RELAX NG Compact Syntax Tutorial

Working Draft 26 March 2003

This version:
   Working Draft: 26 March 2003

Editors:
   James Clark<jjc@jclark.com>, John Cowan<jcowan@reutershealth.com>, MURATA Makoto<EB2M-MRT@asahi-net.or.jp>


This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to OASIS, except as needed for the purpose of developing OASIS specifications, in which case the procedures for copyrights defined in the OASIS Intellectual Property Rights document must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Abstract

RELAX NG is a simple schema language for XML, based on [RELAX] and [TREX]. A RELAX NG schema specifies a pattern for the structure and content of an XML document. A RELAX NG schema thus identifies a class of XML documents consisting of those documents that match the pattern.

Two syntaxes have been defined for RELAX NG. The original syntax uses XML; with this syntax an RELAX NG schema is itself an XML document. Subsequently, a compact non-XML syntax has been
defined.

This document is a tutorial for RELAX NG version 1.0 using the compact syntax.

**Status of this Document**

This is a working draft constructed by the editors. It is not an official committee work product and may not reflect the consensus opinion of the committee. Comments on this document may be sent to relax-ng-comment@lists.oasis-open.org.

**Table of Contents**

1. Getting started
2. Choice
3. Attributes
4. Named patterns
5. Datatyping
6. Enumerations
7. Lists
8. Interleaving
9. Modularity
   - 9.1. Referencing external patterns
   - 9.2. Combining definitions
   - 9.3. Merging grammars
   - 9.4. Replacing definitions
10. Namespaces
   - 10.1. Qualified names
   - 10.2. Default namespace
11. Name classes
12. Internationalization
13. Annotations
   - 13.1. Applying annotations
   - 13.2. String literal syntax
   - 13.3. Documentation syntax
   - 13.4. Grouping definitions
14. Nested grammars
15. Non-restrictions
16. Advanced features
   - 16.1. inherit keyword
   - 16.2. Grammar-level annotations
   - 16.3. >> annotations
17. Further information

**Appendixes**

A. List of keywords
B. Comparison with XML DTDs

References
1. Getting started

Consider a simple XML representation of an email address book:

```xml
<addressBook>
  <card>
    <name>John Smith</name>
    <email>js@example.com</email>
  </card>
  <card>
    <name>Fred Bloggs</name>
    <email>fb@example.net</email>
  </card>
</addressBook>
```

The DTD (as an internal subset) would be as follows:

```xml
<!DOCTYPE addressBook [ 
<!ELEMENT addressBook (card*)> 
<!ELEMENT card (name, email)> 
<!ELEMENT name (#PCDATA)> 
<!ELEMENT email (#PCDATA)> ]>
```

A RELAX NG pattern for this could be written as follows:

```xml
element addressBook {
  element card {
    element name { text },
    element email { text }
  }*
}
```

If the `addressBook` is required to be non-empty, then we can use `+` instead of `*`:

```xml
element addressBook {
  element card {
    element name { text },
    element email { text }
  }+
}
```

Now let's change it to allow each `card` to have an optional `note` element:

```xml
element addressBook {
  element card {
    element name { text },
    element email { text },
    element note { text }?
  }*
}
```

Note that the `text` pattern matches arbitrary text, including empty text. Note also that whitespace separating tags is ignored when matching against a pattern.

Comments start with a `#` and continue to the end of the line:

```xml
# A RELAX NG compact syntax pattern
# for an address book.
element addressBook {
```
# an entry in the address book
element card {
    element name { text },
    element email { text }  # an email address
}*
}

Comments starting with ## are treated specially; see Section 13, “Annotations”.

## Choice

Now suppose we want to allow the name to be broken down into a givenName and a familyName, allowing an addressBook like this:

```xml
<addressBook>
  <card>
    <givenName>John</givenName>
    <familyName>Smith</familyName>
    <email>js@example.com</email>
  </card>
  <card>
    <name>Fred Bloggs</name>
    <email>fb@example.net</email>
  </card>
</addressBook>
```

We can use the following pattern:

```xml
element addressBook {
    element card {
        element name { text }
        | (element givenName { text },
          element familyName { text })),
        element email { text },
        element note { text }?
    }*
}
```

This corresponds to the following DTD:

```xml
<!DOCTYPE addressBook [
  <!ELEMENT addressBook (card*)>
  <!ELEMENT card ((name | (givenName, familyName)), email, note?)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT givenName (#PCDATA)>
  <!ELEMENT familyName (#PCDATA)>
  <!ELEMENT note (#PCDATA)>
]>
```

Just as with DTDs, there is no implicit precedence between connectors. For example, $x \lor y, z$ is not allowed; the precedence must be made explicit by using $(x \lor y), z$ or $x \lor (y, z)$ must be used.

