, we need to redefine the named pattern `foreign-attributes` to remove the name "dc:Riggs" or even all the namespace for Dublin Core elements:

```
default namespace lib = "http://eric.van-der-vlist.com/ns/library"
namespace dc="http://purl.org/dc/elements/1.1/"
namespace local = ""

include "book.rnc" {
 foreign-attributes = attribute * - (local:* | lib:* | dc:* ) { text }*
}

book.content &= attribute dc:rights { text } ?
```

Constraints on lists

Lists work on text nodes by splitting them into tokens which are handled as text nodes. It's therefore not possible to find elements, attributes in a list. Texts nodes and embedded lists would be confusing and are forbidden too:

- List patterns cannot have as their descendants any `list`, `ref` (remember that after simplification, the access to elements is done as references to named patterns), `attribute`, `text`. The `interleave` pattern is also forbidden as descendant of `list` patterns because it would complicate the implementations and has been considered not worth of it.

Example: list and interleave

Let's say we'd like to define a price element as allowing either a numeric followed by a token, such as:

```
<price>1 Euro</price>
```

or a token followed by a numeric:

```
<price>1 Euro</price>
```

We might be tempted to write:

```
element price {
  list { xsd:decimal & xsd:token }
}
```

But this would be invalid because `interleave` is forbidden in a `list`. To workaround this limitation, we need to give all the possible combinations, which is easy on this example but can rapidly grow out of control:

```
element price {
  list { (xsd:decimal, xsd:token) | (xsd:token, xsd:decimal) }
}
```

Constraints on `except` patterns

Except patterns (i.e. `except` elements used in a `data` pattern) are about single data.

- An `except` element with a `data` parent can only contain `data`, `value` and `choice` elements.

Constraints on `start` patterns.

After simplification, a start pattern describes the list of possible root elements. You can thus find only combinations of choices between `ref` elements.

Constraints on content models

Relax NG defines three different content models for an element:

- Empty when the element has only attributes.
- Simple when the element has only attributes and has been described using `data`, `value` or `list` patterns.
- Complex in the other cases.

Note that this is identical to the definition given by W3C XML Schema and similar but somewhat different from the definition of these terms in "plain" XML: an element expressed as "<foo>bar</foo>" is considered by Relax NG as complex content if its content has been described using a `text` pattern and as a simple content if its content has been described using other patterns. This means that it's not enough for an element that it contains only a text node to have a simple content but that it is

also necessary that this element has been described with a `data` orientation. When it's not the case i.e. when the `text` pattern has been used, the element is considered as "document" oriented and a special case of mixed content where no elements are included.

The restriction on the content model is expressed saying that empty content can be grouped with any other content models but that simple and complex content models can't be grouped together (through `group` or `interleave` patterns): they can only appear under the definition of the same element as alternatives. In other words, for each alternative, you need to choose if you are `data` or `text` oriented but can't mix both mindsets.

We have already mentioned the practical consequence of this restriction on mixed content model in "Chapter 7: Constraining Text Values": it is not possible to use `data` patterns to specify constraints on the text nodes occurring in mixed elements.

Limitations on `interleave`.

The last two limitations are on `interleave` and their goal is to facilitate the implementation of this feature which is lacking so much in other schema languages... These two limitations are defined to reduce the number of combinations that Relax NG processors need to explore to support `interleave`:

- Elements combined through interleave must have name classes without overlap: we have already seen a similar restriction with attributes which are always combined through interleave.
- There must be at most one `text` pattern in each set of patterns combined by `interleave`.

These limitations don't impact the expressive power of Relax NG (i.e. the set of content models which can be written with Relax NG): even if we may hit them from time to time, schemas can always be rewritten to work around them; but they are a nuisance when combining existing patterns with mixed content models.

They are needed for the different algorithms currently used to implement Relax NG and James Clark thinks that they could be removed in future versions of Relax NG: "Hopefully better algorithms will be developed that will allow this restriction to be removed in future versions."

Example: at most one text pattern in interleave

We may have the following schema to describe our books:

```
start = book
book = element book { book.content }
book.content =
  attribute id { text },
  attribute available { text },
  element isbn { text },
  title
title = element title { title.attributes, title.content }
title.attributes = attribute xml:lang { xsd:language }
title.content = text
```

(book.rnc)

To add the XHTML `Inline.model` to `title.content` we could be tempted to write:

```
include "book.rnc"
```

```
include "xhtml-attribs-2.rnc"
include "xhtml-inltext-2.rnc"
include "xhtml-datatypes-2.rnc"

title.content &= Inline.model
```

Unfortunately, `Inline.model` already contains a `text` pattern and this gets simplified as:

```
title-id2635741 =
 element title {
   attribute lang { xsd:language },
   (text
    & (empty
       | (empty
          | (text
             | ((((((((((((abbr-id2636549 | cite-id2636578)
                           | code-id2636607)
                          | dfn-id2636636)
                         | em-id2636664)
                        | kbd-id2636693)
                       | l-id2636721)
                      | quote-id2636750)
                     | samp-id2636778)
                    | span-id2636807)
                   | strong-id2636836)
                  | sub-id2636865)
                 | sup-id2636893)
                | var-id2636922)))+))
 }
```

Where we find `text` patterns within `interleave`.

To fix this, be need to replace our combination by a redefinition of `title.content`:

```
include "book.rnc" {
  title.content = Inline.model
}
include "xhtml-attribs-2.rnc"
include "xhtml-inltext-2.rnc"
include "xhtml-datatypes-2.rnc"
include "book.rnc" {
  title.content = Inline.model
}
include "xhtml-attribs-2.rnc"
include "xhtml-inltext-2.rnc"
include "xhtml-datatypes-2.rnc"
```

We see that we have not lost in expressive power (we are able to describe what we wanted to describe) but in modularity: changes done to `title.content` in "book.rnc" would now have to be manually added to our derived schema.

Chapter 17. Chapter 16: Determinism and Datatype Assignment

One of the strengths of Relax NG is its flexibility in supporting scaring concepts called "ambiguous content models" (SGML world) "non-deterministic content models" (XML DTDs) or "Unique Particle Attribution rule" and "Consistent Declaration rule" (W3C XML Schema).

Before you read on into this chapter, let's make it clear that as far as validation only is concerned, it's perfectly fine with Relax NG to write ambiguous schemas.

That being said, when type assignment or data binding is involved, ambiguity may become a problem and we will see in this chapter how Relax NG can be used to be "type assignment friendly".

What are we talking about?

The thing we need to do first is to try to clarified these notions which are blurred in many papers and discussions and are not as obscure as people often think.

Ambiguity versus determinism

The first distinction to make is to differentiate what's called ambiguity (or rather unambiguity) and what's called determinism. These two terms have been given precise definitions by regular expressions and hedge grammars theoreticians and part of the confusion around these notions comes from the fact that they are often misused.

A schema is said to be ambiguous when a document may be valid through different pattern alternatives. A trivial example is:

```
element foo{empty} | element foo{empty}
```

When an empty element named `foo` is found in an instance document, there is no way to say if it is valid per the left or right definition of "element foo{empty}" in the schema.

There are, of course, more complex cases of ambiguity and we'll see some of them in the next sections, but this is the general idea behind ambiguity.

Ambiguity (or unambiguity) is independent of any implementation or algorithm. It's a property of the schema itself and without rewriting it a schema is either ambiguous or not.

On the contrary, determinism has been introduced to facilitate implementation of schema processors. The basic idea beyond determinism is that at each point when matching an instance document against a schema, the schema processor has at most one possible choice. This is making the life easier for implementers which can safely rely on well known algorithm such as automatons (also called Finite State Machines or FSMs) and be sure that their computation times will not grow up exponentially. This is also a major constraint imposed on schema authors.

An ambiguous schema is always non deterministic, but the opposite is far from being true. Consider for instance:

```
element foo{empty} | (element foo{empty}, element bar{empty})
```

This is not ambiguous since after having read the element after an empty element named `foo` a schema processor is able to say if the right or left alternative is being used (or none if the document is invalid)

but this is non deterministic since when a schema processor is matching an empty element named `foo` it has two different choices and cannot choose between them without looking ahead at the next element.

Ambiguous schemas are not a problem as long as validation only is concerned: their validation reports are consistent and we don't care why a document is valid or not as long as the answer (valid or invalid) is reliable. The only real downside about ambiguous schemas is for applications performing datatype assignment (or more generally instance document annotation) through validation and we will see more about these issues in the next sections of this chapter.

The main issue with schema languages requiring deterministic schemas is that some content models are fundamentally non deterministic and cannot be rewritten in a deterministic form. Such schema languages are not only adding restrictions on the forms to use to write a schema but their expressive power is limited and they cannot describe all the content models allowed in well formed XML. We will see examples of content models impossible to describe in a deterministic form in the section about compatibility with W3C XML Schema.

Different types of ambiguities

In a Relax NG schema, we can distinguish four different types of ambiguities: regular expression ambiguities, hedge grammar ambiguities, name classes ambiguity and datatype ambiguities and we'll briefly introduce them since they have slightly different behaviors.

Regular expression ambiguities

Note that in this chapter we are using the term "regular expression" as used in the math behind Relax NG. The term "regular expression" that you'll find in this chapter should thus not be confused with the regular expressions which we've seen in the W3C XML Schema `pattern` facet.

After a schema has been simplified, we can make a clear distinction between the definition of each element (embedded in its own named pattern) and the grammar which combines these definitions. What's called a regular expression ambiguity is an ambiguity which resides within the definition of an element.

Theoreticians have demonstrated that any ambiguous regular expressions may be rewritten in an unambiguous way and these ambiguities may be considered as unlucky variations over unambiguous schemas.

A basic example of such a choice between a pattern and itself is:

```
<choice>
 <ref name="pattern"/>
 <ref name="pattern"/>
</choice>
```

or:

```
pattern|pattern
```

Obvious in this case, the unambiguous form is more or less difficult to find when the ambiguous pattern gets more complex. For instance, the following pattern:

```
<choice>
 <group>
```

```
  <optional>
   <ref name="first"/>
  </optional>
  <ref name="second"/>
 </group>
 <group>
  <ref name="second"/>
  <optional>
   <ref name="third"/>
  </optional>
 </group>
</choice>
```

or:

```
(first?,second)|(second,third?)
```

Is ambiguous because an instance nodeset matching only the named pattern `second` without the leading `first` nor the ending `third` is valid per the two alternative of the choice. It can be rewritten by removing the option of matching only the `second` pattern from one of the alternatives:

or:

```
 <group>
  <optional>
   <ref name="first"/>
  </optional>
  <ref name="second"/>
 </group>
 <group>
  <ref name="second"/>
  <ref name="third"/>
 </group>
</choice>
```

or:

```
(first?,second)|(second,third)
```

Algorithms have been developed to rewrite ambiguous regular expressions in their unambiguous forms and it would be really useful if XML development tools could implement them to propose non ambiguous alternatives for ambiguous patterns when they exist. Until this happens, the best thing to do when you are confronted with an ambiguous pattern to disambiguate is to take a step back, grab a cup of tea or coffee and calmly write the different combinations expressed by the schema to combine them differently till the combination isn't ambiguous any longer.

Note that explicit choices aren't the only pattern which can lead to ambiguous schemas. Consider this simple pattern:

```
<group>
 <optional>
  <ref name="pattern"/>
```

```
  </optional>
  <optional>
   <ref name="pattern"/>
  </optional>
 <group>
```

or:

```
 pattern?, pattern?
```

If we have a content model which matches only one pattern, we cannot know if it will match it for the first or the second occurrence of the pattern and this schema can be considered as ambiguous. To rewrite it as a non ambiguous schema, we could write:

```
  <optional>
   <ref name="pattern"/>
   <optional>
    <ref name="pattern"/>
   </optional>
  </optional>
```

or:

```
 (pattern, pattern?)?
```

Although the way leading to rewritings may look opaque, the math behind Relax NG can help us like high school algebra helps us factorize mathematical expressions. As an exercise, let's decompose the chain of factorizations and simplifications to rewrite "pattern?, pattern?" as "(pattern, pattern?)?".

The first step relies on the fact that "pattern?" is equivalent to "empty|pattern":

```
 pattern?, pattern?
```

is equivalent to:

```
 (empty|pattern), (empty|pattern)
```

which can be factorized as:

```
 (empty,empty)|(empty,pattern)|(pattern,empty)|(pattern,pattern)
```

which can be simplified as:

```
 empty|pattern|(pattern,pattern)
```

which is equivalent to:

```
empty|(pattern,(empty|pattern))
```

which is equivalent to

```
(pattern, pattern?)?
```

We could argue whether the unambiguous forms are clearer, more logical an easier to read than the ambiguous forms or not, but I think that the answer would be very subjective. These different forms are highly dependent of the perspective from which we have analyzed the content of instance documents. There isn't a good nor a bad form and working with a schema language such as Relax NG which supports all of these forms does save a lot of time: you don't have to take a perspective imposed by the language!

A last thing to note is that disambiguating regular expressions does not significantly change the structure or the style of your schema since the changes are limited to the regular expression itself and this will not be the case of ambiguous regular hedge grammars.

Ambiguous regular hedge grammars

In the case of a Relax NG schema, we've defined a regular expression ambiguity as an ambiguity which resides within the definition of an element. Ambiguous regular hedge grammars are on the contrary ambiguities spread over element definitions which play the role of "hedges" in a Relax NG schema. A example of an ambiguous regular hedge grammar is:

```
    <choice>
      <ref name="pattern1"/>
      <ref name="pattern2"/>
    </choice>
  .../...
  <define name="pattern1">
    <element name="foo">
      <empty/>
    </element>
  </define>
  <define name="pattern2">
    <element name="foo">
      <empty/>
    </element>
  </define>
```

or:

```
pattern1|pattern2
.../...
pattern1=element foo{empty}
pattern2=element foo{empty}
```

This example is ambiguous because when we find an empty element `foo` we can't tell if it's been validated through `pattern1` or `pattern2` and it's an ambiguous hedge grammar (rather than an ambiguous regular expression) because the ambiguity is spread over two hedges, i.e. two definitions of the element `foo`.

Again, it has been demonstrated that ambiguous regular hedge grammars can be rewritten in unambiguous forms but the disambiguation must be done at the level of the grammar itself and does often requires heavy changes to the structure of the schema.

The exercise of disambiguating regular hedge grammars can get significantly complicated when compositions of named patterns and grammars are involved. For instance, maintaining non ambiguous patterns while combining definitions by choice means that you need to exclude all the instance nodesets valid per the original definition from the pattern given as a choice and this isn't always possible without modifying the included schema. Consider for instance this pattern:

```
<define name="namedPattern">
 <ref name="first"/>
</define>
```

or:

```
namedPattern=first
```

If we need to add an optional `second` pattern it may seem natural to combine it by choice as:

```
<define name="namedPattern" combine="choice">
 <ref name="first"/>
 <optional>
  <ref name="second"/>
 </optional>
</define>
```

or:

```
namedPattern|=first,second?
```

The result of the combination is equivalent to:

```
<define name="namedPattern">
 <choice>
  <ref name="first"/>
  <group>
   <ref name="first"/>
   <optional>
    <ref name="second"/>
   </optional>
  </group>
 </choice>
</define>
```

or:

```
namedPattern=first|(first,second?)
```

This gives us an ambiguous pattern. Of course, outside the context of a pattern combination, this would be trivial to rewrite as:

```
<define name="namedPattern">
 <ref name="first"/>
 <optional>
  <ref name="second"/>
 </optional>
</define>
```

or:

```
namedPattern=first,second?
```

but in this case, we won't get there directly by pattern combination and we need to take the problem under a different angle and consider that we must leave in the alternative to the original definition only things which would are not already allowed. In other words, we need to remove from our target of "the first pattern followed by an optional second pattern" the case where the first pattern is not followed by the second one. The alternative will thus be between the first pattern alone and the first pattern followed by a second one:

```
<define name="namedPattern" combine="choice">
 <choice>
  <ref name="first"/>
  <group>
   <ref name="first"/>
   <ref name="second"/>
  </group>
 </choice>
</define>
```

or:

```
namedPattern=first|(first,second)
```

With this target in mind, we can rewrite our combination as:

```
<define name="namedPattern" combine="choice">
 <ref name="first"/>
 <ref name="second"/>
</define>
```

or:

```
namedPattern|=first,second
```

If we want to avoid ambiguous hedge grammars, we also need to be careful when combining named patterns by choice: without knowing how pattern1 and pattern2 are defined, it's just impossible to say if:

```
<choice>
 <ref name="pattern1"/>
 <ref name="pattern2"/>
</choice>
```

or:

```
pattern1|pattern2
```

is ambiguous or not.

Name class ambiguity

Another source of ambiguity is when name classes used in different alternatives of a choice overlap. Again, a schematic example of such overlap would be:

```
<choice>
  <element name="foo">
    <empty/>
  </element>
  <element>
    <anyName/>
    <empty/>
  </element>
</choice>
```

or:

```
element foo{empty} | element * {empty}
```

This is ambiguous since the name class `anyName` includes the name class matching only the name `foo` and an element `foo` would be valid per both branches of the choice pattern.

The `except` name class does save our lives for name class ambiguity since it lets us remove the overlap from one of the alternatives and this pattern can easily be rewritten as a non ambiguous choice pattern:

```
<choice>
  <element name="foo">
    <empty/>
  </element>
  <element>
    <anyName>
      <except>
        <name>foo</name>
      </except>
    </anyName>
    <empty/>
  </element>
</choice>
```

or:

```
element foo{empty} | element * - foo {empty}
```

Or more simply:

```
<element>
  <anyName>
    <except>
      <name>foo</name>
    </except>
  </anyName>
  <empty/>
</element>
```

or:

```
element * {empty}
```

Note that the fact that name classes overlap is not enough to make an ambiguous pattern. For instance:

```
<choice>
  <element name="foo">
    <attribute name="bar">
      <empty/>
    </attribute>
  </element>
  <element>
    <anyName/>
    <empty/>
  </element>
</choice>
```

or: element foo{attribute bar{empty}} | element * {empty}}

is no ambiguous since the content model of the elements with the two name class do not overlap. Making our `bar` attribute optional:

```
<choice>
  <element name="foo">
    <optional>
      <attribute name="bar">
        <empty/>
      </attribute>
    </optional>
  </element>
  <element>
    <anyName/>
    <empty/>
```

```
      </element>
    </choice>
```

or: element foo{attribute bar{empty}?} | element * {empty}}

is enough to make our pattern ambiguous. However, this pattern is strictly equivalent to the preceding one which means that we know how to rewrite it in a non ambiguous way.

Finally, note that name class ambiguity may be considered as an extension to regular hedge grammar ambiguity. When we have been writing:

```
element foo{empty} | element foo{empty}
```

which after simplification is an example of regular hedge grammar ambiguity, the ambiguity comes from the fact that the name classes for both alternatives are the single value `foo` and thus do overlap.

Ambiguous datatypes

Datatype ambiguity is the one which is the most difficult to handle with Relax NG and that doesn't come from Relax NG itself but rather from the fact that datatype libraries are not built-in and are kind of opaque and less flexible than other patterns or name classes.

A basic example of ambiguous datatypes is:

```
<element name="foo">
  <choice>
    <data type="boolean"/>
    <data type="integer"/>
  </choice>
</element>
```

or:

```
element foo{xsd:boolean|xsd:integer}
```

Since the lexical space of the two possible datatypes do overlap (0 and 1 are valid W3C XML Schema boolean and integers), there is no way to determine what is the datatype an element foo with a value `0` or `1`. We have no except pattern available to remove the lexical space of a datatype from the lexical space of another datatype and, the only way to disambiguate such pattern is using parameters when the datatype library provides any. In the case of W3C XML Schema, the parameter to use if we want to work on the lexical space is the pattern parameter and we could remove `0` and `1` from the lexical space of the boolean datatype by either specifying the lexical space of boolean as being explicitly `true` or `false`:

```
<element name="foo">
  <choice>
    <data type="boolean">
      <param name="pattern">true|false</param>
    </data>
    <data type="integer"/>
  </choice>
</element>
```

or:

```
element foo{
  xsd:boolean{pattern="true|false"}
  |xsd:integer
}
```

Or by removing the lexical space of integer from boolean:

```
    <element name="foo">
      <choice>
        <data type="boolean">
          <param name="pattern">[^0-9]*</param>
        </data>
        <data type="integer"/>
      </choice>
    </element>
```

or:

```
element foo{
 xsd:boolean{pattern="[^0-9]*"}
 |xsd:integer
}
```

That's not much fun for more complex cases, but that's the only hope we have to disambiguate such ambiguities.

