Balisage: The Markup Conference 2016
Proceedings

FOXpath - an expression language for selecting files and folders

Hans-Jürgen Rennau
Senior Java developer
Traveltainment GmbH
<hrennau@yahoo.de>

Balisage: The Markup Conference 2016
August 2 - 5, 2016

How to cite this paper

Rennau, Hans-Jürgen. “FOXpath - an expression language for selecting files and folders.” Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016. In *Proceedings of Balisage: The Markup Conference 2016*. Balisage Series on Markup Technologies, vol. 17 (2016). https://doi.org/10.4242/BalisageVol17.Rennau01.

Abstract

A new expression language (FOXpath, short for folder XPath) enables XPath-like addressing of files and folders in a file system. The first version of the language is a modified copy of XPath 3.0, with node navigation removed and file system navigation added. The language is based on the data model XDM 3.0, without assuming any modifications of the model. In a second step, the language was merged back into XPath 3.0, resulting in FOXpath 3.0, which is a superset of XPath 3.0. The new expression language supports node navigation, file system navigation and a free combination of both functionalities within a single path expression. A reference implementation is described, and the possibility of extending the new functionality beyond file systems is discussed.

Table of Contents

Introduction

XPath is an expression language for selecting content from XML documents. The introduction of XPath 3.0 (8) states:

> ***The primary purpose of XPath is to address the nodes of [XML 1.0] or [XML 1.1] trees. XPath gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document.***

Key concepts of the language are the definition of item selection as a sequence of steps and the modelling of a step as a "spatial" movement across a tree, following a specified axis. These are combined into the concept of a navigation path, which enables item selection in a very elegant, concise and yet readable way. For example, the following expression

```
//animals/fox[not(trail)]
```

selects all `fox` elements found under an `animals` element, but not containing a `trail` element. It is not difficult to imagine similar possibilities for the selection of items from other, non-XML types of tree-structured information – file systems, URI trees, web site maps, object type hierarchies, object instance trees, etc. Yet other expression languages enabling XPath-like navigation of non-XML tree structures do not seem to exist, or are not popular enough to be easily discovered, excepting JSONPath (5) and JXPath (6).

An interesting alternative to new XPath-like languages would be extensions of the XPath language itself: the addition of new kinds of expressions enabling a traversal of non-XML trees along navigation axes, filtered by item tests and predicates. In this context, file systems (or, more generally, resource name trees) deserve particular interest. The selection of files and folders is of course an important operation in its own right - XPath-like `ls` (Unix) or `dir` (Windows) commands could be quite useful. But if available within XPath, navigation of resource name trees could complement node navigation. The addition might enable composite navigation, starting with the selection of resources and continuing with the selection of nodes within the selected resources. Ideally, such two-level navigation might even be expressed by a single path expression.

This paper reports the design and implementation of FOXpath, a new expression language for selecting files and folders (FOXpath = folder XPath). The language can be characterized as a modified copy of XPath 3.0, in which node navigation is replaced by URI navigation. The paper then proceeds to describe the integration of FOXpath into XPath 3.0, resulting in FOXpath 3.0, which is a superset of XPath 3.0. It extends the ***path expression*** by URI navigation, enabling expressions like the following:

```
   \projects\\parks[not(ancestor~::private]]\* //animals/fox[not(trail)]
```

which merges file system navigation (addressing files contained by a `parks` folder which itself is not contained by a `private` folder) and node navigation into a single, continuous path.

The FOXpath language

This section describes the first version of the FOXpath language, which is a modified copy of XPath 3.0, not a superset. The subsequent section (section “FOXpath 3.0”) will describe FOXpath 3.0, the result of integrating FOXpath into XPath 3.0

Overview

FOXpath is an expression language designed to enable elegant and fine-grained selection of files and folders. The goal is to achieve maximum similarity with XPath. The goal includes both, syntactic similarity as well as equivalent levels of expressiveness. The FOXpath language is a modified copy of the XPath 3.0 language:

- It is an expression language based on the data model XDM 3.0 (13).
- It has the same processing model as XPath 3.0 (8).
- Its grammar is a modified copy of the XPath 3.0 grammar:
  - The sole kind of expression which has been removed is the path expression.
  - The sole kind of expression which has been added is the ***foxpath expression***.
  - The syntax of the new foxpath expression resembles the syntax of the path expression of XPath 3.0.
- Its semantics are a modified copy of the semantics of XPath 3.0:
  - The semantics of all retained XPath 3.0 expressions are retained or extended (see below).
  - The semantics of the new foxpath expression resemble the semantics of the path expression of XPath 3.0.

The term ***extended semantics*** is used for a modification of the evaluation rules which in particular cases avoids a type error and defines a successful evaluation instead. Such a modification can be viewed as an extension as it does not change the evaluation unless the original rules prescribe a type error. FOXpath applies such extensions to several operators as well as to the definition of the ***effective boolean value***. An example is an additional rule defining the evaluation of the `except` operator when an operand contains atomic items.

Retained XPath 3.0 expressions

Except for the path expression, all basic kinds of expression retain the syntax and semantics as defined by XPath 3.0. In a few cases the semantics were extended by additional rules enabling a successful evaluation where XPath 3.0 evaluation raises a type error. The following table summarizes the XPath 3.0 expressions retained by the FOXpath language.

Table I

The XPath 3.0 expressions and operators retained by the FOXpath language. The `Semantics` column indicates whether expression semantics of XPath 3.0 have been retained (=) or extended (E). When several examples are given, they are seperated by a semicolon.

Expression or operator	Semantics	Example
Arithmetic expression	=	$a + $b ; $a - $b ; $a * $b ; $a div $b ; $a idiv $b ; $a mod $b ; -$a
Cast expression	=	$a cast as xs:integer*
Comma operator	=	$x, $y
Conditional expression	=	if ($a instance of xs:integer) then $a else "?"
Context item expression	=	.
Except operator	E	$nodes1 except $nodes2
Filter expression	=	$names[not(starts-with(., 'test-'))]
For expression	=	for $a in $as, $b in $bs return concat($a, b)
Function call (constructor function)	=	xs:date($s)
Function call (dynamic)	=	$f($s)
Function call (static)	=	lower-case($s)
General comparison	=	$a = $b ; $a != $b ; $a < $b ; $a <= $b ; $a > $b ; $a >= $b
Inline function expression	=	function($n, $v) {concat($n, ': ', $v)}
Instance of expression	=	$a instance of xs:integer*
Intersect operator	E	$nodes1 intersect $nodes2
Let expression	=	let $a := 1, $b := 2 return $a + $b
Logical expression	=	$a or $b and $c
Named function reference	=	replace#3
Node comparison	=	$a << $b ; $a >> $b ; $a is $b
Numeric literal	=	123 ; 5.1 ; 1.1E5
Parenthesized expression	=	($x, $y)
Quantified expression	=	some $a in $as satisfies $a lt 0 ; every $a in $as satisfies $a lt 0
Range expression	=	1 to 10
Simple map operator	=	$uris ! doc(.)
String concatenation expression	=	$name || ': ' || $value
String literal	=	'London' ; "Paris"
Treat expression	=	$a treat as xs:integer
Union operator	E	$nodes1 | $nodes2 ; $nodes1 union $nodes2
Value comparison	=	$a eq $b ; $a ne $b ; $a lt $b ; $a le $b ; $a gt $b ; $a ge $b
Variable reference	=	$x

Semantic extensions of retained XPath 3.0 expressions

Several semantic extensions were made in order to streamline the processing of URI sequences. Most importantly, XPath does not define the effective boolean value of a sequence of several atomic items, and an attempt to evaluate the effective boolean value of such a sequence raises a type error. In XPath, therefore, a predicate expression which may resolve to a sequence of several URIs must be avoided, whereas support for such predicate expressions is highly desirable when navigating the URI tree of a file system. The extensions are detailed below.