## Attributes

Suppose we want the card element to have attributes rather than child elements. The DTD might look like this:
Just change each element pattern to an attribute pattern:

```xml
<addressBook>
  <card>
    <name>John Smith</name>
    <email>js@example.com</email>
  </card>
</addressBook>
```

In XML, the order of attributes is traditionally not significant. RELAX NG follows this tradition. The above pattern would match both

```xml
<card name="John Smith" email="js@example.com"/>
```

and

```xml
<card email="js@example.com" name="John Smith"/>
```

In contrast, the order of elements is significant. The pattern

```xml
<card>
  <name>John Smith</name>
  <email>js@example.com</email>
</card>
```

would not match

```xml
<card><email>js@example.com</email><name>John Smith</name></card>
```

Note that an attribute pattern by itself indicates a required attribute, just as an element pattern by itself indicates a required element. To specify an optional attribute, use `?` just as with element:

```xml
<addressBook>
  <card>
    <name>John Smith</name>
    <email>js@example.com</email>
    <note>My note</note>
  </card>
</addressBook>
```

The `,` and `|` connectors can be applied to attribute patterns in the same way they are applied to element patterns. For example, if we wanted to allow either a `name` attribute or both a `givenName` and a `familyName` attribute, we can specify this in the same way that we would if we were using elements:

```xml
<addressBook>
  <card>
    (attribute name { text } | (attribute givenName { text }, attribute familyName { text })),
    attribute email { text }?
  </card>
</addressBook>
```
The | and | connectors can combine element and attribute patterns without restriction. For example, the following pattern would allow a choice of elements and attributes independently for both the name and the email part of a card:

```
<addressBook>
  <card>
    <name>John Smith</name>
    <email>js@example.com</email>
  </card>
</addressBook>
```

element addressBook {
  element card {
    (element name { text } | attribute name { text }),
    (element email { text } | attribute email { text })
  }*
}

As usual, the relative order of elements is significant, but the relative order of attributes is not. Thus the above would match any of:

```
<card name="John Smith" email="js@example.com"/>
<card email="js@example.com" name="John Smith"/>
<card email="js@example.com"> <name>John Smith</name> </card>
<card name="John Smith"> <email>js@example.com</email> </card>
<card> <name>John Smith</name> <email>js@example.com</email> </card>
```

However, it would not match

```
<card> <email>js@example.com</email> <name>John Smith</name> </card>
```

because the pattern for card requires any email child element to follow any name child element.

When an element pattern does not contain any patterns matching attributes, then an element that matches the pattern cannot have any attributes. Similarly, when any element pattern does not contain any patterns matching elements or strings, then an element that matches the pattern cannot have any children. This can be made more explicit by using the empty pattern. For example,

```
element card {
  attribute email { text } ,
  empty
}
```

is equivalent to

```
element card {
  attribute email { text }
}
```

The use of the empty pattern is necessary only when an element has neither attributes nor children. For example,

```
element addressBook {
  element card {
    element name { text },
    element email { text },
    element prefersHTML { empty }?
  }*
}
```

4. Named patterns

For a non-trivial RELAX NG pattern, it is often convenient to be able to give names to parts of the
pattern. Instead of

element addressBook {
    element card {
        element name { text },
        element email { text }
    }*
}

we can write

grammar {
    start =
        element addressBook {
            element card { cardContent }*
        }
    cardContent =
        element name { text },
        element email { text }
}

A grammar pattern contains one or more definitions. Each definition associates a name with a pattern. Inside a grammar, a pattern consisting of just a name references the definition of that name in the grammar. The name start is special. A grammar pattern is matched by matching the definition of start. A grammar pattern must define start.

We can use the grammar pattern to write RELAX NG in a style similar to DTDs:

grammar {
    start = AddressBook
    AddressBook = element addressBook { Card* }
    Card = element card { Name, Email }
    Name = element name { text }
    Email = element email { text }
}

The opening grammar { and closing } are required only when a grammar pattern is nested within another pattern. In the typical case, when the grammar pattern is the outermost pattern, they can be omitted. For example, the above pattern can be written as:

    start = AddressBook
    AddressBook = element addressBook { Card* }
    Card = element card { Name, Email }
    Name = element name { text }
    Email = element email { text }

Recursive references are allowed. For example,

    inline =
        (text
         | element bold { inline }
         | element italic { inline }
         | element span {
             attribute style { text }?,
             inline
         })*

However, recursive references must be within an element pattern. Thus, the following is not allowed:

    inline =
        (text
To use a keyword such as `element`, `attribute`, `text`, `empty`, `grammar` as the name of a definition, it must be quoted with `\`. For example,

```
start = \element
\element = element element { text }
```

is equivalent to

```
start = e
e = element element { text }
```

Note that keywords need not be quoted when specifying element or attribute names. A complete list of keywords is in Appendix A, List of keywords.

5. Datatyping

RELAX NG allows patterns to reference externally-defined datatypes. RELAX NG implementations may differ in what datatypes they support. You can only use datatypes that are supported by the implementation you plan to use. The most commonly used datatypes are those defined by [W3C XML Schema Datatypes].

