Datenbanksysteme 2015
Kapitel 10

Einführung in XML, XPath und XQuery

8.6.2015

Martin Gieseking

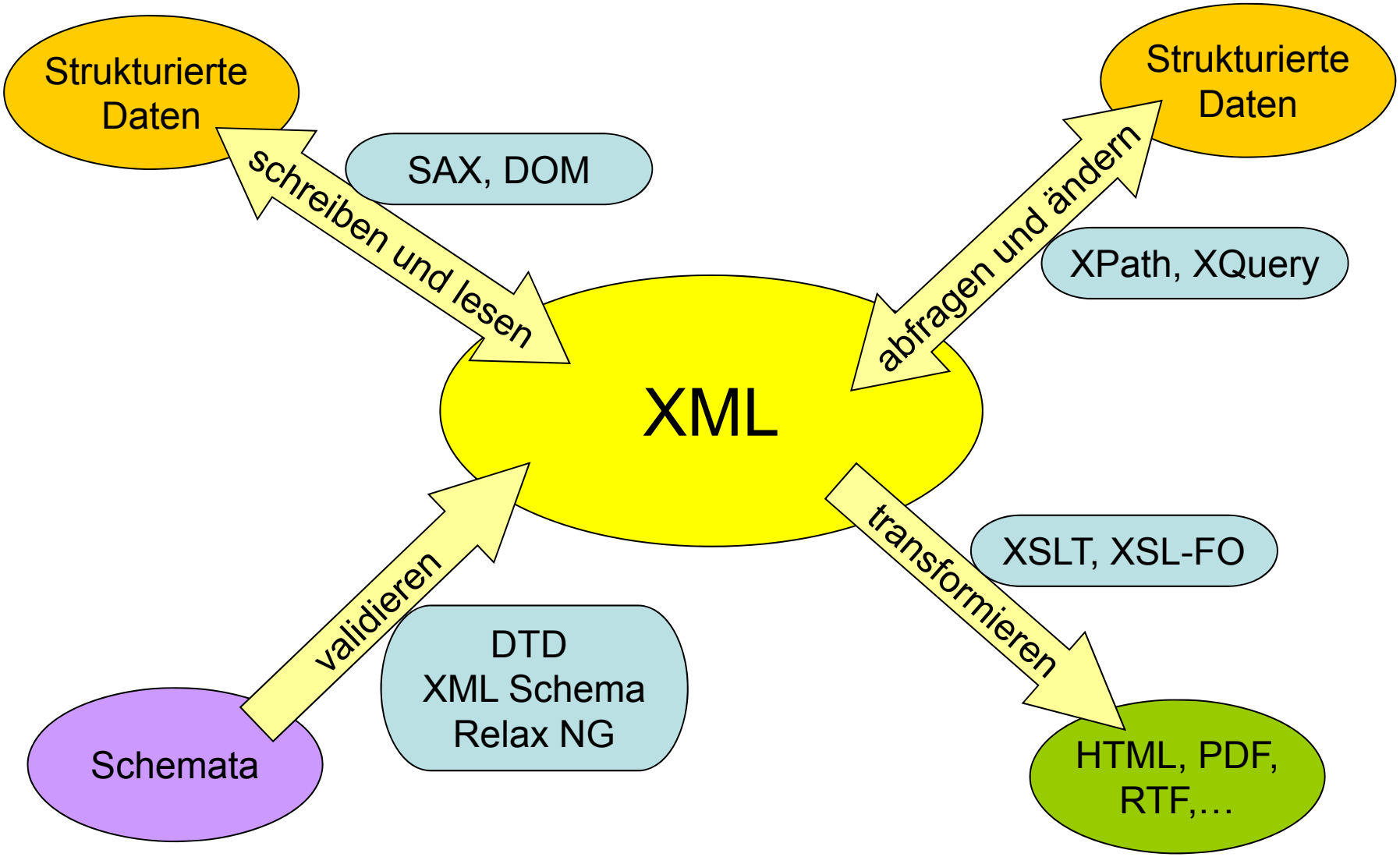
Was ist XML?