Effective boolean value

If the operand is a sequence of several items which starts with an atomic item, XPath 3.0 mandates the raising of a type error. In FOXpath, the effective boolean value of the operand is equal to the effective boolean value obtained for a value consisting of the first item of the operand.

The operators **`Union`**, **`Intersect`** and **`Except`**

If an operand of the `Union` operator contains an atomic item, XPath 3.0 mandates the raising of a type error. In FOXpath, the expression returns the value `fn:distinct-values((E1, E2))`, where `E1` and `E2` denote the operands.

If an operand of the `Intersect` operator contains an atomic item, XPath 3.0 mandates the raising of a type error. In FOXpath, the expression returns the value `fn:distinct-values(E1[. = E2])`, where `E1` and `E2` denote the operands.

If an operand of the `Except` operator contains an atomic item, XPath 3.0 mandates the raising of a type error. In FOXpath, the expression returns the value `fn:distinct-values(E1[not(. = E2)])`, where `E1` and `E2` denote the operands.

Foxpath expression

The FOXpath language modifies the XPath 3.0 language by removing the path expression and adding a new kind of expression, the **foxpath expression**. This expression resembles the path expression syntactically and has semantics which appear like a translation of node navigation into the navigation of a file system. The similarity can be illustrated by considering the following definition from XPath 3.0 (8):

> ***[Definition: A path expression can be used to locate nodes within trees. A path expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//".]***

Compare this to the definition of a foxpath expression:

> ***[Definition: A foxpath expression can be used to locate files and folders in a file system. A foxpath expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//", or with "/" or "//" preceded by a drive letter and a colon.]***

Foxpath operator

As a path expression combines successive steps using the path operator (/), the foxpath expression combines successive steps using the foxpath operator (/ in FOXpath, but \ in FOXpath 3.0, see section “Syntactic modifications of the FOXpath language”). The semantics of the foxpath operator can be regarded as a modified copy of the semantics of the path operator, designed to support step-wise navigation of the file system.

XPath 3.0 defines the path operator as follows (8).

> XPath 3.0:
>
> ***The path operator "/" is used to build expressions for locating nodes within trees. Its left-hand side expression must return a sequence of nodes. The operator returns either a sequence of nodes, in which case it additionally performs document ordering and duplicate elimination, or a sequence of non-nodes. Each operation `E1/E2` is evaluated as follows: Expression `E1` is evaluated, and if the result is not a (possibly empty) sequence `S` of nodes, a type error is raised [err:XPTY0019]. Each node in `S` then serves in turn to provide an inner focus (...)***

> ***for an evaluation of*** *`E2`****, as described in 2.1.2 Dynamic Context. The sequences resulting from all the evaluations of*** *`E2`* ***are combined as follows:***
>
> 1. ***If every evaluation of*** *`E2`* ***returns a (possibly empty) sequence of nodes, these sequences are combined, and duplicate nodes are eliminated based on node identity. The resulting node sequence is returned in document order.***
> 2. ***If every evaluation of*** *`E2`* ***returns a (possibly empty) sequence of non-nodes, these sequences are concatenated, in order, and returned.***
> 3. ***If the multiple evaluations of*** *`E2`* ***return at least one node and at least one non-node, a type error is raised [err:XPTY0018].***

The definition of the foxpath operator is a modified copy of this definition.

> FOXpath:
>
> ***The foxpath operator is used to build expressions for locating files and folders in a file system. Each operation*** *`E1/E2`* ***is evaluated as follows: Expression*** *`E1`* ***is evaluated and the result is atomized, resulting in a sequence*** *`s`*. ***Every item in*** *`s`* ***then serves in turn to provide an inner focus for an evaluation of*** *`E2`*. ***The sequences resulting from all the evaluations of*** *`E2`* ***are combined as follows:***
>
> 1. ***If every evaluation of*** *`E2`* ***returns a (possibly empty) sequence of atomic items, these sequences are concatenated, items are cast to*** *`xs:string`*, ***duplicate items are eliminated and the remaining items are returned in sorted order.***
> 2. ***If the multiple evaluations of*** *`E2`* ***return at least one node, the result sequences are concatenated, in order, and returned.***

The evaluation of the foxpath operation `E1/E2` (syntax changed to `E1\E2` in FOXpath 3.0) is summarized by the following pseudo-code:

```
let $items := data(E1) ! E2
return
   if (every $item in $items satisfies $item instance of xs:anyAtomicType)
   then
      sort(distinct-values($items ! string(.)))
   else
      $items
```

Steps

Whereas a step in XPath 3.0 is either an axis step or a postfix expression, a step in FOXpath is either a fox axis step or a postfix expression.

Fox axis steps

FOXpath introduces a new kind of expression, the **fox axis step**. Its definition is derived from the definition of an axis step (8), repeated here for the reader's convenience:

> XPath 3.0:
>
> ***[Definition: An axis step returns a sequence of nodes that are reachable from the context node via a specified axis. Such a step has two parts: an axis, which defines the "direction of movement" for the step, and a node test, which selects nodes based on their kind, name, and/or type annotation.]***

A fox axis step is defined analogously:

> FOXpath:
>
> ***[Definition: A fox axis step returns a sequence of URI references that are reachable from the URI reference provided by the context item via a specified fox***

axis. Such a step has two parts: a fox axis, which defines the "direction of movement" for the step, and a name test, which selects URI references based on the resource name, defined as the final step of the URI path.]

Fox axes

The definition of a fox axis is derived from the definition of XPath axes. The following table summarizes all fox axes. Note that there is no `following` axis and no `preceding` axis, as these were considered to be without practical value.

Table II

The fox axes defined by the FOXpath language. The string `name` represents a name test combined with the navigation axis.