A pattern consisting of a name qualified with a prefix matches a string that represents a value of a named datatype. The prefix identifies the library of datatypes being used and the rest of the name specifies the name of the datatype in that library. The prefix `xsd` identifies the datatype library defined by [W3C XML Schema Datatypes]. Assuming your RELAX NG implementation supports this library (most do), you could use:

```
element number { xsd:integer }
```

If the children of an element or an attribute match a datatype pattern, then the complete content of the element or attribute must match that datatype pattern. It is not permitted to have a pattern which allows part of the content to match a datatype pattern, and another part to match another pattern. For example, the following pattern is not allowed:

```
element bad {
    xsd:int,
    element note { text }
}
```

However, this would be fine:

```
element ok {
    xsd:int,
    attribute note { text }
}
```

Note that this restriction does not apply to the `text` pattern.
Datatypes may have parameters. For example, a string datatype may have a parameter controlling the length of the string. The parameters applicable to any particular datatype are determined by the datatyping vocabulary. In the case of [W3C XML Schema Datatypes], the applicable parameters correspond to the facets defined in [W3C XML Schema Datatypes] with the exception of the enumeration and whiteSpace facets. Parameters are specified by following the datatype name with a list of one or more name=value parameter assignments in braces. For example, the following constrains the email element to contain a string at least 6 characters long and at most 127 characters long:

```xml
element email {
    xsd:string { minLength = "6" maxLength = "127" }
}
```

The value of a parameter is a string literal. As in XML, string literals can be delimited using either " or '.

A companion document, [Guidelines], describes exactly how the datatypes defined in [W3C XML Schema Datatypes] can be used as a RELAX NG datatype library.

To use a datatype pattern with a prefix other than xsd, a datatypes declaration must be added to the beginning of the file. The datatypes declaration associates the prefix with the URI of a datatype library. The URI of datatype library identified by xsd prefix is http://www.w3.org/2001/XMLSchema-datatypes. So, for example:

```xml
datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"

element number { xs:integer }
```

is equivalent to

```xml
element number { xsd:integer }
```

### 6. Enumerations

Many markup vocabularies have attributes whose value is constrained to be one of a set of specified strings. A pattern consisting of a literal string matches that string. For example,

```xml
element card {
    attribute name { text },
    attribute email { text },
    attribute preferredFormat { "html" | "text" }
}
```

allows the preferredFormat attribute to have the value html or text. This corresponds to the DTD:

```xml
<!DOCTYPE card [
<!ELEMENT card EMPTY>
<!ATTLIST card
    name CDATA #REQUIRED
    email CDATA #REQUIRED
    preferredFormat (html|text) #REQUIRED>
]>
```

Literal string patterns are not restricted to attribute values. For example, the following is allowed:

```xml
element card {
    element name { text },
    element email { text },
```
The prohibition against a datatype pattern's matching only part of the content of an element also applies to literal string patterns.

By default, a literal string pattern will consider the string in the pattern to match the string in the document if the two strings are the same after the whitespace in both strings is normalized. Whitespace normalization strips leading and trailing whitespace characters, and collapses sequences of one or more whitespace characters to a single space character. This corresponds to the behaviour of an XML parser for an attribute that is declared as other than CDATA. Thus the above pattern will match any of:

```xml
<card name="John Smith" email="js@example.com" preferredFormat="html"/>
<card name="John Smith" email="js@example.com" preferredFormat=" html "/>
```

The way that a literal string pattern compares the pattern string with the document string can be controlled by preceding the literal string with a prefixed name, which identifies a datatype in the same way as for the datatype pattern. The pattern string matches the document string if they both represent the same value of the specified datatype. Thus, whereas a datatype pattern matches an arbitrary value of a datatype, a literal string pattern matches a specific value of a datatype.

There are two datatypes built-in to every RELAX NG implementation. These are named `string` and `token`:

- `token` corresponds to the default comparison behavior of a literal string pattern;
- `string` compares strings without any whitespace normalization (other than the end-of-line and attribute value normalization automatically performed by an XML processor). For example,

```xml
element card {
    attribute name { text },
    attribute email { text },
    attribute preferredFormat { string "html" | string "text" }
}
```

will not match

```xml
<card name="John Smith" email="js@example.com" preferredFormat=" html "/>
```

### 7. Lists

The `list` pattern matches a whitespace-separated sequence of tokens; it contains a pattern that the sequence of individual tokens must match. The `list` pattern splits a string into a list of strings, and then matches the resulting list of strings against the pattern inside the `list` pattern.