- XML (Extensible Markup Language) ist eine Meta-Auszeichnungssprache zur textbasierten Beschreibung hierarchisch strukturierter Daten
  - Spezifikation des World Wide Web Consortiums (W3C)
  - XML definiert kein konkretes Dateiformat sondern legt fest, welche syntaktischen Bestandteile verwendet und wie sie kombiniert werden dürfen
  - auf Grundlage der Regeln können konkrete Dateiformate mit definierter Struktur und Semantik abgeleitet werden
- ein konkretes Dateiformat, das auf Grundlage der XML-Spezifikation festgelegt wurde, wird allgemein **XML-Format** genannt
  - entsprechend nennt man Dateien in diesem Format **XML-Dateien**
  - Beispiele für XML-Formate sind *RSS*, *MathML*, *SVG*, *XHTML*

Beispiel einer XML-Datei

```
<?xml version="1.0"?>
<!-- Vorlesungsverzeichnis Sommersemester 2015 -->
<vorlesungsverzeichnis>
    <abschnitt titel="Mathematik &amp;amp; Informatik">
        <abschnitt titel="Grundstudium">
            <veranstaltung nr="1.234" typ="v">
                <dozent>
                    <titel>Prof. Dr.</titel>
                    <vname>Klaus</vname>
                    <nname>Meier</nname>
                </dozent>
                <titel>Einführung in die monotone Algebra I</titel>
                <zeit>Mo 10:00-12:00</zeit>
                <raum>08/15</raum>
            </veranstaltung>
            <veranstaltung nr="1.247" typ="ü">
                <dozent>
                    <vname>Sandra</vname>
                    <nname>Schmidt</nname>
                </dozent>
                <titel>Übung zur Einführung in die monotone Algebra I</titel>
                <zeit>Mi 8:00-12:00</zeit>
                <raum>47/11</raum>
            </veranstaltung>
        </abschnitt>
    </abschnitt>
</vorlesungsverzeichnis>
```