Axis	Fox axis step syntax	Abbreviated syntax	
		syntax	equivalence
self	self~::name	-	-
child	child~::name	name	child~::name
descendant	descendant~::name	-	-
descendant-or-self	descendant-or-self~::name	...//...	.../descendant-or-self~::*/...
parent	parent~::name	..	parent~::*
ancestor	ancestor~::name	...name	ancestor~::name
ancestor-or-self	ancestor-or-self~::name	-	-
following-sibling	following-sibling~::name	-	-
preceding-sibling	preceding-sibling~::name	-	-

Fox name test

XPath combines a navigation axis with a node test, which may be either a name test or a kind test. Similarly to this, a fox axis is always associated with a fox name test. A fox name test constrains the name of files and folders to be selected. Contrary to an XPath name test, a fox name test may contain wildcard characters:

- Character `*` represents zero or more arbitrary characters
- Character `?` represents exactly one arbitrary character

If these characters should appear in a name as such, they must be escaped by a preceding tilde (~).

A fox name test can use one of two alternative syntaxes:

- Canonical syntax
- Abbreviated syntax

Using the ***canonical syntax***, a fox name test consists of a name representation surrounded by backquotes. Within the name representation, two adjacent backquotes are interpreted as a single backquote character. Apart from the doubling of backquotes and the escaping of literal wildcard characters and of the tilde character by a preceding tilde (~), no other escaping is necessary or allowed.

The ***abbreviated syntax*** of a fox name test consists of a name representation without surrounding delimiters. Within the name representation, several non-wildcard characters must be escaped by a preceding tilde (~) in order to avoid syntactical ambiguities:

```
   ~ [ ]   \ /   <>   ( )   = !|,  WHITESPACE
```

The initial character of a name test is subject to additional constraints: if the initial character of matching names should be a digit, dot (.) or backquote (`), the name representation must escape the digit, dot or backquote by a preceding tilde (~).

See Appendix A for the ENBF production of a fox name test (rules [908a] - [912a]). The following table shows a few examples.

Table III

Examples of fox name tests using the canonical or abbreviated syntax.

Canonical syntax	Abbreviated syntax
\`foo\`	foo
\`.git\`	~.git
\`2016\`	~2016
\`\`\`foo\`	~\`foo
\`foo+bar\`	foo~+bar
\`foo(1)\`	foo~(1~)
\`foo bar\`	foo~ bar
\`foo\`\`bar\`	foo\`bar

Function library

The FOXpath language supports all functions supported by XPath 3.0 (10) as well as a few additional functions expected to be useful when selecting files and folders from a file system. The additional functions are summarized by the following table.

Table IV

Functions supported by the FOXpath language and not supported by XPath 3.0.

Name	Meaning
bslash	Returns the argument with forward slashes replaced by back slashes.
eval-xpath	Returns the value of the argument string evaluated as an XPath 3.0 expression.
file-contains	Returns `true` if the file contains a pattern specified in glob syntax.
file-date	Returns the date and time of the last modification.
file-lines	Returns the lines of a text file, optionally filtered by a pattern using glob syntax (4).
file-name	Returns the file name, which is the last step of the file URI.
file-size	Returns the file size as number of bytes.
has-xatt	Returns `true` if the context item is the URI of an XML document containing an attribute whose name, value and parent element name match the specified constraints.
has-xelem	Returns `true` if the context item is the URI of an XML document containing an element whose name and text content match the specified constraints.
has-xroot	Returns `true` if the context item is the URI of an XML document having a root element with a name matching the specified constraints.
is-dir	Returns `true` if the argument is the URI of a directory.
is-file	Returns `true` if the argument is the URI of a file, rather than a directory.
matches-xpath	Returns the effective boolean value of the XPath expression supplied as an argument and evaluated in the context of the document whose URI is either specified as second argument or provided by the context item.

xroot	Returns the local name of the root element of the document with a URI specified by argument or provided by the context item; returns the empty sequence if the URI does not reference an XML document.

Examples

Here are some examples of valid expressions of the FOXpath language. All examples refer to a folder `wild902` containing an installation of the WildFly application server (version 9.0.2.Final), downloaded from (7). For further examples, requiring FOXpath 3.0, see (section “Examples”).

```
# Child axis   (top level files and folders)
/wildfly902/*
=>
/wildfly902/.installation
/wildfly902/appclient
/wildfly902/bin
/wildfly902/copyright.txt
/wildfly902/docs
/wildfly902/domain
...

# Child axis, filtered   (top level files)
/wildfly902/*[is-file(.)]
=>
/wildfly902/copyright.txt
/wildfly902/jboss-modules.jar
/wildfly902/LICENSE.txt
/wildfly902/README.txt

# Descendant axis   (count all XML files) :)
count(/wildfly902/descendant~::*.xml)
=> 375

# Embedded //   (count folders and folders)
count(/wildfly902//*[is-dir()]), count(/wildfly902//*[is-file()])
=> 891 1261

# Multiple embedded //
/wildfly902//layers//*sql*//*.xml
=> /wildfly902/modules/system/layers/base/javax/sql/api/main/module.xml

# Parent axis   (all folders containing html files) :)
/wildfly902//*.html/parent~::*
=> /wildfly902/welcome-content

/wildfly902//*.html/..
=> /wildfly902/welcome-content

# Ancestor axis   (all top-level folders containing directly or indirectly XSD files)
/wildfly902//*.xsd/ancestor~::*[parent~::wildfly902]
=> /wildfly902/docs

/wildfly902//*.xsd/...*[parent~::wildfly902]
=> /wildfly902/docs

# Preceding-sibling axis   (top level files and folders, before 'docs' :)
/wildfly902/docs/preceding-sibling~::*
=>
/wildfly902/.installation
/wildfly902/appclient
/wildfly902/bin
/wildfly902/copyright.txt

# Following-sibling axis   (top level files and folde4rs, after 'docs' :)
/wildfly902/docs/following-sibling~::*
=>
/wildfly902/domain
/wildfly902/jboss-modules.jar
/wildfly902/LICENSE.txt
/wildfly902/modules
/wildfly902/README.txt
/wildfly902/standalone
/wildfly902/welcome-content

# Position predicate (foward axis)
/wildfly902/descendant~::*.xml[1]
=> /wildfly902/appclient/configuration/appclient.xml

/wildfly902/descendant~::*.xml[last()]
=> /wildfly902/standalone/configuration/standalone_xml_history/standalone.last.xml

# Position predicate (reverse axis)
/wildfly902//*standalone.last.xml/ancestor~::*[1]
=> /wildfly902/standalone/configuration/standalone_xml_history

/wildfly902//*standalone.last.xml/ancestor~::*[2]
=> /wildfly902/standalone/configuration
```

```
    # Parenthesized step   (count all XML or XSD  files)
    count(/wildfly902//(*.xml, *.xsd))
    => 758

    # Filtering by attribute name, value and parent name
    /wildfly902//sql//*.xml[has-xatt('name','javax/sql*', 'path')]
    =>
    /wildfly902/modules/system/layers/base/javax/sql/api/main/module.xml

    # Filtering by element name and value
    /wildfly902//*.xml[has-xelem('*property-type','java*')]
    =>
    /wildfly902/modules/system/layers/base/org/jboss/genericjms/main/META-INF/ra.xml

    # Filtering by root name
    /wildfly902//*.xml[has-xroot('connector')]
    =>
    /wildfly902/modules/system/layers/base/org/jboss/genericjms/main/META-INF/ra.xml

    # Filtering by matching XPath
    /wildfly902//*.xml[matches-xpath('count(//*:subsystem) > 100')]
    =>
    /wildfly902/domain/configuration/domain.xml

    # Final step a concatenation   (file path + file size)
    /wildfly902//*[is-file(.)][file-size(.) le 50]/concat(., ' (', file-size(.), ')')
    =>
    /wildfly902/modules/system/layers/base/org/jboss/as/jdr/main/resources/plugins.properties (40)
    /wildfly902/modules/system/layers/base/sun/jdk/main/service-loader-resources/META-
INF/services/java.sql.Driver (29)

    # Final step an edited path   (file extension)
    sort(distinct-values(/wildfly902//*[is-file(.)]/replace(., '.*\.', '')), lower-case#1)
    => bat conf css Driver dtd exe gif html ico jar jbossclirc log MF png properties ps1 ScriptEngineFactory
sh so txt xml xsd

    # Empty directories
    fox  "/wildfly902//*[is-dir(.)][empty(*)]"
    =>
    /wildfly902/.installation
    /wildfly902/domain/data/content
    /wildfly902/domain/tmp/auth
    ...

    # A quantified expression   (checking that all XML and XSD documents are wellformed)
    every $doc in /wild902//(*.xml, *.xsd) satisfies doc-available($doc)
    => true
```