For example, suppose we want to have a `vector` element that contains two floating point numbers separated by whitespace. We could use `list` as follows:

```xml
element vector {
    list { xsd:float, xsd:float }
}
```

Or suppose we want the `vector` element to contain a list of one or more floating point numbers separated by whitespace:

```xml
element vector {
    list { xsd:double+ }
}
```
Or suppose we want a path element containing an even number of floating point numbers:

```
element path {
    list { (xsd:double, xsd:double)+ }
}
```

### 8. Interleaving

In addition to the , and | connectors, RELAX NG provides the & connector. This is useful when child elements are allowed in any order. For example, the following would allow the card element to contain the name and email elements in any order:

```
element addressBook {
    element card {
        element name { text }
        & element email { text }
    }*
}
```

The & connector is called the interleave connector because of how it works with patterns that match more than one element. Suppose we want to write a pattern for the HTML head element which requires exactly one title element, at most one base element and zero or more style, script, link and meta elements and suppose we are writing a grammar pattern that has one definition for each element. Then we could define the pattern for head as follows:

```
head = element head { title & base? & style* & script* & link* & meta* }
```

Suppose we had a head element that contained a meta element, followed by a title element, followed by a meta element. This would match the pattern because it is an interleaving of a sequence of two meta elements, which match the child pattern

```
meta*
```

and a sequence of one title element, which matches the child pattern

```
title
```

The semantics of the & connector are that a sequence of elements matches a pattern $x \& y$ if it is an interleaving of a sequence that matches $x$ and a sequence that matches $y$. Note that this is different from the & connector in SGML: $A^* \& B$ matches the sequence of elements $A A B$ or the sequence of elements $B A A$ but not the sequence of elements $A B A$.

One special case of interleaving is very common: interleaving text with a pattern $p$ represents a pattern that matches what $p$ matches but also allows characters to occur as children. The mixed pattern is a shorthand for this.

```
mixed { p }
```

is short for

```
text & p
```

### 9. Modularity
9.1. Referencing external patterns

The `external` pattern can be used to reference a pattern defined in a separate file. The `external` keyword is followed by a quoted string specifying the URL of a file containing the pattern. The `external` pattern matches if the pattern contained in the specified URL matches. Suppose for example, you have a RELAX NG pattern that matches HTML inline content stored in `inline.rnc`:

```
start = inline
inline =
  (text
   | element code { inline }
   | element em { inline }
   # etc )
```

Then we could allow the `note` element to contain inline HTML markup by using `external` as follows:

```
element addressBook {
  element card {
    element name { text },
    element email { text },
    element note { external "inline.rnc" }?
  }*
}
```

For another example, suppose you have two RELAX NG patterns stored in files `pattern1.rnc` and `pattern2.rnc`. Then the following is a pattern that matches anything matched by either of those patterns:

```
element addressBook {
  element card {
    element name { text },
    element email { text },
    element note { external "pattern1.rnc" | external "pattern2.rnc" }?
  }*
}
```

9.2. Combining definitions

If a grammar contains multiple definitions with the same name, then the definitions must specify how they are to be combined into a single definition by using `|=` or `&=` instead of `=`. For example,

```
inline.class |= element bold { inline }
inline.class |= element italic { inline }
```

is equivalent to

```
inline.class =
  element bold { inline }
| element italic { inline }
```

When combining attributes, `&=` is typically used. For example,

```
start =
element addressBook {
  element card { card.attlist }*
}
card.attlist &= attribute name { text }
card.attlist &= attribute email { text }
```

is equivalent to

```
start =
element addressBook {
  element card { card.attlist }*
}
```
which is equivalent to

```xml
start =
  element addressBook {
    element card { card.attlist }*
  }
```

since combining attributes with `&` has the same effect as combining them with `,`

It is an error for the same name to be defined using both `&=` and `|=`. Note that the order of definitions within a grammar is not significant.

### 9.3. Merging grammars

The `include` directive allows grammars to be merged together. Along with definitions, a grammar pattern contain `include` directives. An `include` directive consists of the `include` keywords followed by a quoted string specifying the URL of a file containing a grammar pattern. The definitions in the referenced grammar pattern will be included in grammar pattern containing the `include` directive.

Both `|=` and `&=` are particularly useful in conjunction with `include`. For example, suppose a RELAX NG pattern `inline.rnc` provides a pattern for inline content, which allows `bold` and `italic` elements arbitrarily nested:

```xml
inline = inline.class*
inline.class =
  text
  | element bold { inline }
  | element italic { inline }
```

Another RELAX NG pattern could use `inline.rnc` and add `code` and `em` to the set of inline elements as follows:

```xml
include "inline.rnc"
start =
  element doc {
    element p { inline }*
  }
inline.class |=
  element code { inline }
  | element em { inline }
```

This would be equivalent to

```xml
inline = inline.class*
inline.class =
  text
  | element bold { inline }
  | element italic { inline }
start =
  element doc {
```
element p { inline }*
}
inline.class |=
   element code { inline }
| element em { inline }

which is equivalent to

inline = inline.class*
inline.class =
   text
   | element bold { inline }
   | element italic { inline }
   | element code { inline }
   | element em { inline }
start =
   element doc {
      element p { inline }*
   }

Note that it is allowed for one of the definitions of a name to use = rather than |= or &=. However, it is an error if there is more than one definition that does so.