XML-Technologien



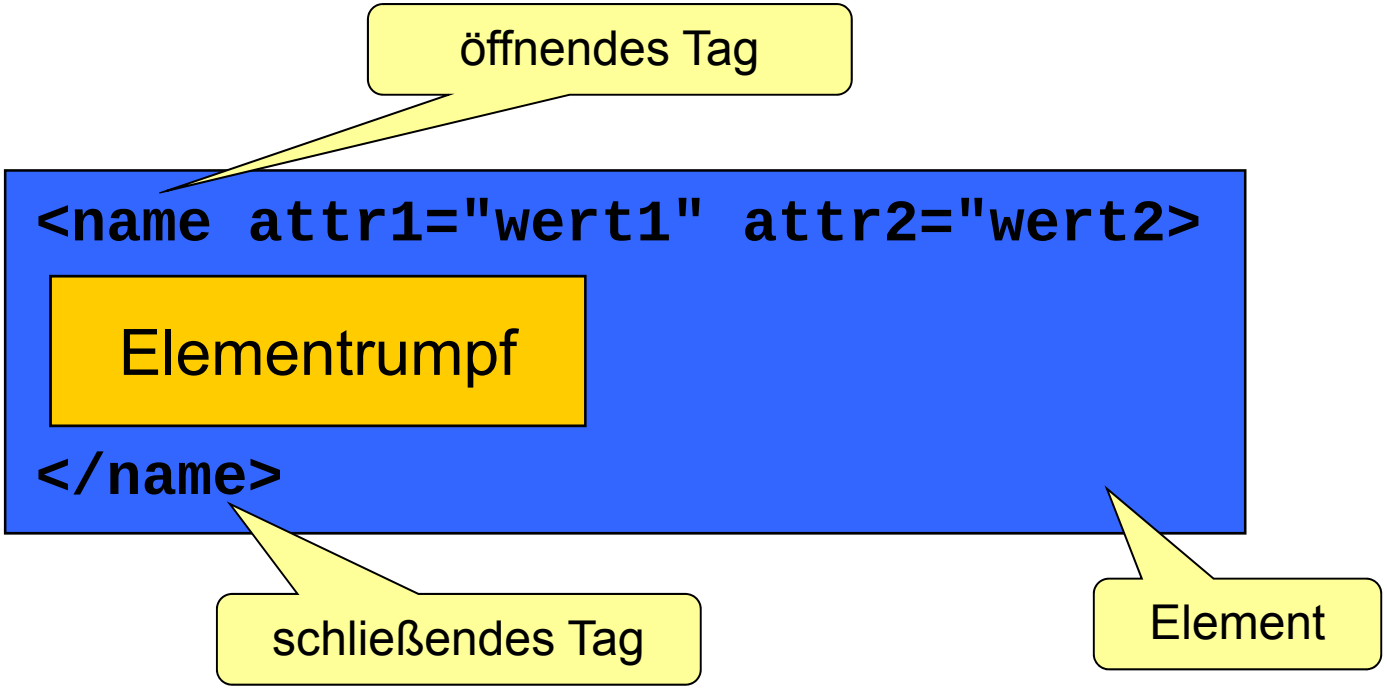
Bausteine einer XML-Datei

- XML-Dateien können aus folgenden Bausteinen zusammengesetzt werden:
  - Elemente
  - Attribute
  - Text und Entities
  - Kommentare
  - Verarbeitungsanweisungen (processing instructions)
  - CDATA-Abschnitte
  - DOCTYPE-Angabe

```
<?xml version="1.0"?>
<!-- Vorlesungsverzeichnis Sommersemester 2015 -->
<vorlesungsverzeichnis>
    <abschnitt titel="Mathematik &amp; Informatik">
        <abschnitt titel="Grundstudium">
            <veranstaltung nr="1.234" typ="v">
                <dozent>
                    <titel>Prof. Dr.</titel>
                    <vname>Klaus</vname>
                    <nname>Meier</nname>
                </dozent>
                <titel>Einführung in die ...</titel>
                <zeit>Mo 10:00-12:00</zeit>
                <raum>12/13</raum>
            </veranstaltung>
        </abschnitt>
    </abschnitt>
</vorlesungsverzeichnis>
```

Elemente

- die strukturierenden Bausteine eines XML-Dokuments werden **Elemente** genannt
- sie bestehen aus einem öffnenden und einem schließenden **Tag** sowie dem **Elementinhalt** (oder **Elementrumpf**)
  - anders als z.B. bei HTML muss jedes Element explizit ein öffnendes und schließendes Tag (Start- und End-Tag) besitzen



Elemente

- **öffnende Tags** haben immer die Form
  `<name attr1="wert1" attr2="wert2" ...>`
  - der Name kann aus Buchstaben, Ziffern und den Zeichen `_ . - ·` bestehen
    - erstes Zeichen muss ein Buchstabe sein
    - es wird zwischen Groß- und Kleinschreibung unterschieden
    - darf nicht mit *xml, Xml, xMl, xmL, XMl, XmL, xML* oder *XML* beginnen
  - zwischen „<“ und *name* darf sich kein Leerzeichen befinden
  - Attribute werden in der angegebenen Form durch Whitespaces getrennt hinter dem Elementnamen aufgelistet
  - für Attributnamen gelten die gleichen Vorgaben wie für Elementnamen
  - Attributwerte werden wahlweise in einfache oder doppelte Anführungszeichen eingeschlossen
  - die Reihenfolge der Attribute ist nicht signifikant
  - jeder Attributname darf in der Attributliste nur einmal auftauchen
- **schließende Tags** haben immer die Form
  `</name>`
  - der Name muss mit dem des zugehörigen öffnenden Tags übereinstimmen
  - schließende Tags enthalten keine weiteren Informationen

Elemente

- der Elementrumpf ist entweder leer oder besteht aus einer Folge von weiteren Elementen, Text, Kommentaren und/oder Verarbeitungsanweisungen
  - Elemente können beliebig tief geschachtelt werden

```
<veranstaltung nr="1.234" typ="v">
  <dozent geschlecht="m">
    <!- ein Kommentar -->
    <titel>Prof. Dr.</titel>
    <vname>Klaus</vname>
    <nname>Meier</nname>
  </dozent>
  Dies ist ein Text.
  <titel>Einführung in die ...</titel>
  <zeit>Mo 10:00-12:00</zeit>
  <raum>12/13</raum>
</veranstaltung>
```

- für Elemente mit leerem Rumpf gibt es die Kurzschreibweise **`<name attr1="val1" attr2="val2" ... />`**

```
<überschrift text="Erstes Kapitel"></überschrift>
```

ist identisch mit

```
<überschrift text="Erstes Kapitel"/>
```

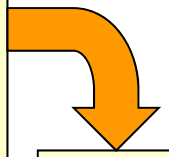
Elemente

- jedes XML-Dokument muss genau ein äußeres Element, das so genannte **Wurzelelement** besitzen
  - es stellt quasi den Rahmen oder den Container für die eigentlichen Daten dar
  - außerhalb des Wurzelelements sind nur Kommentare, Verarbeitungsanweisungen, eine DOCTYPE-Anweisung und Whitespace erlaubt

Fehler: kein (eindeutiges) Wurzelelement

```
<?xml version="1.0"?>
<veranstaltung nr="1.234">
  ...
</veranstaltung>

<veranstaltung nr="4.321">
  ...
</veranstaltung>
```



```
<?xml version="1.0"?>
<veranstaltungen>
  <veranstaltung nr="1.234">
    ...
  </veranstaltung>
  <veranstaltung nr="4.321">
    ...
  </veranstaltung>
</veranstaltungen>
```