Implementation

A reference implementation of the FOXpath language is available, written in the XQuery language, version 3.1. The implementation is an integral part of the implementation of the FOXpath 3.0 language. See section “Implementation” for further information.

FOXpath 3.0

Overview

The FOXpath 3.0 language is not a modified copy of XPath 3.0, but a ***superset*** of XPath 3.0: every valid XPath 3.0 expression is also a valid FOXpath 3.0 expression. This was achieved by “disassembling” the new expression kind (the foxpath expression) and integrating its parts (foxpath operator and fox axis step expression) into the path expression of the XPath language. This integration required syntactic changes of the new parts compared to the original FOXpath language in order to avoid syntactic ambiguity. Specifically, whereas in FOXpath the path operator (/) is effectively redefined in order to support the navigation of resource name trees, FOXpath 3.0 ***restores*** the original path operator and ***adds*** a foxpath operator (\) represented by a backslash, rather than a forward slash. Besides, extra-grammatical constraints were added which achieve the disambiguation of node name tests versus fox name tests using the abbreviated syntax and not preceded by an explicit fox axis.

Syntactic modifications of the FOXpath language

Integration of the FOXpath language into XPath involves two syntactic modifications:

- The foxpath operator is represented by a backslash `\`, not by a slash `/`.
- Dependent on the location within the expression tree, a fox name test can be constrained to use the canonical syntax, rather than the abbreviated syntax.

Use of the abbreviated fox name syntax is controlled by a new ***extra-grammatical constraint***. It allows the abbreviated syntax only in places where the context item is known to originate from a fox axis step and hence can be assumed to be a resource URI.

Context-dependent parsing

The benefit of supporting the abbreviated syntax of a fox name test is regarded as important enough to justify a context dependency of expression parsing. Consider how the abbreviated syntax makes file system navigation as elegant as node tree navigation, comparing this example

```
    \projects\\offline\*\(config, src)[flag.xml]
```

with the equivalent expression using canonical syntax:

```
    \`projects`\`offline`\\`*`\(`config`, `src`)[`flag.xml`]
```

Independently of the backquotes, none of the name tests could be a node name test, rather than a fox name test:

- The name tests `projects`, `offline` and `*` appear behind the foxpath operator and are therefore evaluated in the context of an item known to represent a resource URI, not a node
- The name tests `config`, `src` and `flag.xml` represent immediate sub expressions of an expression apprearing behind the foxpath operator

The rules formalizing such inference rely on a pseudo-function `context-is-URI(E)` which takes an expression `E` from the expression tree of a query and returns `true` or `false`, according to these rules:

- If `E` is the top-level expression of the query, `context-is-URI(E)` is `false`
- If `E` is the right-hand operand of the foxpath operator, `context-is-URI(E)` is `true`
- If `E` is the right-hand operand of the path operator, `context-is-URI(E)` is `false`
- If `E` is the right-hand operand of the simple map operator (!), `context-is-URI(E)` is `false`
- Otherwise, `context-is-URI(E)` is equal to `context-is-URI(`parent expression of `E`)

The parsing of an expression `E` depends on `context-is-URI(E)` as follows:

- If `E` matches the abbreviated syntax of a fox name test, it is parsed as a fox name test if and only if `context-is-URI(E)` is `true`.
- If the text of `E` is two adjacent dots (`..`), it is parsed as an abbreviated fox axis step if `context-is-URI(E)` is `true`, and it is parsed as an abbreviated node axis step otherwise.

To illustrate the effect of `context-is-URI(E)`, consider the following examples in which `E` denotes an expression:

```
\a\E
\a\\E
\a\(b, E)
\a\b[E]
\a\(b, c[d[E]])
```

In all cases, `context-is-URI(E)` is `true` so that `E` may use the abbreviated fox name syntax.

Extended path expression

Overview

The path expression of FOXpath 3.0 is an extended version of the path expression of XPath 3.0. Remember that in the FOXpath language the foxpath expression replaces the path expression of XPath 3.0. In the FOXpath 3.0 language, the foxpath expression is in turn replaced by an ***extended path expression***, which merges the constituents of both, the original path expression and the new foxpath expression. As a point of reference, let us once more consider the definition of a path expression given in XPath 3.0 (8):

> XPath 3.0
>
> ***[Definition: A path expression can be used to locate nodes within trees. A path expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//".]***

In FOXpath 3.0, the following definition holds:

> FOXpath 3.0
>
> ***[Definition: A path expression can be used to locate nodes within trees, or files and folders within a file system. A path expression consists of a series of one or more steps, separated by "/", "//", "\" or "\\", and optionally beginning with "/", "//", "\" or "\\", or with "\" or "\\" preceded by a drive letter and a colon.]***

Whereas in XPath 3.0 a step is either a postfix expression or an axis step, in FOXpath 3.0 a step is either a postfix expression or an axis step or a fox axis step.

Initial operators (/, //, \, \\)

In an extended path expression, initial "/" or "//" has the same semantics as in XPath 3.0. Initial "\" or "\\" (optionally preceded by a drive letter and a colon) has similar semantics, but referring to the file system. Their definition references a new function `fox:root-URI()`, which returns the root folder of the file system. As some file systems define several root folders distinguished by a "drive letter", a second variant of the function accepts a single parameter which is interpreted as drive letter and returns the corresponding root folder.