The downsides of ambiguous and non deterministic content models

Again, if you're only interested in using a Relax NG schema for validation which, after all, is the primary goal of Relax NG, it is perfectly fine to design and use non deterministic and even ambiguous schemas. The downsides of ambiguous schemas appear when we want to use Relax NG schemas for adding validation information to the instance documents or use a Relax NG schema for guided editing and the downsides of non deterministic schemas only appear when we want to be able to translate our schemas into a W3C XML Schema.

Instance annotations

What I'll be calling instance annotation in this book is the ability to attach to the instance document information gathered during the validation to facilitate its processing. Instance annotation is probably one of the most promising paths to automating XML document processing and its applications cover domains such as datatype assignment (which is one of the basis of XQuery 1.0, XPath 2.0 and XSLT 2.0), data binding (probably the only way to automate the creation of objects from XML documents and the creation of XML documents from objects) and XML guided editing.

Some tools may have more stringent requirements depending on their algorithms (for instance, a SAX based streaming tool might want to impose deterministic schemas), but in theory (and in general), it

is sufficient for the applications of instance annotations to insure that the annotations are consistent and this can be achieved if the schema is unambiguous.

Note that even this condition isn't always required and that these requirements are application dependents. Consider for instance a databinding application which needs to know the content model of each element. This application might be in trouble to determine which content model to use if it finds a pattern such as:

```
element foo {first?,second}
|element foo {second,third?}
```

```
first=element first{xsd:integer}
second=element second{xsd:token}
third=element third{xsd:boolean}
```

and an element foo with a content pattern matching the `second` pattern. Should it bind it into an object allowing an optional `first` or into an object allowing an optional `third`? Such ambiguity is likely to be an issue for this application. On the other hand, if all you need is to perform simple type assignment, this schema is perfectly fine since even though it is ambiguous, there is no ambiguity on datatype assignment.

As a bottom line, we can say that chasing ambiguity in your Relax NG schemas may be considered a good practice if you have in mind instance annotation applications at large, you must also check the tools which you will be using since they can have either more stringent or more relaxed requirements.

Compatibility with W3C XML Schema

I have promised to give an example of unambiguous patterns which is not deterministic and can't be rewritten in a deterministic form and here it is! Let's consider a pattern describing a book as a sequence of odd and even pages:

```
<zeroOrMore>
  <ref name="odd"/>
  <ref name="even"/>
</zeroOrMore>
<optional>
  <ref name="odd"/>
</optional>
```

or:

```
(odd, even)*, odd?
```

This pattern is not ambiguous since for any valid combinations of odd and even pages it is possible to know which pattern has matched each of the pages. It can't be deterministic since for each odd page, you need to look ahead at the next one to see if it is the last before knowing if an even page is required in next position.

W3C XML Schema requires deterministic content models under the name of "Unique Particle Attribution" and "Consistent Declaration" rules and just can't describe such a simple and useful content model!

Another example of non deterministic pattern is:

```
<choice>
 <element name="foo">
  <attribute name="bar"/>
 </element>
 <element name="foo">
  <element name="bar">
   <text/>
  </element>
 </element>
</choice>
```

or:

```
element foo {attribute bar} | element foo {element bar {text}}
```

This one would seem easier to translate. At least, it can be factorized and rewritten as a deterministic pattern in Relax NG as:

```
<element name="foo">
 <choice>
  <attribute name="bar"/>
  <element name="bar">
   <text/>
  </element>
 </choice>
</element>
```

or:

```
element foo {attribute bar| element bar {text}}
```

Unfortunately, this doesn't help to translate our schema into W3C XML Schema since this language doesn't know how to handle this type of situations mixing constraints on sub elements and attributes without using dark hacks with key definitions which don't work in all the cases.

Making sure that your schemas are deterministic is thus a good practice when you plan to translate your schemas into W3C XML Schema schemas but unfortunately not a guarantee that they will translate gracefully. The only rule I can give if you want to make sure that your schemas will be easy to translate is to check the result of translation frequently as you write your schema and hope that James Clark will continue to improve the conversion algorithm!

On the other hand, note that W3C XML Schema deals nicely with datatype ambiguities. If we take or example of datatype ambiguity:

```
element foo{xsd:boolean|xsd:integer}
```

you will be surprised to know that it translates gracefully into:

```
<xs:element name="foo">
  <xs:simpleType>
    <xs:union memberTypes="xs:boolean xs:integer"/>
  </xs:simpleType>
</xs:element>
```

and that this is not considered as ambiguous by W3C XML Schema which has added a rule to say that when several datatypes where grouped "by union" which is basically what our choice between datatype does, a processor should stop after the first type which matches and not evaluate the next alternatives.

Some ideas to make disambiguation easier

To close this chapter I'd like to present some ideas which would facilitate our lives in disambiguating schemas.

Generalized `except` pattern

In the different forms of ambiguity, name classes has been the easiest one to disambiguate. Why is this? Not because name classes are inherently simpler than regular expressions or datatypes: all of them are about defining sets of things that can happen in a XML documents and I would argue that they are very similar. The reason why name classes have been easier to disambiguate is because they have a first class `except` operator and if we had the same level of support for patterns and datatypes, we could more easily disambiguate them.

Applied to datatypes, this is already possible to some extend and away to disambiguate our example:

```
element foo{xsd:boolean|xsd:integer}
```

is to write:

```
element foo{ (xsd:boolean - xsd:integer) |xsd:integer}
```

A value which is only integer will obviously match only the right alternative. A value which is only boolean (i.e. `true` and `false`) will match the left alternative and a value which is both a boolean and an integer (i.e. `0` and `1`) will match the first condition of the left alternative (`xsd:boolean`) but will not match the exception clause.

Unfortunately, this can't be generalized out of the scope of `data` patterns (note that the examples given below with the except (`-`) operator are not valid Relax NG).

If this could be generalized, applied to an ambiguous regular expression such as:

```
two|(one?,two+,three*)
```

We would be able to write:

```
two|((one?,two+,three*)-two)
```

Of course, this can be developed and rewritten with the existing Relax NG patterns, but that would give a new level of flexibility to the language.

Explicit disambiguation rules

The second idea I'd like to give is far less disruptive and is just the realization that these ambiguities are just ambiguous because we have not decided anything to rule them out. There are plenty of examples in other computer languages of ambiguities which have been partially or fully ruled out such as for XSLT templates, order of evaluation of statements in programming languages or as we've seen in the section about W3C XML Schema union of datatypes.

There is absolutely nothing preventing writing a specification defining a priority for the alternatives to be used by applications interested in instance annotation at large when they encounter ambiguities.

This specification wouldn't need to apply to Relax NG processors interested only in validation and would not compromise their optimizations. It would only apply to NG processors performing instance annotation and guarantee a consistent and interoperable type annotation for schema which are today considered as ambiguous.

The rule could be as simple as "use the first alternative in document order" or it could also take into account additional factors such as giving a lesser precedence to included grammars like XSLT does with stylesheet imports.

Accepting ambiguity

The third idea has been proposed by Jeni Tennison on the xml-dev mailing list: instead of trying to fight against ambiguity, why not accept it? Why couldn't we acknowledge that something can have several datatypes (or model) and have at the same type a datatype "A" and "B"? Why a value couldn't be at the same type an `integer` and a `boolean`?

This idea would have a serious impact on specifications such as XPath 2.0 which assign a single datatype to each simple type element and attribute, but that would be much more compatible with the principle of markup which is only the projection of a structure over a document. It often happen that a piece of text may have several meaning, acknowledging that elements and attributes may have belong to datatypes at the same time just seems something obvious to do.

Part III. Short reference guide

This part is a concise reference guide covering the elements of the RELAX NG XML syntax, the commented EBNF of the non XML syntax, a glossary and a short reference guide for the W3C XML Schema datatypes (adapted from my book about W3C XML Schema).

Chapter 18. Elements reference guide

This short reference guide to Relax NG elements presents each of the elements composing the XML syntax for Relax NG by alphabetical order. Note that the synopsis given for each element is generated from the Relax NG schema for Relax NG and dooesn't capture the restrictions applied after simplification. The simplification process and restrictions are detailed in "Chapter 15: Simplification And Restrictions". The main restrictions are also mentioned for each element in this chapter in the section titled "Restrictions".

Elements

Name

anyName — Name class accepting any name.

Class:

name-class

Synopsis

```
element anyName
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( ( element * - rng:* { ... }* ) & element except { ... }? )
}
```

May be included in:

`attribute`, `choice`, `element`, `except`.

Compact syntax equivalent:

*-nameClass

Description:

The `anyName` name class matches any name from any namespace. This wild spectrum may be restricted by embedding `except` name classes.

Restrictions:

Within the scope of an element, the name classes of attributes cannot overlap. The same restriction applies to name classes of elements when these elements are combined by `interleave`.

Example:

```
<element>
 <anyName/>
 <ref name="anything"/>
</element>

<element>
 <anyName>
   <except>
     <nsName ns=""/>
     <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
     <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
   </except>
 </anyName>
 <ref name="anything"/>
</element>
```

```
<attribute>
  <anyName/>
</attribute>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited. Note that although `datatypeLibrary` is allowed in `anyName` to maintain a coherence with other Relax NG elements, it has no direct consequence on `anyName` itself nor on the name class definitions which might be embedded.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited. Note that although `ns` is allowed in `anyName` it has no direct consequence on `anyName` itself which does always allow any name from any namespace and may only have a consequence on name class definitions embedded in this one."

Name

attribute — Pattern matching an attribute.

Class:

pattern

Synopsis

```
element attribute
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      attribute name { xsd:QName }
    | (
         ( element * - rng:* { ... }* )
       & (
            element name { ... }
          | element anyName { ... }
          | element nsName { ... }
          | element choice { ... }
         )
      )
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )?
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

attribute

Description:

The `attribute` pattern matches an attribute. The name of the attribute may be defined either through a `name` attribute or through a name class.

Restrictions:

<itemizedList>
<listItem>

After simplification, attributes patterns can only contain patterns relevant for text nodes.
</listItem>
<listItem>

Attributes cannot be duplicated, either directly or through overlaping name classes.
</listItem>
<listItem>

Attributes which have an infinite name class (`anyName` or `nsName`) must be enclosed in a `oneOrMore` (or `zeroOrMore` before simplification) pattern.
</listItem>
</itemizedList>

Example:

```
<attribute name="id"/>

<attribute name="xml:lang">
  <data type="language"/>
 </attribute>

<attribute>
  <anyName/>
</attribute>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`name`	When `name` is specified, the `attribute` pattern matches only attributes with this name (`name` is a shortcut to define a single name as a name class for the `attribute` pattern). Both `name` and the definition of a name class cannot be specified (they are exclusive options).

`ns`	The ns attribute defines the namespace of the attribute. Note that in the context of the `ns` pattern, the ns attribute is not inherited.

Name

choice (in the context of a name-class) — Choice between name classes

Class:

name-class

Synopsis

```
element choice
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element name { ... }
       | element anyName { ... }
       | element nsName { ... }
       | element choice { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `element`, `except`.

Compact syntax equivalent:

nameClass|nameClass

Description:

The `choice` name class performs a choice between several name classes: a name will match `choice` if and only if it matches at least one of the sub-name classes.

Example:

```
<element>
  <choice>
    <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
    <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
  </choice>
  <ref name="anything"/>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is

	inherited. Note that although `datatypeLibrary` is allowed in `choice` to maintain a coherence with other Relax NG elements, it has no direct consequence on `choice` itself nor on the name class definitions which might be embedded.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

choice (in the context of a pattern) — `choice` pattern

Class:

pattern

Synopsis

```
element choice
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

pattern|pattern

Description:

The `choice` pattern defines a choice between different patterns: it matches a node if and only if at least one of its sub-pattern matches this node.

Example:

```
<element name="name">
 <choice>
  <text/>
  <group>
   <element name="first"><text/></element>
   <optional>
     <element name="middle"><text/></element>
   </optional>
   <element name="last"><text/></element>
  </group>
 </choice>
</element>

<attribute name="available">
  <choice>
   <value>true</value>
   <value>false</value>
   <value>who knows?</value>
  </choice>
 </attribute>

 <start>
   <ref name="libraryElement"/>
   <ref name="bookElement"/>
 </start>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

data — `data` pattern

Class:

pattern

Synopsis

```
element data
{
   attribute type { xsd:NCName },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & ( element param { ... }*, element except { ... }? )
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

datatypeName param exceptPattern

Description:

The `data` pattern matches a single text node and gives the possibility to restrict its values. This is different from the `text` pattern which matches zero or more text nodes and doesn't give any possibility to restrict the values of these text nodes. The restrictions are applied through the `type` attribute which defines the datatype and the `param` and `except` children patterns.

Restrictions:

The `data` pattern is meant for data oriented applications and can't be used in mixed content models.

Example:

```
<attribute name="see-also">
 <list>
  <data type="token"/>
 </list>
</attribute>

<attribute name="id">
 <data type="NMTOKEN">
   <param name="maxLength">16</param>
```

```
 </data>
</attribute>

<element name="isbn">
 <data type="token">
  <except>
   <value>0836217462</value>
  </except>
 </data>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.
`type`	The `type` attribute specifies the datatype used for evaluating the `data` pattern. Any text node which value isn't valid per this datatype fails to match the `data` pattern.

Name

define — Named pattern definition

Class:

define-element

Synopsis

```
element define
{
   attribute name { xsd:NCName },
   ( attribute combine { "choice" | "interleave" }? ),
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`div`, `grammar`, `include`.

Compact syntax equivalent:

identifier assignMethod pattern

Description:

When `define` is embedded in a grammar, it defines a named pattern or combines a new definition with an existing one. Named pattern are global to a `grammar` and can be referenced by `ref` in the

scope of their `grammar` and by `parentRef` in the scope of the grammars directly embedded in their `grammar`.

When `define` is embedded in `include`, the new definition is a redefinition and replaces the definitions from the included grammar unless a `combine` attribute is specified in which case the definitions are combined.

Restrictions:

Named patterns are always global and apply only to patterns and it is not possible to define and make reference to non patterns such as class names or datatype parameters.

Example:

```
<define name="born-element">
  <element name="born">
    <text/>
  </element>
</define>

<define name="book-content" combine="interleave">
  <attribute name="id"/>
  <attribute name="available"/>
  <ref name="isbn-element"/>
  <ref name="title-element"/>
  <zeroOrMore>
    <ref name="author-element"/>
  </zeroOrMore>
  <zeroOrMore>
    <ref name="character-element"/>
  </zeroOrMore>
</define>

<define name="isbn-element" combine="choice">
  <notAllowed/>
</define>
```

Attributes:

`combine`

The `combine` attribute specifies how multiple definitions of a named pattern should be combined together. The possible values are `choice` and `interleave`.

When the `combine` attribute is specified and set to `choice`, multiple definitions of a named pattern are combined in a `choice` pattern. When the `combine` attribute is specified and set to `interleave`, multiple definitions of a named pattern are combined in an `interleave` pattern.

Note that it is forbidden to specify more than one `define` with the same name and no `combine` attribute or multiple `define` with different values of `combine` attribute.

`datatypeLibrary`

The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.

`name`	The `name` attribute specifies the name of the named pattern.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

div (in the context of a grammar-content) — Division (in the context of a `grammar`)

Class:

grammar-content

Synopsis

```
element div
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         ( element start { ... } )
       | ( element define { ... } )
       | element div { ... }
       | element include { ... }
      )*
   )
}
```

May be included in:

`div`, `grammar`.

Compact syntax equivalent:

div

Description:

The `div` element is provided to define logical divisions in Relax NG schemas. It has no effect on the validation and its purpose is to define a group of definitions within a `grammar` which may be annotated as a whole.

In the context of a `grammar`, the content of a `div` element is the same than the content of a `grammar` (this means that `div` elements may be embedded in other `div` elements.

Example:

```
  <grammar xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns="http://relaxng.org
 .../...
   <div>
     <xhtml:p>The content of the book element has been split in two named patte
     <define name="book-start">
       <attribute name="id"/>
       <ref name="isbn-element"/>
       <ref name="title-element"/>
       <zeroOrMore>
```

```
        <ref name="author-element"/>
      </zeroOrMore>
    </define>
    <define name="book-end">
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
      <attribute name="available"/>
    </define>
  </div>
.../...
</grammar>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

div (in the context of a include-content) — Division (in the context of an `include`)

Class:

include-content

Synopsis

```
element div
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         ( element start { ... } )
       | ( element define { ... } )
       | element div { ... }
      )*
   )
}
```

May be included in:

`div`, `include`.

Compact syntax equivalent:

div

Description:

The `div` element is provided to define logical divisions in Relax NG schemas. It has no effect on the validation and its purpose is to define a group of definitions within an `include` which may be annotated as a whole.

In the context of an `include`, the content of a `div` element is the same than the content of a `include` (this means that `div` elements may be embedded in other `div` elements.

Example:

```
  <include href="common.rng">
 .../...
   <div>
     <xhtml:p>The content of the book element has been split in two named patte
     <define name="book-start">
       <attribute name="id"/>
       <ref name="isbn-element"/>
       <ref name="title-element"/>
       <zeroOrMore>
         <ref name="author-element"/>
```

```
      </zeroOrMore>
    </define>
    <define name="book-end">
      <zeroOrMore>
        <ref name="author-element"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="character-element"/>
      </zeroOrMore>
      <attribute name="available"/>
    </define>
  </div>
 .../...
</include>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

element — Pattern matching an element

Class:

pattern

Synopsis

```
element element
{
   (
      attribute name { xsd:QName }
    | (
         ( element * - rng:* { ... }* )
       & (
            element name { ... }
          | element anyName { ... }
          | element nsName { ... }
          | element choice { ... }
         )
      )
   ),
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

element

Description:

The `element` pattern matches an element. The name of the element may be defined either through a `name` attribute or through a name class.

Example:

```
<element name="born">
 <text/>
</element>

<element name="character">
  <attribute name="id"/>
  <element name="name">
   <text/>
  </element>
  <element name="born">
   <text/>
  </element>
  <element name="qualification">
   <text/>
  </element>
</element>

<element>
  <anyName/>
  <ref name="anything"/>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`name`	When `name` is specified, the `element` pattern matches only elements with this name (`name` is a shortcut to define a single name as a name class for the `element` pattern). Both `name` and the definition of a name class cannot be specified (they are exclusive options).
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

empty — Empty content

Class:

pattern

Synopsis

```
element empty
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

empty

Description:

The `empty` patterns is used to define elements which are empty, i.e. which have no children elements, text nor attributes. Note that it is mandatory to use this pattern in such case (`<element name="foo"/>` is forbidden) and that there is no such thing as empty attributes (an attribute such as `foo=""` is considered as having a value which is the empty string rather than be considered as being empty, i.e. having no value).

Example:

```
<element name="pageBreak">
  <empty/>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

except (in the context of a except-name-class) — Remove a name class from another

Class:

except-name-class

Synopsis

```
element except
{
   (
      ( element * - rng:* { ... }* )
    & (
         element name { ... }
       | element anyName { ... }
       | element nsName { ... }
       | element choice { ... }
      )+
   )
}
```

May be included in:

`anyName`, `nsName`.

Compact syntax equivalent:

-nameClass

Description:

The `except` name class is used to remove a name class from another. Note that this name class has no attributes.

Restrictions:

It is impossible to use `except` to produce empty name classes by including "anyName" in an "except" name class or "nsName" in an "except" name class included in another "nsName".

Example:

```
<element>
  <anyName>
    <except>
      <nsName ns=""/>
      <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
      <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
    </except>
  </anyName>
  <ref name="anything"/>
</element>

<element>
```

```
 <nsName ns=ns="http://eric.van-der-vlist.com/ns/person"/>
   <except>
     <name>lib:name</name>
     <name>hr:name</name>
   <except>
 </nsName>
 <ref name="anything"/>
</element>
```

Attributes:

None.