Kommentare und Entities

- neben Text und Elementen spezifiziert XML noch weitere Konstrukte
  - Kommentare
    <!-- dies ist ein Kommentar -->
    - die Zeichenfolge **--** ist innerhalb von Kommentaren nicht erlaubt
  - Entity-Referenzen
    &name; &#dez; &#xhex;
    - bezeichnen ein einzelnes Zeichen oder eine Zeichenfolge
    - die Zeichen **<** und **&** haben in der XML-Syntax eine besondere Bedeutung (Metazeichen), und dürfen nicht als normaler Textbestandteil verwendet werden
    - zahlreiche Zeichen können oft nur umständlich direkt eingegeben werden, z.B. mathematische Symbole, Ligaturen, griechische Buchstaben, asiatische Zeichen usw.

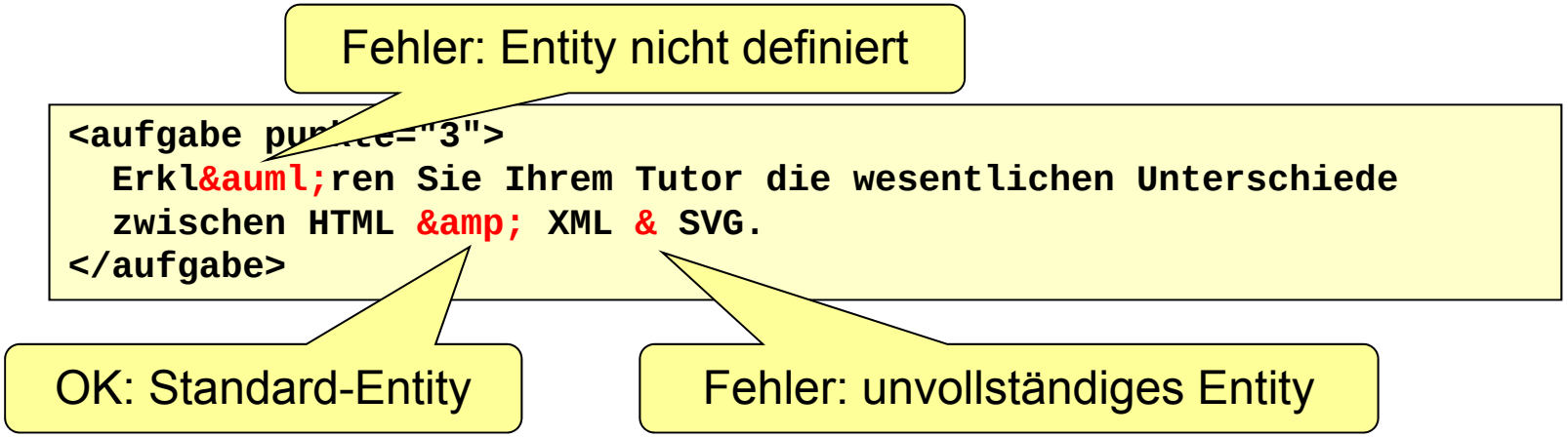
<	<
>	>
'	'
"	&quote;
&	&

```
<!- ein einfaches Entity-Beispiel -->
<code language="java">
  int a=0, b=0;
  if (a == 0 &amp;&amp; b &lt; 1)
    ...
</code>
```

Entity-Referenzen

- im Gegensatz zu HTML definiert XML nur fünf benannte Entities
- alle anderen aus HTML bekannten Entities, wie z.B. **`&auml; &szlig;  `** usw. sind in XML nicht definiert und führen bei Verwendung zu einem Fehler
- es gibt die Möglichkeit, neue benannte Entities zu definieren

<	<
>	>
'	'
"	&quote;
&	&



- numerische Entity-Referenzen bezeichnen jeweils ein einzelnes Unicode-Zeichen
  - Eurozeichen: **`&#8364;`** (dezimal), **`&#x20AC;`** (hexadezimal)

