XML and Semantic Web Technologies

II. XML / 4. XML Path Language (XPath)

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Economics and Information Systems
& Institute of Computer Science
University of Hildesheim
http://www.ismll.uni-hildesheim.de

II. XML / 4. XML Path Language (XPath)

XPath Specification

XML Path Language is an expression language for XSLT & XQuery consisting of

1. XQuery 1.0 and XPath 2.0 Data Model (Rec-2007/01/23),
2. XML Path Language (XPath) 2.0 (Rec-2007/01/23),
3. XQuery 1.0 and XPath 2.0 Functions and Operators (Rec-2007/01/23)

as well as further documents (Formal Semantics, Requirements, Use Cases, etc.).

XPath 2.0 is a superset of XPath 1.0 (REC-1999/11/16) that improves by

- using (node) sequences instead of node sets,
- exploiting type information available through XML Schema,
- adding some powerful language constructs (e.g., if- and for-expressions).

XPath 2.0 is implemented, e.g., in Saxon (but not yet in Xalan).

Abstract Types in XML Schema

In XML Schema types can serve two different purposes:

- as types to associate information items with,
- as basetypes for derived types.

If a type should only be used as basetype, it can be declared **abstract**.

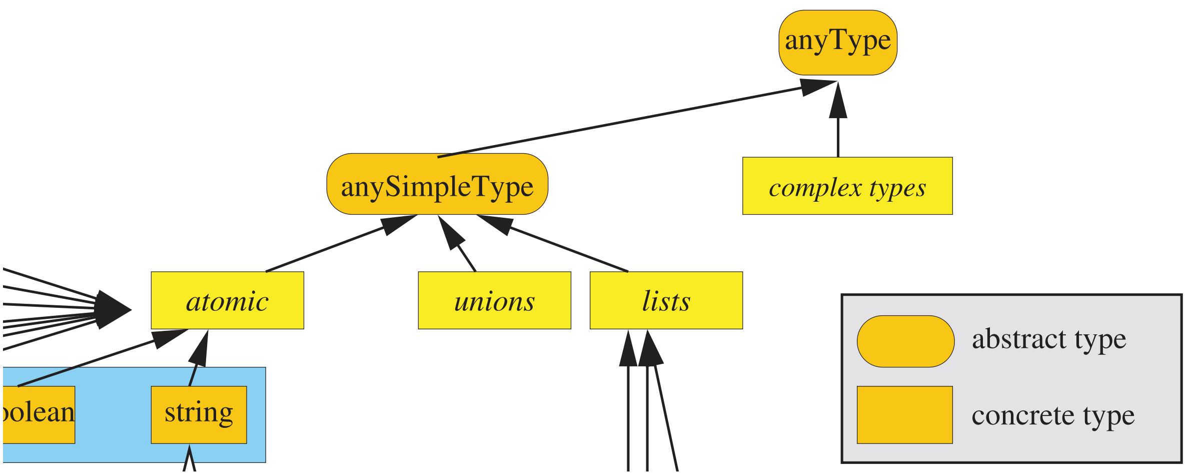

Figure 1: Abstract basetypes in XML Schema type hierarchy.

Additional Datatypes in XPath

There are 5 new datatypes defined in the XPath namespace

http://www.w3.org/2003/11/xpath-datatypes

- `untyped`,
- `anyAtomicType` (abstract) and `untypedAtomic`,
- and two duration types `dayTimeDuration` and `yearMonthDuration`.

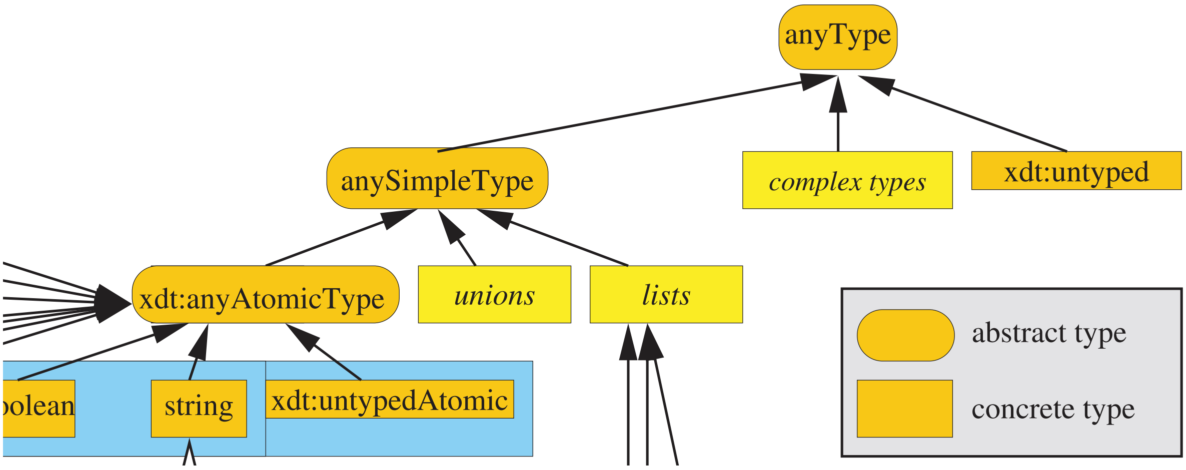

Figure 2: Additional types from XPath.

