

# Relax NG Compact Syntax (RNC)



A new way to specify XML document types

John W. Shipman

2009-10-09 12:03

## Abstract

Describes the Relax NG Compact Syntax, a schema language for specifying XML document types.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to [tcc-doc@nmt.edu](mailto:tcc-doc@nmt.edu).

## Table of Contents

1. What is RNC? .....	1
1.1. The schema wars .....	2
2. A small example .....	3
3. Named patterns: factoring the schema .....	3
4. The content patterns .....	4
5. The definition patterns .....	6
6. The <code>xsd:</code> datatypes .....	7
6.1. The basic <code>xsd:</code> types .....	8
6.2. Parameters to the <code>xsd:</code> types .....	11
6.3. Regular expression syntax for <code>xsd:</code> .....	13

## 1. What is RNC?

Relax NG is a *schema language*, that is, it defines an XML *document type*. The reader is assumed to be familiar with these concepts:

- XML, the Extensible Markup Language. For more information see the TCC Help System page for XML<sup>3</sup>.
- The difference between a *well-formed* XML document and a *valid* one. The difference is that a well-formed document need only follow the general rules for XML, while a valid file must conform to a schema as well as being well-formed.

Indeed, the whole point of a schema is to provide a way to mechanically validate the structure of an XML file.

<sup>1</sup> <http://www.nmt.edu/tcc/help/pubs/rnc/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/pubs/rnc/rnc.pdf>

<sup>3</sup> <http://www.nmt.edu/tcc/help/xml/>

## 1.1. The schema wars

XML has a variety of ways to express a schema. To understand why this is, and why we recommend RNC, here is a brief history of the Schema Wars:

### 1.1.1. DTD: Document Type Definition

Historically, the first way to express a schema for an XML document type was through a Document Type Definition (DTD), which is part of the XML standard<sup>4</sup>.

Such schemas are universally supported today, but the DTD syntax is limited in what it can express. Many users also wanted to see more features for the validation of common content types (e.g., floating point numbers or URLs). Also, because of XML's heritage in SGML, DTD schemas use their own syntax, which does not resemble XML.

### 1.1.2. XSchema: The W3C weighs in

XML Schema<sup>5</sup> is a W3C Recommendation. There are three parts to it:

- a. Part 0: Primer<sup>6</sup> is an introductory section.
- b. Part 1: Structures<sup>7</sup> covers XML Schema's method for describing the structure of documents. A schema written in this method is itself a valid XML document, which addresses one of the objections to DTDs. However, it is quite complex.
- c. Part 2: Datatypes<sup>8</sup> proposed a large, rich set of datatypes to assist applications in validating content.

### 1.1.3. RNG: Relax NG

Relax NG<sup>9</sup> was a reaction against the complexity of XML Schema. Written by James Clark and Makoto Murata, this standard is maintained at *Organization for the Advancement of Structured Information Standards (OASIS)*<sup>10</sup>, an open-source standards organization. It is being developed as an ISO standard, ISO/IEC 19757-2<sup>11</sup>.

The author considers Relax NG to be a nearly perfect schema language. It has sufficient structural complexity to handle a huge range of applications, but is easy to learn. Most implementations use Part 2 of the XML Schema work to provide a rich and standardized set of datatypes. Like XML Schema, a Relax NG schema is itself a valid XML document.

### 1.1.4. RNC: Relax NG Compact Format

The author feels that although the XML syntax for Relax NG is “almost perfect,” James Clark made a quantum improvement when he invented RNC (Relax NG Compact Syntax). Although an RNC schema is not an XML data type, it is much less verbose.