Wohlgeformte XML-Dokumente

- ein XML-Dokument heißt **wohlgeformt**, wenn es die Syntax- und Strukturvorgaben der XML-Spezifikation einhält
  - über 100 Regeln, die größtenteils intuitiv aus den bisher beschriebenen Aspekten hervorgehen
- die Wohlgeformtheit kann ohne Kenntnis der in einem Dokument zulässigen Elemente überprüft werden

nicht wohlgeformt

```
<?xml version="1.0"?>
<blatt nr="1">
  <aufgabe punkte="4">
    <list>
      <li>Punkt 1</li>
      <li>Punkt 2
  </aufgabe>
      </li>
    </list>
</Blatt>

<blatt nr="2">
  <aufgabe PUNKTE="6"/>
</blatt>
```

wohlgeformt

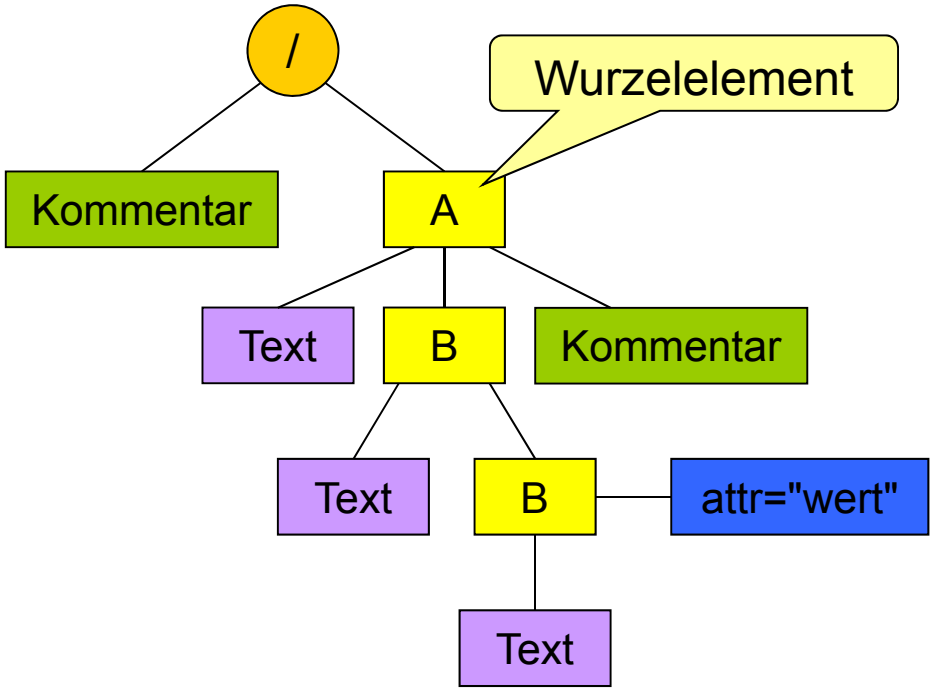
```
<?xml version="1.0"?>
<blattsammlung>
  <blatt nr="1">
    <aufgabe punkte="4">
      <list>
        <li>Punkt 1</li>
        <li>Punkt 2</li>
      </list>
    </aufgabe>
  </blatt>

  <blatt nr="2">
    <aufgabe PUNKTE="6"/>
  </blatt>
</blattsammlung>
```

Struktur von XML-Dokumenten

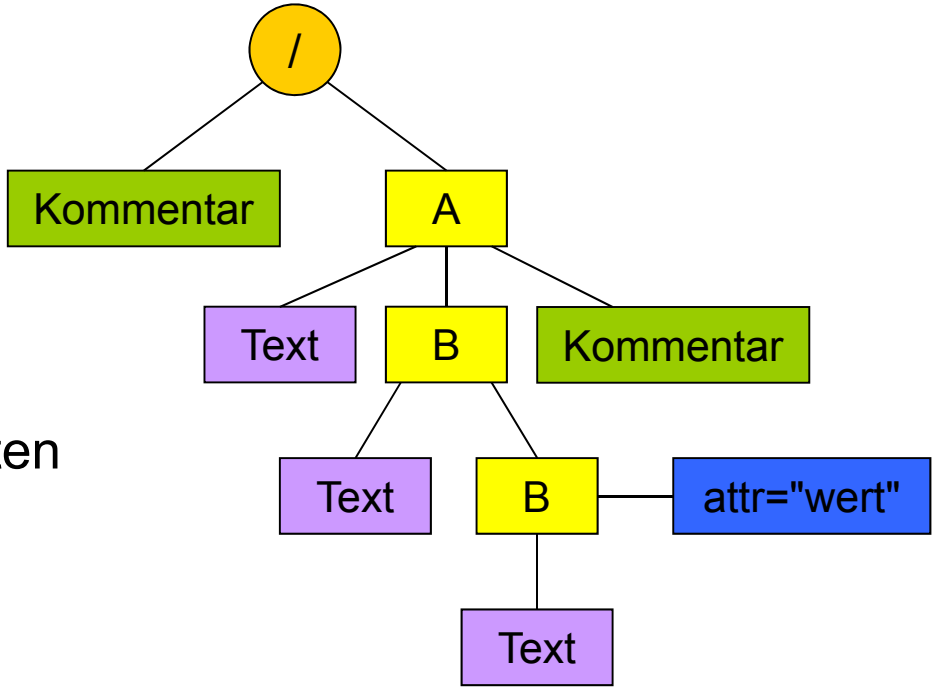
- XML-Dokumente besitzen grundsätzlich eine Baumstruktur
  - es gibt genau einen Wurzelknoten vom Typ *document*
  - bis auf den **Dokumentknoten** haben alle Knoten einen eindeutigen Elternknoten
  - Attribute sind sog. **assoziierte Knoten**, die zwar ein Elternelement besitzen, selbst aber keine Kinder ihrer Elemente sind
- der Dokumentknoten muss genau einen Element-Kindknoten, das **Wurzelelement**, besitzen

```
<?xml version="1.0"?>
<!- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B attr="wert">
      dritter Textteil
    </B>
  </B>
  <!- zweiter Kommentar -->
</A>
```