Node Kinds

The XPath Data Model describes a XML document as a tree with nodes of 7 different kinds:

document node unique root node of the tree
(≠ root element of the XML document !),

element node for each element,

text node for character data in element contents,

processing-instruction node for each PI,

comment node for each comment,

attribute node for each attribute of each element
(in most contexts not regarded as node, e.g., `node()`),

namespace node for each xmlns-attribute of each element
(no longer exposed in XPath 2.0).

Only element nodes can occur as interior nodes of the tree.

```
<?xml version="1.1"?>
<!-- first ideas -->
<?xml-stylesheet href='article.css' type='text/css'?>
<article author="John Doe" version="2004/06/07">
  <title>What <em>others</em> say</title>
  A <em>short</em><!-- 20 pages--> overview ...
</article>
```

Figure 3: Sample document

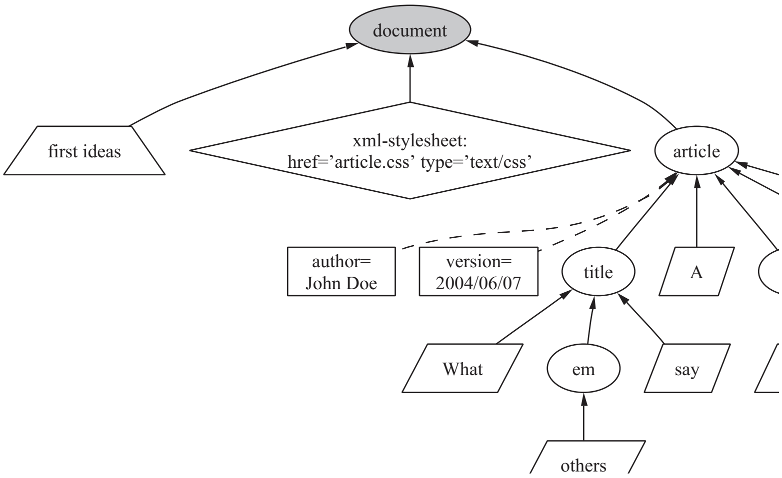

Figure 4: Document tree of the sample document.

Document Order

The set of nodes carries a total order called **document order** (that is partially implementation-dependent).
For two nodes x, y:

$x \prec y$:⇔ x is the parent of y,
or x and y are siblings and
(x is a namespace and y is not
or x is an attribute and y is neither a namespace nor an attribute
or x, y are elements, PIs, comments or text and x occurs in XML before y)

Document order is any total order that extends the transitive hull of $\prec$,
i.e., the order of

- two namespace nodes or
- two attribute nodes

of the same element is implementation-dependent.

11 Accessors

	document	element	attribute	namespace	PI	comment	text
node-kind	document	element	attribute	namespace	processing-instruction	comment	text
base-uri	base-uri/p	base-uri/p	/p	–	base-uri/p	/p	/p
parent	–	g	g	g	g	g	g
node-name	–	name	name	prefix	target	–	–
type	–	type	type	–	–	–	uA
string-value	cc	value/cc	value	ns-uri	content	content	content
typed-value	as uA	value/cc	value	as uA	as xs:string	as xs:string	as uA
children	g	g	–	–	–	–	–
attributes	–	g	–	–	–	–	–
namespaces	–	g	–	–	–	–	–
nilled	–	g	–	–	–	–	–

g = given / stored property, /p = or property of parent, – = empty list (), uA = xdt:untypedAtomic,
cc = concatenation of the contents of all its text-node descendants in document order.

Accessors / typed-value

For element or attribute nodes x:

$$\text{type}(x) := \begin{cases} \textit{QName of type of } x, & \text{if } x \text{ is schema-validated,} \\ \text{xdt:untypedAny}, & \text{if } x \text{ is an element node,} \\ \text{xdt:untypedAtomic}, & \text{if } x \text{ is an attribute node} \end{cases}$$

$$\text{string-value}(x) := \begin{cases} \textit{string representation of the value of } x, \\ \quad \text{if } x \text{ is of simple type or complex type / simple content} \\ \textit{concatenation of the contents of all its text-node descendants}, \\ \quad \text{otherwise} \end{cases}$$

$$\text{typed-value}(x) := \begin{cases} \textit{value of } x, \\ \quad \text{if } x \text{ is of simple type or complex type / simple content} \\ \text{string-value}(x) \textit{ as } \text{xdt:untypedAtomic}, \\ \quad \text{if type}(x) = \text{xdt:untypedAny or complex type/mixed content} \\ \textit{error}, \\ \quad \text{if } x \text{ is of of complex type / complex content} \end{cases}$$

II. XML / 4. XML Path Language (XPath)

Axis Steps / Node Tests

⟨*PathExpr*⟩ := (`/` ⟨*RelativePathExpr*⟩?) | ⟨*RelativePathExpr*⟩

⟨*RelativePathExpr*⟩ := ⟨*StepExpr*⟩ (`/` ⟨*StepExpr*⟩)*

⟨*StepExpr*⟩ := ⟨*Axis*⟩ `::` ⟨*NodeTest*⟩ ⟨*Predicates*⟩ /* axis step */
| ⟨*PrimaryExpr*⟩ ⟨*Predicates*⟩ /* filter step */