The document you are now reading is a compact reference to RNC syntax. The author recommends these documents for full details:

---

<sup>4</sup> <http://www.w3.org/TR/REC-xml/>

<sup>5</sup> <http://www.w3.org/XML/Schema>

<sup>6</sup> <http://www.w3.org/TR/xmlschema-0/>

<sup>7</sup> <http://www.w3.org/TR/xmlschema-1/>

<sup>8</sup> <http://www.w3.org/TR/xmlschema-2/>

<sup>9</sup> [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=relax-ng](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=relax-ng)

<sup>10</sup> <http://www.oasis-open.org/>

<sup>11</sup> <http://xml.coverpages.org/ni2002-05-24-c.html>

- RNC Tutorial, Working Draft, 26 March 2003<sup>12</sup>
- RNC Committee Specification, 21 November 2002<sup>13</sup>

## 2. A small example

---

Suppose you wish to define a small XML document type to keep track of hiking trails inside a park.

You want to record the name of the park, and for each trail you want to specify whether it is an easy, medium, or hard climb, and the distance traveled. Here is an example of such a document showing a park with two trails:

```
<park name="Lincoln Natural Forest">
  <trail distance="3400" climb="medium">Canyon Trail</trail>
  <trail climb="easy" distance="1200">Pickle Madden Trail</trail>
</park>
```

Here's a small RNC schema for this document type:

```
element park
{ attribute name { text },
  element trail
  { attribute climb { "easy" | "medium" | "hard" },
    attribute distance { text },
    text
  }*
}
```

The `element` pattern specifies the content of an element such as `<park>...</park>`. The `attribute` pattern specifies what attributes can occur inside an element. The keyword `text` matches any text. The `**` suffix specifies that the previous element can occur any number of times, including zero.

## 3. Named patterns: factoring the schema

---

Nesting `element` patterns inside other `element` patterns, as the example above shows, is not the best way to write a complex schema. This style is hard to understand for the same reason monolithic programs are hard to understand: too much complexity in one place.

Fortunately, RNC provides an easy, natural way to factor the schema into small, easily understood parts. When you write a schema, you can define *named patterns*, and assemble them like building blocks into a complete schema.

This RNC construct defines a named pattern:

```
name = pattern
```

where the *name* follows XML naming rules (letters, digits, underscore, dollar sign, hyphen, and period).

Here's a rewrite of the example pattern above using named patterns. We start by defining a named pattern named `start` that describes the pattern of the whole document:

```
## Describes one park; the root element.
start = park
```

<sup>12</sup> <http://relaxng.org/compact-tutorial-20030326.html>

<sup>13</sup> <http://relaxng.org/compact-20021121.html>

This defines the special pattern named `start` as having the same structure as another pattern named `park`, which we define next:

```
park = element park
{ attribute name { text },
  trail*
}
```

The four lines above define the named pattern `park` as containing an element named `park`, which has a required text attribute called `name`, and contains zero or more `trail` patterns.

Note that `park` is used in two different ways in the lines above. It is the name of a pattern we're defining, but it is also the name of an element. We don't have to use the same name in both places. However, in many cases, if a named pattern is the same as an element, using the same name for both is clear.

RNC schemas use two different *namespaces*: pattern names exist only inside the schema file, but element names correspond to the names of elements in an XML document.

Next we define the `trail` pattern:

```
## Each trail element describes one trail in the park.
trail = element trail
{ attribute climb { "easy" | "medium" | "hard" },
  attribute distance { text },
  text
}
```

The lines above define the content of the named `trail` pattern as a single `trail` element, which has two attributes named `climb` and `distance`. The content of the `trail` element can be any text.

## 4. The content patterns

---

RNC notation is an expression syntax. We will introduce its pieces one by one, but an RNC file is in general rather like a syntax description in BNF (Backus-Naur Form): it follows the rules of a context-free grammar.

Operators have no innate precedence. If you use two different operators in an expression, you must parenthesize part of the expression to force precedence.

### **text**

A pattern that matches text (not containing any tags). For example, this pattern specifies that a `horse` element can contain only text:

```
element horse { text }
```

### ***p*\***

The pattern *p* can occur zero or more times. For example, this schema fragment says that a `corral` element can contain zero or more `horse` elements:

```
element corral
{ element horse { text }*
}
```

### ***p*+**

The pattern *p* must occur one or more times. For example:

```
element forest {
  { element tree { text }+ }
```

### ***p*?**

Pattern *p* is optional. Example:

```
element verification {
  element claim { text },
  element result { text }?
}
```

This pattern says that a `verification` element must contain exactly one `claim`, optionally followed by a `result`.