XML-Knotentypen

- entsprechend der syntaktischen Elemente einer XML-Datei wird zwischen verschiedenen Knotentypen unterschieden
  - Dokumentknoten
  - Elementknoten
  - Textknoten
  - Kommentarknoten
  - Attributknoten
  - Namensraumknoten
  - Verarbeitungsanweisungsknoten



XPath: Navigieren in XML-Bäumen

- eine zentrale Operation auf XML-Bäumen ist das Auswählen von einzelnen Knoten oder Knotenmengen nach bestimmten Kriterien

- die Lokatorsprache **XPath** stellt einen Formalismus zur Adressierung von Knoten in XML-Bäumen bereit
  - zusätzlich werden mathematische und logische Operatoren sowie eine überschaubare Anzahl von Funktionen bereitgestellt (XPath 1.0)

- XPath ist integraler Bestandteil von weiteren XML-Technologien, wie XSLT und XQuery (Obermenge von XPath 2.0/3.0)

XPath-Software (Open Source)

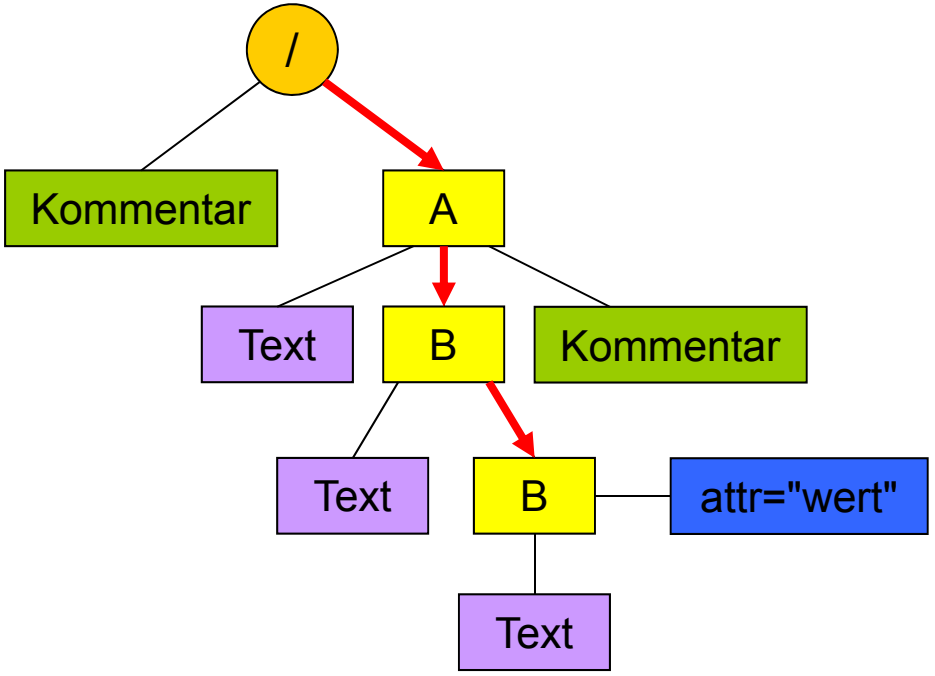
- Firebug- und FirePath-Erweiterung für Firefox
  - Firefox unterstützt wie alle anderen Browser nur XPath 1.0
- Libxml2 (http://xmlsoft.org)
  - Kommandozeilen-Programm **xmllint** mit Option **`--xpath`**