⟨*Axis*⟩ := `self`
| `child` | `descendant` | `descendant-or-self`
| `following-sibling` | `following`
| `parent` | `ancestor` | `ancestor-or-self`
| `preceding-sibling` | `preceding`
| `attribute`

⟨*NodeTest*⟩ := ⟨*QName*⟩ | `*` | (⟨*NCName*⟩ `:` `*`) | (`*` `:` ⟨*NCName*⟩) /* NameTest */
| ⟨*KindTest*⟩

⟨*Predicates*⟩ := (`[` ⟨*Expr*⟩ `]`)*

Axis Steps / Axes

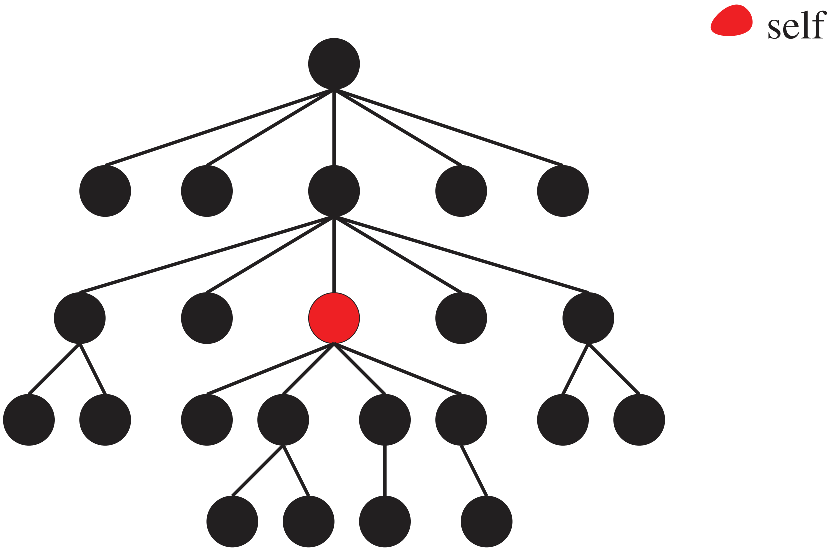

Figure 5: Self axis.

Axis Steps / Axes

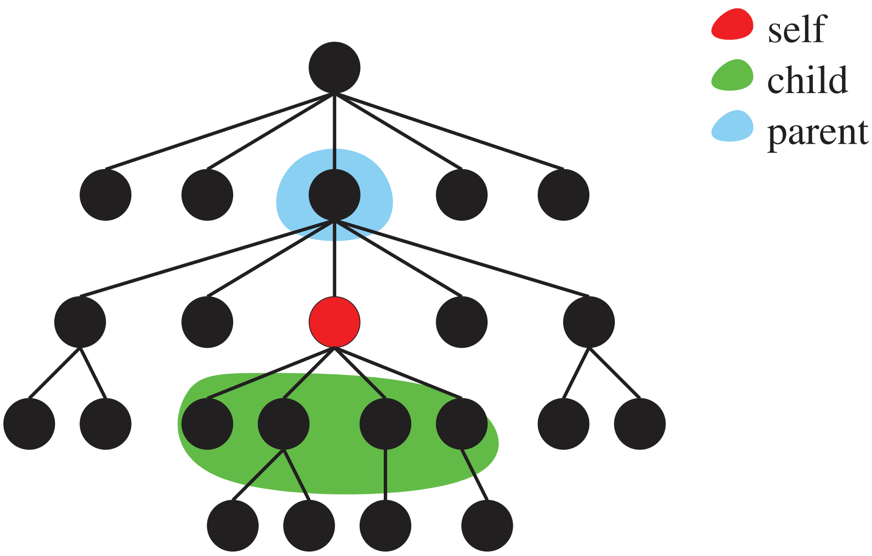

Figure 6: Child and parent axis.

Axis Steps / Axes

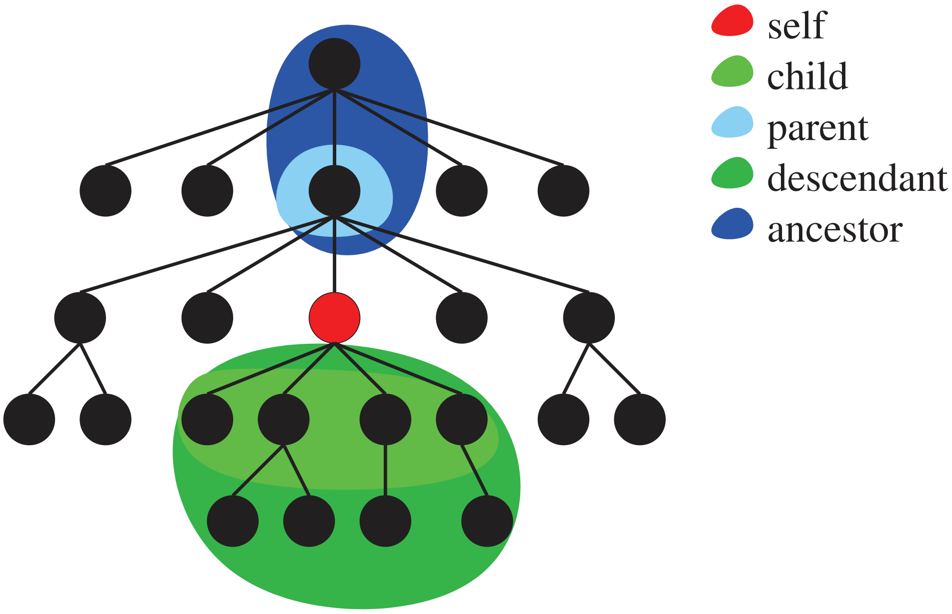

Figure 7: Descendant and ancestor axis.