### **xsd:datatype**

Specifies a predefined data type defined in XML Schema Part 2: Datatypes<sup>14</sup>. For a list of these datatypes, see Section 6, "The `xsd: datatypes`" (p. 7).

For example, this pattern specifies a `head-count` element whose content must be an integer greater than zero:

```
element head-count { xsd:positiveInteger }
```

### ***p* | *q***

A pattern that matches either *p* or *q*. For example:

```
element vehicle {
  attribute vkind { "car" | "bus" | "truck" } ... }
```

This allows only three valid `vkind` attributes in a `vehicle` element: `vkind="car"`, `vkind="bus"`, or `vkind="truck"`.

You can use such an enumeration to restrict the content of elements as well as attributes.

### ***p*, *q***

Matches *p* followed by *q*.

### ***p* & *q***

Matches pattern *p* and pattern *q*, but those patterns can occur in any order.

For example, this pattern says that a `part` element must contain exactly one each of the patterns `itemName`, `unitCost`, `amount`, and `stockNumber`, but they can occur in any order:

```
element part { itemName & unitCost & amount & stockNumber }
```

This kind of pattern is called an *interleave*.

### ***name* |= *p***

Defines the given named pattern *name* as *o* | *p*, where *o* is the old content of *name*.

### ***name* &= *p***

Defines the given named pattern *name* as *o* & *p*, where *o* is the old content of *name*.

### **notAllowed**

A special keyword that never matches. You can use this pattern as a placeholder for other content to be added later using the `&=` or `|=` operator.

<sup>14</sup> <http://www.w3.org/TR/xmlschema-2/>

\*

In a context where a name is expected, \* matches any name.

***n1 - n2***

If *n1* and *n2* are name classes, the result is all the names in class *n1* except for the names in *n2*. For example:

```
element notes {
  attribute * - (xsl:* | fo:*) { text }*,
  text
}
```

This specifies that a `notes` element can have any attribute name that is not in namespace `xsl:` or `fo:`, that the attributes can have any values, and that the `notes` element's content can be any text.

## 5. The definition patterns

---

To write an RNC schema, use `start` construct to define what content can go into the root element. The remainder of the schema can be placed inside the `start` construct, or you can break the schema up into named parts and use references to those names to connect the parts of the schema to the `start` construct.

**namespace *ns* = "URI"**

Define a namespace prefix *ns* using the given URI. For example:

```
namespace xsl = "http://www.w3.org/1999/XSL/Transform"
```

After the above definition, a tag name such as `xsl:comment` would match only tags in the namespace given by that URI.

**default namespace = "URI"**

Define the default namespace.

**grammar { *definition* ... }**

The top level of schema definitions. Should include a `start` definition.

**start = *definition***

Declares the content of the root element of the document type.

***name* = *definition***

Names a pattern. You can use such names anywhere a pattern or definition is allowed.

**element *name* { *p* }**

Specifies that in this document type, the content of element *name* must match *p*.

An `element` construct is just another pattern, which you can nest. For example:

```
element name {
  element first { text },
  element last { text }
}
```

This specifies that a `name` element must contain a `first` element followed by a `last` element.

You can also use the postfix operators `*`, `+`, and `?` after the closing brace, for example:

```
element q-and-a {
  element question { text },
```

```
element answer { text }*
```

This rule defines the content of a `q-and-a` element as one `question` element followed by zero or more `answer` elements.

### **attribute *aname* { *p* }**

Used inside an `element` construct to specify that the element must have an attribute named *aname*.

### **list { *p* }**

Matches a whitespace-separated list of patterns that match *p*.

For example, this pattern would require that a `scores` element contain a space-separated list `scores="i0 f0 i1 f1 ..."`, where each *i<sub>k</sub>* is an integer, and each *f<sub>k</sub>* is a float (type `xsd:double`):

```
attribute scores {  
  list { (xsd:int, xsd:double)+ }  
}
```

### **mixed { *p* }**

Matches any mixture of `text` and pattern *p*.