The notAllowed pattern is useful when merging grammars. The notAllowed pattern never matches anything. Just as combining a pattern with empty using the , connector does not change the semantics of the pattern, so combining a pattern with notAllowed using the | connector also does not change the semantics of the pattern. It is typically used to allow an including pattern to specify additional choices with |=. For example, if inline.rnc were written like this:

inline =
   (text
      | element bold { inline }
      | element italic { inline }
      | inline.extra)*
inline.extra = notAllowed

then it could be customized to allow inline code and em elements as follows:

include "inline.rnc"
start =
   element doc {
      element p { inline }*
   }
inline.extra |=
   element code { inline }
| element em { inline }

9.4. Replacing definitions

The include directive may be followed by a list of definitions in braces. These definitions replace definitions in the included grammar pattern.

Suppose the file addressBook.rnc contains:

start =
   element addressBook {
      element card { cardContent }*
   }
cardContent =
   element name { text },


Suppose we wish to modify this pattern so that the card element contains an emailAddress element instead of an email element. Then we could replace the definition of cardContent as follows:

```xml
include "addressBook.rnc" {
  cardContent =
    element name { text },
    element emailAddress { text }
}
```

This would be equivalent to

```xml
start =
  element addressBook {
    element card { cardContent }*
  }
cardContent =
  element name { text },
  element emailAddress { text }
```

Definitions of start can be replaced in exactly the same way as other definitions.

## 10. Namespaces

### 10.1. Qualified names

The name following an element or attribute keyword may be qualified with a prefix. Each such prefix must be associated with a namespace URI using a namespace declaration. Namespace declarations occur at the beginning of the file, before the pattern. For example,

```xml
namespace ab = "http://www.example.com/address"
```

```xml
element ab:addressBook {
  element ab:card {
    element ab:name { text },
    element ab:email { text }
  }*
}
```

Multiple namespace declarations are allowed:

```xml
namespace a = "http://www.example.com/address"
namespace ab = "http://www.example.com/addressBook"
```

```xml
element ab:addressBook {
  element ab:card {
    element a:name { text },
    element a:email { text }
  }*
}
```

When an element or attribute pattern is matched against an element or attribute in the XML document, namespace URIs rather than prefixes are used. Thus,

```xml
namespace eg = "http://www.example.com"
```

```xml
element eg:foo { empty }
```
would match any of

```
<foo xmlns="http://www.example.com"/>
<e:foo xmlns:e="http://www.example.com"/>
<eg:foo xmlns:eg="http://www.example.com"/>
<example:foo xmlns:example="http://www.example.com"/>
```

but not any of

```
<foo/>
<eg:foo xmlns:eg="http://www.example.com/example"/>
<eg:foo xmlns:eg="http://WWW.EXAMPLE.COM"/>
<example:foo xmlns:example="http://www.example.net"/>
```

The prefix `xml` is predeclared as in XML: no namespace declaration is required for the `xml` prefix.

Namespace declarations and datatypes declarations can be mixed together at the beginning of the file in any order.

Unlike in XML, namespace declarations cannot be nested. A prefix is therefore always consistently bound to a single namespace URI throughout an entire file.

Namespace declarations apply only to the file in which they occur. A file referenced using `include` or `external` must declare whatever prefixes occur in that file; it cannot take advantage of the namespace declarations in the referencing file.

### 10.2. Default namespace

A single default namespace can be declared. For example,

```
default namespace = "http://www.example.com/address"

element addressBook {
    element card {
        element name { text },
        element email { text }
    }*
}
```

is equivalent to

```
namespace ab = "http://www.example.com/address"

element ab:addressBook {
    element ab:card {
        element ab:name { text },
        element ab:email { text }
    }*
}
```

As with XML, the default namespace does not apply to attribute patterns. Thus,

```
default namespace = "http://www.example.com/address"

element addressBook {
    element card {
        attribute name { text },
        attribute email { text }
    }*
}```
is equivalent to

namespace ab = "http://www.example.com/address"

element ab:addressBook {
    element ab:card {
        attribute name { text },
        attribute email { text }
    }*
}

and so will match

<addressBook xmlns="http://www.example.com">
    <card name="John Smith" email="js@example.com"/>
</addressBook>
or

<example:addressBook xmlns:example="http://www.example.com">
    <example:card name="John Smith" email="js@example.com"/>
</example:addressBook>

but not

<example:addressBook xmlns:example="http://www.example.com">
    <example:card example:name="John Smith" example:email="js@example.com"/>
</example:addressBook>

Default namespace declarations can be mixed with normal namespace declarations. For example,

default namespace = "http://www.example.com/address"
namespace ab = "http://www.example.com/addressBook"

element ab:addressBook {
    element ab:card {
        element name { text },
        element email { text }
    }*
}

is equivalent to

namespace a = "http://www.example.com/address"
namespace ab = "http://www.example.com/addressBook"

element ab:addressBook {
    element ab:card {
        element a:name { text },
        element a:email { text }
    }*
}

A default namespace declaration and a normal declaration for the same URI can be combined into a single declaration:

default namespace eg = "http://www.example.com"

is equivalent to
default namespace = "http://www.example.com"
namespace eg = "http://www.example.com"

If a file does not declare a default namespace and is referenced from another file using `include` or `external`, then it inherits the default namespace of the referencing file. Thus, if `address.rnc` contains