- BaseX (http://basex.org)
  - XML-Datenbank mit grafischer Oberfläche und XQuery-Schnittstelle
  - XQuery 1.0 ist eine Obermenge von XPath 2.0
- XQilla (http://xqilla.sourceforge.net/HomePage)
  - XQuery-Prozessor
  - C-Bibliothek und Kommandozeilen-Programm
- Zorba (http://zorba.28.io)
  - XQuery- und JSONiq-Prozessor
  - C++-Bibliothek und Kommandozeilen-Programm
  - Language-Bindings für weitere Programmiersprachen

Beispiel: einfache Pfadangabe

- Knoten werden in XPath mit Hilfe von **Pfadangaben** ausgewählt
  - einfache XPath-Pfadangaben erinnern an Unix-Pfade zur Navigation im Dateisystem
  - ausgehend von der Dokumentwurzel können die Kindknoten schrittweise ausgewählt werden
- das Ergebnis einer Pfadangabe ist immer eine **Knotenmenge**
  - eine geordnete Liste mit Referenzen auf alle passenden Knoten ohne Dopplungen
- Beispiel: **`/A/B/B`**

```
<?xml version="1.0"?>
<!- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B attr="wert">
      dritter Textteil
    </B>
  </B>
  <!- zweiter Kommentar -->
</A>
```



Pfadangaben und Location-Steps

- jede Pfadangabe besteht aus einer Folge sogenannter **Location-Steps**
  - werden durch Slashes (`/`) voneinander getrennt
  - Beispiel: **`step1/step2/step3`**
  - jeder Location-Step wählt relativ zur Ergebnismenge des vorangehenden Location-Steps bzw. zum Kontextknoten eine Knotenmenge des XML-Baums aus

- jeder Location-Step besteht aus maximal drei Komponenten
  - einem Knotentest
  - einem optionalen Achsenbezeichner
  - einem oder mehreren optionalen Prädikat(en)
  - Syntax: **`achenbezeichner::knotentest[prädikat]`**
  - Beispiel: **`ancestor::A[B]`**