### **empty**

Used to declare that an element has no content. For example:

```
element newPage { empty }
```

### **external "*filename*"**

Matches the RNC patterns defined in file *filename*.

### **include "*filename*"**

Merges with the current grammar the grammar from the named file.

### **include "*filename*" { *defs* }**

Merge the named file into the current grammar, but replace any definitions from that file with *defs*, the definitions between the braces.

### **# *comment***

"#" is the comment character in RNC. Any characters on a line following # will be ignored.

### **## *annotation***

Lines starting with ## are special comments that are treated as *annotations* to the material that follows. Some tools may allow you to retrieve the annotations for a given pattern.

## 6. The `xsd:` datatypes

Although Relax NG is an alternative to the XSchema standard, it does use an important part of XSchema: the predefined set of standard datatypes.

This defines a plethora of types, including common ones like `xsd:string`, `xsd:int`, `xsd:unsigned-Int`, and `xsd:double` (which is a long floating point number).

You can supply additional parameters to these types by following the type name with a parameter set in this syntax:

```
xsd:datatype { pname=pvalue ... }
```

The *pname* is the parameter name, and the *pvalue* the corresponding value.

For example, this pattern would match any string whose length is between 7 and 25 characters:

```
xsd:string { minLength="7" maxLength="25" }
```

## 6.1. The basic xsd: types

Here is a list of the basic datatypes in the `xsd:` namespace. Many of the basic types are *derived* from other types. For example, the `decimal` type includes numbers with decimal points, such as "3.14". The `integer` type is derived from `decimal`, including only those values without decimal points, such as "42". To see what parameters are allowed for a given type, see Section 6.2, "Parameters to the `xsd:` types" (p. 11).

### anyURI

The data must conform to the syntax of a *Uniform Resource Identifier (URI)*, as defined in RFC 2396<sup>15</sup> as amended by RFC 2732<sup>16</sup>. Example: "http://www.nmt.edu/tcc/" is the URI for the New Mexico Tech Computer Center's index page.

### base64Binary

Represents a sequence of binary octets (bytes) encoded according to RFC 2045<sup>17</sup>, the standard defining the MIME types (look under "6.8 Base64 Content-Transfer-Encoding").

### boolean

A Boolean true or false value. Representations of true are "true" and "1"; false is denoted as "false" or "0".

### byte

A signed 8-bit integer in the range [-128, 127]. Derived from the `short` datatype.

### date

Represents a specific date. The syntax is the same as that for the date part of `dateTime`, with an optional time zone indicator. Example: "1889-09-24".

### dateTime

Represents a specific instant of time. It has the form `YYYY-MM-DDThh:mm:ss` followed by an optional time-zone suffix.

`YYYY` is the year, `MM` is the month number, `DD` is the day number, `hh` the hour in 24-hour format, `mm` the minute, and `ss` the second (a decimal and fraction are allowed for the seconds part).

The optional zone suffix is either "Z" for Universal Coordinated Time (UTC), or a time offset of the form "`±hh:mm`", giving the difference between UTC and local time in hours and minutes.

Example: "2004-10-31T21:40:35.5-07:00" is a time on Halloween 2004 in Mountain Standard time. The equivalent UTC would be "2004-11-01T04:40:35.5Z".

### decimal

Any base-10 fixed-point number. There must be at least one digit to the left of the decimal point, and a leading "+" or "-" sign is allowed. Examples: "42", "-3.14159", "+0.004".

### double

A 64-bit floating-point decimal number as specified in the IEEE 754-1985 standard. The external form is the same as the `float` datatype.