A "\" at the beginning of a path expression is an abbreviation for the initial step `fox:root-URI()\` (however, if the "\" is the entire path expression, the trailing "\" is omitted from the expansion.) The effect of this initial step is to begin the path at the root folder of the file system. Similarly, a "x:\" at the beginning of a path expression has the effect to begin the path at the root folder returned by the function call `fox:root-URI("x")`).

A "\\" at the beginning of a path expression is an abbreviation for the initial steps `fox:root-URI()\descendant-or-self~::*\` (however, "\\" by itself is not a valid path expression.) The effect of these initial steps is to establish an initial URI sequence comprising the URIs of all files and folders in the file system. This URI sequence is used as the input to subsequent steps in the path expression. Similarly, a "x:\\" at the beginning of a path expression establishes an initial URI sequence comprising the URIs of all files and folders found in the file system identified by drive letter "x".

Extended semantics of axis steps

The semantics of an axis step is extended in order to enable seamless combination of fox axis steps and axis steps. Whereas the semantics of XPath 3.0 prescribe a type error if the left-hand operand of the path operator returns atomic values, the semantics of FOXpath 3.0 avoid the type error by prescribing a "nodification" of any atomic values: the atomic value is replaced by the document node obtained by calling the `fn:doc` function with the atomic value as argument. (If, however, the function call raises an error, the path expression raises an error.) Thanks to this extension, expressions like the following:

```
\projects\niem\\*.xsd /xs:schema/xs:element
```

can be evaluated, as the first axis step (reading from left to right) is applied to the result of parsing each resource URI returned by the preceding step into document nodes.

Preserved semantics of fox axis steps

The semantics of fox axis steps are not changed compared with the semantics defined by the FOXpath language. In summary, a fox axis step consists of a navigation axis and a name test and optional predicates. The evaluation of predicates is governed by the same rules as the evaluation of predicates in node axis steps.

Heterogeneous navigation

The semantics of axis steps and fox axis steps imply that both kinds of steps can be mixed without restriction. In the typical case, all fox axis steps precede the first node axis step, selecting the resources into which the node axis steps navigate, for example:

```
\projects\\*.xsd /xs:schema/@targetNamespace
```

However, different patterns are also possible. For example, initial node axis steps might navigate into a catalog document, arriving at items containing the URIs of folders. Subsequent fox axis steps may navigate down into those folders (or anywhere into the file system, starting at those folders):

```
doc("catalog.xml")//projectHome/@uri \\*.xml
```

Examples

All examples of FOXpath expressions shown in (section "Examples") can be converted into examples of FOXpath 3.0 by replacing each / operator by its new syntax which is a backslash (\). Here come a few further examples demonstrating the merging of fox steps and node steps into a single path expression.

```
    # All root element names
    sort(distinct-values(\wildfly902\\*.xml /local-name(*)))
    =>
    connector
    domain
    host
    jboss-cli
    module
    module-alias
    server

    # All XSDs with a top-level element declaration 'Claims'
    \wildfly902\\*.xsd[/xs:schema/xs:element/@name = 'Claims']
    =>
    /wildfly902/docs/schema/ws-trust-1.3.xsd
    /wildfly902/docs/schema/wstrust/v1_3/ws-trust-1.3.xsd

    # All XSDs with a target namespace containing 'jaxws'
    \wildfly902\\*.xsd[contains(./*/@targetNamespace, 'jaxws')]
    =>
    /wildfly902/docs/schema/jbossws-jaxws-config_4_0.xsd

    # For each found XML document a sorted list of all element names
    \wildfly902\\bind\\*.xml\concat(., ': ', string-join(sort(distinct-values(//local-name(.))), ' '))
    =>
    /wildfly902/modules/system/layers/base/com/sun/xml/bind/main/module.xml: dependencies module properties
property resource-root resources
```

```
   /wildfly902/modules/system/layers/base/javax/xml/bind/api/main/module.xml: dependencies module resource-root
resources
```

Generalization

The file system is just an instance of a larger abstraction – a tree of resource URIs[1], or "resource tree" for short. Examples of such trees include:

- Resource URIs exposed by a RESTful web service
- The URIs of documents stored in a NOSQL database
- The URIs of resources managed by a version control system

The navigation of URI references supported by the FOXpath language is not restricted to the file system - any other type of resource tree can be included, for which two basic navigation functions are available, from which the functionality of foxpath navigation may be derived ***completely***:

```
    fox:child-uri-collection($uri as xs:string) as xs:anyURI*
    fox:root-uri($uri as xs:string) as xs:anyURI?
```

While these functions are sufficient to enable foxpath navigation of a resource tree, the efficiency of navigation may be greater if `fox:child-uri-collection()` supports a second parameter specifying a name pattern, which corresponds to a fox name test. A further increase of efficiency may be provided by an additional function returning all descendant URIs, rather than only child URIs. This leads us to the following set of functions enabling efficient navigation of resource trees:

```
    fox:child-uri-collection($uri as xs:string, $namePattern as xs:string?) as xs:anyURI*
    fox:descendant-uri-collection($uri as xs:string, $namePattern as xs:string?) as xs:anyURI*
    fox:root-uri($uri as xs:string) as xs:anyURI?
```

Implementations of these functions will tend to be specific for a particular type of resource tree. The FOXpath language can support navigation of ***several*** types of resource trees if a function is available which maps a given URI to the appropriate instances of those basic navigation functions:

```
    fox:get-function-child-uri-collection($uri as xs:string) as function(xs:string, xs:string?) as xs:anyURI*
    fox:get-function-descendant-uri-collection($uri as xs:string) as function(xs:string, xs:string?) as
xs:anyURI*
    fox:get-function-root-uri($uri as xs:string) as function(xs:string) as xs:anyURI?
```

The appropriate instances are those applicable to the type of the resource tree to which the input URI belongs. In the case of the file system, instances of the first two of these functions are provided by (partial applications of) the EXPath defined function `file:list`, and an instance of the third function can be implemented by a simple string manipulation extracting from a file system path the initial slash and the drive letter optionally preceding it.

Note the difference between a resource tree type and a resource tree instance. The current version of the FOXpath language supports a single resource tree type which is the file system. However, the implementation supports the distinction of multiple resource trees via drive letters as used by the Windows file system.

Implementation

A reference implementation of the FOXpath 3.0 language is available, written in the XQuery language, version 3.1. The implementation consists of five XQuery library modules, summarized in the following table.

Table V

The XQuery library modules implementing the FOXpath 3.0 language.

Module	Purpose
foxpath.xqm	Resolves a foxpath expression to an XDM value.
foxpath-parser.xqm	Parses a foxpath expression into a tree-structured representation.
foxpath-util.xqm	Provides utilities used by the parser and the resolver.
foxpath-processorDependent.xqm	Encapsulates the dependency on a particular XQuery processor.
foxpath-resourceTreeTypeDependent.xqm	Encapsulates the dependency on particular types of resource trees - file system and (later) possibly others.