Axis Steps / Axes

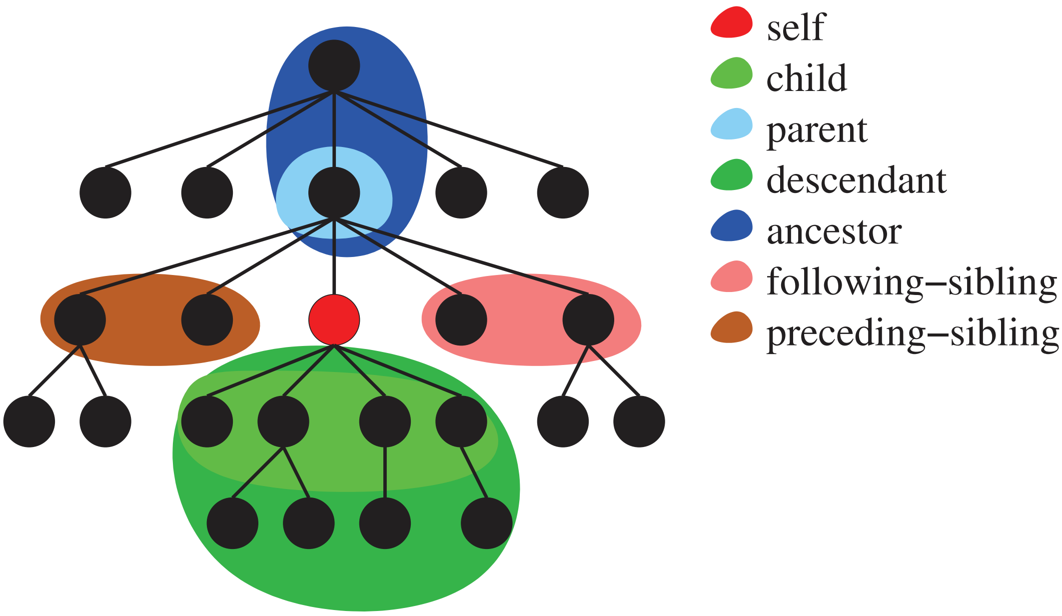

Figure 8: Following-sibling and preceding-sibling axis.

Axis Steps / Axes

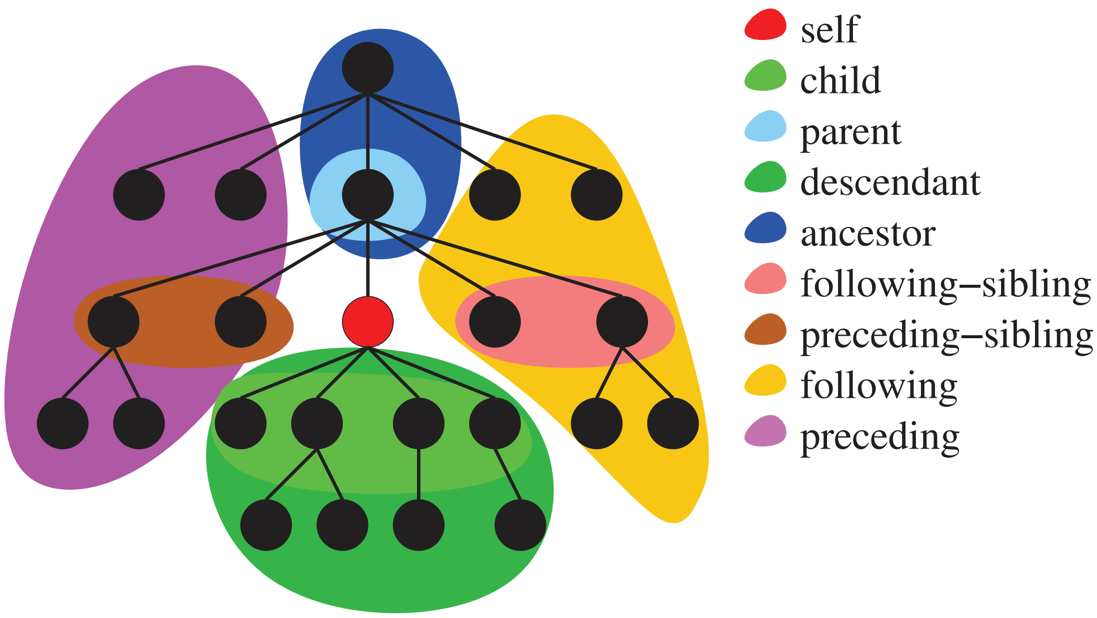

Figure 9: Following and preceding axis.

Axis Steps / Node Tests

Absolute path expressions start with the document node as **context node**, for **relative path expressions** the context node is set by the host language.

Step expressions successively shift the context node.