<sup>15</sup> <http://www.ietf.org/rfc/rfc2396.txt>

<sup>16</sup> <http://www.ietf.org/rfc/rfc2732.txt>

<sup>17</sup> <http://www.ietf.org/rfc/rfc2045.txt>

## duration

Represents a duration of time, as a composite of years, months, days, hours, minutes, and seconds. The syntax of a duration value has these parts:

- If the duration is negative, it starts with "-".
- A capital "P" is always included.
- If the duration has a years part, the number of years is next, followed by a capital "Y".
- If there is a months part, it is next, followed by capital "M".
- If there is a days part, it is next, followed by capital "D".
- If there are any hours, minutes, or seconds in the duration, a capital "T" comes next; otherwise the duration ends here.
- If there is an hours part, it is next, followed by capital "H".
- If there is a minutes part, it is next, followed by capital "M".
- If there is a seconds part, it is next, followed by capital "S". You can use a decimal point and fraction to specify part of a second.

Missing parts are assumed to be zero. Examples: "P1347Y" is a duration of 1347 Gregorian years; "P1Y2MT2H5.6S" is a duration of one year, two months, two hours, and 5.6 seconds.

## float

A 32-bit floating-point decimal number as specified in the IEEE 754-1985 standard. Allowable values are the same as in the `decimal` type, optionally followed by an exponent, or one of the special values "INF" (positive infinity), "-INF" (negative infinity), or "NaN" (not a number).

The exponent starts with either "e" or "E", optionally followed by a sign, and one or more digits.

Example: "6.0235e-23".

## gDay

A day of the month in the Gregorian calendar. The syntax is "--DD" where *DD* is the day of the month. Example: the 27th of each month would be represented as "--27".

## gMonth

A month number in the Gregorian calendar. The syntax is "--MM--", where *MM* is the month number. For example, "--06--" represents the month of June.

## gMonthDay

A Gregorian month and day as "--MM-DD". Example: "--07-04" is the Fourth of July.

## gYear

A Gregorian year, specified as *YYYY*. Example: "1889".

## gYearMonth

A Gregorian year and month. The syntax is *YYYY-MM*. Example: "1995-08" represents August 1995.

## hexBinary

Represents a sequence of octets (bytes), each given as two hexadecimal digits. Example: "0047dedbef" is five octets.

## ID

A unique identifier as in the ID attribute type from the XML standard<sup>18</sup>.

<sup>18</sup> <http://www.xml.com/axml/testaxml.htm>

Derived from the `NCName` datatype.

### **IDREF, IDREFS**

An `IDREF` value is a reference to a unique identifier as defined under attribute types in the XML standard<sup>19</sup>. An `IDREFS` value is a space-separated sequence of such references.

Derived from the `NCName` datatype.

### **int**

Represents a 32-bit signed integer in the range [-2,147,483,648, 2,147,483,647]. Derived from the `long` datatype.

### **integer**

Represents a signed integer. Values may begin with an optional "+" or "-" sign. Derived from the `decimal` datatype.

### **language**

One of the standardized language codes defined in RFC 1766<sup>20</sup>. Example: "fj" for Fijian. Derived from the `token` type.

### **long**

A signed, extended-precision integer; at least 18 digits are guaranteed. Derived from the `integer` datatype.

### **Name**

A name as defined in the XML standard<sup>21</sup>. The first character can be a letter or underbar "\_", and the remaining characters may be letters, underbars, hyphen "-", period ".", or colon ":".

Derived from the `token` datatype.

### **NCName**

The local part of a qualified name. See the `NCName` definition in the document Namespaces in XML<sup>22</sup>.

Derived from the `name` datatype.

### **negativeInteger**

Represents an integer less than zero. Derived from the `nonPositiveInteger` datatype.

### **NMTOKEN, NMTOKENS**

Any sequence of name characters, defined in the XML standard<sup>23</sup>: letters, underbars "\_", hyphen "-", period ".", or colon ":".

A `NMTOKENS` data value is a space-separated sequence of `NMTOKEN` values.

Derived from the `NMTOKEN` datatype.

### **nonNegativeInteger**

An integer greater than or equal to zero. Derived from the `integer` datatype.

### **nonPositiveInteger**

An integer less than or equal to zero. Derived from the `integer` datatype.

### **normalizedString**

This datatype describes a "normalized" string, meaning that it cannot include newline (LF), return (CR), or tab (HT) characters.