```xml
<element addressBook {
  <element card {
    <element name { text },
    <element email { text }
  }*
}

then

default namespace = "http://www.example.com/address"

external "address.rnc"

is equivalent to

default namespace = "http://www.example.com/address"

<element addressBook {
  <element card {
    <element name { text },
    <element email { text }
  }*
}

If a file does not declare a default namespace and is a top-level file that is not referenced from another file using `include` or `external`, then the default namespace is the absent or null namespace. Thus, a top-level file containing

```xml
<element foo { empty }
```

matches any of:

```xml
<foo xmlns=""/>
<foo/>
```

but not any of:

```xml
<foo xmlns="http://www.example.com"/>
<e:foo xmlns:e="http://www.example.com"/>
```

A namespace declaration can refer to the null or absent namespace by using a namespace URI of "" (like with the `xmlns` attribute). A file can ensure that its default namespace will be the null or absent namespace and will not be inherited from any referencing file by explicitly declaring the default namespace as "":

```xml
default namespace = ""
```

### 11. Name classes

In all the examples up to now, the `element` and `attribute` keywords have been followed by a name, possibly qualified with a prefix. However, in general, the `element` and `attribute` keywords are
followed by a name-class. A name is one particular simple kind of a name-class: a name specifies a name-class with that name as its only member. An element or attribute pattern will only match an element or attribute in the XML document if the name of the element of attribute is a member of the name-class in the pattern. Another simple kind of name-class is * which contains all names, regardless of their local name and namespace URI. For example, the following pattern matches any well-formed XML document:

```
start = anyElement
anyElement =
    element * {
        (attribute * { text } |
        | text |
        | anyElement)*
    }
```

A name-class ns:* contains all names with the namespace URI declared for the prefix ns.

Name-classes can be combined using the | connector. A name-class x | y contains the union of x and y. In other words, a name is a member of x | y if it is a member of x and/or a member of y.

Name-classes can also be combined using the - connector. A name-class x - y contains the difference of x and y. In other words, a name is a member of x - y if it is a member of x but not a member of y. The left-hand name-class to be combined with the - connector must be a * or ns:* name class. As with patterns, there is no implicit precedence between connectors and parentheses must be used to make precedence explicit. For example,

```
namespace local = ""
default namespace ex = "http://www.example.com"

element card {
    attribute * - (ex:* | local:*) { text }*,
    text
}
```

would allow the card element to have any number of namespace-qualified attributes provided that they were qualified with namespace other than that of the card element.

Note that an attribute pattern matches a single attribute even if it has a name-class that contains multiple names. To match zero or more attributes, * must be used.

Some schema languages have a concept of lax validation, where an element or attribute is validated against a definition only if there is one. We can implement this concept in RELAX NG with name classes that use the - connector. Suppose, for example, we wanted to allow an element to have any attribute with a qualified name, but we still wanted to ensure that if there was an xml:space attribute, it had the value default or preserve. It wouldn't work to use

```
element example {
    attribute * { text }*,
    attribute xml:space { "default" | "preserve" }?
}
```

because an xml:space attribute with a value other than default or preserve would match

```
attribute * { text }
```

even though it did not match
attribute xml:space { "default" | "preserve" }

The solution is to use the - connector:

element example {
    attribute * - xml:space { text }*,
    attribute xml:space { "default" | "preserve" }?
}

Note that definitions cannot define name-classes; they can only define patterns.

12. Internationalization

In the absence of externally supplied information, a RELAX NG Compact Syntax file will be assumed to be in Unicode using either the UTF-8 or UTF-16 encoding. RELAX NG processors can automatically choose between UTF-8 and UTF-16 by using the byte order mark that almost all text editors automatically put at the beginning of a UTF-16 file. Although particular RELAX NG processors may allow you to use a legacy encoding, it is best to use UTF-8 or UTF-16 for interchange.

Unicode characters can be entered using an escape sequence of the form \x{N}, where N is the hex code of the character. For example, \x{A9} can be used to represent the copyright sign. Unlike XML character references, the \x escape sequence can be used anywhere, even in names of elements, attributes and definitions. For example,

element \xE14\xE35 { empty }

13. Annotations

When a RELAX NG pattern is to be used for purposes other than validation, it is often desirable to be able to annotate it with additional information. For example, if a RELAX NG pattern is intended to be read by a human, it is desirable to be able to annotate it with documentation; when a RELAX NG pattern is converted into another schema language, it is desirable to be able to annotate it with information to guide the conversion.