Name

except (in the context of a pattern) — Remove a set of values from a `data`

Class:

pattern

Synopsis

```
element except
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`data`.

Compact syntax equivalent:

-pattern

Description:

The `except` pattern is used to remove a set of values from a `data` pattern.

Restrictions:

The `except` pattern can only be used in the context of data and can only contain `data`, `value` and `choice` elements.

Example:

```
  <element name="isbn">
 <data type="token">
  <except>
   <value>0836217462</value>
  </except>
 </data>
</element>

<attribute name="available">
 <data type="token">
  <except>
   <choice>
    <value type="string">true</value>
    <value type="string">false</value>
   </choice>
  </except>
 </data>
</attribute>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

externalRef — Reference to an external schema

Class:

pattern

Synopsis

```
element externalRef
{
   attribute href { xsd:anyURI },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

external

Description:

The `externalRef` pattern is a reference to an external schema. This has the same effect than replacing the `externalRef` pattern by the external schema considered as a pattern.

Example:

```
<element name="book">
  <externalRef href="book.rng"/>
</element>

<element xmlns="http://relaxng.org/ns/structure/1.0" name="university">
 <element name="name">
  <text/>
 </element>
 <externalRef href="flat.rng"/>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.

`href`	The `href` attribute defines the location of the external schema.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

grammar — Grammar pattern

Class:

pattern

Synopsis

```
element grammar
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         ( element start { ... } )
       | ( element define { ... } )
       | element div { ... }
       | element include { ... }
      )*
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

grammar

Description:

The `grammar` pattern encapsulates the definitions of `start` and named patterns.

The most common use of `grammar` is to validate XML documents and in this case the `start` pattern defines which elements may be used as the document root element. The `grammar` pattern may also be used as a way to write modular schemas and in this case the `start` pattern defines which nodes must be matched by the `grammar` at the location where it appears in the schema.

In every case, the named patterns defined in a `grammar` are considered to be local to this `grammar`.

Example:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
 <start>
  <element name="library">
   <oneOrMore>
    <ref name="book-element"/>
   </oneOrMore>
```

```
  </element>
 </start>
 <define name="author-element">
 .../...
 </define>
</grammar>

 <define name="author-element">
   <grammar>
     <start>
       <element name="author">
         <attribute name="id"/>
         <ref name="name-element"/>
         <parentRef name="born-element"/>
         <optional>
           <ref name="died-element"/>
         </optional>
       </element>
     </start>
     <define name="name-element">
       <element name="name">
         <text/>
       </element>
     </define>
     <define name="died-element">
       <element name="died">
         <text/>
       </element>
     </define>
   </grammar>
 </define>

 <element xmlns="http://relaxng.org/ns/structure/1.0" name="university">
  <element name="name">
    <text/>
  </element>
  <grammar>
    <include href="flat.rng"/>
  </grammar>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

group — `group` pattern

Class:

pattern

Synopsis

```
element group
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

pattern,pattern

Description:

The `group` pattern defines an ordered group of sub patterns (note that when `attribute` patterns are included in such a group, their order is not enforced). `group` patterns are implicit with `element` and `define` patterns.

Example:

```
  <element name="name">
   <choice>
    <text/>
    <group>
     <element name="first"><text/></element>
     <optional>
       <element name="middle"><text/></element>
     </optional>
     <element name="last"><text/></element>
    </group>
   </choice>
  </element>

 <element name="foo">
  <interleave>
   <element name="out"><empty/></element>
   <group>
    <element name="in1"><empty/></element>
    <element name="in2"><empty/></element>
   </group>
  </interleave>
 </element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

include — Grammar merge

Class:

grammar-content

Synopsis

```
element include
{
   attribute href { xsd:anyURI },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         ( element start { ... } )
       | ( element define { ... } )
       | element div { ... }
      )*
   )
}
```

May be included in:

`div`, `grammar`.

Compact syntax equivalent:

include

Description:

The `include` pattern includes a grammar and merges its definitions with the definitions of the current grammar. The definitions of the included grammar may be redefined and overridden by the definitions embedded in the `include` pattern. Note that a schema must contain an explicit `grammar` definition in order to be included.

Example:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
 <start>
  <element name="library">
   <oneOrMore>
    <ref name="book-element"/>
   </oneOrMore>
  </element>
 </start>
 <include href="included.rng"/>
 .../...
</grammar>
```

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
 <include href="flat.rng">
   <define name="book-element">
     <element name="book">
       <attribute name="id"/>
       <attribute name="available"/>
       <ref name="isbn-element"/>
       <ref name="title-element"/>
       <ref name="description-element"/>
       <zeroOrMore>
         <ref name="author-element"/>
       </zeroOrMore>
     </element>
   </define>
 </include>
 <define name="description-element">
   <element name="description">
     <text/>
   </element>
 </define>
</grammar>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`href`	The `href` attribute defines the location of the schema which grammar should be included.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

interleave — `interleave` Pattern

Class:

pattern

Synopsis

```
element interleave
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

pattern&pattern

Description:

The `interleave` pattern "interleaves" sub patterns, i.e. allows their leaves to be mixed in any relative order.

`interleave` is more than defining unordered groups as we can see on the following example: consider element "a" and the ordered group of element "b1" and "b2". An unordered group of these two patterns would only allow element "a" followed by elements "b1" and "b2" or elements "b1" and "b2 followed by element "a". An `interleave` of these two patterns does allow these two combinations but also element "b1" followed by "a" followed by "b2", i.e. a combination where the element "a" has been "interleaved" between elements "b1" and "b2".

The `interleave` behavior is the behavior applied to `attribute` patterns even when they are embedded in (ordered) `group` patterns (the reason for this is that XML 1.0 specifies that the relative order of attributes is not significant).

Another case where `interleave` patterns are often needed is to described mixed content models, i.e. content models where `text` are interleaved between elements. A shortcut (the `mixed` pattern) has been defined for this case.

Restrictions:

<itemizedList>
<listItem>

The `interleave` pattern cannot be used within a `list`.
</listItem>
<listItem>

Elements within a `interleave` pattern cannot have overlapping name classes.
</listItem>
<listItem>

There must be at most one "text" pattern in each set of patterns combined by `interleave`
</listItem>
</itemizedList>

Example:

```
 <element name="character">
  <interleave>
   <attribute name="id"/>
   <element name="name">
    <text/>
   </element>
   <element name="born">
    <text/>
   </element>
   <element name="qualification">
    <text/>
   </element>
  </interleave>
 </element>

<element name="title">
 <interleave>
  <attribute name="xml:lang"/>
  <zeroOrMore>
   <element name="a">
    <attribute name="href"/>
    <text/>
   </element>
  </zeroOrMore>
```

```
    <text/>
   </interleave>
  </element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

list — Text node split

Class:

pattern

Synopsis

```
element list
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

list

Description:

The `list` pattern splits a text node into tokens separated by white spaces to allow the validation of these tokens separately. This is most useful for validating lists of values.

Restrictions:

<itemizedList>
<listItem>

`interleave` cannot be used within `list`.
</listItem>
<listItem>

The content of a `list` is only about data: it's forbidden to define `element`, `attribute` or `text` there.
</listItem>
<listItem>

It's forbidden to embed `list` into `list`.
</listItem>
</itemizedList>

Example:

```
<attribute name="see-also">
 <list>
  <zeroOrMore>
   <data type="token"/>
  </zeroOrMore>
 </list>
</attribute>

<attribute name="dimensions">
 <list>
  <data type="xs:decimal"/>
  <data type="xs:decimal"/>
  <data type="xs:decimal"/>
  <choice>
   <value>inches</value>
   <value>cm</value>
   <value>mm</value>
  </choice>
 </list>
</attribute>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

mixed — Pattern for mixed content models

Class:

pattern

Synopsis

```
element mixed
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

mixed

Description:

The `mixed` pattern is a shortcut for `interleave` with an embedded `text` pattern. It describes unordered content models where a text node may be included before and after each element. Note that Relax NG does not allow to add constraints on these text nodes.

Restrictions:

The limitations of `interleave` apply here:
<itemizedList>
<listItem>

The `mixed` pattern cannot be used within a `list`.
</listItem>
<listItem>

Elements within a `mixed` pattern cannot have overlapping name classes.
</listItem>
<listItem>

There must no other "text" pattern in each set of patterns combined by `mixed`
</listItem>
</itemizedList>

Example:

```
    <element name="title">
     <mixed>
      <attribute name="xml:lang"/>
      <zeroOrMore>
       <element name="a">
        <attribute name="href"/>
        <text/>
       </element>
      </zeroOrMore>
     </mixed>
    </element>

is equivalent to:

    <element name="title">
     <interleave>
      <text/>
      <group>
        <attribute name="xml:lang"/>
        <zeroOrMore>
         <element name="a">
          <attribute name="href"/>
          <text/>
        </element>
        </zeroOrMore>
      </group>
     </interleave>
    </element>

which itself is equivalent to:

    <element name="title">
     <interleave>
      <text/>
      <attribute name="xml:lang"/>
      <zeroOrMore>
       <element name="a">
```

```
      <attribute name="href"/>
      <text/>
    </element>
    </zeroOrMore>
  </interleave>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

name — Name class for a single name

Class:

name-class

Synopsis

```
element name
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   xsd:QName
}
```

May be included in:

`attribute`, `choice`, `element`, `except`.

Compact syntax equivalent:

name

Description:

The `name` name class defines a class with a single name.

Example:

```
<element>
  <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
    <except>
      <name>lib:name</name>
      <name>hr:name</name>
    <except>
  </nsName>
  <ref name="anything"/>
</element>

<element>
  <choice>
    <name>lib:name</name>
    <name>hr:name</name>
  </choice>
  <ref name="name-content"/>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

notAllowed — Not allowed

Class:

pattern

Synopsis

```
element notAllowed
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

notAllowed

Description:

The `notAllowed` pattern always fails. It can be used to provide abstract definitions which must be overridden before they can be used in a schema.

Example:

```
<define name="isbn-element" combine="choice">
  <notAllowed/>
</define>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

nsName — Name class for any name in a namespace

Class:

name-class

Synopsis

```
element nsName
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( ( element * - rng:* { ... }* ) & element except { ... }? )
}
```

May be included in:

`attribute`, `choice`, `element`, `except`.

Compact syntax equivalent:

nsName exceptNameClass

Description:

The `nsName` name class allows any name in a specific namespace.

Restrictions:

Within the scope of an element, the name classes of attributes cannot overlap. The same restriction applies to name classes of elements when these elements are combined by `interleave`. It is impossible to use `nsName` to produce empty name classes by including `nsName` in an `except` name class included in another `nsName`.

Example:

```
<element>
  <choice>
    <nsName ns="http://eric.van-der-vlist.com/ns/library"/>
    <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
  </choice>
  <ref name="anything"/>
</element>

<element>
  <nsName ns="http://eric.van-der-vlist.com/ns/person"/>
    <except>
      <name>lib:name</name>
      <name>hr:name</name>
    <except>
```

```
    </nsName>
    <ref name="anything"/>
  </element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

oneOrMore — `oneOrMore` pattern

Class:

pattern

Synopsis

```
element oneOrMore
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

```
pattern+
```

Description:

The `oneOrMore` pattern specifies that its sub patterns considered as an ordered group must be matched one or more time.

Restrictions:

The `oneOrMore` pattern cannot contain attribute definitions.

Example:

```
<element name="library">
 <oneOrMore>
  <element name="book">
  .../...
  </element>
 </oneOrMore>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

optional — `optional` pattern

Class:

pattern

Synopsis

```
element optional
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

```
pattern?
```

Description:

The `optional` pattern specifies that its sub-patterns considered as an ordered group is optional, i.e. must be matched zero or one time.

Example:

```
  <element name="author">
   <attribute name="id"/>
   <element name="name">
    <text/>
   </element>
   <element name="born">
    <text/>
   </element>
   <optional>
    <element name="died">
     <text/>
    </element>
   </optional>
  </element>

<element name="name">
 <choice>
  <text/>
  <group>
   <element name="first"><text/></element>
   <optional>
     <element name="middle"><text/></element>
   </optional>
   <element name="last"><text/></element>
  </group>
 </choice>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

param — Datatype parameter

Class:

pattern

Synopsis

```
element param
{
   attribute name { xsd:NCName },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   text
}
```

May be included in:

`data`.

Compact syntax equivalent:

param

Description:

The `param` element defines parameters passed to the datatype library to determine if a value is valid per a datatype. When the datatype library is the W3C XML Schema datatypes, these parameters are the facets of the datatype and they define additional restrictions to be applied. The name of the parameter is defined by the `name` attribute and its value is the content of the `param` element.

Example:

```
<element name="book">
 <attribute name="id">
  <data type="NMTOKEN">
    <param name="maxLength">16</param>
  </data>
 </attribute>
 <attribute name="available">
  <data type="boolean"/>
 </attribute>
 <element name="isbn">
  <data type="NMTOKEN">
    <param name="pattern">[0-9]{9}[0-9x]</param>
  </data>
 </element>
 <element name="title">
  <attribute name="xml:lang">
   <data type="language">
    <param name="length">2</param>
```

```
 </data>
</attribute>
<data type="token">
  <param name="maxLength">255</param>
</data>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`name`	The `name` attribute specifies the name of the parameter.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

parentRef — Reference to a named pattern from the parent grammar

Class:

pattern

Synopsis

```
element parentRef
{
   attribute name { xsd:NCName },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

parent

Description:

The `parentRef` pattern is a reference to a named pattern belonging to the parent `grammar`, i.e. the `grammar` in which the current `grammar` is included. The scope of named pattern is usually limited to the `grammar` in which they are defined and the `parentRef` pattern provides a way to extend this scope and refer named pattern defined in the parent `grammar`.

Example:

```
<define name="born-element">
  <element name="born">
    <text/>
  </element>
</define>
<define name="author-element">
  <grammar>
    <start>
      <element name="author">
        <attribute name="id"/>
        <ref name="name-element"/>
        <parentRef name="born-element"/>
        <optional>
          <ref name="died-element"/>
        </optional>
      </element>
    </start>
```

```
      <define name="name-element">
        <element name="name">
          <text/>
        </element>
      </define>
      <define name="died-element">
        <element name="died">
          <text/>
        </element>
      </define>
    </grammar>
  </define>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`name`	The `name` attribute specifies the name of the named pattern which is referenced.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

ref — Reference to a named pattern

Class:

pattern

Synopsis

```
element ref
{
   attribute name { xsd:NCName },
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

Name without a colon

Description:

The `ref` pattern defines a reference to a named pattern defined in the current `grammar`.

Example:

```
<element name="book">
 <ref name="book-start"/>
 <ref name="book-end"/>
</element>

  <element name="library">
   <oneOrMore>
    <ref name="book-element"/>
   </oneOrMore>
  </element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`name`	The `name` attribute specifies the name of the named pattern which is referenced.

`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

start — Start of a grammar

Class:

start-element

Synopsis

```
element start
{
   ( attribute combine { "choice" | "interleave" }? ),
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )
   )
}
```

May be included in:

`div`, `grammar`, `include`.

Compact syntax equivalent:

start

Description:

The `start` pattern defines the "start" of a `grammar`. When this grammar is used to validate a complete document, the `start` pattern defines which elements may be used as the document (root) element. When this grammar is embedded in another `grammar`, the `start` pattern describes which pattern should be applied at the location where the `grammar` is embedded. Like named pattern

definitions, start patterns may be combined by `choice` or `interleave` and redefined when they are included in `include` patterns.

Example:

```
<start>
 <element name="library">
  <oneOrMore>
   <ref name="book-element"/>
  </oneOrMore>
 </element>
</start>
<start combine="choice">
  <ref name="book-element"/>
</start>

<define name="author-element">
  <grammar>
    <start>
      <element name="author">
        <attribute name="id"/>
        <ref name="name-element"/>
        <ref name="born-element"/>
        <optional>
          <ref name="died-element"/>
        </optional>
      </element>
    </start>
    <define name="name-element">
      <element name="name">
        <text/>
      </element>
    </define>
    <define name="born-element">
      <element name="born">
        <text/>
      </element>
    </define>
    <define name="died-element">
      <element name="died">
        <text/>
      </element>
    </define>
  </grammar>
</define>
```

Attributes:

`combine`

The `combine` attribute specifies how multiple definitions of `start` pattern should be combined together. The possible values are `choice` and `interleave`.

When the `combine` attribute is specified and set to `choice`, multiple definitions of a `start` pattern are combined in a

	`choice` pattern. When the `combine` attribute is specified and set to `interleave`, multiple definitions of a `start` pattern are combined in an `interleave` pattern. Note that it is forbidden to specify more than one `start` with the same name and no `combine` attribute or multiple `start` with different values of `combine` attribute.
`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

text — Pattern matching text nodes

Class:

pattern

Synopsis

```
element text
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   ( element * - rng:* { ... }* )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

text

Description:

The `text` pattern matches zero or more text nodes. The fact that a `text` pattern matches more than one text node has no effect when it is used in ordered content models (the data model used by Relax NG for XML documents is similar to the data model of XPath 1.0 and two text nodes cannot be adjacent) but makes a difference when a `text` pattern is used in `interleave`:adding a single `text` pattern in an `interleave` pattern has the effect of allowing any number of text nodes which can interleave before and after each element (note that the `mixed` pattern is provided as a shortcut to define these content models).

Restrictions:

No more than one `text` pattern can be included in an `interleave` pattern.

Example:

```
<element name="first"><text/></element>

 <element name="name">
  <choice>
   <text/>
   <group>
    <element name="first"><text/></element>
    <optional>
      <element name="middle"><text/></element>
    </optional>
```

```
      <element name="last"><text/></element>
    </group>
  </choice>
</element>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Name

value — Match a text node and a value

Class:

pattern

Synopsis

```
element value
{
   attribute type { xsd:NCName }?,
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   text
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

datatypeName literal

Description:

The `value` pattern matches a text node against a value using the semantic of a specified datatype to perform the comparison.

Restrictions:

The `value` pattern is meant for data oriented applications and can't be used in mixed content models.