The implementation depends on several functions of the `file` module (3) defined by the EXPath initiative (2). Currently, the implementation can only be executed using the BaseX processor (1), because in a few cases the BaseX extension function `xquery:eval` is used in order to evaluate expressions whose evaluation in pure XQuery appeared to be difficult (or tedious) beyond proportion:

- Partial function application
- Instance of expression
- Treat expression
- Castable expression
- Cast expression

This dependency on the BaseX processor will be removed as soon as there will be an EXPath defined (or W3C defined) function for the dynamic evaluation of XPath expressions. The current dependency is factored out into a single function:

```
    declare function f:xquery($xquery as xs:string, $context as map(*)?) as item()* {
       if (exists($context)) then xquery:eval($xquery, $context)
       else xquery:eval($xquery)
};
```

The code can thus easily be adapted for execution by a different XQuery processor, which meets the following conditions:

- supports XQuery, version >= 3.1
- supports all extension functions of the EXPath defined `file` module, version >= 1.0
- supplies an extension function for the dynamic evaluation of XPath expressions

The implementation of the FOXpath 1.0 language is an integral part of the implementation of the FOXpath 3.0 language (see section “FOXpath 3.0”). Whether the code behaves as an implementation of the FOXpath 1.0 language or as an implementation of FOXpath 3.0 is controlled by an external variable, defaulting to version 3.0.

The implementation can be downloaded from here: https://github.com/hrennau/foxpath .

Issues and features at risk

A crucial aspect of FOXpath 3.0 is the syntax of fox name tests, as it impacts the “look and feel” of file system navigation and its combination with node navigation into a single navigation model. The abbreviated fox name syntax is an important feature as it maximizes elegance and the similarity between node tree and URI tree navigation. But the feature raises two issues: (a) it introduces a context-dependent parsing rule (the same expression syntax can be parsed as a node name test or a fox name test), (b) it burdens the grammar with complex escaping rules (many characters must be escaped, and even more characters when used at the beginning of the name test). Such escaping rules could be removed by adopting an alternative syntax rule suggested by the lookup operator of XPath 3.1 (9). The alternative rule would restrict the use of abbreviated syntax to NCNames and NCNames with inserted wildcard characters. In all other cases, the canonical syntax would be mandatory.

Dependent on feedback of users and perhaps other implementers, the feature of an abbreviated fox name syntax might be removed or modified to admit abbreviated syntax only for NCNames and NCNames with inserted wildcard characters.

Discussion

The practical usefulness of an expression language for file system navigation is fairly obvious. A simple shell script can make the functionality immediately available and thus serve as a far more powerful alternative to shell commands like the Unix `ls` command or the Windows `dir` command. Command-line tools may use the FOXpath language as the syntax for command-line options selecting input files. Any program language may profit from an API function enabling expression-based selection of resources.

Especially great is the potential benefit of FOXpath for XPath itself and languages built upon it (XQuery (12), XSLT (14), XProc (11)). This is due to the fact that the selection of resources and the selection of nodes within resources are two phases of the same operation, which is the selection of information items from a system of resources. The seamless integration of FOXpath into XPath 3.0 is therefore felt to lift the usefulness of FOXpath to a significantly higher level.

It may be asked if the first version of FOXpath, which is like XPath minus node navigation plus file system navigation, is of any interest when there is FOXpath 3.0 which is XPath ***plus*** file system navigation. Though less interesting from a conceptual point of view, FOXpath 1.0 may nevertheless be an interesting option for implementers and users who do not wish to deal with the complexities of node navigation and perhaps are simply not interested in the navigation of XML nodes.

Another question worth asking is whether integration of FOXpath and XPath may be achieved in a simpler way than provided by FOXpath 3.0. An extension of the path expression as introduced by FOXpath 3.0 is a far-reaching extension of the XPath language itself which may be regarded as a risk better not taken. As a conceivable alternative, a simple and risk-less integration is provided by a plain extension function

```
fox:foxpath($foxpath as xs:string) as item()*
```

which consumes a FOXpath expression and returns an XDM value. In comparison to the seamless integration of FOXpath into XPath provided by FOXpath 3.0, such a function offers a comparable gain of functionality, though with less elegance. Comparing these expressions,

FOXpath 3.0:

```
\projects\\parks[not(ancestor~::private]]\*
//animals/fox[not(trail)]
```

XPath + `fox:foxpath()`:

```
fox:foxpath(
"\projects\\parks[not(ancestor~::private]]\* ")/doc(.)
//animals/fox[not(trail)])
```

one may be inclined to start the introduction of FOXpath into XPath with an extension function, and postpone the integration of FOXpath into the expression architecture of the XPath language until the usefulness has been consolidated and any major issues revealed by field experience have been addressed.

While such a cautious approach to adoption is appealing, it should not blind us to the remarkable chances which a full integration (integration on the level of the expression language) promises. As discussed above (section “Generalization”), a file system is just a particular type of resource tree, whereas the concept of foxpath navigation refers to resource trees in general, rather than only the file system. This insight might boost our motivation to make resource tree navigation an integral part of the XPath language.

Appendix A. Grammar of the FOXpath 3.0 language

The grammar of the FOXpath 3.0 language is a modified copy of the XPath 3.0 grammar. Rule numbers with appended "a" (e.g. [901a]) identify a ***new*** rule. Rule numbers with appended "m" (e.g. [35m]) identify a rule obtained by ***modifying*** a rule of the XPath 3.0 grammar; the rule number of the original rule in the XPath 3.0 grammar is equal to the rule number of the modified rule without the "m" postfix: for instance, rule [35m] is a modified copy of XPath 3.0 rule [35]. Numbers without postfix "a" or "m" identify rules which have been defined by the XPath 3.0 grammar and are retained by the FOXPath 3.0 grammar.

Comments or extra-grammatical constraints on grammar productions are between /* and */ symbols.

