Extending XQuery with pattern matching

Benito van der Zander

Universität zu Lübeck
Motivation

Regular expressions are a standard way to validate/query strings.

Example

fooba+r

matches foobar
or foobaar
or foobaaar
but not foobr

Can we use something similar to validate/query XML?
Regular expressions are a standard way to validate/query strings.

Example

\((.*)://([^/]+)(/(.*)?\)?

matches https://example.org

or http://balisage.net/2014/Program.html

Can we use something similar to validate/query XML?
Motivation

Regular expressions are a standard way to validate/query strings.

Example

Can we use something similar to validate/query XML?
Motivation

Example

```
<xml>(.*)</xml>
```

matches `<xml>data</xml>` :)
but also `<xml>data</xml><!-<xml>->` :

We need regular expressions that understand the XML structure.
Motivation

But XPath is great. Why would we need regular expressions?

Example

No, it is really bad for siblings, as they often are on webpages:

```html
<html>
  <h2>Section start</h2>
  <p>data we want</p>
  <p>continued</p>
  <h2>Section end</h2>
</html>

/html/h2[. eq 'Section start']/following-sibling::p intersect /html/h2[. eq 'Section end']/preceding-sibling::p

<html>
  <h2>Section start</h2>
  (<p>.*</p>)*
  <h2>Section end</h2>
</html>
```
Motivation

Example

XPath does not check for input errors

```
/html/h2[. eq 'Section start']/following-sibling::*
intersect /html/h2[. eq 'Section end']/preceding-sibling::*

applied to

<xml>something entirely different</xml>

Query succeeds and returns: ()
```

Exception: invalid input
Overview

1. Motivation
2. Previous work
3. Pattern syntax
   - Basic pattern syntax
   - Selecting data
   - Advanced pattern syntax
   - JSONiq
4. Integration in XQuery
   - As a function
   - Extended XQuery syntax
5. Practical application
6. Conclusion
A really old query language (1998):

Example

```
WHERE <foo> $t </foo>
IN <foo> hello world </foo>
CONSTRUCT $a
```

returns `hello world`

(Was there ever a public implementation?)
Previous work

Scala

Example

In Scala we can do:

```scala
<mystery> match {
  case { txt } => println("foo:" + txt)
  case { txt } => println("bar:" + txt)
}

prints "foo:mystery"
```
Previous work
Scala

Example
Not so nice with attributes:

```scala
<foo id="bar"/> match {
  case n @ <foo/>
    if (n \ "@id" text) == "bar"
    => println("bar")
  case n @ <foo/>
    if (n \ "@id" text) == "baz"
    => println("baz")
}

prints "bar"
```
Basic pattern syntax

- A `text node` matches another `text node` (with "matching" string value)
- An `attribute="value"` matches another `attribute="value"` (with same name and "matching value")
- An `<element/>` E matches another `<element/>` F (with same name) that has
  - matching attributes (every attribute of E matches an attribute of F)
  - matching descendants (every `child` of E matches an `descendant` of F, such that the order of matched descendants of F is the same as the order of children of E)
Example

A simple pattern:

```xml
<element foo="bar">
    cat<meow/>
</element>
```

matches

```xml
<element foo="bar">
    cat<meow/>
</element>
```

or

```xml
<element foo="bar" att="value">
    <p>cat<call><meow loudness="60 dB"/></call></p>
</element>
```
Example

A simple pattern:

```xml
<element foo="bar">
    cat<meow/>
</element>
```

matches **not**

```xml
<element>
    <meow/>cat
</element>
```

**nor**

```xml
<element foo="bar">
    <meow/>cat
</element>
```
Selecting data

We allow embedding of arbitrary XQuery expressions.

Example

```
<a><b>{.}</b></a>
```

matched against

```
<a><b>some text</b></a>
```

returns . evaluated with the context item

```
<b>some text</b>
```

= `some text`

(Alternative syntax: replace `.{}` with `<t:s>{}</t:s>` or `<template:s>{}</template:s>` )
Selecting data

We allow embedding of arbitrary XQuery expressions.

Example

```xml
<html>
  <h2>A</h2>
  <a>{@href}</a>
  <h2>B</h2>
</html>
```

matched against

```xml
<html>
  <a href="http://wrong">link</a>
  <h2>A</h2>
  <a href="http://example.org">link</a>
  <h2>B</h2>
  <a href="http://wrong">link</a>
</html>
```

returns

```
href="http://example.org"
```
Selecting data

How to return multiple values?

Allow variable assignments as named return values.