Example:

```
<attribute name="see-also">
 <list>
  <oneOrMore>
   <choice>
    <value>0836217462</value>
    <value>0345442695</value>
    <value>0449220230</value>
    <value>0449214044</value>
    <value>0061075647</value>
   </choice>
  </oneOrMore>
 </list>
</attribute>
```

```
<attribute name="available">
 <data type="boolean">
  <except>
   <value>0</value>
   <value>1</value>
  </except>
 </data>
</attribute>

<attribute name="available">
 <data type="boolean">
  <except>
   <value type="boolean">false</value>
  </except>
 </data>
</attribute>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.
`type`	The `type` attribute specifies the datatype to use to perform the comparison. Note that this is not an inherited attribute and that when it is not specified, the comparison is done using the default datatype which is the `token` datatype of the Relax NG built in type library which means that a string comparison will be done on the values after space normalization.

Name

zeroOrMore — `zeroOrMore` pattern

Class:

pattern

Synopsis

```
element zeroOrMore
{
   (
      attribute ns { text }?,
      attribute datatypeLibrary { xsd:anyURI }?,
      attribute * - (rng:* | local:*) { text }*
   ),
   (
      ( element * - rng:* { ... }* )
    & (
         element element { ... }
       | element attribute { ... }
       | element group { ... }
       | element interleave { ... }
       | element choice { ... }
       | element optional { ... }
       | element zeroOrMore { ... }
       | element oneOrMore { ... }
       | element list { ... }
       | element mixed { ... }
       | element ref { ... }
       | element parentRef { ... }
       | element empty { ... }
       | element text { ... }
       | element value { ... }
       | element data { ... }
       | element notAllowed { ... }
       | element externalRef { ... }
       | element grammar { ... }
      )+
   )
}
```

May be included in:

`attribute`, `choice`, `define`, `element`, `except`, `group`, `interleave`, `list`, `mixed`, `oneOrMore`, `optional`, `start`, `zeroOrMore`.

Compact syntax equivalent:

pattern*

Description:

The `zeroOrMore` pattern specifies that its sub patterns considered as an ordered group must be matched zero or more time.

Restrictions:

The `zeroOrMore` pattern cannot contain attribute definitions.

Example:

```
<define name="book-element">
 <element name="book">
  <attribute name="id"/>
  <attribute name="available"/>
  <ref name="isbn-element"/>
  <ref name="title-element"/>
  <zeroOrMore>
   <ref name="author-element"/>
  </zeroOrMore>
  <zeroOrMore>
   <ref name="character-element"/>
  </zeroOrMore>
 </element>
</define>
```

Attributes:

`datatypeLibrary`	The `datatypeLibrary` attribute defines the default datatype library. The value of `datatypeLibrary` is inherited.
`ns`	The `ns` attribute defines the default namespace for the elements defined in a portion of schema. The value of `ns` is inherited.

Chapter 19. Compact syntax reference guide

Compact syntax reference guide

Introduction

This quick reference guide is following the formal description of the compact syntax described as an EBNF grammar in its documentation. Each definition of the EBNF grammar is documented and when these definitions include a long list of alternatives (such as it is the case for `pattern`, `nameClass` or `literalSegment`), each alternative is documented separately. The grammar from the specification has also been slightly simplified to suppress definitions which were used only once but its meaning has been kept unchanged. Note that this grammar is a summary which does not include the definition of annotations.

Here is the full EBNF which has been used as a basis for this guide:

```
topLevel            ::= decl* (pattern|grammarContent*)
decl                ::= "namespace" identifierOrKeyword "=" namespaceURILiteral
                        |"default" "namespace" [identifierOrKeyword] "=" namespa
                        |"datatypes" identifierOrKeyword "=" literal
pattern             ::= "element" nameClass "{" pattern "}"
                        |"attribute" nameClass "{" pattern "}"
                        |pattern ("," pattern)+
                        |pattern ("&" pattern)+
                        |pattern ("|" pattern)+
                        |pattern "?"
                        |pattern "*"
                        |pattern "+"
                        |"list" "{" pattern "}"
                        |"mixed" "{" pattern "}"
                        |identifier
                        |"parent" identifier
                        |"empty"
                        |"text"
                        |[datatypeName] literal
                        |datatypeName ["{" param* "}"] [exceptPattern]
                        |"notAllowed"
                        |"external" literal [inherit]
                        |"grammar" "{" grammarContent* "}"
                        |"(" pattern ")"
param               ::= identifierOrKeyword "=" literal
exceptPattern       ::= "-" pattern
grammarContent      ::= start
                        |define
                        |"div" "{" grammarContent* "}"
                        |"include" literal [inherit] ["{" includeContent* "}"]
includeContent      ::= define
                        |start
                        |"div" "{" includeContent* "}"
start               ::= "start" assignMethod pattern
define              ::= identifier assignMethod pattern
assignMethod        ::= "="
                        |"|="
```

```
                        |"&="
nameClass           ::= name
                        |NCName ":*" [exceptNameClass]
                        |"*" [exceptNameClass]
                        |nameClass "|" nameClass
                        |"(" nameClass ")"
name                ::= identifierOrKeyword
                        |CName
exceptNameClass     ::= "-" nameClass
datatypeName        ::= CName
                        |"string"
                        |"token"
namespaceURILiteral ::= literal
                        |"inherit"
inherit             ::= "inherit" "=" identifierOrKeyword
identifierOrKeyword ::= identifier
                        |keyword
identifier          ::= (NCName - keyword)
                        |quotedIdentifier
quotedIdentifier    ::= "\" NCName
CName               ::= NCName ":" NCName
literal             ::= literalSegment ("~" literalSegment)+
literalSegment      ::= """ (Char - (""" newline))* """
                        |"'" (Char - ("'" newline))* "'"
                        |"""""" (["""] ["""] (Char - """))* """""
                        |"''''" (["'"] ["'"] (Char - "'"))* "''''
keyword             ::= "attribute"
                        |"default"
                        |"datatypes"
                        |"div"
                        |"element"
                        |"empty"
                        |"external"
                        |"grammar"
                        |"include"
                        |"inherit"
                        |"list"
                        |"mixed"
                        |"namespace"
                        |"notAllowed"
                        |"parent"
                        |"start"
                        |"string"
                        |"text"
                        |"token"
```

Note that this EBNF dooesn't capture the restrictions applied after simplification. The simplification process and restrictions are detailed in "Chapter 15: Simplification And Restrictions". The main restrictions are also mentioned for each element in this chapter in the section titled "Restrictions".

EBNF production quick reference

Name

"""..."""" — Literal segment enclosed in three double quotes

Class:

literalSegment

Synopsis

```
"""""" (["""] ["""] (Char - """))* """""" 
```

May be included in:

`datatypeName` `literal`, `datatypes`, `external`, `include`.

XML syntax equivalent:

none

Description:

The `"""..."""` production describes literal segments enclosed in three double quotes. These segments can include any character except sequences of three double quotes.

Name

"..." — Literal segment enclosed in double quotes

Class:

literalSegment

Synopsis

```
""" (Char - (""" newline))* """
```

May be included in:

`datatypeName literal`, `datatypes`, `external`, `include`.

XML syntax equivalent:

none

Description:

The `"..."` production describes literal segments enclosed in double quotes. These segments can include any character except newlines and double quotes.

Name

'''...''' — Literal segment enclosed in three single quotes

Class:

literalSegment

Synopsis

```
"'''" (["'"] ["'"] (Char - "'"))* "'''"
```

May be included in:

`datatypeName` `literal`, `datatypes`, `external`, `include`.

XML syntax equivalent:

none

Description:

The `'''...'''` production describes literal segments enclosed in three single quotes. These segments can include any character except sequences of three single quotes.

Name

'...' — Literal segment enclosed in single quotes

Class:

literalSegment

Synopsis

```
"'" (Char - ("'" newline))* "'"
```

May be included in:

`datatypeName` `literal`, `datatypes`, `external`, `include`.

XML syntax equivalent:

none

Description:

The `'...'` production describes literal segments enclosed in single quotes. These segments can include any character except newlines and single quotes.

Name

(nameClass) — Container

Class:

nameClass

Synopsis

```
"(" nameClass ")"
```

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

none

Description:

The `(nameClass)` container is useful to group together name classes combined through "`|`" (choice). This container is a name class and may be combined with other name classes.

Even when such a container is not required, it may often be used to improve the readability of a schema.

Example:

```
element hr:* - ( hr:author | hr:name | hr:born | hr:died ) { anything }
```

Name

(pattern) — Container

Class:

pattern

Synopsis

`"(" pattern ")"`

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

none

Description:

The `(pattern)` container is useful to group together patterns combined through "`,`" (ordered group), "`|`" (choice) or "`&`" (interleave). This container is a pattern and may be combined with other patterns or quantified using qualifiers.

The operator "`,`", "`|`" or "`&`" used within the `(pattern)` container defines how the sub patterns are combined and different operators cannot be mixed at the same level.

Even when such a container is not required, it may often be used to improve the readability of a schema.

Example:

```
element name {
 text|(
  element first{text},
  element middle{text}?,
  element last{text}
 )}

element foo {
 element out {empty} &
 (
  element in1 {empty},
  element in2 {empty}
 )
}
```

Name

*-nameClass — Name class accepting any name.

Class:

nameClass

Synopsis

```
"*" [exceptNameClass]
```

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

anyName

Description:

The `anyName` name class matches any name from any namespace. This wild spectrum may be restricted by embedding `except` name classes.

The set of these names can be restricted using the optional `exceptNameClass` production.

Restrictions:

Within the scope of an element, the name classes of attributes cannot overlap. The same restriction applies to name classes of elements when these elements are combined by `interleave`.

Example:

```
foreign-elements = element * - (local:* | lib:* | hr:*) { anything }*
```

Name

-nameClass — Remove a name class from another

Class:

Synopsis

```
exceptNameClass     ::= "-" nameClass
```

May be included in:

`*-nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

except

Description:

The `except` name class is used to remove a name class from another.

Restrictions:

It is impossible to use `-nameClass` to produce empty name classes by including "anyName" in an "except" name class or "nsName" in an "except" name class included in another "nsName".

Example:

```
element hr:* - ( hr:author | hr:name | hr:born | hr:died ) { anything }
```

Name

-pattern — Remove a set of values from a `data`

Class:

Synopsis

```
exceptPattern       ::= "-" pattern
```

May be included in:

`datatypeName param exceptPattern`.

XML syntax equivalent:

except

Description:

The `except` pattern is used to remove a set of values from a "`datatypeName param exceptPattern`" pattern.

Restrictions:

The `-pattern` pattern can only be used in the context of data and can only contain `data`, `value` and `choice` elements.

Example:

```
attribute available {xs:boolean - (xs:boolean "false")}
```

Name

CName — Colonized names

Class:

Synopsis

```
CName                ::= NCName ":" NCName
```

May be included in:

`(nameClass)`, `attribute`, `datatypeName literal`, `datatypeName param exceptPattern`, `element`, `nameClass|nameClass`.

XML syntax equivalent:

none

Description:

The `CName` production describes colonized names (i.e. names containing a colon) as two non colonized names separated by a colon (`":"`).

Name

QuotedIdentifier — Quoted identifier

Class:

Synopsis

```
quotedIdentifier    ::= "\" NCName
```

May be included in:

`(pattern)`, `attribute`, `datatypes`, `default namespace`, `element`, `list`, `mixed`, `namespace`, `parent`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

none

Description:

The `QuotedIdentifier` production describes quoted identifiers, i.e. non colonized names preceded by a \. This is needed to allow names which are the same than the keywords of the compact syntax.

Name

Top level — Top level

Class:

Synopsis

```
topLevel            ::= decl* (pattern|grammarContent*)
```

May be included in:

XML syntax equivalent:

none

Description:

Start symbol for the Relax NG compact syntax EBNF. The `Top level` production describes the top level structure of a Relax NG compact syntax document composed of an optional declaration section and of the actual schema composed of either a single `pattern` or a more complete `grammarContent`.

Name

assignMethod — Define how to assign a content to `start` and named patterns.

Class:

Synopsis

```
assignMethod        ::= "="
                        |"|="
                        |"&="
```

May be included in:

`div`, `grammar`, `include`.

XML syntax equivalent:

none

Description:

The `assignMethod` how the content of `start` and named patterns are affected by a new definition. `assignMethod` which may take the values: "`=`" (definition), "`&=`" (combination by interleave) or "`|=`" (combination by choice).

Name

attribute — Pattern matching an attribute.

Class:

pattern

Synopsis

```
"attribute" nameClass "{" pattern "}"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

attribute

Description:

The `attribute` pattern matches an attribute. The name of the attribute is defined through a nameClass which may be either a single name or name class. Note unlike that, for the XML syntax, the content of an attribute is not defaulted to `text` and must always be explicitly defined.

Restrictions:

<itemizedList>
<listItem>

After simplification, attributes patterns can only contain patterns relevant for text nodes.
</listItem>
<listItem>

Attributes cannot be duplicated, either directly or through overlaping name classes.
</listItem>
<listItem>

Attributes which have an infinite name class (`anyName` or `nsName`) must be enclosed in a `oneOrMore` (or `zeroOrMore` before simplification) pattern.
</listItem>
</itemizedList>

Example:

```
attribute available { text }

attribute xml:lang { xsd:language }

attribute * - (local:* | lib:* | hr:*) { text }
```

Name

datatypeName — Datatype name

Class:

Synopsis

```
datatypeName        ::= CName
                        |"string"
                        |"token"
```

May be included in:

`datatypeName literal`, `datatypeName param exceptPattern`.

XML syntax equivalent:

none

Description:

The `datatypeName` production defines what is a valid datatype name. `CName` (Colonized names) must be used for any datatype library except for the built in type library which has only two datatypes (`string` and `token`).

Name

datatypeName literal — Match a text node and a value

Class:

pattern

Synopsis

```
[datatypeName] literal
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

value

Description:

The `datatypeName literal` pattern matches a text node against a value using the semantic of a specified datatype to perform the comparison.

When datatypeName is omitted, the default datatype (which is the token datatype from the Relax NG built in library) is used.

Restrictions:

The `datatypeName literal` pattern is meant for data oriented applications and can't be used in mixed content models.

Example:

```
"0"

xs:integer "0"

xs:boolean "false"

attribute available {xs:boolean "true"}
```

Name

datatypeName param exceptPattern — `data` pattern

Class:

pattern

Synopsis

`datatypeName` `["{"` *`param*`* `"}"] [`*`exceptPattern`*`]`

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

data

Description:

The `datatypeName param exceptPattern` pattern matches a single text node and gives the possibility to restrict its values. This is different from the `text` pattern which matches zero or more text nodes and doesn't give any possibility to restrict the values of these text nodes.

In this construction, the restrictions are applied through `datatypeName` which defines the datatype, the optional `param` which define additional parameters passed to the datatype library (when the datatype library is W3C XML Schema datatypes, these parameters are the W3C XML Schema facets) and the optional `exceptPattern` which defines exceptions, i.e. a set of values which are excluded by the `exceptPattern`.

Restrictions:

The `datatypeName param exceptPattern` pattern is meant for data oriented applications and can't be used in mixed content models.

Example:

```
attribute available {xs:boolean - (xs:boolean "false")}

 element born {xs:date {
   minInclusive = "1900-01-01"
   maxInclusive = "2099-12-31"
   pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
 }}
```

Name

datatypes — Namespace declaration (to identify datatype libraries)

Class:

decl

Synopsis

```
"datatypes" identifierOrKeyword "=" literal
```

May be included in:

XML syntax equivalent:

xmlns:

Description:

The `datatypes` declaration assigns a prefix to a datatype library for the compact syntax like xmlns:xxx attributes in XML. Note that unlike XML namespace declarations, declarations for the Relax NG compact syntax in general (and `datatypes` declarations in particular) are global to a schema and cannot be redefined. The prefix `xsd` is predefined and bound to "http://www.w3.org/2001/XMLSchema-datatypes".

Example:

```
datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"
```

Name

decl — Declarations

Class:

Synopsis

```
decl                ::= "namespace" identifierOrKeyword "=" namespaceURILiteral
                        |"default" "namespace" [identifierOrKeyword] "=" namespa
                        |"datatypes" identifierOrKeyword "=" literal
```

May be included in:

XML syntax equivalent:

none

Description:

Declarations section of a Relax NG compact syntax schema. These declarations are global and common to the whole schema and include the namespace and datatype libraries declarations.

Name

default namespace — Default namespace declaration

Class:

decl

Synopsis

```
"default" "namespace" [identifierOrKeyword] "=" namespaceURILiteral
```

May be included in:

XML syntax equivalent:

xmlns

Description:

The `default namespace` declaration defines the default namespace for the compact syntax like `xmlns` attributes in XML. An optional prefix may be assigned to the default namespace which may then be explicitly referenced. Note that unlike XML default namespace declarations, declarations for the Relax NG compact syntax in general (and `default namespace` declarations in particular) are global to a schema and cannot be redefined. A prefix can be assigned to the lack of namespace using the value `""`.

Example:

```
default namespace = "http://eric.van-der-vlist.com/ns/library"
default namespace local = ""
```

Name

div — Division (in the context of a `grammar`)

Class:

grammarContent

Synopsis

```
"div" "{" grammarContent* "}"
```

May be included in:

`div`, `grammar`.

XML syntax equivalent:

div

Description:

The `div` element is provided to define logical divisions in Relax NG schemas. It has no effect on the validation and its purpose is to define a group of definitions within a `grammar` which may be annotated as a whole.

In the context of a `grammar`, the content of a `div` element is the same than the content of a `grammar` (this means that `div` elements may be embedded in other `div` elements.

Example:

```
[
  xhtml:p [
    "The content of the book element has been split in two named patterns:"
  ]
]
div {
  book-start =
    attribute id { text },
    isbn-element,
    title-element,
    author-element*
  book-end =
    author-element*,
    character-element*,
    attribute available { text }
}
```

Name

element — Pattern matching an element

Class:

pattern

Synopsis

```
"element" nameClass "{" pattern "}"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName  param  exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

element

Description:

The `element` pattern matches an element. The name of the element is defined through a nameClass which may be either a single name or name class.

Example:

```
element isbn { text }

element hr:born { text }

element title { attribute xml:lang { text }, text }

element * - (local:* | lib:* | hr:*) { anything }
```

Name

empty — Empty content

Class:

pattern

Synopsis

```
"empty"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

empty

Description:

The `empty` patterns is used to define elements which are empty, i.e. which have no children elements, text nor attributes. Note that it is mandatory to use this pattern in such case (`element foo{}` is not forbidden) and that there is no such thing as empty attributes (an attribute such as `foo=""` is considered as having a value which is the empty string rather than be considered as being empty, i.e. having no value).

Example:

```
element foo {
 element out {empty} &
 (
  element in1 {empty},
  element in2 {empty}
 )
}
```

Name

external — Reference to an external schema

Class:

pattern

Synopsis

```
"external" literal [inherit]
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

externalRef

Description:

The `external` pattern is a reference to an external schema. This has the same effect than replacing the `external` pattern by the external schema considered as a pattern.

Example:

```
element university { element name { text }, external "flat.rnc" }

element book { external "book.rnc" }
```

Name

grammar — Grammar pattern

Class:

pattern

Synopsis

```
"grammar" "{" grammarContent* "}"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName  param  exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

grammar

Description:

The `grammar` pattern encapsulates the definitions of `start` and named patterns.

The most common use of `grammar` is to validate XML documents and in this case the `start` pattern defines which elements may be used as the document root element. The `grammar` pattern may also be used as a way to write modular schemas and in this case the `start` pattern defines which nodes must be matched by the `grammar` at the location where it appears in the schema.

In every case, the named patterns defined in a `grammar` are considered to be local to this `grammar`.

Example:

```
grammar {

 author-element= element author {
  attribute id {text},
  name-element,
  born-element,
  died-element?
 }

 book-element = element book {
  attribute id {text},
  attribute available {text},
  isbn-element,
  title-element,
  author-element *,
  character-element*
 }

 born-element = element born {text}

 character-element =  element character {
```

```
  attribute id {text},
  name-element,
  born-element,
  qualification-element
 }

 died-element = element died {text}

 isbn-element = element isbn {text}

 name-element = element name {text}

 qualification-element = element qualification {text}

 title-element = element title {attribute xml:lang {text}, text}

 start = element library {
  book-element +
 }
}


author-element =
  grammar
  {
     start =
        element author
        {
           attribute id { text },
           name-element,
           born-element,
           died-element?
        }
     name-element = element name { text }
     born-element = element born { text }
     died-element = element died { text }
  }
```

Name

grammarContent — Content of a grammar

Class:

Synopsis

```
grammarContent      ::= start
                        |define
                        |"div" "{" grammarContent* "}"
                        |"include" literal [inherit] ["{" includeContent* "}"]
```

May be included in:

`div`, `grammar`.

XML syntax equivalent:

none

Description:

The `grammarContent` production defines the content of a `grammar`.

Name

identifier — Identifier

Class:

Synopsis

```
identifier          ::= (NCName - keyword)
                        |quotedIdentifier
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `datatypes`, `default namespace`, `div`, `element`, `external`, `grammar`, `include`, `list`, `mixed`, `namespace`, `parent`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

none

Description:

The `identifier` production describes valid `identifier`s for the compact syntax, i.e. either quoted identifiers or the non colonized names which are not keywords.

Name

identifier assignMethod pattern — Named pattern definition

Class:

Synopsis

```
define              ::= identifier assignMethod pattern
```

May be included in:

`div`, `grammar`, `include`.

XML syntax equivalent:

define

Description:

When `identifier assignMethod pattern` is embedded in a grammar, it defines a named pattern or combines a new definition with an existing one. Named pattern are global to a `grammar` and can be referenced by `ref` in the scope of their `grammar` and by `parentRef` in the scope of the grammars directly embedded in their `grammar`.

When `identifier assignMethod pattern` is embedded in `include`, the new definition is a redefinition and replaces the definitions from the included grammar unless a `combine` attribute is specified in which case the definitions are combined.

The combination is defined through the `assignMethod` which may take the values: "`=`" (definition), "`&=`" (combination by interleave) or "`|=`" (combination by choice).

Restrictions:

Named patterns are always global and apply only to patterns and it is not possible to define and make reference to non patterns such as class names or datatype parameters.