Axis selects a sequence of nodes relative to the context node ("scope").

Node tests allow to choose a subsequence of these nodes by tests on names or types / kinds.

Predicates allow more complex choices of subsequences of these nodes.

Sequences of nodes are always in document order.
Context positions are assigned starting from 1

- in document order for forward axes and
- in reverse document order for reverse axes.

Axis Steps / Node Tests / Example

```
<?xml version="1.1"?>
<books>
 <book>
  <author>R.E.</author><author>S.E.</author>
  <title>XML und DM</title></book>
 <book>
  <author>E.R.</author><title>Learning XML</title></book>
 <book>
  <author>N.W.</author><author>L.M.</author>
  <title>DocBook</title></book>
</books>
```

Figure 10: An abreviated books document `books-short.xml`.

Axis Steps / Node Tests / Example

Query: /descendant-or-self::title

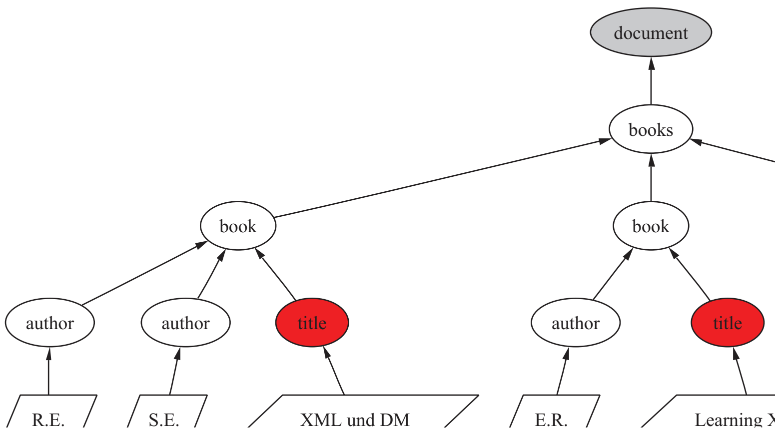

Figure 11: Result of XPath query /descendant-or-self::title.

Axis Steps / Node Tests / Example

Query: /descendant-or-self::title[contains(string(.),"XML")]

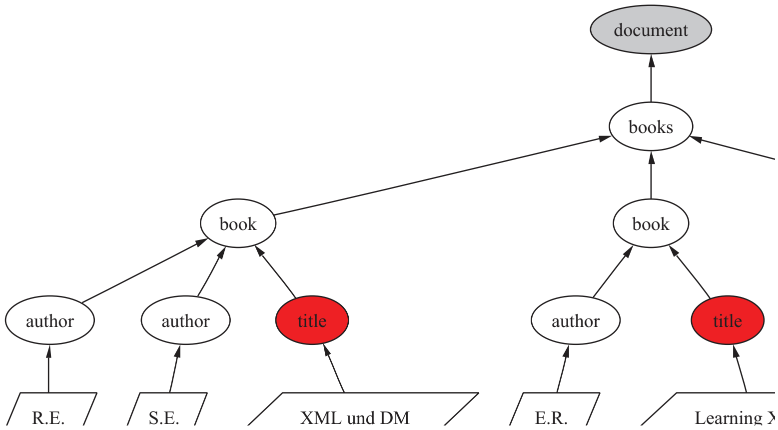

Figure 12: Result of XPath query /descendant-or-self::title[contains(string(.),"XML")].

Axis Steps / Node Tests / Example

Query: /descendant-or-self::title[contains(string(.),"XML")]/parent::node()

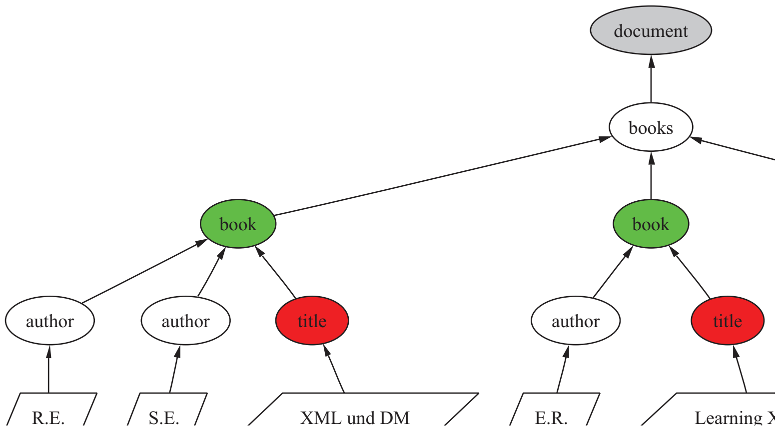

Figure 13: Result of XPath query /descendant-or-self::title[contains(string(.),"XML")]/parent::node().