<sup>19</sup> <http://www.xml.com/axml/testaxml.htm>

<sup>20</sup> <http://www.ietf.org/rfc/rfc1766.txt>

<sup>21</sup> <http://www.xml.com/axml/testaxml.htm>

<sup>22</sup> <http://www.w3.org/TR/1999/REC-xml-names-19990114/>

<sup>23</sup> <http://www.xml.com/axml/testaxml.htm>

Derived from the `string` type.

### **positiveInteger**

An extended-precision integer greater than zero. Derived from the `nonNegativeInteger` datatype.

### **QName**

An XML qualified name, such as `"xsl:stylesheet"`.

### **short**

A 16-bit signed integer in the range `[-32,768, 32,767]`. Derived from the `int` datatype.

### **string**

Any sequence of zero or more characters.

### **time**

A moment of time that repeats every day. The syntax is the same as that for `dateTime`, omitting everything up to and including the separator `"T"`. Examples: `"00:00:00"` is midnight, and `"13:04:00"` is an hour and four minutes after noon.

### **token**

The values of this type represent tokenized strings. They may not contain newline (LF) or tab (HT) characters. They may not start or end with whitespace. The only occurrences of whitespace allowed inside the string are single spaces, never multiple spaces together. Derived from `normalizedString`.

### **unsignedByte**

An unsigned 16-bit integer in the range `[0, 255]`. Derived from the `unsignedShort` datatype.

### **unsignedInt**

An unsigned 32-bit integer in the range `[0, 4,294,967,295]`. Derived from the `unsignedLong` datatype.

### **unsignedLong**

An unsigned, extended-precision integer. Derived from the `nonNegativeInteger` datatype.

### **unsignedShort**

An unsigned 16-bit integer in the range `[0, 65,535]`. Derived from the `unsignedInt` datatype.

## 6.2. Parameters to the `xsd:` types

You can apply parameters to your datatypes to further constrain what values are legal. Parameters come in four groups:

- The length group. The `length` parameter specifies the exact length of valid values, no more and no less. The `minLength` specifies the minimum length, and the `maxLength` parameter specifies the maximum length.

For example, a value declared as

```
xsd:string { length="4" }
```

can have any characters in it, so long as there are exactly four of them. A value declared this way

```
xsd:integer { minLength="5" maxLength="9" }
```

must be an integer with between 5 and 9 characters. To specify that a string cannot exceed 20 characters:

```
xsd:string { maxLength="20" }
```

- The range group. These attributes restrict the values that a number can have. There are four: `minInclusive="N"` means the number must be greater than or equal to *N*, and `minExclusive="N"` means the number must be greater than *N*. Similarly, `maxInclusive="N"` restricts values to those less than or equal to *N*, and `maxExclusive="N"` means values must be strictly less than *N*.

For example, to specify a decimal float that is greater than 0.0 but less than or equal to 10.0:

```
xsd:decimal { minExclusive="0.0" maxInclusive="10.0" }
```

- The digits group. The `totalDigits="N"` attribute limits the total number of digits in a number to *N*. The `fractionDigits="N"` attribute limits the total number of digits after the decimal point to *N*.

Here's an example. Suppose you want to restrict the values of a number to have no more than three digits after the decimal. You could do this with:

```
xsd:decimal { fractionDigits="3" }
```

- The `pattern="P"` attribute allows you to restrict values using a regular expression syntax. See Section 6.3, “Regular expression syntax for xsd: ” (p. 13).

Here is a table showing the attribute groups that are allowed for each of the basic types.