Example:

```
date-element = element born { xsd:data }

date-element |= element died { xsd:date }
```

Name

identifierOrKeyword — Identifier or keyword

Class:

Synopsis

```
identifierOrKeyword ::= identifier
                        |keyword
```

May be included in:

`(nameClass)`, `attribute`, `datatypeName param exceptPattern`, `datatypes`, `default namespace`, `element`, `external`, `include`, `nameClass|nameClass`, `namespace`.

XML syntax equivalent:

none

Description:

The `identifierOrKeyword` production either a valid `identifier` or a `keyword`.

Name

include — Grammar merge

Class:

grammarContent

Synopsis

```
"include" literal [inherit] ["{" includeContent* "}"]
```

May be included in:

`div`, `grammar`.

XML syntax equivalent:

include

Description:

The `include` pattern includes a grammar and merges its definitions with the definitions of the current grammar. The definitions of the included grammar may be redefined and overridden by the definitions embedded in the `include` pattern. Note that a schema must contain an explicit `grammar` definition in order to be included.

The optional `inherit` production defines which namespaces are inherited from the included schema and `includeContent` allows to redefine definitions from the included schema.

Example:

```
include "included.rnc"

include "flat.rnc" { start = book-element }
```

Name

includeContent — Content of an `include` pattern.

Class:

Synopsis

```
includeContent      ::= define
                        |start
                        |"div" "{" includeContent* "}"
```

May be included in:

`include`.

XML syntax equivalent:

none

Description:

The `includeContent` production defines the content of an `include`. The only difference with `grammarContent` is that `includeContent` doesn't allow embedded `include`.

Name

inherit — Namespace inheritance

Class:

Synopsis

```
inherit             ::= "inherit" "=" identifierOrKeyword
```

May be included in:

`external`, `include`.

XML syntax equivalent:

none

Description:

The `inherit` production is used in `external` and `include` statements to specify the prefixes of the namespaces which are inherited by the included file.

Name

keyword — Keywords

Class:

Synopsis

```
keyword             ::= "attribute"
                      |"default"
                      |"datatypes"
                      |"div"
                      |"element"
                      |"empty"
                      |"external"
                      |"grammar"
                      |"include"
                      |"inherit"
                      |"list"
                      |"mixed"
                      |"namespace"
                      |"notAllowed"
                      |"parent"
                      |"start"
                      |"string"
                      |"text"
                      |"token"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName` `param` `exceptPattern`, `datatypes`, `default` `namespace`, `element`, `external`, `include`, `list`, `mixed`, `namespace`, `parent`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

none

Description:

The `keyword` production gives the list of keywords for the Relax NG compact syntax. Note that these keywords are reserved only when there is a risk of confusion and that they can be used, for instance, as element or attribute names without being quoted. When they are reserved, they can still be used as identifiers but need to be quoted.

Name

list — Text node split

Class:

pattern

Synopsis

```
"list" "{" pattern "}"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

list

Description:

The `list` pattern splits a text node into tokens separated by white spaces to allow the validation of these tokens separately. This is most useful for validating lists of values.

Restrictions:

<itemizedList>
<listItem>

`interleave` cannot be used within `list`.
</listItem>
<listItem>

The content of a `list` is only about data: it's forbidden to define `element`, `attribute` or `text` there.
</listItem>
<listItem>

It's forbidden to embed `list` into `list`.
</listItem>
</itemizedList>

Example:

```
attribute see-also {list {token*}}

attribute dimensions {list {xsd:decimal, xsd:decimal, xsd:decimal, ("inches"|"
```

Name

literal — Literal

Class:

Synopsis

```
literal             ::= literalSegment ("~" literalSegment)+
```

May be included in:

`datatypeName` `literal`, `datatypeName` `param` `exceptPattern`, `datatypes`, `default` `namespace`, `external`, `include`, `namespace`.

XML syntax equivalent:

none

Description:

The `literal` production describes literals as several segments of literals contained by the "~" sign.

Name

literalSegment — Literal segment

Class:

Synopsis

```
literalSegment      ::= """ (Char - (""" newline))* """
                        |"'" (Char - ("'" newline))* "'"
                        |"""" (["""] ["""] (Char - """))* """"
                        |"''" (["'"] ["'"] (Char - "'"))* "''"
```

May be included in:

`datatypeName literal`, `datatypeName param exceptPattern`, `datatypes`, `default namespace`, `external`, `include`, `namespace`.

XML syntax equivalent:

none

Description:

The `literalSegment` production describes literal segments as strings enclosed either in simple or double quotes or enclosed in three double or single quotes using a pythonic syntax.

Name

mixed — Pattern for mixed content models

Class:

pattern

Synopsis

```
"mixed" "{" pattern "}"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName  param  exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

mixed

Description:

The `mixed` pattern is a shortcut for `interleave` with an embedded `text` pattern. It describes unordered content models where a text node may be included before and after each element. Note that Relax NG does not allow to add constraints on these text nodes.

Restrictions:

The limitations of `interleave` apply here:
<itemizedList>
<listItem>

The `mixed` pattern cannot be used within a `list`.
</listItem>
<listItem>

Elements within a `mixed` pattern cannot have overlapping name classes.
</listItem>
<listItem>

There must no other "text" pattern in each set of patterns combined by `mixed`
</listItem>
</itemizedList>

Example:

```
 element title {
  mixed {
   attribute xml:lang {text}&
   element a {attribute href {text}, text} *
  }
 }
```

is equivalent to:

```
 element title {
  ( text & (
    attribute xml:lang {text}&
    element a {attribute href {text}, text} *
  )
 }
```

which itself is equivalent to:

```
 element title {
    text &
    attribute xml:lang {text}&
    element a {attribute href {text}, text} *
 }
```

Name

name — Define a set of names that must be matched by an element or attribute.

Class:

Synopsis

```
name                ::= identifierOrKeyword
                        |CName
```

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

none

Description:

The `name` name class defines sets of names which are singletons i.e. that match only one name. There is no other restriction than those of XML 1.0 and namespaces in XML 1.0 on such names and they can be either `CName` or `identifierOrKeyword` (in particular, even `keyword`s can be used as `name`s).

Name

nameClass — Define a set of names that must be matched by an element or attribute.

Class:

Synopsis

```
nameClass           ::= name
                        |NCName ":*" [exceptNameClass]
                        |"*" [exceptNameClass]
                        |nameClass "|" nameClass
                        |"(" nameClass ")"
```

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

none

Description:

The `nameClass` production defines sets of names that must be matched by elements and attributes. Its simplest expression is to define a single `name` but specific wildcards can also be expressed as `nameClass`.

Name

nameClass|nameClass — Choice between name classes

Class:

nameClass

Synopsis

nameClass "|" *nameClass*

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

choice

Description:

The `nameClass|nameClass` production performs a choice between two name classes: a name will match `nameClass|nameClass` if and only if it matches at least one of the two alternatives.

Example:

```
element lib:* | hr:* { anything }
```

Name

namespace — Namespace declaration

Class:

decl

Synopsis

```
"namespace" identifierOrKeyword "=" namespaceURILiteral
```

May be included in:

XML syntax equivalent:

xmlns:

Description:

The `namespace` declaration defines namespace prefixes for the compact syntax like xmlns:xxx attributes in XML. Note that unlike XML namespace declarations, declarations for the Relax NG compact syntax in general (and `namespace` declarations in particular) are global to a schema and cannot be redefined. A prefix can be assigned to the lack of namespace using the value `""`. The `xml` prefix is predefined.

Example:

```
namespace hr = "http://eric.van-der-vlist.com/ns/person"
namespace local = ""
```

Name

namespaceURILiteral — Namespace URI Literal

Class:

Synopsis

```
namespaceURILiteral ::= literal
                        |"inherit"
```

May be included in:

`default namespace`, `namespace`.

XML syntax equivalent:

none

Description:

The `namespaceURILiteral` production is used to specify a namespace URI and can either be a `literal` or the value "`inherit` to specify that the namespace URI is inherited from the including file.

Name

notAllowed — Not allowed

Class:

pattern

Synopsis

```
"notAllowed"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

notAllowed

Description:

The `notAllowed` pattern always fails. It can be used to provide abstract definitions which must be overridden before they can be used in a schema.

Example:

```
isbn-element |= notAllowed
```

Name

nsName exceptNameClass — Name class for any name in a namespace

Class:

nameClass

Synopsis

NCName ":*" [*exceptNameClass*]

May be included in:

`(nameClass)`, `*-nameClass`, `attribute`, `element`, `nameClass|nameClass`, `nsName exceptNameClass`.

XML syntax equivalent:

nsName

Description:

The `nsName exceptNameClass` name class allows any name in a specific namespace.

The namespace is defined by the `nsName` production and the set of these names can be restricted using the `exceptNameClass` production.

Restrictions:

Within the scope of an element, the name classes of attributes cannot overlap. The same restriction applies to name classes of elements when these elements are combined by `interleave`. It is impossible to use `nsName exceptNameClass` to produce empty name classes by including `nsName exceptNameClass` in an `except` name class included in another `nsName`.

Example:

```
element lib:* { anything }

element hr:* - ( hr:author | hr:name | hr:born | hr:died ) { anything }
```

Name

param — Datatype parameter

Class:

Synopsis

```
param                ::= identifierOrKeyword "=" literal
```

May be included in:

`datatypeName param exceptPattern`.

XML syntax equivalent:

param

Description:

The `param` production defines parameters passed to the datatype library to determine if a value is valid per a datatype. When the datatype library is the W3C XML Schema datatypes, these parameters are the facets of the datatype and they define additional restrictions to be applied. The name of the parameter is defined by `identifierOrKeyword` and its value defined by `literalparam`.

Example:

```
element born {xs:date {
  minInclusive = "1900-01-01"
  maxInclusive = "2099-12-31"
  pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}"
}}
```

Name

parent — Reference to a named pattern from the parent grammar

Class:

pattern

Synopsis

```
"parent" identifier
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

parentRef

Description:

The `parent` pattern is a reference to a named pattern belonging to the parent `grammar`, i.e. the `grammar` in which the current `grammar` is included. The scope of named pattern is usually limited to the `grammar` in which they are defined and the `parent` pattern provides a way to extend this scope and refer named pattern defined in the parent `grammar`.

Example:

```
 born-element = parent born-element

start =
   attribute id { parent id-content },
   attribute available { parent available-content },
   element isbn { parent isbn-content },
   element title { parent title-content },
   element author { parent author-content }*,
   element character { parent character-content }*
```

Name

pattern — Pattern

Class:

Synopsis

```
pattern             ::= "element" nameClass "{" pattern "}"
                        |"attribute" nameClass "{" pattern "}"
                        |pattern ("," pattern)+
                        |pattern ("&" pattern)+
                        |pattern ("|" pattern)+
                        |pattern "?"
                        |pattern "*"
                        |pattern "+"
                        |"list" "{" pattern "}"
                        |"mixed" "{" pattern "}"
                        |identifier
                        |"parent" identifier
                        |"empty"
                        |"text"
                        |[datatypeName] literal
                        |datatypeName ["{" param* "}"] [exceptPattern]
                        |"notAllowed"
                        |"external" literal [inherit]
                        |"grammar" "{" grammarContent* "}"
                        |"(" pattern ")"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `div`, `element`, `grammar`, `include`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

none

Description:

A pattern is an atom of Relax NG schema which is matched against nodes from the instance document (elements, attributes, text nodes or token resulting of a split through `list`).

Name

pattern&pattern — `interleave` Pattern

Class:

pattern

Synopsis

pattern ("&" *pattern*)+

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

interleave

Description:

The `interleave` pattern "interleaves" sub patterns, i.e. allows their leaves to be mixed in any relative order.

`interleave` is more than defining unordered groups as we can see on the following example: consider element "a" and the ordered group of element "b1" and "b2". An unordered group of these two patterns would only allow element "a" followed by elements "b1" and "b2" or elements "b1" and "b2 followed by element "a". An `interleave` of these two patterns does allow these two combinations but also element "b1" followed by "a" followed by "b2", i.e. a combination where the element "a" has been "interleaved" between elements "b1" and "b2".

The `interleave` behavior is the behavior applied to `attribute` patterns even when they are embedded in (ordered) `group` patterns (the reason for this is that XML 1.0 specifies that the relative order of attributes is not significant).

Another case where `interleave` patterns are often needed is to described mixed content models, i.e. content models where `text` are interleaved between elements. A shortcut (the `mixed` pattern) has been defined for this case.

Any number of patterns may be combine through the & operator using this construct, but one should note that different operators (`,`, `|` and &) cannot be mixed at the same level.

Restrictions:

<itemizedList>
<listItem>

The `pattern&pattern` pattern cannot be used within a `list`.
</listItem>
<listItem>

Elements within a `pattern&pattern` pattern cannot have overlapping name classes.
</listItem>
<listItem>

There must be at most one "text" pattern in each set of patterns combined by `pattern&pattern`
</listItem>
</itemizedList>

Example:

```
  element character {
   attribute id {text}&
   element name {text}&
   element born {text}&
   element qualification {text}}

element foo {
 element out {empty} &
 (
  element in1 {empty},
  element in2 {empty}
 )
}
```

Name

pattern* — `zeroOrMore` pattern

Class:

pattern

Synopsis

pattern "*"

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

zeroOrMore

Description:

A pattern qualified as `zeroOrMore` must be matched zero or more times (i.e. any number of times).

Restrictions:

The `pattern*` pattern cannot contain attribute definitions.

Example:

```
 element author {
  attribute id {text},
  element name {text},
  element born {text},
  element died {text}?}*

book-element = element book {
 attribute id {text},
 attribute available {text},
 isbn-element,
 title-element,
 author-element *,
 character-element*
}
```

Name

pattern+ — `oneOrMore` pattern

Class:

pattern

Synopsis

pattern "+"

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

oneOrMore

Description:

A pattern qualified as `oneOrMore` must be matched one or more times.

Restrictions:

The `pattern+` pattern cannot contain attribute definitions.

Example:

```
start = element library {
 book-element +
}

 attribute see-also {list {("0836217462"|"0345442695"|"0449220230"|"044921404
```

Name

pattern,pattern — `pattern,pattern` pattern

Class:

pattern

Synopsis

pattern ("," *pattern*)+

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

group

Description:

The `group` pattern defines an ordered group of sub patterns (note that when `attribute` patterns are included in such a group, their order cannot be guaranteed). Any number of patterns may be combine through the `,` operator using this construct, but one should note that different operators (`,`, `|` and `&`) cannot be mixed at the same level.

Example:

```
element author {
 attribute id {text},
 element name {text},
 element born {text},
 element died {text}?}*

element lib:title { attribute xml:lang { text }, text }

attribute dimensions {list {token, token, token, ("inches"|"cm"|"mm")}}
```

Name

pattern? — `optional` pattern

Class:

pattern

Synopsis

pattern "?"

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

optional

Description:

A pattern qualified as `optional` is optional, i.e. must be matched zero or one time.

Example:

```
  element died {text}?

attribute see-also {list {token, token?, token?, token?}}
```

Name

pattern|pattern — `choice` pattern

Class:

pattern

Synopsis

pattern ("|" *pattern*)+

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

choice

Description:

The `choice` pattern defines a choice between different patterns: it matches a node if and only if at least one of its sub-pattern matches this node.

Any number of patterns may be combine through the `|` operator using this construct, but one should note that different operators (`,` , `|` and &) cannot be mixed at the same level.

Example:

```
element name {
 text|(
  element first{text},
  element middle{text}?,
  element last{text}
 )}

  attribute available {"true"|"false"|"who knows?"}
```

Name

start — Start of a grammar

Class:

Synopsis

```
start                ::= "start" assignMethod pattern
```

May be included in:

`div`, `grammar`, `include`.

XML syntax equivalent:

start

Description:

The `start` pattern defines the "start" of a `grammar`. When this grammar is used to validate a complete document, the `start` pattern defines which elements may be used as the document (root) element. When this grammar is embedded in another `grammar`, the `start` pattern describes which pattern should be applied at the location where the `grammar` is embedded. Like named pattern definitions, start patterns may be combined by `choice` or `interleave` and redefined when they are included in `include` patterns.

The combination is defined through the `assignMethod` which may take the values: "`=`" (definition), "`&=`" (combination by interleave) or "`|=`" (combination by choice).

Example:

```
start = element library {
 book-element +
}

start |= book-element
```

Name

text — Pattern matching text nodes

Class:

pattern

Synopsis

```
"text"
```

May be included in:

`(pattern)`, `attribute`, `datatypeName param exceptPattern`, `element`, `list`, `mixed`, `pattern&pattern`, `pattern*`, `pattern+`, `pattern,pattern`, `pattern?`, `pattern|pattern`.

XML syntax equivalent:

text

Description:

The `text` pattern matches zero or more text nodes. The fact that a `text` pattern matches more than one text node has no effect when it is used in ordered content models (the data model used by Relax NG for XML documents is similar to the data model of XPath 1.0 and two text nodes cannot be adjacent) but makes a difference when a `text` pattern is used in `interleave`:adding a single `text` pattern in an `interleave` pattern has the effect of allowing any number of text nodes which can interleave before and after each element (note that the `mixed` pattern is provided as a shortcut to define these content models).

Restrictions:

No more than one `text` pattern can be included in an `interleave` pattern.

Example:

```
element author {
 attribute id {text},
 element name {text},
 element born {text},
 element died {text}?}?
```

Chapter 20. Datatype Reference Guide

This chapter provides a quick reference to all of the datatypes W3C XML Schema defines. Each datatype is listed with the list of its Relax NG datatype parameters (this list correspond to the list of W3C XML Schema facets available for the datatype with the exception of the `whiteSpace` facet which is not supported by Relax NG), as well as information about what it represents and how. For the so called "secondary datatypes" (i.e. the W3C XML Schema builtin types which are derived from another builtin type), the synopsis shows the formal definition of the type using W3C XML Schema syntax. Examples are given for all these datatypes.

Name

xsd:anyURI — URI (Uniform Resource Identifier).

Derived from:

xsd:anySimpleType

Primary:

xsd:anyURI

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="anyURI" id="anyURI">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction></xsd:simpleType>
```

Description

This datatype corresponds normatively to the XLink `href` attribute. Its value space includes the URIs defined by the RFCs 2396 and 2732, but its lexical space doesn't require the character escapes needed to include non-ASCII characters in URIs.

Restrictions

Relative URIs are not "absolutized" by W3C XML Schema. A pattern defined as:

```
<data type="xsd:anyURI">
  <choice">
    <value type="xsd:anyURI">http://www.w3.org/TR/xmlschema-0/</value>
    <value type="xsd:anyURI">http://www.w3.org/TR/xmlschema-1/</value>
    <value type="xsd:anyURI">http://www.w3.org/TR/xmlschema-2/</value>
  </choice>
</data>
```

should not match the `href` attribute in this instance element:

```
<a xml:base="http://www.w3.org/TR/" href="xmlschema-1/">
  XML Schema Part 2: Datatypes
</a>
```

The Recommendation states that "it is impractical for processors to check that a value is a context-appropriate URI reference," freeing schema processors from having to validate the correctness of the URI.

Example

```
<define name="httpURI">
  <data type="xsd:anyURI">
    <param name="pattern">http://.*<param>
  </data>
</define>
```

Name

xsd:base64Binary — Binary content coded as "base64".

Derived from:

xsd:anySimpleType

Primary:

xsd:base64Binary

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="base64Binary" id="base64Binary">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:base64Binary` is the set of arbitrary binary contents. Its lexical space is the same set after base64 coding. This coding is described in Section 6.8 of RFC 2045.

Restrictions

RFC 2045 has been defined to transfer binary contents over text-based mail systems. It imposes a line break at least every 76 characters to avoid the inclusion of arbitrary line breaks by the mail systems. Sending base64 content without line breaks is nevertheless a common usage for applications such as SOAP and the W3C XML Schema Working Group. After a request from other W3C Working Groups, the W3C XML Schema Working Group decided to remove the obligation to include these line breaks from the constraints on the lexical space. (This decision was made after the publication of the W3C XML Schema Recommendation and has been included in a release of the errata.)

Example

```
<define name="picture">
  <attribute name="type">
    <ref name="graphicalFormat"/>
  </attribute>
  <data type="xsd:base64Binary">
</define>
```

Name

xsd:boolean — Boolean (true or false).

Derived from:

xsd:anySimpleType

Primary:

xsd:boolean

Known subtypes:

none

Data parameters (facets):

`pattern`.

Synopsis

```
<xsd:simpleType name="boolean" id="boolean">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:boolean` is "true" and "false," and its lexical space accepts true, false, and also "1" (for true) and "0" (for false).