Axis Steps / Node Tests / Example

Query: /descendant-or-self::title[contains(string(.),"XML")]/parent::node()/child::auth

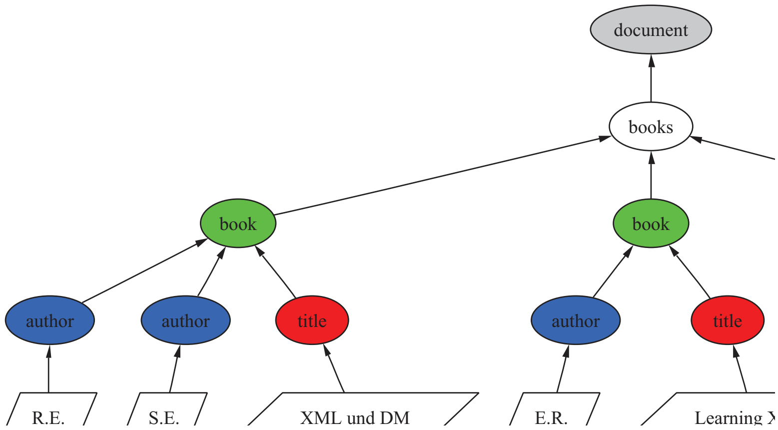

Figure 14: Result of XPath query /descendant-or-self::title[contains(string(.),"XML")]/parent::node()/ch

Performing XPath Queries by Saxon

XPath queries can be performed, e.g., by Saxon.

```
/descendant-or-self::title[contains(string(.),"XML")]/parent::node()/child::author
```

Figure 15: File `books.xpath` containing an XPath query.

call (with saxon.jar in classpath):

```
java net.sf.saxon.Query -s books-short.xml books.xpath
```

```
<?xml version="1.0" encoding="UTF-8"?>
<author>R.E.</author>
<?xml version="1.0" encoding="UTF-8"?>
<author>S.E.</author>
<?xml version="1.0" encoding="UTF-8"?>
<author>E.R.</author>
```

Figure 16: Result of the XPath query above.

Axis Steps / Kind Tests

⟨*KindTest*⟩ := `document-node (` ElementTest? `)`
| ElementTest
| `text ( )`
| `processing-instruction (` (⟨*NCName*⟩)? `)`
| `comment ( )`
| AttributeTest
| `node ( )`

⟨*ElementTest*⟩ := `element (` ((⟨*SchemaContextPath*⟩ ⟨*QName*⟩)
| ((⟨*QName*⟩ | `*`) (`,` (⟨*QName*⟩ | `*`) `nillable`?)?))? `)`

⟨*AttributeTest*⟩ := `attribute (` ((⟨*SchemaContextPath*⟩ ⟨*QName*⟩)
| (⟨*QName*⟩ | `*`) (`,` (⟨*QName*⟩ | `*`))?))? `)`

⟨*SchemaContextPath*⟩ := (⟨*QName*⟩ | (`type (` ⟨*QName*⟩ `)`)) `/` (⟨*QName*⟩ `/`)*

Axis Steps / Abbreviated Syntax

abbreviation	meaning
no axis name e.g., section/para	child:: axis child::section/child::para
@ as axis name e.g., section/@no	attribute:: axis child::section/attribute::no
// e.g., section//para	/descendant-or-self::node()/ child::section/descendant-or-self::node()/child::para
.. e.g., ../section	parent::node() parent::node()/child::section
[number] e.g., section[1]	[position()=number] section[position()=1]

/descendant-or-self::title[contains(string(.),"XML")]/parent::node()/
child::author[position()=1]

can be written more compactly as

//title[contains(string(.),"XML")]/../author[1]

Axis Steps / Abbreviated Syntax

Do not confuse

//section[1]
= /descendant-or-self::node()/child::section[1]

with

/descendant::section[1]

Accessors

Most accessors of the XPath data model can be queried:

accessor	XPath expression
node-kind	[Node-kind tests]
base-uri	base-uri(x)
parent	x/..
node-name	node-name(x)
	local-name(x)
	namespace-uri(x)
type	[castable-as and instance-of tests]
string-value	string(x)
typed-value	data(x)
children	x/node()
attributes	x/@*
namespaces	get-in-scope-prefixes(x)
	get-namespace-uri-for-prefix(prefix)
nilled	

If a sequence of atomic values is expected in a context,
then the typed value `data(x)` of a node is returned (**atomization**).

II. XML / 4. XML Path Language (XPath)

1. **XPath Data Model**

2. **XPath Path Expressions**

3. **XPath Expressions**

Expressions

$\langle$*Expr*$\rangle$:= $\langle$*ExprSingle*$\rangle$ (`,` $\langle$*ExprSingle*$\rangle$)*

$\langle$*ExprSingle*$\rangle$:= $\langle$*PrimaryExpr*$\rangle$
| $\langle$*Expr*$\rangle$ $\langle$*Operator*$\rangle$ $\langle$*Expr*$\rangle$
| $\langle$*PathExpr*$\rangle$
| $\langle$*ForExpr*$\rangle$
| $\langle$*QuantifiedExpr*$\rangle$
| $\langle$*IfExpr*$\rangle$
| $\langle$*TypeExpr*$\rangle$

$\langle$*ExprComment*$\rangle$:= `(:` ($\langle$*ExprCommentContent*$\rangle$ | $\langle$*ExprComment*$\rangle$)* `:)`

Primary expressions

⟨*PrimaryExpr*⟩ := ⟨*IntegerLiteral*⟩ | ⟨*DecimalLiteral*⟩ | ⟨*DoubleLiteral*⟩
| ⟨*StringLiteral*⟩
| `$` ⟨*QName*⟩ /* variable reference */
| `.` /* context item */
| ⟨*QName*⟩ `(` (⟨*ExprSingle*⟩ (`,` ⟨*ExprSingle*⟩)*)? `)`
/* function call */
| `(` ⟨*Expr*⟩? `)`

Variables can be bound by

- for-expressions,
- quantified expressions, and
- the host language (XSL, XQuery).