Type	pattern	length, minLength, maxLength	minInclusive, minExclusive, maxExclusive maxInclusive	fractionDigits, totalDigits
anyURI	x	x		
base64Binary	x	x		
boolean	x			
byte	x		x	x
date	x		x	
dateTime	x		x	
decimal	x		x	x
double	x		x	
duration	x		x	
float	x		x	
gDay	x		x	
gMonth	x		x	
gMonthDay	x		x	
gYear	x		x	
gYearMonth	x		x	
hexBinary	x	x		
ID	x	x		
IDREF	x	x		
IDREFS	x	x		
int	x		x	x

Type	pattern	length, minLength, maxLength	minInclusive, minExclusive, maxExclusive maxInclusive	fractionDigits, totalDigits
integer	x		x	x
language	x	x		
long	x		x	x
Name	x	x		
NCName	x	x		
negativeInteger	x		x	x
NMTOKEN	x	x		
NMTOKENS	x	x		
nonNegativeInteger	x		x	x
nonPositiveInteger	x		x	x
normalizedString	x	x		
positiveInteger	x		x	x
QName	x	x		
short	x		x	x
string	x	x		
time	x		x	
token	x	x		
unsignedByte	x		x	x
unsignedInt	x		x	x
unsignedLong	x		x	x
unsignedShort	x		x	x

### 6.3. Regular expression syntax for xsd:

The regular expression syntax is fairly similar to that of Perl. Refer to the Appendix F of the XML Schema Datatypes specification<sup>24</sup> for a complete definition of the regular expressions allowed in the `pattern` parameter of any of the `xsd:` datatypes.

#### Note

If you are working with Unicode, you should read the full specification, as there are a number of advanced features, not discussed here, that are most useful in Unicode work.

Here is a summary of most of the commonly used features.

$p   q$	Either pattern $p$ or pattern $q$ .
$pq$	Pattern $p$ followed by pattern $q$ .

<sup>24</sup> <http://www.w3.org/TR/xmlschema-2/#regexs>

$p?$	Matches pattern $p$ or nothing at all. You could think of it as saying “ $p$ occurs optionally here.”
$p^*$	Matches zero or more occurrences of $p$ .
$p^+$	Matches one or more occurrences of $p$ .
$p\{n\}$	Matches exactly $n$ occurrences of pattern $p$ .
$p\{n,m\}$	Matches at least $n$ occurrences, but no more than $m$ occurrences, of pattern $p$ .
$p\{n,\}$	Matches $n$ or more occurrences of pattern $p$ .
$[c_1c_2\dots]$	Matches any single character from inside the square brackets. For example, the pattern “ <code>xsd:string { pattern=' [abc] ' }</code> ” matches any of the characters $a$ , $b$ , or $c$ . You can specify ranges of characters as “ $[c_1-c_2]$ ”. For example, the pattern “ $[a-zA-Z]$ ” matches any letter, lowercase or uppercase.
$[^c_1c_2\dots]$	Matches any single character <i>except</i> those enumerated inside the square brackets. For example, the regular expression “ <code>xsd:string { pattern=' [^abc] ' }</code> ” matches any single character <i>except</i> $a$ , $b$ , or $c$ .
$(p)$	Parentheses may be used for grouping. For example, pattern “ $(ab)^+$ ” matches “ $ab$ ”, “ $abab$ ”, “ $ababab$ ”, and so on.
$\r$	Matches the carriage return (ASCII CR) character.
$\n$	Matches the newline (ASCII LF) character.
$.$	Matches any character except newline or carriage return.
$\t$	Matches the tab (ASCII HT) character.
$\backslash$	Any of the following characters must be escaped by preceding them with a backslash: “ $\backslash   . - ^ ? * + \{ \} ( ) [ ]$ ”. For example, “ <code>pattern=' \[ \* \] '</code> ” matches the string “ $[*]$ ”.
$\s$	Matches a <i>whitespace character</i> : space, tab, newline, or carriage return.
$\S$	Matches any character except a whitespace character.
$\w$	Matches a <i>name start character</i> : a letter, “ $_$ ”, or “ $:$ ”.
$\W$	Matches any character except a name start character.
$\w$	Matches a <i>name character</i> , that is, a name start character or digit.
$\W$	Matches any character except a name character.
$\d$	Matches a decimal digit (same as “ $[0-9]$ ”).
$\D$	Matches any character except a decimal digit.

Here's an example of a pattern for a U. S. Postal Service zip code:

```
xsd:string { pattern=' [0-9]{5}(-[0-9]{4})?' }
```

That is, five digits, optionally followed by a hyphen and four more digits.