Restrictions

This datatype cannot be localized—for instance, to accept "vrai" and "faux" instead of "true" and "false".

Example

```
<book id="b0836217462" available="true"/>
```

Name

xsd:byte — Signed value of 8 bits.

Derived from:

xsd:short

Primary:

xsd:decimal

Known subtypes:

none

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="byte" id="byte">
  <xsd:restriction base="xsd:short">
    <xsd:minInclusive value="-128"/>
    <xsd:maxInclusive value="127"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:byte` is the integers between -128 and 127, i.e., the signed values that can fit in a word of 8 bits. Its lexical space allows an optional sign and leading zeros before the significant digits.

Restrictions

The lexical space does not allow values expressed in other numeration bases (such as hexadecimal, octal, or binary).

Example

Valid values for byte include 27, -34, +105, and 0.

Invalid values include 0A, 1524, and INF.

Name

xsd:date — Gregorian calendar date.

Derived from:

xsd:anySimpleType

Primary:

xsd:date

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="date" id="date">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

This datatype is modeled after the calendar dates defined in Chapter 5.2.1 of ISO 8601. Its value space is the set of Gregorian calendar dates as defined by this standard; i.e., a one-day-long period of time. Its lexical space is the ISO 8601 extended format "[-]CCYY-MM-DD[Z|(+|-)hh:mm]" with an optional timezone. Timezones that are not specified are considered "undetermined."

Restrictions

The basic format of ISO 8601 calendar dates "CCYYMMDD" is not supported.

The other forms of dates available in ISO 8601—ordinal dates defined by the year and the number of the day in the year and dates identified by calendar week and day numbers—are not supported.

As the value space is defined by reference to ISO 8601, there is no support for any calendar system other than Gregorian.

As the lexical space is also defined as reference to ISO 8601, there is no support for any localization such as different orders for date parts or named months.

The order relation between dates with and without timezone is partial: they can be compared only outside of a +/- 14 hours interval.

There is a dissension between ISO 8601 which defines a day as a period of time of 24 hours, and W3C XML Schema, which indicates that a date is a "one-day long, non-periodic instance . . . independent of how many hours this day has." Even though technically right (some days do not last exactly 24 hours

because of leap seconds), this definition is not coherent with the definition of `xsd:duration` for which a day is always exactly 24 hours long.

Example

Valid values include: `"2001-10-26"`, `"2001-10-26+02:00"`, `"2001-10-26Z"`, `"2001-10-26+00:00"`, `"-2001-10-26"`, or `"-20000-04-01"`.

The following values would be invalid: `"2001-10"` (all the parts must be specified), `"2001-10-32"` (the days part (32) is out of range), `"2001-13-26+02:00"` (the month part (13) is out of range), or `"01-10-26"` (the century part is missing).

Name

xsd:dateTime — Instant of time (Gregorian calendar).

Derived from:

xsd:anySimpleType

Primary:

xsd:dateTime

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="dateTime" id="dateTime">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

This datatype describes instants identified by the combination of a date and a time. Its value space is described as a "combination of date and time of day" in Chapter 5.4 of ISO 8601. Its lexical space is the extended format "[-]CCYY-MM-DDThh:mm:ss[Z|(+|-)hh:mm]". The timezone may be specified as "Z" (UTC) or "(+|-)hh:mm." Timezones that are not specified are considered "undetermined."

Restrictions

The basic format of ISO 8601 calendar datetimes "CCYYMMDDThhmmss" is not supported.

The other forms of datetimes available in ISO 8601—ordinal dates defined by the year and the number of the day in the year and dates identified by calendar week and day numbers—are not supported.

As the value space is defined by reference to ISO 8601, there is no support for any calendar system other than Gregorian.

As the lexical space is also defined as reference to ISO 8601, there is no support for any localization such as different orders for date parts or named months.

The order relation between datetimes with and without timezone is partial: they can be compared only outside of a +/- 14 hours interval.

Example

Valid values for `xsd:dateTime` include: `"2001-10-26T21:32:52"`, `"2001-10-26T21:32:52+02:00"`, `"2001-10-26T19:32:52Z"`,

`"2001-10-26T19:32:52+00:00"`, `"-2001-10-26T21:32:52"`, or `"2001-10-26T21:32:52.12679"`.

The following values would be invalid: `"2001-10-26"` (all the parts must be specified), `"2001-10-26T21:32"` (all the parts must be specified), `"2001-10-26T25:32:52+02:00"` (the hours part (25) is out of range), or `"01-10-26T21:32"` (all the parts must be specified).

Name

xsd:decimal — Decimal numbers.

Derived from:

xsd:anySimpleType

Primary:

xsd:decimal

Known subtypes:

xsd:integer

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="decimal" id="decimal">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

`xsd:decimal` is the datatype that represents the set of all the decimal numbers with arbitrary lengths. Its lexical space allows any number of insignificant leading and trailing zeros (after the decimal point).

Restrictions

The decimal separator is always a point (".") and no thousand separator may be added. There is no support for scientific notations.

Example

Valid values include: `"123.456"`, `"+1234.456"`, `"-1234.456"`, `"-.456"`, or `"-456"`.

The following values would be invalid: `"1  234.456"` (spaces are forbidden), `"1234.456E+2"` (scientific notation (`"E+2"`) is forbidden), `"+  1234.456"` (spaces are forbidden), or `"+1,234.456"` (delimiters between thousands are forbidden).

Name

xsd:double — IEEE 64 bit floating point.

Derived from:

xsd:anySimpleType

Primary:

xsd:double

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="double" id="double">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:double` is "double" (64 bits) floating-point numbers as defined by the IEEE. The lexical space uses a decimal format with optional scientific notation. The match between lexical (powers of 10) and value (powers of 2) spaces is approximate and done on the closest value.

This datatype differentiates positive (0) and negative (-0) zeros, and includes the special values "-INF" (negative infinity), "INF" (positive infinity) and "NaN" (Not a Number).

Note that the lexical spaces of `xsd:float` and `xsd:double` are exactly the same; the only difference is the precision used to convert the values in the value space.

Restrictions

The decimal separator is always a point (".") and no thousands separator may be used.

Examples

Valid values include: `"123.456"`, `"+1234.456"`, `"-1.2344e56"`, `"-.45E-6"`, `"INF"`, `"-INF"`, or `"NaN"`.

The following values would be invalid: `"1234.4E 56"` (spaces are forbidden), `"1E+2.5"` (the power of 10 must be an integer), `"+INF"` (positive infinity doesn't expect a sign), or `"NAN"` (capitalization matters in special values).

Name

xsd:duration — Time durations.

Derived from:

xsd:anySimpleType

Primary:

xsd:duration

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="duration" id="duration">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

Durations may be expressed using all the parts of a datetime (from year to fractions of second) and are, therefore, defined as a "six-dimensional space." Note that because the relation between some of these date parts (such as the number of days in a month) is not fixed, the order relationship between durations is only partial and the result of a comparison between two durations may be undetermined.

The lexical space of `xsd:duration` is the format defined by ISO 8601 under the form "PnYnMnDTnHnMnS," in which the capital letters are delimiters and can be omitted when the corresponding member is not used.

Although some durations are undetermined, this is fixed as soon as a starting point is fixed for the duration. W3C XML Schema relies on this feature to define the algorithm to use to compare two durations. Four datetimes have been chosen, which produce the greatest deviations when durations are added. A duration will be considered bigger than another when the result of its addition to these four dates is consistently bigger than the result of the addition of the other duration to these same four datetimes. These datetimes are: "1696-09-01T00:00:00Z", "1697-02-01T00:00:00Z," "1903-03-01T00:00:00Z," and "1903-07-01T00:00:00Z."

Restrictions

The lexical space cannot be customized.

Example

Valid values include `"PT1004199059S"`, `"PT130S"`, `"PT2M10S"`, `"P1DT2S"`, `"-P1Y"`, or `"P1Y2M3DT5H20M30.123S"`.

The following values would be invalid: `"1Y"` (leading `"P"` is missing), `"P1S"` (`"T"` separator is missing), `"P-1Y"` (all parts must be positive), `"P1M2Y"` (parts order is significant and `Y` must precede `M`), or `"P1Y-1M"` (all parts must be positive).

Name

xsd:ENTITIES — Whitespace separated list of unparsed entity references.

Derived from:

xsd:ENTITY

Primary:

none

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`.

Synopsis

```
<xsd:simpleType name="ENTITIES" id="ENTITIES">
  <xsd:restriction>
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:restriction base="xsd:ENTITY"/>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

`xsd:ENTITIES` is derived by a list from `xsd:ENTITY`. It represents lists of unparsed entity references. Each part of this entity reference is a nonqualified name (`xsd:NCName`) and must be declared as an unparsed entity in an internal or external DTD.

Restrictions

Unparsed entities have been defined in XML 1.0 as a way to include non-XML content in a XML document, but most of the applications prefer to define links (such as those defined in (X)HTML to include images or other multimedia objects).

W3C XML Schema does not provide alternative ways to declare unparsed entities. A DTD is needed to do so.

Name

xsd:ENTITY — Reference to an unparsed entity.

Derived from:

xsd:NCName

Primary:

xsd:string

Known subtypes:

xsd:ENTITIES

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="ENTITY" id="ENTITY">
  <xsd:restriction base="xsd:NCName"/>
</xsd:simpleType>
```

Description

`xsd:ENTITY` is an entity reference, i.e., a nonqualified name (`xsd:NCName`) that has been declared as an unparsed entity in an internal or external DTD.

Restrictions

Unparsed entities are defined in XML 1.0 as a way to include non-XML content in XML document, but most of the applications prefer to define links (such as those defined in (X)HTML to include images or other multimedia objects).

W3C XML Schema does not provide alternative ways to declare unparsed entities. A DTD is needed to do so.

Name

xsd:float — IEEE 32 bit floating point.

Derived from:

xsd:anySimpleType

Primary:

xsd:float

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="float" id="float">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:float` is "float" (32 bits) floating-point numbers as defined by the IEEE. The lexical space uses a decimal format with optional scientific notation. The match between lexical (powers of 10) and value (powers of 2) spaces is approximate and is done on the closest value.

This datatype differentiates positive (0) and negative (-0) zeros, and includes the special values "-INF" (negative infinity), "INF" (positive infinity), and "NaN" (Not a Number).

Note that the lexical spaces of `xsd:float` and `xsd:double` are exactly the same; the only difference is the precision used to convert the values in the value space.

Restrictions

The decimal separator is always a point (".") and no thousands separator may be added.

Example

Valid values include: `"123.456"`, `"+1234.456"`, `"-1.2344e56"`, `"-.45E-6"`, `"INF"`, `"-INF"`, or `"NaN"`.

The following values would be invalid: `"1234.4E 56"` (spaces are forbidden), `"1E+2.5"` (the power of 10 must be an integer), `"+INF"` (positive infinity doesn't expect a sign), or `"NAN"` (capitalization matters in special values).

Name

xsd:gDay — Recurring period of time: monthly day.

Derived from:

xsd:anySimpleType

Primary:

xsd:gDay

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="gDay" id="gDay">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:gDay` is the periods of one calendar day recurring each calendar month (such as the third of the month); its lexical space follows the ISO 8601 syntax for such periods (i.e., "---DD") with an optional timezone.

When needed, days are reduced to fit in the length of the months, so `---31` would occur on the the 28th of February of nonleap years.

Restrictions

The period (one month) and the duration (one day) are fixed, and no calendars other than the Gregorian are supported.

Example

Valid values include `"---01"`, `"---01Z"`, `"---01+02:00"`, `"---01-04:00"`, `"---15"`, or `"---31"`.

The following values would be invalid: `"--30-"` (the format must be `"---DD"`), `"---35"` (the day is out of range), `"---5"` (all the digits must be supplied), or `"15"` (missing leading `"---"`).

Name

xsd:gMonth — Recurring period of time: yearly month.

Derived from:

xsd:anySimpleType

Primary:

xsd:gMonth

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="gMonth" id="gMonth">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:gMonth` is the period of one calendar month recurring each calendar year (such as the month of April); its lexical space should follow the ISO 8601 syntax for such periods (i.e., " -- MM") with an optional timezone. However, there is a typo in the W3C XML Schema Recommendation where the format is defined as " -- MM -- -- . Even though an erratum should be published to bring the W3C XML Schema inline with ISO 8601, most of the current schema processors will expect the (bogus) format " -- MM -- -- . In the example, we follow the correct ISO 8601 format.

Restrictions

The period (one year) and the duration (one month) are fixed, and no calendars other than the Gregorian are supported.

Because of the typo in the W3C XML Schema Specification, users must choose between a bogus format, which works on the current version of the tools, or a correct format, which conforms to ISO 8601.

Example

Valid values include `"--05"`, `"--11Z"`, `"--11+02:00"`, `"--11-04:00"`, or `"--02"`.

The following values would be invalid: `"-01-"` (the format must be `"--MM"`), `"--13"` (the month is out of range), `"--1"` (both digits must be provided), or `"01"` (leading " -- " are missing).

Name

xsd:gMonthDay — Recurring period of time: yearly day.

Derived from:

xsd:anySimpleType

Primary:

xsd:gMonthDay

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="gMonthDay" id="gMonthDay">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:gMonthDay` is the period of one calendar day recurring each calendar year (such as the third of April); its lexical space follows the ISO 8601 syntax for such periods (i.e., " --MM-DD") with an optional timezone.

When needed, days are reduced to fit in the length of the months, so `--02-29` would occur on the the 28th of February of nonleap years.

Restrictions

The period (one year) and the duration (one day) are fixed, and no calendars other than the Gregorian are supported.

Example

Valid values include `"--05-01"`, `"--11-01Z"`, `"--11-01+02:00"`, `"--11-01-04:00"`, `"--11-15"`, or `"--02-29"`.

The following values would be invalid: `"-01-30-"` (the format must be `"--MM-DD"`), `"--01-35"` (the day part is out of range), `"--1-5"` (the leading zeros are missing), or `"01-15"` (the leading `"--"` are missing).

Name

xsd:gYear — Period of one year.

Derived from:

xsd:anySimpleType

Primary:

xsd:gYear

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="gYear" id="gYear">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:gYear` is the period of one calendar year (such as the year 2002); its lexical space follows the ISO 8601 syntax for such periods (i.e., "YYYY") with an optional timezone.

Restrictions

The duration (one year) is fixed, and no calendars other than the Gregorian are supported.

Example

Valid values include `"2001"`, `"2001+02:00"`, `"2001Z"`, `"2001+00:00"`, `"-2001"`, or `"-20000"`.

The following values would be invalid: `"01"` (the century part is missing) or `"2001-12"` (month parts are forbidden).

Name

xsd:gYearMonth — Period of one month.

Derived from:

xsd:anySimpleType

Primary:

xsd:gYearMonth

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="gYearMonth" id="gYearMonth">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:gYearMonth` is the period of one calendar month (such as the month of February 2002); its lexical space follows the ISO 8601 syntax for such periods (i.e., "YYYY-MM") with an optional timezone.

Restrictions

The duration (one month) is fixed, and no calendars other than the Gregorian are supported.

Example

Valid values include `"2001-10"`, `"2001-10+02:00"`, `"2001-10Z"`, `"2001-10+00:00"`, `"-2001-10"`, or `"-20000-04"`.

The following values would be invalid: `"2001"` (the month part is missing), `"2001-13"` (the month part is out of range), `"2001-13-26+02:00"` (the month part is out of range), or `"01-10"` (the century part is missing).

Name

xsd:hexBinary — Binary contents coded in hexadecimal.

Derived from:

xsd:anySimpleType

Primary:

xsd:hexBinary

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="hexBinary" id="hexBinary">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:hexBinary` is the set of all binary contents; its lexical space is a simple coding of each octet as its hexadecimal value.

Restrictions

This datatype should not be confused with another encoding called BinHex that is not supported by W3C XML Schema. Other popular binary text encodings (such as uuXXcode, Quote Printable, BinHex, aencode, or base85, to name few) are not supported by the W3C XML Schema.

The expansion factor is high since each binary octet is coded as two characters (i.e., four octets if the document is encoded with UTF-16).

Example

A UTF-8 XML header such as:

```
"<?xml version="1.0" encoding="UTF-8"?>"
```

encoded would be:

```
"3f3c6d78206c657673726f693d6e3122302e20226e656f636964676e223d54552d4622383e3f"
```

Name

xsd:ID — Definition of unique identifiers.

Derived from:

xsd:NCName

Primary:

xsd:string

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="ID" id="ID">
  <xsd:restriction base="xsd:NCName"/>
</xsd:simpleType>
```

Description

The purpose of the `xsd:ID` datatype is to define unique identifiers that are global to a document and emulate the ID attribute type available in the XML DTDs.

Unlike their DTD counterparts, W3C XML Schema ID datatypes can be used to define not only attributes, but also simple element content.

For both attributes and simple element content, the lexical domain of these datatypes is the lexical domain of XML nonqualified names (`xsd:NCName`).

Identifiers defined using this datatype are global to a document and provide a way to uniquely identify their containing element, whatever its type and name is.

The constraint added by this datatype beyond the `xsd:NCName` datatype from which it is derived is that the values of all the attributes and elements that have an ID datatype in a document must be unique.

Note that this datatype is laxer than the ID datatype from the DTD compatibility datatype library and does allow both using this datatypes for elements and defining multiple type assignment to attributes defined as ID depending on their location in the schema.

Restrictions

Applications that need to maintain a level of compatibility with DTDs should not use this datatype for elements but should reserve it for attributes.

The lexical domain (`xsd:NCName`) of this datatype doesn't allow the definition of numerical identifiers or identifiers containing whitespaces.

Example

```
<element name="book">
  <element name="isbn">
    <data type="xsd:int"/>
  </element>
  <element name="title">
    <data type="xsd:string"/>
  </element>
  <element name="author-ref">
    <attribute name="ref">
      <data type="xsd:IDREF"/>
    </attribute>
  </element>
  <element name="character-refs">
    <data type="xsd:IDREFS"/>
  </element>
  <attribute name="identifier">
    <data type="xsd:ID"/>
  </attribute>
</element>
```

Name

xsd:IDREF — Definition of references to unique identifiers.

Derived from:

xsd:NCName

Primary:

xsd:string

Known subtypes:

xsd:IDREFS

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="IDREF" id="IDREF">
  <xsd:restriction base="xsd:NCName"/>
</xsd:simpleType>
```

Description

The `xsd:IDREF` datatype defines references to the identifiers defined by the ID datatype and, therefore, emulates the IDREF attribute type of the XML DTDs, even though it can be used for simple content elements as well as for attributes.

The lexical space of `xsd:IDREF` is, like the lexical space of `xsd:ID`, nonqualified XML names (NCName).

The constraint added by this datatype beyond the `xsd:NCName` datatype from which it is derived is the values of all the attributes and elements that have a `xsd:IDREF` datatype must match an ID defined within the same document.

Restrictions

Applications that need to maintain a level of compatibility with DTDs should not use this datatype for elements but should reserve it for attributes.

The lexical domain (NCName) of this datatype doesn't allow definition of numerical key references or references containing whitespaces.

Example

```
<element name="book">
  <element name="isbn">
    <data type="xsd:int"/>
  </element>
  <element name="title">
    <data type="xsd:string"/>
```

```
  </element>
  <element name="author-ref">
    <attribute name="ref">
      <data type="xsd:IDREF"/>
    </attribute>
  </element>
  <element name="character-refs">
    <data type="xsd:IDREFS"/>
  </element>
  <attribute name="identifier">
    <data type="xsd:ID"/>
  </attribute>
</element>
```

Name

xsd:IDREFS — Definition of lists of references to unique identifiers.

Derived from:

xsd:IDREF

Primary:

none

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`.