Working with Numbers

XPath has the usual operators for numerical values (+, -, *, `mod`).

Division is written as `div` (as `/` is already used for step-expressions).
`idiv` is used for interger division.

XPath has the basic functions `abs`, `ceiling`, `floor`, `round`.

function	returns
string-length(x)	length of string x
substring(x, f, l)	substring of x starting at f and of length l.
concat$(x, y, ...)$	concatenation of two or more strings
string-join(x, s)	concatenation of the strings in sequence x using separator s.
normalize-space(x)	whitespace-normalization of x.
upper-case(x)	upper-cased value of x.
lower-case(x)	lower-cased value of x.
translate(x, y, z)	x with all occurrences of characters in y replaced by characters in z at same position.
contains(x, y)	true, if x contains y.
starts-with(x, y)	true, if x starts with y.
ends-with(x, y)	true, if x ends with y.
substring-before(x, y)	substring of x before first occurrence of y.
substring-after(x, y)	substring of x after first occurrence of y.
matches(x, r)	true, if x matches the regular expression r.
replace(x, r, q)	x with all substrings matched by the regexp. r replaced by q.
tokenize(x, r)	a sequence of substrings of x separated by substrings of x that match the regexp. r.

Working with Sequences

Sequences can be explicitly constructed by the concatenation operator ",".

function	returns
count(s)	length of sequence s.
avg(s), sum(s), min(s), max(s)	average, sum, minimum, maximum of sequence s
zero-or-one(s), one-or-more(s), exactly-one(s)	s, if count$(s) \in \{0, 1\}$, ≥ 1, $= 1$.
distinct-values(s)	sequence containing each element of s exactly on
insert-before(s, i, t)	s with t inserted at position i.
remove(s, i)	s without item at position i.
reverse(s)	s in reverse order.
subsequence(s, f, l)	subsequence of s starting at f and of length l.
index-of(s, x)	sequence of positions at which x occurs in s.
empty(s), exists(s)	true, if count$(s) = 0$, $\neq 0$.

Strings are not sequences but atomic types !

Working with Sequence / Filter Steps

So called filter steps implement indexed access to sequences:

- $x[i]$ returns the i-th element of the sequence x.
  (with i a numeric expression).

- $x[b]$ returns all items of sequence x for which b evaluates to true
  (with b a boolean expression that may contain the context item ".").

"Filter steps" cannot be chained by "/" (contrary to axis steps).
But predicates "[...]" can be chained.

XPath expression	result
(1,3,2)[2]	3
(1,3,2)[. ge 2]	3,2
tokenize("The quick brown fox jumps over the lazy dog.", " ")[string-length(.) < 4]	"The", "fox", "the"
(1,3,2)[. ge 2][. lt 3]	2

Working with Sequence / Comparison Operators

XPath has 3 different sets of comparison operators:

value comparison: eq, ne, lt, le, gt, and ge.
Operands must be atomic, otherwise a type error is raised.

general comparison: =, !=, <, <=, >, and >=.
Operands may be sequences.
The comparison evaluates to true, if it holds between any two items in the respective sequences
(existentially quantification).

node comparison: is, <<, >>.
Operands must be single nodes.
"is" checks node identity, << and >> document order.

Sample expressions applied to `books-short.xml`:

XPath expression	result
//book[2]/author eq "E.R"	true
//book[1]/author eq "R.E"	[ERROR]
//book[1]/author = "R.E"	true

Working with Sequences of Nodes

expression	result
x union y, $x\|y$	sequence containing nodes in x or in y exactly once in document order
x intersect y	sequence containing nodes in x and in y exactly once in document order
x except y	sequence containing nodes in x but not in y exactly once in document order

These operators do not work for sequences of atomic values.

Sample expressions applied to `books-short.xml`:

expression	result
(//book[1]/author) union (//book[2]/author)	<author>R.E.</author> <author>S.E.</author> <author>E.R.</author>
(//book[2]/author) union (//book[2]/author)	<author>E.R.</author>

Loop Expressions (for)

⟨*ForClause*⟩ := `for $` ⟨*QName*⟩ `in` ⟨*ExprSingle*⟩
(`, $` ⟨*QName*⟩ `in` ⟨*ExprSingle*⟩)*
`return` ⟨*ExprSingle*⟩

for returns a sequence where each item is
the result of the evaluation of the return-expression
for the variables bound to the items of the for-expressions successively.

XPath variables are "read-only" and cannot be modified.

Variables bound by XPath expressions (as by for) are of local scope of that expressions.

Variables also can be bound by constructs of the host language (XSL, XQuery).