Example

```html
<a> { $target := data(@href), $caption := text() } </a>
```

matched against

```html
<a href="http://example.org">an example</a>
```

returns $target = http://example.org

and $caption = an example
Selecting data

Possible abbreviations:

\[
\begin{align*}
\langle \text{element} \rangle \{ \$\text{variable} \}\rangle \\
\Rightarrow \langle \text{element} \rangle \{ \$\text{variable} := . \}\rangle \\
\langle \text{element} \rangle \text{ attribute} = "{\$\text{variable}}"/\rangle \\
\Rightarrow \langle \text{element} \rangle \{ \$\text{variable} := @\text{attribute} \}\rangle \\
\text{ (actually } \langle \text{element} \rangle \{ @\text{attribute} / (\$\text{variable} := .)\}\rangle )
\end{align*}
\]
Advanced pattern syntax

Regular expressions have more operators: ?, +, *, |, []

What do our patterns have?
Advanced pattern syntax

Optional elements:

\[ \texttt{<element/>?} \]

Are ignored in the pattern, if they do not exist in the data

(Alternative syntax: add attribute \texttt{t:optional="true"} or \texttt{template:optional="true"} )
Advanced pattern syntax

Repeated elements:

```
<element/>+ or <element/>*
```

Are repeated as long as possible

Example

```
<a>{data(@href)}</a>+
```

matched against
```
<html>  <a href="#1">1</a>  <a href="#2">2</a>  </html>
```

returns ("#1","#2")

(Alternative syntax: surround with
```
<t:loop min="" max="">...
```
or
```
<template:loop ....
```
Advanced pattern syntax

Alternative elements
Multiple elements are accepted

Example

```xml
<t:switch>
  <a>{text()}</a>
  <b>{text()}</b>
</t:switch>
```

matched against either `<a>foobar</a>` or `<b>foobar</b>`
returns `foobar`

*(Not an alternative syntax: (..|..). Should it be one?)*
Advanced pattern syntax

Many more:

\[<t:\text{loop}><t:\text{switch}> \text{ Used together } \Rightarrow \text{ unordered matching}\]
\[<t:\text{switch prioritized}="\text{true}"> \text{ use element order of pattern}\]
\[\text{instead order of data to find the “first” match}\]
\[t:\text{condition= arbitrary XQuery condition for testing matching}\]
\[<t:\text{if}> \text{ skip/use parts of the pattern depending on conditions}\]
\[<t:\text{else}> \text{ opposite of } <t:\text{if}>\]
\[t:\text{test=} \text{ abbreviation for } <t:\text{if}>\]
\[<t:\text{meta}> \text{ change various matching options, e.g. exact match}\]
\[\text{vs. regex match for text nodes}\]
Pattern matching is also useful for JSONiq types:

- atomar values match other atomar values that have the same value
- object O matches object P, if for every property of O there is a property of P with the same name and matching value
  e.g. \{"a": 1\} matches \{"a": 1, "b": 2\}
- array A matches array B, if there is a subsequence of B with the same length as A, where the i-th member of A matches the i-th member of the subsequence
  e.g. \[1,2,3\] matches \[1,2,"xxxx",3,4,5\]

Selector expressions could be contained in <t:s> nodes or dynamic functions
Example

\{
  "a": [1,2,3],
  "b": null,
  "c": \lt;\text{s}>.</\text{s}\gt
\}

applied to

\{
  "a": [1,"u",2,"v",3],
  "b": null,
  "c": [7,8,9],
  "d": 17
\}

would return

[7,8,9]
Integration in XQuery

A function for pattern matching

Every pattern is an XML/JSONiq element, so it can be stored in a variable. So we can make a function:

pxp:match($pattern as item(), $data as item()*)

Returning a map of all variables of the pattern, combining multiple assignments to a single sequence
Integration in XQuery

A function for pattern matching

Example

```
pxp:match(<ul> <li>{$var := text()}}</li>+ </ul>,
        <ul> <li>1</li> <li>2</li> </ul>)
```

returns a map

```
{"var": ("1", "2")}
```
Integration in XQuery

But calling a function in every query is inconvenient. So we add the patterns to the let, typeswitch and for expression:

Example

```
let <html><h2>section 1</h2>
    <p>{$var}</p>
    <h2>section 2</h2></html>
:= <html><h2>section 1</h2>
    <p>a</p>
    <p>b</p>
    <h2>section 2</h2>
    <p>c</p></html>
return $var / string ()
```

would return

```
("a", "b")
```
The `typeswitch` extensions is like the `let` extensions, but multiple patterns can be given in case clauses:

**Example**

```xquery
typeswitch (<html><b>foobar</b></html>)
    case <a>{.}</a> return "a link to " || @href
    case <b>{$v}</b> return "bold text: " || $v
    default return "unknown element"

would return

"bold text: foobar"
```
The `for` extension iterates over all assignments in that merging them:

```xml
for <ul> <li>{.}</li>+</ul>
in <ul> <li>1</li> <li>2</li> </ul>
return "li: " || .
```

would return

```
("li: 1", "li: 2")
```
Practical application

We have created an app/wrapper for 175 libraries by creating patterns for 15 different web catalogues/OPACs.

library 1.a
library 1.b
...
library 2.a
library 2.b
...
library 3.a
library 3.b
...

system 1
system 2
system 3

Desktop program

Android app

Common variables
Practical application

Creating a pattern:

Take a webpage:
Practical application

Creating a pattern:

Take a webpage’s source:

```html
<table border="0" cellspacing="1" cellpadding="1" width="100%">
<tr valign="top">
    <td width="90" nowrap align="left">
        Titel
    </td>
    <td width="10">
    </td>
    <td width="16">
        <img align="middle" title="Buch" alt="Buch" src='IMG/MAT/book.gif'>
    </td>
    <td>
    </td>
</tr>
<tr valign="top">
    <td width="90" nowrap align="left">
        Verfasser
    </td>
    <td width="10">
    </td>
    <td>
    </td>
    <td>
        Butcher, Jim
    </td>
</tr>
</table>

&lt; The &gt; Dresden files / 1. Storm front

( Bd. : &nbsp; 1 )
```

```html
</td>
</tr>
<tr valign="top">
    <td width="90" nowrap align="left">
        Verfasser
    </td>
    <td width="10">
    </td>
    <td>
    </td>
    <td>
    </td>
</tr>
```

```html
</table>

<spam class="darkLink">
    Butcher, Jim
</spam>
```
Practical application
Creating a pattern:

Remove the nonsense (formatting, whitespace):

...  
<table>
<tr>
<td>Title</td><td>1</td><td>IMG</td><td>2</td><td>3</td><td><a>The Dresden files / 1. Storm front (Bd. 1)</a></td>
</tr>
<tr>
<td>Verfasser</td><td>10</td><td>:</td><td>Butcher, Jim</td>
</tr>
<tr>
<td>Verlag</td><td>16</td><td>:</td><td>Orbit</td>
</tr>
<tr>
<td>Signatur</td><td>ENGL BUT 12/1:1</td>
</tr>
<tr>
<td>Jahr</td><td>2011</td>
</tr>
</table>

...
Practical application

Creating a pattern:

Remove the data and replace it with selection annotations:

...  
<\text{table}>
<\text{tr}>
  <\text{td}>Titel</\text{td}><\text{td}>:</\text{td}>\text{td}</\text{td}><\text{td}><\text{td}><\text{IMG}</\text{td}>
  <\text{td}><\text{a}>{\$book.title}</\text{a}</\text{td}</\text{tr}>

<\text{tr}><\text{td}>Verfasser</\text{td}><\text{td} width="10">:</\text{td}>\text{td}</\text{td}</\text{td}</\text{td}>
  <\text{td}><\text{span}>{\$book.author}</\text{span}</\text{td}</\text{tr}>

<\text{tr}><\text{td}>Verlag</\text{td}><\text{td} width="16">:\text{td}</\text{td}</\text{td}</\text{td}>
  <\text{td}>{\$book.publisher}</\text{td}</\text{tr}>

<\text{tr}><\text{td}><\text{b}>Signatur</\text{b}</\text{td}><\text{td}>:</\text{td}>\text{td}</\text{td}</\text{td}</\text{td}>
  <\text{td}>{\$book.id}</\text{td}</\text{tr}>

<\text{tr}><\text{td}>Jahr</\text{td}><\text{td}>:</\text{td}>\text{td}</\text{td}</\text{td}</\text{td}>
  <\text{td}>{\$book.year}</\text{td}</\text{tr}>
</\text{table}>

⇒ Finished pattern

Rarely takes more than a few minutes for a page (although it does not always then work then, e.g. due to javascript, invalid html, ... )
Conclusion

- Pattern matching is for XML/HTML what regular expressions are for strings
- Very useful to query schema-less data where the ordering matters
- XQuery can be used in patterns
- And patterns can be used in XQuery
- Patterns can be created with ease
- Implementation as standalone XQuery interpreter available at http://xidel.sourceforge.net
- Would be better in the standard

Questions?