Synopsis

```
<xsd:simpleType name="IDREFS" id="IDREFS">
  <xsd:restriction>
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:restriction base="xsd:IDREF"/>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

`xsd:IDREFS` is derived as a list from `xsd:IDREF` and, thus, represents whitespace-separated lists of references to identifiers defined using the ID datatype.

The lexical space of `xsd:IDREFS` is the lexical space of a list of `xsd:NCName` values with a minimum length of one element (`xsd:IDREFS` cannot be empty lists).

`xsd:IDREFS` emulates the `IDREFS` attribute type of the XML DTDs, even though it can be used to define simple content elements as well as attributes.

Restrictions

Applications that need to maintain a level of compatibility with DTDs should not use this datatype for elements but should reserve it for attributes.

The lexical domain (lists of `xsd:NCName`) of this datatype doesn't allow definition of lists of numerical key references or references containing whitespaces.

Example

```
<element name="book">
```

```
  <element name="isbn">
    <data type="xsd:int"/>
  </element>
  <element name="title">
    <data type="xsd:string"/>
  </element>
  <element name="author-ref">
    <attribute name="ref">
      <data type="xsd:IDREF"/>
    </attribute>
  </element>
  <element name="character-refs">
    <data type="xsd:IDREFS"/>
  </element>
  <attribute name="identifier">
    <data type="xsd:ID"/>
  </attribute>
</element>
```

Name

xsd:int — 32 bit signed integers.

Derived from:

xsd:long

Primary:

xsd:decimal

Known subtypes:

xsd:short

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="int" id="int">
  <xsd:restriction base="xsd:long">
    <xsd:minInclusive value="-2147483648"/>
    <xsd:maxInclusive value="2147483647"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:int` is the set of common single size integers (32 bits), i.e., the integers between -2147483648 and 2147483647, its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

-0 and +0 are considered equal, which is different from the behavior of `xsd:float` and `xsd:double`.

Example

Valid values include `"-2147483648"`, `"0"`, `"-00000000000000000005"`, or `"2147483647"`.

Invalid values include `"-2147483649"` and `"1."`.

Name

xsd:integer — Signed integers of arbitrary length.

Derived from:

xsd:decimal

Primary:

xsd:decimal

Known subtypes:

xsd:nonPositiveInteger, xsd:long, xsd:nonNegativeInteger

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="integer" id="integer">
  <xsd:restriction base="xsd:decimal">
    <xsd:fractionDigits value="0" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:integer` includes the set of all the signed integers, with no restriction on range. Its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

-0 and +0 are considered equal, which is different from the behavior of `xsd:float` and `xsd:double`.

Example

Valid values for `xsd:integer` include `"-1234567890123456789012345678901234567890"`, `"2147483647"`, `"0"`, or `"-00000000000000000000005"`.

Invalid values include `"1."`, `"2.6"`, and `"A"`.

Name

xsd:language — RFC 1766 language codes.

Derived from:

xsd:token

Primary:

xsd:string

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="language" id="language">
  <xsd:restriction base="xsd:token">
     <xsd:pattern
      value="([a-zA-Z]{2}|[iI]-[a-zA-Z]+|[xX]-[a-zA-Z]{1,8})(-[a-zA-Z]{1,8})*"
      />
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:language` are the set of language codes defined by the RFC 1766.

Restrictions

Although the schema for schema defines a minimal test to perform expressed as patterns (see the Definition), the lexical space is the set of existing language codes.

Example

Some valid values for this datatype are: `"en"`, `"en-US"`, `"fr"`, or `"fr-FR"`.

Name

xsd:long — 64 bit signed integers.

Derived from:

xsd:integer

Primary:

xsd:decimal

Known subtypes:

xsd:int

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="long" id="long">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="-9223372036854775808"/>
    <xsd:maxInclusive value="9223372036854775807"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:long` is the set of common double-size integers (64 bits), i.e., the integers between -9223372036854775808 and 9223372036854775807; its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values for `xsd:long` include `"-9223372036854775808"`, `"0"`, `"-00000000000000000005"`, or `"9223372036854775807"`.

Invalid values include `"9223372036854775808"` and `"1."`.

Name

xsd:Name — XML 1.O names.

Derived from:

xsd:token

Primary:

xsd:string

Known subtypes:

xsd:NCName

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="Name" id="Name">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="\i\c*"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:Name` are the tokens (`NMTOKEN`) that conform to the definition of a name in XML 1.0.

Restrictions

Following XML 1.0, those names may contain colons (":"), but no special meaning is attached to these characters. Another datatype (`xsd:QName`) should be used for qualified names when they use namespaces prefixes.

Example

Valid values include `"Snoopy"`, `"CMS"`, or `"_1950-10-04_10:00"`.

Invalid values include `"0836217462"` (a `xsd:Name` cannot start with a number) or `"bold,brash"` (commas are forbidden).

Name

xsd:NCName — Unqualified names.

Derived from:

xsd:Name

Primary:

xsd:string

Known subtypes:

xsd:ID, xsd:IDREF, xsd:ENTITY

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="NCName" id="NCName">
  <xsd:restriction base="xsd:Name">
    <xsd:pattern value="[\i-[:]][\c-[:]]*"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:NCName` are the names (`Name`) that conform to the definition of a `NCName` in the Recommendation "Namespaces in XML 1.0"—i.e., all the XML 1.0 names that do not contain colons (":").

Restrictions

This datatype allows characters such as "-" and may need additional constraints to match the notion of name in your favorite programming language or database system.

Example

Valid values include `"Snoopy"`, `"CMS"`, `"_1950-10-04_10-00"`, or `"bold_brash"`.

Invalid values include `"_1950-10-04:10-00"` or `"bold:brash"` (colons are forbidden).

Name

xsd:negativeInteger — Strictly negative integers of arbitrary length.

Derived from:

xsd:nonPositiveInteger

Primary:

xsd:decimal

Known subtypes:

none

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="negativeInteger" id="negativeInteger">
  <xsd:restriction base="xsd:nonPositiveInteger">
    <xsd:maxInclusive value="-1"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:negativeInteger` includes the set of all the strictly negative integers (excluding zero), with no restriction of range. Its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values for `xsd:negativeInteger` include `"-1234567890123456789012345678901234567890"`, `"-1"`, or `"-00000000000000000000005"`.

Invalid values include `"0"` or `"-1."`.

Name

xsd:NMTOKEN — XML 1.0 name token (NMTOKEN).

Derived from:

xsd:token

Primary:

xsd:string

Known subtypes:

xsd:NMTOKENS

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="NMTOKEN" id="NMTOKEN">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="\c+"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:NMTOKEN` are the set of XML 1.0 "name tokens," i.e., tokens composed of characters, digits, ".", ":", "-", and the characters defined by Unicode, such as "combining" or "extender".

Restrictions

This type is usually called a "token."

Example

Valid values include `"Snoopy"`, `"CMS"`, `"1950-10-04"`, or `"0836217462"`.

Invalid values include `"brought classical music to the Peanuts strip"` (spaces are forbidden) or `"bold,brash"` (commas are forbidden).

Name

xsd:NMTOKENS — List of XML 1.0 name token (NMTOKEN).

Derived from:

xsd:NMTOKEN

Primary:

none

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`.

Synopsis

```
<xsd:simpleType name="NMTOKENS" id="NMTOKENS">
  <xsd:restriction>
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN"/>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

`xsd:NMTOKENS` is derived by list from `xsd:NMTOKEN` and represents whitespace-separated lists of XML 1.0 name tokens.

Restrictions

None.

Example

Valid values include `"Snoopy"`, `"CMS"`, `"1950-10-04"`, `"0836217462 0836217463"`, or `"brought classical music to the Peanuts strip"` (note that, in this case, the sentence is considered as a list of words).

Invalid values include `"brought classical music to the "Peanuts" strip"` (quotes are forbidden) or `"bold,brash"` (commas are forbidden).

Name

xsd:nonNegativeInteger — Integers of arbitrary length positive or equal to zero.

Derived from:

xsd:integer

Primary:

xsd:decimal

Known subtypes:

xsd:unsignedLong, xsd:positiveInteger

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="nonNegativeInteger" id="nonNegativeInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:nonNegativeInteger` includes the set of all the integers greater than or equal to zero, with no restriction of range. Its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"+12345678901234567890123456789O"`, `"0"`, `"00000000000000000005"`, or `"2147483647"`.

Invalid values include `"1."` or `"-1."`.

Name

xsd:nonPositiveInteger — Integers of arbitrary length negative or equal to zero.

Derived from:

xsd:integer

Primary:

xsd:decimal

Known subtypes:

xsd:negativeInteger

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:maxInclusive value="0"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:nonPositiveInteger` includes the set of all the integers less than or equal to zero, with no restriction of range. Its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"-1234567890123456789012345678890"`, `"0"`, `"-00000000000000000000005"`, or `"-2147483647"`.

Invalid values include `"-1."` or `"1."`.

Name

xsd:normalizedString — Whitespace-replaced strings.

Derived from:

xsd:string

Primary:

xsd:string

Known subtypes:

xsd:token

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="normalizedString" id="normalizedString">
  <xsd:restriction base="xsd:string">
    <xsd:whiteSpace value="replace"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical space of `xsd:normalizedString` is unconstrained (any valid XML character may be used), and its value space is the set of strings after whitespace replacement (i.e., after any occurrence of `#x9` (tab), `#xA` (linefeed), and `#xD` (carriage return) have been replaced by an occurrence of `#x20` (space) without any whitespace collapsing).

Restrictions

This is the only datatype that performs whitespace replacement without collapsing. When whitespaces are not significant, `xsd:token` is preferred.

This datatype corresponds to neither the XPath function `normalize-space( )` (which performs whitespace trimming and collapsing) nor to the DOM "normalize" method (which is a merge of adjacent text objects).

Example

The value of the element:

```
<title lang="en">
  Being a Dog Is
  a Full-Time Job
</title>"
```

is the string: " `Being` a `Dog` `Is` a `Full-Time` `Job` ", where all the whitespaces have been replaced by spaces if the title element is a type `xsd:normalizedString`.

Name

xsd:NOTATION — Emulation of the XML 1.0 feature.

Derived from:

xsd:anySimpleType

Primary:

xsd:NOTATION

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="NOTATION" id="NOTATION">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

For W3C XML Schema, the value and lexical spaces of `xsd:NOTATION` are references to notations declared though the `xsd:notation` element. This element doesn't exist in Relax NG where this datatype can be seen as a synonym for `xsd:QName` with backward compatibility for W3C XML Schema.

Restrictions

Notations are very seldom used in real-world applications.

One cannot use `xsd:notation` directly, but must derive it as shown in the Example.

Example

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xsd:notation name="jpeg" public="image/jpeg"
    system="file:///usr/bin/xv"/>
  <xsd:notation name="gif" public="image/gif"
    system="file:///usr/bin/xv"/>
  <xsd:notation name="png" public="image/png"
    system="file:///usr/bin/xv"/>
  <xsd:notation name="svg" public="image/svg"
    system="file:///usr/bin/xsmiles"/>
  <xsd:notation name="pdf" public="application/pdf"
    system="file:///usr/bin/acroread"/>
```

```
  <xsd:simpleType name="graphicalFormat">
    <xsd:restriction base="xsd:NOTATION">
      <xsd:enumeration value="jpeg"/>
      <xsd:enumeration value="gif"/>
      <xsd:enumeration value="png"/>
      <xsd:enumeration value="svg"/>
      <xsd:enumeration value="pdf"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="picture">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:base64Binary">
          <xsd:attribute name="type" type="graphicalFormat"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Name

xsd:positiveInteger — Strictly positive integers of arbitrary length.

Derived from:

xsd:nonNegativeInteger

Primary:

xsd:decimal

Known subtypes:

none

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="positiveInteger" id="positiveInteger">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:minInclusive value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:positiveInteger` includes the set of the strictly positive integers (excluding zero), with no restriction of range. Its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"12345678901234567890123456789O"`, `"1"`, or `"00000000000000000005"`.

Invalid values include `"0"` or `"1."`.

Name

xsd:QName — Namespaces in XML qualified names.

Derived from:

xsd:anySimpleType

Primary:

xsd:QName

Known subtypes:

none

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="QName" id="QName">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical space of `xsd:QName` is the qualified names per Namespace in XML, i.e., a local name (which is a `xsd:NCName`) with an optional prefix (itself a `xsd:NCName`), separated by a colon (":"), where the prefix is declared a namespace prefix in the scope of the element carrying the value. Its value space comprises the pairs (namespace URI, local name) in which the namespace URI is the URI associated to the prefix in the namespace declaration.

Restrictions

It is impossible to apply a pattern on the namespace URI.

The usage of `QNames` in elements and attributes is controversial since it creates a dependency between the content of the document and its markup. However, the official position of the W3C doesn't discourage this practice.

Example

W3C XML Schema itself has already given us some examples of QNames. When we wrote `"<xsd:attribute name="lang" type="xsd:language"/>"`, the type attribute was a `xsd:QName` and its value was the tuple `{"http://www.w3.org/2001/XMLSchema", "language"}` because the URI `"http://www.w3.org/2001/XMLSchema"` had been assigned to the prefix `"xsd:"`. If there had been no namespace declaration for this prefix, the type attribute would have been considered invalid.

Name

xsd:short — 32 bit signed integers.

Derived from:

xsd:int

Primary:

xsd:decimal

Known subtypes:

xsd:byte

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="short" id="short">
  <xsd:restriction base="xsd:int">
    <xsd:minInclusive value="-32768"/>
    <xsd:maxInclusive value="32767"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:short` is the set of common short integers (16 bits), i.e., the integers between -32768 and 32767; its lexical space allows any number of insignificant leading zeros.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"-32768"`, `"0"`, `"-00000000000000000005"`, or `"32767"`.

Invalid values include `"32768"` and `"1."`.

Name

xsd:string — Any string.

Derived from:

xsd:anySimpleType

Primary:

xsd:string

Known subtypes:

xsd:normalizedString

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="string" id="string">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="preserve"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:string` are the set of all possible strings composed of any character allowed in a XML 1.0 document without any treatment done on whitespaces.

Restrictions

This is the only datatype that leaves all the whitespaces. When whitespaces are not significant, `xsd:token` is preferred.

Example

The value of the following element:

```
<title lang="en">
  Being a Dog Is
  a Full-Time Job
</title>
```

is the full string `"Being a Dog Is a Full-Time Job"`, with all its tabulations and CR/LF if the `title` element is a `xsd:string` type .

Name

xsd:time — Point in time recurring each day.

Derived from:

xsd:anySimpleType

Primary:

xsd:time

Known subtypes:

none

Data parameters (facets):

`enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`.

Synopsis

```
<xsd:simpleType name="time" id="time">
  <xsd:restriction base="xsd:anySimpleType">
    <xsd:whiteSpace value="collapse" fixed="true"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical space of `xsd:time` is identical to the time part of `xsd:dateTime` ("hh:mm:ss[Z|(+|-)hh:mm]"); and its value space is the set of points in time recurring daily.

Restrictions

The period (one day) is fixed and no calendars other than the Gregorian are supported.

Example

Valid values include `"21:32:52"`, `"21:32:52+02:00"`, `"19:32:52Z"`, `"19:32:52+00:00"`, or `"21:32:52.12679"`.

Invalid values include `"21:32"` (all the parts must be specified), `"25:25:10"` (the hour part is out of range), `"-10:00:00"` (the hour part is out of range), or `"1:20:10"` (all the digits must be supplied).

Name

xsd:token — Whitespace-replaced and collapsed strings.

Derived from:

xsd:normalizedString

Primary:

xsd:string

Known subtypes:

xsd:language, xsd:NMTOKEN, xsd:Name

Data parameters (facets):

`enumeration`, `length`, `maxLength`, `minLength`, `pattern`.

Synopsis

```
<xsd:simpleType name="token" id="token">
  <xsd:restriction base="xsd:normalizedString">
    <xsd:whiteSpace value="collapse"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The lexical and value spaces of `xsd:token` are the sets of all the strings after whitespace replacement —i.e., after any occurrence of `#x9` (tab), `#xA` (linefeed), and `#xD` (carriage return) is replaced by an occurrence of `#x20` (space) and collapsing (i.e., the contiguous occurrences of spaces are replaced by a single space, and leading and trailing spaces are removed).

More simply said, `xsd:token` is the most appropriate datatype to use for strings that do not care about whitespaces.

Restrictions

The name `xsd:token` is misleading since whitespaces are allowed within `xsd:token`. `xsd:NMTOKEN` is the type corresponding to what is usually called "tokens."

Example

The element:

```
<title lang="en">
  Being a Dog Is
  a Full-Time Job
</title>
```

is a valid `xsd:token` and its value is the string `"Being a Dog Is a Full-Time Job"`, where all the whitespaces have been replaced by spaces, leading and trailing spaces have been removed and contiguous sequences of spaces have been replaced by single spaces.

Name

xsd:unsignedByte — Unsigned value of 8 bits.

Derived from:

xsd:unsignedShort

Primary:

xsd:decimal

Known subtypes:

none

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="unsignedByte" id="unsignedBtype">
  <xsd:restriction base="xsd:unsignedShort">
    <xsd:maxInclusive value="255"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:unsignedByte` is the integers between 0 and 255, i.e., the unsigned values that can fit in a word of 8 bits. Its lexical space allows an optional "+" sign and leading zeros before the significant digits.

Restrictions

The lexical space does not allow values expressed in other numeration bases (such as hexadecimal, octal, or binary).

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"255"`, `"0"`, `"+00000000000000000005"`, or `"1"`.

Invalid values include `"-1"` and `"1."`.

Name

xsd:unsignedInt — Unsigned integer of 32 bits.

Derived from:

xsd:unsignedLong

Primary:

xsd:decimal

Known subtypes:

xsd:unsignedShort

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="unsignedInt" id="unsignedInt">
  <xsd:restriction base="xsd:unsignedLong">
    <xsd:maxInclusive value="4294967295"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:unsignedInt` is the integers between 0 and 4294967295, i.e., the unsigned values that can fit in a word of 32 bits. Its lexical space allows an optional "+" sign and leading zeros before the significant digits.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"4294967295"`, `"0"`, `"+00000000000000000005"`, or `"1"`.

Invalid values include `"-1"` and `"1."`.

Name

xsd:unsignedLong — Unsigned integer of 64 bits.

Derived from:

xsd:nonNegativeInteger

Primary:

xsd:decimal

Known subtypes:

xsd:unsignedInt

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="unsignedLong" id="unsignedLong">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:maxInclusive value="18446744073709551615"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:unsignedLong` is the integers between 0 and 18446744073709551615, i.e., the unsigned values that can fit in a word of 64 bits. Its lexical space allows an optional "+" sign and leading zeros before the significant digits.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"18446744073709551615"`, `"0"`, `"+00000000000000000005"`, or `"1"`.

Invalid values include `"-1"` and `"1."`.

Name

xsd:unsignedShort — Unsigned integer of 16 bits.

Derived from:

xsd:unsignedInt

Primary:

xsd:decimal

Known subtypes:

xsd:unsignedByte

Data parameters (facets):

`enumeration`, `fractionDigits`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, `totalDigits`.

Synopsis

```
<xsd:simpleType name="unsignedShort" id="unsignedShort">
  <xsd:restriction base="xsd:unsignedInt">
    <xsd:maxInclusive value="65535"/>
  </xsd:restriction>
</xsd:simpleType>
```

Description

The value space of `xsd:unsignedShort` is the integers between 0 and 65535, i.e., the unsigned values that can fit in a word of 16 bits. Its lexical space allows an optional "+" sign and leading zeros before the significant digits.

Restrictions

The decimal point (even when followed only by insignificant zeros) is forbidden.

Example

Valid values include `"65535"`, `"0"`, `"+00000000000000000005"`, or `"1"`.

Invalid values include `"-1"` and `"1."` .

Glossary

A

ambiguous

A pattern is ambiguous when a fragment of an instance document may be valid through several alternatives in its `choice` patterns. Relax NG allows ambiguous patterns but they can be a problem for annotation and datatype assigment.