13.1. Applying annotations

RELAX NG allows an annotation to be placed in square brackets immediately preceding the construct to be annotated. Abstractly, an annotation is a fragment of XML consisting of zero or more attributes followed by zero or more elements. An attribute is written in a similar way to XML. For example,

namespace doc = "http://www.example.com/documentation"

[doc:href="address.html#addressBook"]
element addressBook {
    [doc:href="address.html#card"]
element card {
        [doc:href="address.html#name"]
element name { text },
        [doc:href="address.html#email"]
element email { text }
    }*
}

An attribute in an annotation must be qualified with a prefix; the prefix must be declared in a
namespace declaration with a non-empty URI.

An element in an annotation consists of the element name followed by the attributes and children in square brackets.

```xml
namespace a = "http://www.example.com/annotation"

element addressBook {
  [ a:documentation [ xml:lang="en" "Information about a single address." ] ]
  element card {
    element name { text },
    element email { text }
  }*
}
```

The constructs that can be annotated are patterns, name classes, parameters, definitions and the include directive.

### 13.2. String literal syntax

String literals that are delimited with ' or " are not allowed to contain unescaped newlines. An escaped newline \x{A} can be used to include a newline in a literal. Alternatively, string literals can be delimited with triple quotes ('''' or """") as in Python. Such string literals are allowed to contain unescaped newlines. String literals can be concatenated using ~. For example,

```
"A string can contain both " ~ 'and ",'
```

is equivalent to

```
"""A string can contain both ' and ",""
```

and

```
"Line 1\x{A}" ~
"Line 2"
```

is equivalent to

```
'''Line 1
Line 2'''
```

### 13.3. Documentation syntax

A companion specification, RELAX NG DTD Compatibility [Compatibility], defines annotations to implement some features of XML DTDs. It also provides a documentation element for use as an annotation. There is a special shorthand syntax for this. Comments starting with ## are equivalent to an annotation consisting of a documentation element from the RELAX NG DTD Compatibility namespace. For example,

```xml
## Represents an address book.

element addressBook {
  element card {
    element name { text },
    element email { text }
  }*
}
```
is equivalent to

```xml
namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
[
    a:documentation [
        "Represents an\x{A}" ~
        "address book."
    ]
] element addressBook {
    element card {
        element name { text },
        element email { text }
    }*
}
```

### 13.4. Grouping definitions

RELAX NG also provides a `div` construct which allows an annotation to be applied to a group of definitions in a grammar. For example, you might want to divide up the definitions of the grammar into modules:

```xml
namespace m = "http://www.example.com/module"
[
    m:name = "inline"
] div {
    code = pattern
    em = pattern
    var = pattern
}
[
    m:name = "block"
] div {
    p = pattern
    ul = pattern
    ol = pattern
}
```

This would allow you easily to generate variants of the grammar based on a selection of modules.

### 14. Nested grammars

There is no prohibition against nesting grammar patterns. A name refers to the definition from the innermost containing grammar pattern. There is also a `parent` pattern that escapes out of the current grammar and references a definition from the parent of the current grammar. A `parent` pattern consists of the `parent` keyword followed by the name of the definition.

Imagine the problem of writing a pattern for tables. The pattern for tables only cares about the structure of tables; it doesn't care about what goes inside a table cell. First, we create a RELAX NG pattern `table.rnc` as follows:

```xml
cell.content = notAllowed
start =
    element table {
        element tr {
            element td { cell.content }+
        }+
    }
```
Patterns that include `table.rnc` must redefine `cell.content`. By using a nested grammar pattern containing a parent pattern, the including pattern can redefine `cell.content` to be a pattern defined in the including pattern's grammar, thus effectively importing a pattern from the parent grammar into the child grammar:

```
start =
  element doc {
    (element p { inline } |
      grammar {
        include "table.rnc" {
          cell.content = parent inline
        }
      })*
  }
inline =
  (text | element em { inline }))*
```

Of course, in a trivial case like this, there is no advantage in nesting the grammars: we could simply have included `table.rnc` within the outer grammar pattern. However, when the included grammar has many definitions, nesting it avoids the possibility of name conflicts between the including grammar and the included grammar.

15. ✡ Non-restrictions

RELAX NG does not require patterns to be "deterministic" or "unambiguous".

Suppose we wanted to write the email address book in HTML, but use class attributes to specify the structure:

```html
element html {
  element head {
    element title { text }
  },
  element body {
    element table {
      attribute class { "addressBook" },
      element tr {
        attribute class { "card" },
        element td {
          attribute class { "name" },
          mixed {
            element span {
              attribute class { "givenName" },
              text
            }?,
            element span {
              attribute class { "familyName" },
              text
            }?
          }
        }
      },
      element td {
        attribute class { "email" },
        text
      }
    }+
  }
}
```
This would match a document such as:

```html
<html>
<head>
  <title>Example Address Book</title>
</head>
<body>
  <table class="addressBook">
    <tr class="card">
      <td class="name">
        <span class="givenName">John</span>
        <span class="familyName">Smith</span>
      </td>
      <td class="email">js@example.com</td>
    </tr>
  </table>
</body>
</html>
```

but not:

```html
<html>
<head>
  <title>Example Address Book</title>
</head>
<body>
  <table class="addressBook">
    <tr class="card">
      <td class="name">
        <span class="givenName">John</span>
        <!-- Note the incorrect class attribute -->
        <span class="givenName">Smith</span>
      </td>
      <td class="email">js@example.com</td>
    </tr>
  </table>
</body>
</html>
```

16. Advanced features

This section describes advanced features, which most users will probably not need. These features exist primarily to ensure equivalence between the XML and compact syntaxes.