- A "xgc:" prefix is an extra-grammatical constraint, explained in Appendix B.
- A "gn:" prefix means a "Grammar Note", and is meant as a clarification for parsing rules.

```
[1m] FOXPath                  ::= Prolog Expr
[901a] Prolog                 ::= ((DefaultNamespaceDecl | NamespaceDecl) Separator)* (VarDecl Seperator)*
[902a] Seperator              ::= ";"
[903a] NamespaceDecl          ::= "declare" "namespace" NCName "=" URILiteral
[904a] DefaultNamespaceDecl ::= "declare" "default" "element" "namespace" URILiteral
[905a] VarDecl                ::= "variable" "$" VarName TypeDeclaration? ((":=" VarValue) | ("external" (":="
VarDefaultValue)?))
[906a] URILiteral             ::= StringLiteral
[907a] VarValue               ::= ExprSingle
[908a] VarDefaultValue        ::= ExprSingle
[2] ParamList                 ::= Param ("," Param)*
[3] Param                     ::= "$" EQName TypeDeclaration?
[4] FunctionBody              ::= EnclosedExpr
[5] EnclosedExpr              ::= "{" Expr "}"
[6] Expr                      ::= ExprSingle ("," ExprSingle)*
[7m] ExprSingle               ::= SimpleFLWORExpr
                                    | QuantifiedExpr
                                    | IfExpr
                                    | OrExpr
[909a] SimpleFLWORExpr        ::= (SimpleForClause | SimpleLetClause)+ "return" ExprSingle
[9] SimpleForClause           ::= "for" SimpleForBinding (","
                                  SimpleForBinding)*
[10] SimpleForBinding         ::= "$" VarName "in" ExprSingle
[11] LetExpr                  ::= SimpleLetClause "return" ExprSingle
[12] SimpleLetClause          ::= "let" SimpleLetBinding (","
                                  SimpleLetBinding)*
[13] SimpleLetBinding         ::= "$" VarName ":=" ExprSingle
[14] QuantifiedExpr           ::= ("some" | "every") "$" VarName "in"
                                  ExprSingle ("," "$" VarName "in"
                                  ExprSingle)* "satisfies" ExprSingle
[15] IfExpr                   ::= "if" "(" Expr ")" "then" ExprSingle "else"
                                  ExprSingle
[16] OrExpr                   ::= AndExpr ( "or" AndExpr )*
[17] AndExpr                  ::= ComparisonExpr ( "and" ComparisonExpr )*
[18] ComparisonExpr           ::= StringConcatExpr ( (ValueComp
                                    | GeneralComp
                                    | NodeComp) StringConcatExpr )?
[19] StringConcatExpr         ::= RangeExpr ( "||" RangeExpr )*
[20] RangeExpr                ::= AdditiveExpr ( "to" AdditiveExpr )?
[21] AdditiveExpr             ::= MultiplicativeExpr ( ("+" | "-")
                                  MultiplicativeExpr )*
[22] MultiplicativeExpr       ::= UnionExpr ( ("*" | "div" | "idiv" | "mod")
                                  UnionExpr )*
[23] UnionExpr                ::= IntersectExceptExpr ( ("union" | "|")
                                  IntersectExceptExpr )*
[24] IntersectExceptExpr      ::= InstanceofExpr ( ("intersect" | "except")
                                  InstanceofExpr )*
[25] InstanceofExpr           ::= TreatExpr ( "instance" "of" SequenceType
                                  )?
[26] TreatExpr                ::= CastableExpr ( "treat" "as" SequenceType
                                  )?
[27] CastableExpr             ::= CastExpr ( "castable" "as" SingleType )?
[28] CastExpr                 ::= UnaryExpr ( "cast" "as" SingleType )?
[29] UnaryExpr                ::= ("-" | "+")* ValueExpr
[30] ValueExpr                ::= SimpleMapExpr
```

```
[31] GeneralComp             ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[32] ValueComp               ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[33] NodeComp                ::= "is" | "<<" | ">>"
[34] SimpleMapExpr           ::= PathExpr ("!" PathExpr)*

[35m] PathExpr               ::= ("/" RelativePathExpr?)
                                  | ("//" RelativePathExpr)
                                  | RelativePathExpr
                                  | (DriveSelector? "\" RelativePathExpr?)
                                  | (DriveSelector? "\\" RelativePathExpr)
[36m] RelativePathExpr       ::= StepExpr (
                                 ("/"
                                  | "//"
                                  | (DriveSelector? "\")
                                  | (DriveSelector? "\\")
                                 ) StepExpr)*`
[37m] StepExpr               ::= PostfixExpr | AxisStep | FoxAxisStep

[38] AxisStep                ::= (ReverseStep | ForwardStep) PredicateList
[39] ForwardStep             ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[40] ForwardAxis             ::= ("child" "::")
                                  | ("descendant" "::")
                                  | ("attribute" "::")
                                  | ("self" "::")
                                  | ("descendant-or-self" "::")
                                  | ("following-sibling" "::")
                                  | ("following" "::")
                                  | ("namespace" "::")
[41] AbbrevForwardStep       ::= "@"? NodeTest
[42] ReverseStep             ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[43] ReverseAxis             ::= ("parent" "::")
                                  | ("ancestor" "::")
                                  | ("preceding-sibling" "::")
                                  | ("preceding" "::")
                                  | ("ancestor-or-self" "::")
[44] AbbrevReverseStep       ::= ".."
[45] NodeTest                ::= KindTest | NameTest
[46] NameTest                ::= EQName | Wildcard
[47] Wildcard                ::= "*"
                                  | (NCName ":" "*")
                                  | ("*" ":" NCName)
                                  | (BracedURILiteral "*")                        /* ws: explicit
                                                                                   */
[910a] FoxAxisStep           ::= (ReverseFoxStep | ForwardFoxStep) PredicateList
[911a] ForwardFoxStep        ::= (ForwardFoxAxis FoxNameTest) | AbbrevForwardFoxStep
[912a] ForwardFoxAxis        ::= ("child" "~::")
                                  | ("descendant" "~::"
                                  | ("self" "~::"
                                  | ("descendant-or-self" "~::")
                                  | ("following-sibling" "~::")
[913a] AbbrevForwardFoxStep ::= FoxNameTest
[914a] ReverseFoxStep        ::= (ReverseFoxAxis FoxNameTest) | AbbrevReverseFoxStep
[915a] ReverseFoxAxis        ::= ("parent" "~::")
                                  | ("ancestor" "~::")
                                  | ("preceding-sibling" "~::")
                                  | ("ancestor-or-self" "~::")
[916a] AbbrevReverseFoxStep ::= ("..." FoxNameTest)
                                  | ".."                                           /* xgc:
                                                                                   only-if-context-is-uri
                                                                                   */
[917a] FoxNameTest           ::= CanonicalFoxNameTest
                                  | AbbrevFoxNameTest
[918a] CanonicalFoxNameTest ::= "`" ([^`]|``)* "`"
[919a] AbbrevFoxNameTest     ::= (  [^~\[\]\\/<>()=!|,.d] | ([~] [~\[\]\\/<>()=!|,.\d])  )
                                  (   [^~\[\]\\/<>()=!|,\s] | ([~] [~\[\]\\/<>()=!|,\s]     )*

                                                                                   /* gn:abbrevFoxNameTest
                         Character sequence in which the following characters are escaped by preceding ~:
                            ~ [] \/ <> () =!|,\s
                         Additional constraint concerning the first character:
                            it must not be a digit or a dot unless escaped by preceding ~
                                                                                   */

[920a] DriveSelector         ::= DriveLetter ":"
[921a] DriveLetter           ::= [a-zA-Z]

[48] PostfixExpr             ::= PrimaryExpr (Predicate | ArgumentList)*
[49] ArgumentList            ::= "(" (Argument ("," Argument)*)? ")"
[50] PredicateList           ::= Predicate*
[51] Predicate               ::= "[" Expr "]"
[52] PrimaryExpr             ::= Literal
                                  | VarRef
                                  | ParenthesizedExpr
                                  | ContextItemExpr
                                  | FunctionCall
                                  | FunctionItemExpr
[53] Literal                 ::= NumericLiteral | StringLiteral
[54] NumericLiteral          ::= IntegerLiteral | DecimalLiteral |
                                 DoubleLiteral
[55] VarRef                  ::= "$" VarName
[56] VarName                 ::= EQName
[57] ParenthesizedExpr       ::= "(" Expr? ")"
[58] ContextItemExpr         ::= "."
[59] FunctionCall            ::= EQName ArgumentList                             /* xgc:
                                                                                   reservedfunctionnames
                                                                                   */
                                                                                   /* gn: parens
```

```
                                                                    */
[60] Argument                 ::= ExprSingle | ArgumentPlaceholder
[61] ArgumentPlaceholder     ::= "?"
[62] FunctionItemExpr        ::= NamedFunctionRef | InlineFunctionExpr
[63] NamedFunctionRef        ::= EQName "#" IntegerLiteral                  /* xgc:
                                                                    reservedfunctionnames
                                                                    */
[64] InlineFunctionExpr      ::= "function" "(" ParamList? ")" ("as"
                                  SequenceType)? FunctionBody
[65] SingleType               ::= SimpleTypeName "?"?
[66] TypeDeclaration          ::= "as" SequenceType
[67] SequenceType             ::= ("empty-sequence" "(" ")")
                                  | (ItemType OccurrenceIndicator?)
[68] OccurrenceIndicator     ::= "?" | "*" | "+"                            /* xgc:
                                                                    occurrenceindicators
                                                                    */
[69] ItemType                 ::= KindTest | ("item" "(" ")") | FunctionTest
                                  | AtomicOrUnionType |
                                  ParenthesizedItemType
[70] AtomicOrUnionType       ::= EQName
[71] KindTest                 ::= DocumentTest
                                  | ElementTest
                                  | AttributeTest
                                  | SchemaElementTest
                                  | SchemaAttributeTest
                                  | PITest
                                  | CommentTest
                                  | TextTest
                                  | NamespaceNodeTest
                                  | AnyKindTest
[72] AnyKindTest              ::= "node" "(" ")"
[73] DocumentTest             ::= "document-node" "(" (ElementTest |
                                  SchemaElementTest)? ")"
[74] TextTest                 ::= "text" "(" ")"
[75] CommentTest              ::= "comment" "(" ")"
[76] NamespaceNodeTest       ::= "namespace-node" "(" ")"
[77] PITest                   ::= "processing-instruction" "(" (NCName |
                                  StringLiteral)? ")"
[78] AttributeTest           ::= "attribute" "(" (AttribNameOrWildcard (","
                                  TypeName)?)? ")"
[79] AttribNameOrWildcard   ::= AttributeName | "*"
[80] SchemaAttributeTest    ::= "schema-attribute" "("
                                  AttributeDeclaration ")"
[81] AttributeDeclaration   ::= AttributeName
[82] ElementTest              ::= "element" "(" (ElementNameOrWildcard (","
                                  TypeName "?"?)?)? ")"
[83] ElementNameOrWildcard  ::= ElementName | "*"
[84] SchemaElementTest       ::= "schema-element" "(" ElementDeclaration
                                  ")"
[85] ElementDeclaration     ::= ElementName
[86] AttributeName           ::= EQName
[87] ElementName              ::= EQName
[88] SimpleTypeName          ::= TypeName
[89] TypeName                 ::= EQName
[90] FunctionTest             ::= AnyFunctionTest
                                  | TypedFunctionTest
[91] AnyFunctionTest         ::= "function" "(" "*" ")"
[92] TypedFunctionTest       ::= "function" "(" (SequenceType (","
                                  SequenceType)*)? ")" "as" SequenceType
[93] ParenthesizedItemType  ::= "(" ItemType ")"
[94] EQName                   ::= QName | URIQualifiedName
```