C

chameleon design

Specifying a namespace in `include`, `externalRef` or `parentRef` to give a namespace to grammars or patterns defined without a namespace is known as "chameleon design." This is because the imported grammar or pattern takes the new namespace like a chameleon takes the color of the environment in which it is placed.

character class

In a regular expression, a character class is an atom matching a set of characters. Character classes may be classical Perl character classes, Unicode character classes, or user-defined character classes.

classical Perl character class

A set of character classes designated by a single letter, for which upper- and lowercases of the same letter are complementary (for instance, "\d" is all the decimal digits, and "\D" is all the characters that are not decimal digits).

compositor

A compositor is a pattern which can be used combine other patterns. Relax NG has three basic compositors: `group`, `choice` and `interleave`. A fourth compositor, `mixed`, which is a shortcup for `interleave` with an embedded `text` pattern.

content model

A description of the structure of children elements and text nodes (independent of attributes). The content model is "simple" when there is a text node but no elements, "complex" when there are element nodes but no text, "mixed" when there are text and element nodes, and "empty" when there are neither text nor element nodes.

D

datatype

A term used by Relax NG to qualify both the content of a simple content element or attribute. Datatypes should not be confused with XML 1.0 element types, which are called element names by Relax NG.

deterministic

A pattern is deterministic if a schema processor can always determine which alternative to follow looking only at the current element under validation. Unlike W3C XML Schema, Relax NG does not require deterministic patterns.

DOM

Document Object Model. An object-oriented model of XML documents, including the definition of the API allowing its manipulation. The third version of DOM (DOM Level 3) will include an API named "Abstract Schemas" to facilitate schema-guided editions of XML documents (see `http://www.w3.org/TR/DOM-Level-3-Core`).

DSDL

Document Schema Definition Languages (DSDL) is a project undertaken by the ISO (ISO/IEC JTC 1/SC 34/WG 1, to be precise) whose objective is "to create a framework within which multiple validation tasks of different types can be applied to an XML document in order to achieve more complete

validation results than just the application of a single technology" (see `http://dsdl.org`).

DTD

Document Type Definition. XML 1.0 DTDs are inherited from SGML, in which rules were included that allow the customization of the markup itself and played a very central role. Because of the syntactical rules included in their DTDs, SGML applications need a DTD to be able to read an SGML document. One of the simplifications of XML is to state that a XML parser should be able to read a document without needing a DTD. DTDs have therefore been simplified over their SGML ancestors and remain the first incarnation of what is today called a XML Schema language.

E

element

One of the basic type of nodes in the tree represented by a XML document. An element is delimited by start and end tags. In the corresponding tree, an element is a nonterminal node, which may have subnodes of type element, character (text), and namespace and attribute, as well as comment and processing instruction nodes.

element type

Term used in the XML 1.0 Recommendation, which is equivalent to the notion of element names in W3C XML Schema and should not be confused with the simple or complex datatype of an element.

empty content

An element that has neither child element nor text nodes (with or without attributes).

F

facet

A constraint added to the lexical or value space of a simple datatype of the W3C XML Schema datatype system. The list of facets that can be used depends on the simple datatype. W3C XML Schema's facets can be used as parameters in Relax NG `data` patterns.

G

Grammar

A grammar is a pattern which is a container for a start pattern and any number of named patterns.

I

Infoset

XML Information Set. A formal description of the information that may be found in a well-formed XML document.

instance document

A XML document that is a candidate to be validated by a schema. Any well-formed XML 1.0 document that conforms to the Namespaces in XML 1.0 Recommendation can be considered a valid or invalid instance document.

L

lexical space

The set of all representations (after parsing and whitespace processing) allowed for a simple datatype.

local name

The name of a component in its namespace, i.e., the part of the qualified name that comes after the namespace prefix.

M

mixed content — The content of an element that contains both child element and text nodes.

N

Named pattern — Named patterns are globally defined in a grammar and may be refered from anywhere in this grammar or in the children grammars.

namespace — A unique identifier that can be associated with a set of XML elements and attributes. This identifier is a URI, which is not required to point to an actual resource but must "belong" to the author of these elements and attributes. Since this full URI can't be included in the name of each element and attribute, a namespace prefix is assigned to the namespace URI through a namespace declaration. This prefix is added to the local name of the elements and attributes to form a qualified name. Namespaces are optional and elements and attributes may have no namespaces attached.

P

pattern — Any part of a Relax NG schema that can be matched against a set of attributes and a sequence of elements and strings is a pattern. With the exception of name classes, all parts (including the whole schema) of a Relax NG schema are patterns.

piece — Regular expressions (or patterns) are composed of pieces. Each piece is itself composed of an atom describing a condition on a substring and an optional quantifier defining the expected number of occurrences of the atom.

Q

qualified name — The complete name of a component, including the prefix associated to its target namespace if one is defined.

R

Recursive content models — Recursive content models are content models in which elements can be included directly or indirectly within themselves (such as XHTML "div" or "span" elements).

recursive patterns — Recursive patterns are named patterns including directly or indirectly references to themselves. Relax NG only allows recursive patterns which describe recursive content models, i.e. for which the definition of the named pattern is isolated from its reference by an element pattern.

regular expression — A syntax to express conditions on strings. The syntax used by the W3C XML Schema for its patterns is very close to the syntax introduced by the Perl programming language. A regular expression is composed of elementary "pieces."

RELAX — A grammar-based XML Schema language developed by Murata Makoto and published in March 2000 as a Japanese ISO Standard (see `http://www.xml.gr.jp/relax`).

RELAX NG — A grammar-based XML Schema language resulting from a merger between RELAX and TREX (see `http://relaxng.org`).

Russian doll design — A schema where the definitions of elements and attributes are embedded one in each other without using named patterns is often refered as having a "Russian doll design".

S

SAX — Simple API for XML. A streaming event-based API used between parsers and applications. Its streaming nature means that pipelines of XML processing may be created using SAX (see `http://www.saxproject.org`).

Schematron — A rule-based XML Schema language, developed by Rick Jelliffe, using XPath expressions to describe validation rules (see `http://www.ascc.net/xml/resource/schematron/schematron.html`).

SGML — Standard Generalized Markup Language. Created in 1980, the ancestor of XML. XML was designed as a simplified subset of SGML to be used on the Web.

simple content — An element has a simple content model when it has a child text node only (and no subelements). A simple content element has a simple type if it has no attributes, and it has a complex type if it has any attributes.

Simplification — Action of simplifying and normalizing a Relax NG schema to remove the syntactical variations and use a few number of basic patterns and name classes. The simplification of Relax NG is described in its specification to.

special character — A character that may be used as an atom after a "\" to accept a specific character, either for convenience or because this character is interpreted differently in the context of a regular expression.

Start pattern — When a grammar is used to validate an instance document, its start pattern is matched against the root element of the instance document. When a grammar is embedded in another grammar, the embedded grammar is replaced by its start pattern during the simplication of the schema.

T

TREX — A grammar-based XML Schema language developed by James Clark (see `http://www.thaiopensource.com/trex`).

U

unambiguous — A pattern is unambiguous when any fragment of instance document which is valid per this pattern is only valid for one of each alternatives. Relax NG does not require unambiguous patterns but they can be considered a good practice for annotation and datatype assigment.

Unicode block — A set of characters classified by their "localization" (Latin, Arabic, Hebrew, Tibetan, and even Gothic or musical symbols).

Unicode category — A set of characters classified by their usage (letters, uppercase, digit, punctuation, etc.).

Unicode character class — A set of character classes defined based on the Unicode blocks and categories.

URI — Uniform Resource Identifier. Defined by the RFCs 2396 and 2732. URIs were created to extend the notion of URLs (Uniform Resource Locators) to include abstract identifiers that do not necessarily need to "locate" a resource.

URL — Uniform Resource Locator, a common identifier used on the Web. URLs are absolute when the full path to the resource is indicated, and relative when a partial path is given that needs to be evaluated in relation with a base URL.

V

valid — A XML document that is well-formed and conforms to a schema (Relax NG, DTD, W3C XML Schema, etc.) of some kind.

value space — The set of all the possible values for a simple datatype, independent of their actual representation in the instance documents.

W

W3C — World Wide Web Consortium. Originally created to settle HTML and HTTP as de facto standards. The main specification body for the core specifications of the World Wide Web and the keeper of the core XML specifications (see `http://www.w3.org`).

well-formed — An XML document that meets the conditions defined in the XML 1.0 Recommendation: it must be readable without ambiguity. Syntax errors will be detected by a XML parser without schema of any type.

whitespace — Characters `#x9` (tab), `#xA` (linefeed), `#xD` (carriage return), and `#x20` (space). These are often used to indent the XML documents to give them a more readable aspect, and are filtered by an operation named "whitespace processing."

X

XInclude — A W3C specification defining a general purpose inclusion mechanism for XML documents (see `http://www.w3.org/TR/xinclude`).

XML — Extensible Markup Language. A subset of SGML created to be used on the Web. Its core specification (XML 1.0) was published by the W3C in February 1998. New specifications have been added since this date, and the W3C considers that, with the addition of W3C XML Schema, the core specifications are now complete.

XPath — A query language used to identify a set of nodes within a XML document. Originally defined to be used with XSLT, it is also used by other specifications such as Schematron, XPointer, W3C XML Schema or XForms (see `http://www.w3.org/TR/xpath`).

XSLT — Extensible Stylesheet Language Transformations. A programming language specialized for the transformation of XML documents (see `http://www.w3.org/TR/xslt`).

Part IV. Appendixes

How does Relax NG fit in the DSDL family?

Chapter 21. Appendix A: DSDL

What's the problem?

Although Relax NG has been started as a standalone project under the auspice of the Organization for the Advancement of Structured Information Standards (OASIS), Relax NG is now been standardized at the ISO (ISO/IEC JTC1 SC34 WG1 to be precise) as a part of a multi-part standard named DSDL (see http://dsdl.org).

Standing for "Document Schema Definition Languages", DSDL is a recognition that the validation of XML documents is a subject too wide and complex to be covered by a single language and that the industry needs a set of simple and dedicated languages to perform different validation tasks and a framework in which these languages may be used together.

There are many different aspects in validating (or schematizing) XML documents which can be categorized into:

- Validating the structure of the document, i.e. checking the imbrication of elements and attributes (this is the domain in which Relax NG is so good).
- Validating the content of each text node and attribute independently of each other (this is where datatype libraries are needed).
- Validating integrity constraints between different elements and attributes.
- Validating any other rules (often called business rules).

All over this book, we've seen how Relax NG can help us to cover an important part of this issue, but we've also seen that Relax NG is simple and efficient because it has been kept focussed in solving one and only one problem and there are huge gaps which cannot be covered by Relax NG. For instance, if a XML vocabulary includes mixed content models, you can't restrict the content of your documents to be ASCII only, nor can you define that the content of your "modeling" element must be spell checked. The goal of DSDL is to provide means to fill out these gaps and to cover the whole domain of document validation.

DSDL can be seen as a framework and set of languages to check the quality of XML documents and this issue appears to be crucial for any XML based application. Recent works such has the presentation given by Simon Riggs at XML Europe 2003 or the work of Isabelle Boydens about the quality of big databases have shown that about 10% of XML documents (or data records) contain at least an error and this level of quality is unacceptable for many applications. DSDL could thus be a technology which is just indispensable for most of XML applications.

A multi part standard

DSDL is still work in progress. It is a multi-part specification, each of the parts presenting a different schema language (except part 1 which is an introduction and part 10 which is the description of the framework itself).

Part 1: Overview

This is a kind of road map describing DSDL itself and introducing each of the parts.

Part 2: Regular-grammar-based Validation

This part is Relax NG itself. It is a rewriting of the Relax NG Oasis Technical Committee specification to meet the requirements of ISO publications. Its wording is more formal than the Oasis specification

but the features of the language is the same and any Relax NG implementation conform the one of these two documents should also be conform to the other.

DSDL Part 2 is now a "Final Draft International Standard" (FDIS), i.e. an official ISO standard.

Part 3: Rule-based Validation

This part will describe the next release of the rule based schema language known as Schematron. Schematron has been defined by Rick Jelliffe and other contributors and its home page is http://www.ascc.net/xml/schematron/.

The current version of Schematron is a language to express sets of rules as XPath expressions (or mode accurately as XSLT expressions since XSLT functions such as `document()` are also supported in XPath expressions).

Without entering in the details of the language, let's say that a Schematron schema is composed of set of rules named "patterns" (these patterns shouldn't be confused with Relax NG patterns). Each pattern includes one or more rules. Each rules sets the context nodes under which tests will be performed and each tests is performed either as `assert` or as `report`. An `assert` is a test which raises an error if it is not verified while a `report` is a test which raises an error if it is specified.

A partial schema for our library could be:

```
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
 <sch:title>Schematron Schema for library</sch:title>
 <sch:pattern>
  <sch:rule context="/">
   <sch:assert test="library">The document element should be "library".</sch:a
  </sch:rule>
  <sch:rule context="/library">
   <sch:assert test="book">There should be at least a book!</sch:assert>
   <sch:assert test="not(@*)">No attribute for library, please!</sch:assert>
  </sch:rule>
  <sch:rule context="/library/book">
   <sch:report test="following-sibling::book/@id=@id">Duplicated ID for this b
   <sch:assert test="@id=concat('_', isbn)">The id should be derived from the
  </sch:rule>
  <sch:rule context="/library/*">
   <sch:assert test="self::book or self::author or self::character">This eleme
  </sch:rule>
 </sch:pattern>
</sch:schema>
```

We see from that simple example that it would be very verbose to write a full schema with Schematron since that would mean writing a rule for each element and in this rule writing all the individual tests checking the content model and eventually the relative order between children elements. We see also that it cannot be beaten to express what is oftent called business rules such as:

```
<sch:assert test="@id=concat('_', isbn)">The id should be derived from the ISB
```

which checks that the `id` attribute of a book should be derived from its ISBN element by adding a leading underscore.

DSDL Part 3, the next version of Schematron should keep this structure and add still more power by allowing to use not only XPath 1.0 expressions but also expressions taken from other languages such as EXSLT (a standard extension library for XSLT), XPath 2.0, XSLT 2.0 and even XQuery 1.0.

Part 4: Selection of Validation Candidates

Although Relax NG provides a way to write and combine modular schemas, it is often the case that you need to validate a composite document against existing schemas which can be written using different languages: you may want for instance to validate XHTML documents with embedded RDF statements. In this case, you need to split your documents into pieces and validate each of these pieces against its own schema.

The first contribution to Part 4 has been an ISO specification known as "Relax Namespace" by Murata Makoto. This contribution has been followed by a couple of others, namely MNS by James Clark and "Namespace Switchboard" by Rick Jelliffe. The latest contribution, "Namespace Routing Language" (NRL) has been made by James Clark in June 2003 and builds on the previous proposals. Although it is too early to say if NRL will become DSDL Part 4, it will most likely influence it heavily. NRL is implemented in the latest versions of Jing.

The first example given in the specification (http://www.thaiopensource.com/relaxng/nrl.html) shows how NRL can be used to validate a SOAP message containing one or more XHTML document:

```
<rules xmlns="http://www.thaiopensource.com/validate/nrl">
 <namespace ns="http://schemas.xmlsoap.org/soap/envelope/">
   <validate schema="soap-envelope.xsd"/>
 </namespace>
 <namespace ns="http://www.w3.org/1999/xhtml">
   <validate schema="xhtml.rng"/>
 </namespace>
</rules>
```

This would split the SOAP messages into its envelope validated against the W3C XML Schema schema "soap-envelope.xsd" and one or more XHTML documents found in the body of the SOAP message which will be validated against the Relax NG schema "xhtml.rng".

More advanced features are available including namespace wildcards, validation modes, open schemas, transparent namespaces and NRL seems to be able to handle the most complex cases until the basic assumption that instance documents may be split according to the namespaces of its elements and attributes is met.

Part 5: Datatypes

The goal of this part is to define a set of primitive datatypes with their constraining facets and the mechanisms to derive new datatypes from this set and it is fair to say that it's probably the least advanced and more complex part of DSDL. While people agree on what shouldn't be done it is difficult to go beyond the criticism of existing systems such as W3C XML Schema datatypes and propose something better.

Some interesting ideas have been raised during the last DSDL meeting in May 2003 which kind of converge with threads discussed on the XML-DEV mailing list in June and we may hope that this should lead to something more constructive in the next DSDL meeting in December 2003.

Part 6: Path-based Integrity Constraints

The goal of this part is basically to define a feature covering W3C XML Schema's xs:unique, xs:key and xs:keyref. Part 6 hasn't seen any contribution yet.

Part 7: Character Repertoire Validation

This part will allow to specify which characters may be used in specific elements and attributes or within entire XML documents. The W3C note "A Notation for Character Collections for the

WWW" (http://www.w3.org/TR/charcol/) is used as an input for Part 7 and the first contribution is "Character Repertoire Validation for XML" (CRVX) (http://dret.net/netdret/docs/wilde-crvx-www2003.html).

A simple example of CRVX is:

```
<crvx xmlns="http://dret.net/xmlns/crvx10">
 <restrict structure="ename aname pitarget" charrep="\p{IsBasicLatin}"/>
 <restrict structure="ename aname" charrep="[^0-9]"/>
</crvx>
```

In this proposal, the structure attribute contains identifiers for "element names" (ename), "attribute names (aname)", Processing Instruction targets "pitarget" and other XML constructions including element and attribute contents. This example would thus impose that element and attribute names and Processing Instruction targets are all using characters from the BasicLatin block and that element and attribute names do not use digits.

There is some overlap between Part 7 and other schema languages such as Part 2 (Relax NG) since you'd just need to take care that your names match the rules defined there and can use `data` pattern to check the content of attributes and simple content elements. However, Part 7 gives you a more focused mean of expressing these rules independently of other schemas and is filling some gaps in such constraints: Relax NG cannot express such constraints on name classes nor on mixed content elements.

Part 8: Declarative Document Architectures

This part is still the most mysterious to me. The idea here is to allow to add information to documents (such as default values) depending on the structure of the document and the only input considered for Part 8 so far is known as "Architectural Forms", an old promising-but-never-used-that-much technology.

Part 9: Namespace and Datatype-aware DTDs

There were plenty of good things in DTDs, especially in SGML DTDs and many people are still using them and do challenge the need to put them to trash and define new schema languages to support namespaces and datatypes. DSDL Part 9 is for these people who would like to rely on years of usage of DTDs without loosing all of the goodies of newer schema languages. Despite a burst of discussion in April 2002, this part hasn't really advanced yet.

Part 10: Validation Management

Last but not least, Part 10 (formerly known as Part 1: Interoperability Framework) is the cement which will let you use together the different parts from DSDL together with external tools such as XSLT, W3C XML Schema or your favorite spell checker to come back to an example given in the introduction to this chapter.

Here again, different contributions have been made, including my own "XML Validation Interoperability Framework" XVIF and Rick Jelliffe's Schemachine and the latest contribution is know (and implemented) as "xvif/outie" (see http://downloads.xmlschemata.org/python/xvif/outie/about.xhtml).

A simple example of a xvif/outie document is:

```
<?xml version="1.0" encoding="utf-8"?>Declarative Document Architectures
```

```
<framework>
 <rule>
  <instance>
   <transform transformation="normalize.xslt"/>
  </instance>
  <assert>
   <isValid schema="schema.rng"/>
   <isValid schema="schema.sch"/>
  </assert>
 </rule>
</framework>
```

This document is defining a rule to be checked on the result of the XSLT transformation "normalize.xslt" applied on the instance document and this rule is that the result of the transformation must be valid per both "schema.rng" and "schema.sch".

What DSDL should bring you

As a Relax NG user, DSDL should bring you all what's Relax NG has left behind to focus on the validation of the structure of XML documents and even more:

- You are already using Part 2 (Relax NG)
- Part 3 (Schematron) gives you the ability to add highly flexible "business rules" to your schemas.
- Part 4 (Selection of Validation Candidates) lets you write and reuse schemas written in any language and combine them to validate composite documents.
- Part 5 (Datatypes) should provide a better alternative to W3C XML Schema datatypes.
- Part 6 (Path-based Integrity Constraints) will let you specify integrity constraints between elements and attributes.
- Part 7 (Character repertoire) will let you specify which characters may be used in your documents.
- Part 8 (Declarative Document Architectures) will let you add the information which had been kept implicit to your documents before validation.
- Part 9 (Namespace and Datatype-aware DTDs) will let you upgrade and reuse your DTDs in the context of newer applications.
- Part 10 (Validation Management) will let you do all this together and plug other transformation and validation tools.

If you like Relax NG, I am sure that you'll enjoy the other members of the DSDL family. They share the same principles of focus to solving a specific issue and this focus keeps them both powerful and easy to use.