16.1. inherit keyword

Namespace inheritance is in fact a little more flexible than described in Section 10.2, “Default namespace”.

The inherited namespace need not be the same as the default namespace. The inherited namespace is referenced by using a namespace declaration that associates a prefix with the special keyword inherit.

So for example, if `address.rnc` contains

```relax-ng
namespace ab = inherit
element ab:addressBook {
```

RELAX NG Compact Syntax Tutorial
http://relaxng.org/compact-tutorial-20030326.html

24 of 28 2/7/2009 12:37 PM
element ab:card {
    element ab:name { text },
    element ab:email { text }
}*
}

then

default namespace = "http://www.example.com/address"

external "address.rnc"

is equivalent to

namespace ab = "http://www.example.com/address"

element ab:addressBook {
    element ab:card {
        element ab:name { text },
        element ab:email { text }
    }*
}

When a file is used as a top-level file rather than being referenced by external or include, then its inherited namespace is the null or absent namespace. We can now describe more simply what happens when a file does not declare the default namespace: what happens is simply that a declaration of

default namespace = inherit

is assumed.

Each include and external can independently determine what namespace is inherited by the referenced file by following the URL with inherit = prefix. Thus, if address.rnc contains

namespace ab = inherit

element ab:addressBook {
    element ab:card {
        element ab:name { text },
        element ab:email { text }
    }*
}

then

namespace a = "http://www.example.com/address"

external "address.rnc" inherit = a

is equivalent to

namespace ab = "http://www.example.com/address"

element ab:addressBook {
    element ab:card {
        element ab:name { text },
        element ab:email { text }
    }*
}

If an external or include does not specify inherit = prefix, then the referenced file inherits the
default namespace of the referencing file.

A prefix used in the name of an attribute or element in an annotation cannot be associated with the `inherit` keyword.

### 16.2. Grammar-level annotations

Grammar patterns can contain element annotations interspersed among the definitions. For example,

```xml
namespace x = "http://www.example.com"
start = foo
x:entity [ name="picture" systemId="picture.jpeg" notation="jpeg" ]
foo = element foo { empty }
```

In the XML syntax, such element annotations will be children of the `grammar` element.

### 16.3. Annotations

The `>>` connector creates a pattern or a name-class by combining a pattern or a name-class with an annotation element. In the XML syntax, such element annotations will appear as following siblings of the element representing the pattern or name-class. For example,

```xml
namespace eg = "http://www.example.com"

element foo {
    text >> x[] >> y[]
}
```

is equivalent to the XML

```
<element name="foo" xmlns:eg="http://www.example.com">
    <text/>
    <eg:x/>
    <eg:y/>
</element>
```

### 17. Further information

The definitive specification of RELAX NG is [Specification], which uses the XML syntax. [Compact] is the definitive specification for the compact syntax, which defines the compact syntax by mapping it to the XML syntax.

A tutorial for the XML syntax is available separately [Tutorial].


### A. List of keywords

The complete list of keywords is:
B. Comparison with XML DTDs

RELAX NG provides functionality that goes beyond XML DTDs. In particular, RELAX NG

- provides a choice of an XML syntax or a compact non-XML syntax to represent schemas
- supports datatyping
- integrates attributes into content models
- supports XML namespaces
- supports unordered content
- supports context-sensitive content models

ID/IDREF validation is not provided by RELAX NG; however, it is provided by a companion specification, RELAX NG DTD Compatibility [Compatibility]. Comprehensive support for cross-reference checking is planned for a future specification.

RELAX NG does not support features of XML DTDs that involve changing the infoset of an XML document. In particular, RELAX NG

- does not allow defaults for attributes to be specified; however, this is allowed by RELAX NG DTD Compatibility [Compatibility]
- does not allow entities to be specified
- does not allow notations to be specified
- does not specify whether whitespace is significant

Also, whereas an XML document can associate itself with a DTD using a DOCTYPE declaration, RELAX NG does not define a way for an XML document to associate itself with a RELAX NG pattern.

References
Compact
Compatibility
Guidelines
RELAX
  MURATA Makoto. *RELAX (Regular Language description for XML)*. INSTAC (Information Technology Research and Standardization Center), 2001.
Specification
TREX
Tutorial
W3C XML Schema Datatypes