Appendix B. Extra-grammatical Constraint

This section defines a constraint on the EBNF productions, which is required to parse syntactically valid sentences. Further extra-grammatical constraints referenced by the productions in (Appendix A) are defined by the XPath 3.0 specification (8) and are not repeated here.

only-if-context-is-uri

The rule or rule branch annotated may only be used in a context where the context item is known to originate from a fox axis step. See section “Context-dependent parsing” for a description how to determine whether the constraint is met by a given expression.

References

[1] BaseX - an open source XML database. Homepage. http://basex.org

[2] EXPath Community Group. Homepage. https://www.w3.org/community/expath/

[3] Gruen, Christian et al, eds. File Module 1.0 ExPath Module 20 February 2015. http://expath.org/spec/file/1.0

[4] glob (programming). Wikipedia article. https://en.wikipedia.org/wiki/Glob_%28programming%29

[5] JSONPath - project homepage on https://code.google.com. https://code.google.com/archive/p/jsonpath/

[6] The JXPath component - project homepage on https://commons.apache.org/proper/commons-jxpath/. https://commons.apache.org/proper/commons-jxpath/

[7] WildFly application server. Homepage. http://wildfly.org/

[8] Robie, Jonathan, et al., eds. XML Path Language (XPath), W3C Recommendation 08 April 2014. https://www.w3.org/TR/2014/REC-xpath-30-20140408/

[9] Robie, Jonathan, et al., eds. XML Path Language (XPath), W3C Candidate Recommendation 17 December 2015. https://www.w3.org/TR/2014/REC-xpath-30-20140408/

[10] Kay, Michael, ed. XPath and XQuery Functions and Operators 3.0. W3C Recommendation 08 April 2014. http://www.w3.org/TR/xpath-functions-31/

[11] Walsh, Norman, et al. XProc: An XML Pipeline Language. W3C Recommendation 11 May 2010. http://www.w3.org/TR/xpath-functions-31/

[12] Robie, Jonathan, Michael Dyck, eds. XQuery 3.1: An XML Query Language. W3C Candidate Recommendation 18 December 2014. http://www.w3.org/TR/xquery-31/

[13] Walsh, Norman et al, eds. XQuery and XPath Data Model 3.0. W3C Recommendation 8. April 2014 http://www.w3.org/TR/xpath-datamodel-30/

[14] Kay, Michael, ed. XSL Transformations (XSLT) Version 3.0. W3C Last Call Working Draft 2 October 2014. http://www.w3.org/TR/xslt-30/

[1] A definition of such a tree might be similar to this: a set of URIs composed of a common prefix followed by one or more steps, meeting the "prefix URIs MUST be folders" constraint not formally defined here.

Author's keywords for this paper: XPath; XML; file system

Balisage Series on Markup Technologies