Loop Expressions (for)

```
<?xml version="1.1"?>
<books>
 <book isbn="0-596-00420-6">
  <author>Erik T. Ray</author><title>Learning XML</title><year>2003</year></bo
 <book isbn="1-565-92580-7">
  <author>Norman Walsh</author><author>Leonard Muellner</author>
  <title>DocBook: The Definitive Guide</title><year>1999</year></book>
 <book isbn="no">
  <author>Jon Doe</author><author>Alice Smith</author><author>Bob Miller</au
  <title>About something</title><year>1990</year></book>
</books>
```

Figure 17: Sample document.

```
for $x in //book return
 concat($x/author[1], ": ", $x/title, ", ", $x/year, ".")
```

Figure 18: Sample XPath query.

```
Erik T. Ray: Learning XML, 2003.
Norman Walsh: DocBook: The Definitive Guide, 1999.
Jon Doe: About something, 1990.
```

Figure 19: Result of the sample query on the sample document.

Conditional Expressions (if)

⟨*IfExpr*⟩ := `if (` ⟨*Expr*⟩ `) then` ⟨*ExprSingle*⟩ `else` ⟨*ExprSingle*⟩

If a boolean value is expected in a context (as here in the if-expression), then its **Effective Boolean Value** is computed:

$$\text{Effective Boolean Value}(x) := \begin{cases} \text{false}, & \text{if } x = \text{false} \\ \text{false}, & \text{if } x = () \text{ is the empty sequence} \\ \text{false}, & \text{if } x = \text{""} \text{ is the empty string} \\ \text{false}, & \text{if } x = 0 \text{ is of numeric type and zero} \\ \text{false}, & \text{if } x = \text{NaN is of type float/double and NaN} \\ \text{true}, & \text{otherwise} \end{cases}$$

There are not boolean literals, but functions `true()` and `false()`.

Conditional Expressions (if)

```
<?xml version="1.1"?>
<books>
  <book isbn="0-596-00420-6">
    <author>Erik T. Ray</author><title>Learning XML</title><year>2003</year></bo
  <book isbn="1-565-92580-7">
    <author>Norman Walsh</author><author>Leonard Muellner</author>
    <title>DocBook: The Definitive Guide</title><year>1999</year></book>
  <book isbn="no">
    <author>Jon Doe</author><author>Alice Smith</author><author>Bob Miller</au
    <title>About something</title><year>1990</year></book>
</books>
```

Figure 20: Sample document.

```
for $x in //book return
  if (count($x/author) ge 3) then
    concat($x/author[1], " et al.")
  else
    string-join($x/author, " and ")
```

Figure 21: Sample XPath query.

```
Erik T. Ray
Norman Walsh and Leonard Muellner
Jon Doe et al.
```

Figure 22: Result of the sample query on the sample document.

Quantified Expressions

```
⟨QuantifiedExpr⟩ := ( some | every )
                    $ ⟨QName⟩ in ⟨ExprSingle⟩
                    ( , $ ⟨QName⟩ in ⟨ExprSingle⟩ )*
                    satisfies ⟨ExprSingle⟩
```

```
//book[some $x in author satisfies contains($x, "R.")]
```

Figure 23: Sample XPath query.

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <author>R.E.</author>
   <author>S.E.</author>
    <title>XML und DM</title>
</book>
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <author>E.R.</author>
   <title>Learning XML</title>
</book>
```

Figure 24: Result of the sample query on the document `books-short.xml`.

Quantified Expressions

```
//book[every $x in author satisfies contains($x, "R.")]
```

Figure 25: Sample XPath query.

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <author>E.R.</author>
   <title>Learning XML</title>
</book>
```

Figure 26: Result of the sample query on the document `books-short.xml`.

Type Expressions (casting)

⟨*TypeExpression*⟩ := ⟨*ExprSingle*⟩
((`instance of` ⟨*SequenceType*⟩)
| (`treat as` ⟨*SequenceType*⟩)
| (`castable as` ⟨*SingleType*⟩)
| (`cast as` ⟨*SingleType*⟩))

⟨*SequenceType*⟩ := (⟨*QName*⟩ | ⟨*KindTest*⟩ | (`item ( )`)) (`?` | `*` | `+`)?)
| (`void ( )`)

⟨*SingleType*⟩ := ⟨*QName*⟩ `?`?

`instance` and `castable` check if a given expression is of given type.

`cast` casts an expression to a given type.

`treat` disables compile-time checks of expression types, but does not cast at runtime
(i.e., will throw an error, if the expression does not happen to be of correct type).

Type Expressions (casting)

To make use of XML Schema types, namespaces have to be declared by means of the host language (XSL, XQuery).

```
declare namespace xs="http://www.w3.org/2001/XMLSchema";

1 castable as xs:string
```

Figure 27: XPath expression using XML schema types, embedded in XQuery.

1 castable as xs:string	true
"Hello" castable as xs:decimal	false
(1,2,3) instance of xs:decimal*	true
(1,2,3) instance of xs:string*	false
concat(11, " is prime.")	[compile ERROR]
concat(11 cast as xs:string, " is prime.")	"11 is prime."
string-join((1 to 10) treat as xs:string*, ", ")	[compiles, but runtime ERROR]

Operator Precedence

prio.	operator	operand types
	/	
	unary +, -	numeric
	cast as	
	castable as	
	treat as	
	instance of	
	intersect, except	node-sequence
	union, |	node-sequence
	*, div, idiv, mod	numeric, durations
	+, -	numeric, dates
	to	integer
	eq, ne, lt, le, gt, ge, =, !=, <, <=, >, >= is, «, »	simple types node
	and	boolean
	or	boolean
	for-in-return, if-then-else, some/every-is-satisfies	
	,	