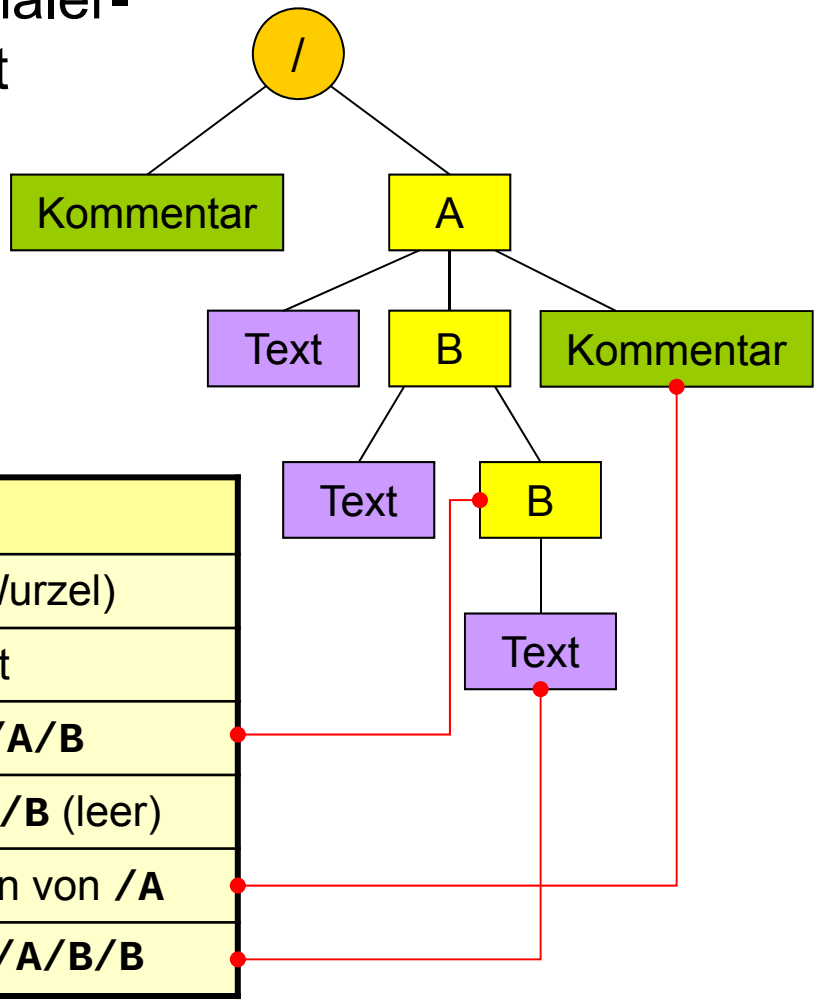
Knotentests

- im einfachsten Fall besteht ein Location-Step nur aus einem **Knotentest**
  - prüft, ob es an der aktuellen Position Knoten eines bestimmten Typs gibt
  - das Ergebnis eines Knotentests ist immer eine Menge aller passenden Knoten
  - falls es keine passenden Knoten gibt, ist die Ergebnismenge leer

Knotentyp	Syntax des Knotentests
Dokumentwurzel	`/`
Element mit dem Namen *A*	`A`
Element mit beliebigem Namen	`*`
Attribut mit dem Namen *attr*	`@attr`
Attribut mit beliebigem Namen	`@*`
Text	`text()`
Kommentar	`comment()`
alle Knotentypen	`node()`

Knotentests

- Die Knotentests **`text()`** und **`comment()`** selektieren Text- bzw. Kommentarknoten (unabhängig von ihrem Inhalt)
- Elemente und Attribute werden normalerweise über ihren Namen ausgewählt
- der Knotentest **`*`** selektiert Elementknoten unabhängig vom Namen
- der Knotentest **`@*`** selektiert Attributknoten unabhängig vom Namen

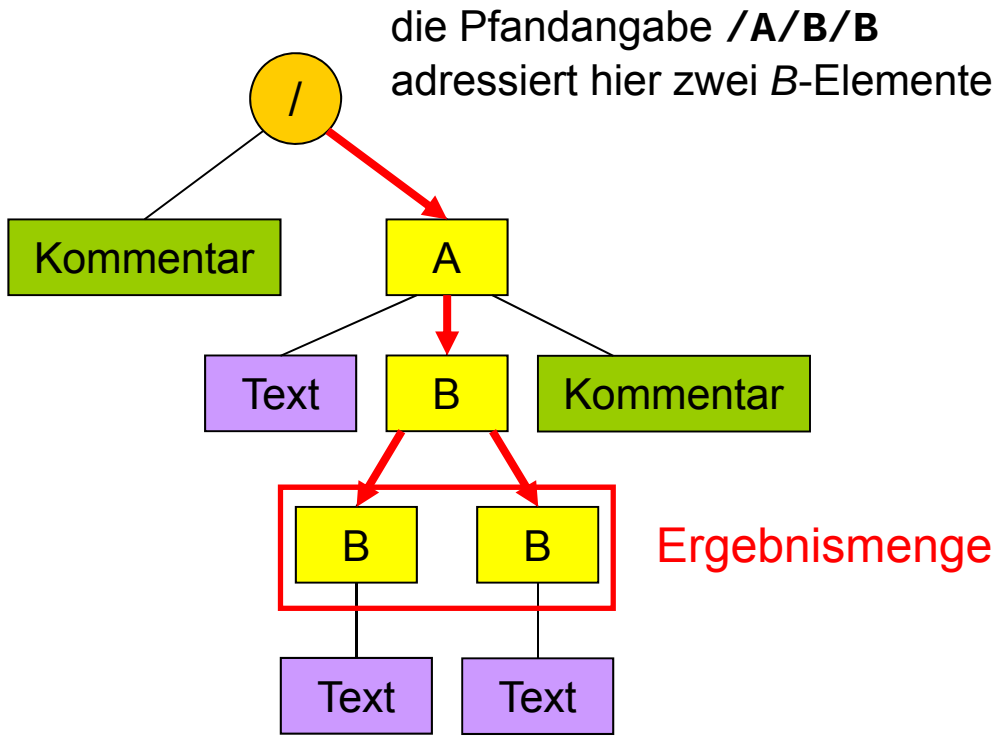


XPath-Ausdruck	Bedeutung
`/`	Dokumentknoten (Wurzel)
`/*`	Wurzelelement
`/A/B/*`	Kindelement von `/A/B`
`/A/B/A`	*A*-Kindelement von `/A/B` (leer)
`/A/comment()`	Kommentar-Kindknoten von `/A`
`/A/B/B/text()`	Text-Kindknoten von `/A/B/B`

Knotenmengen

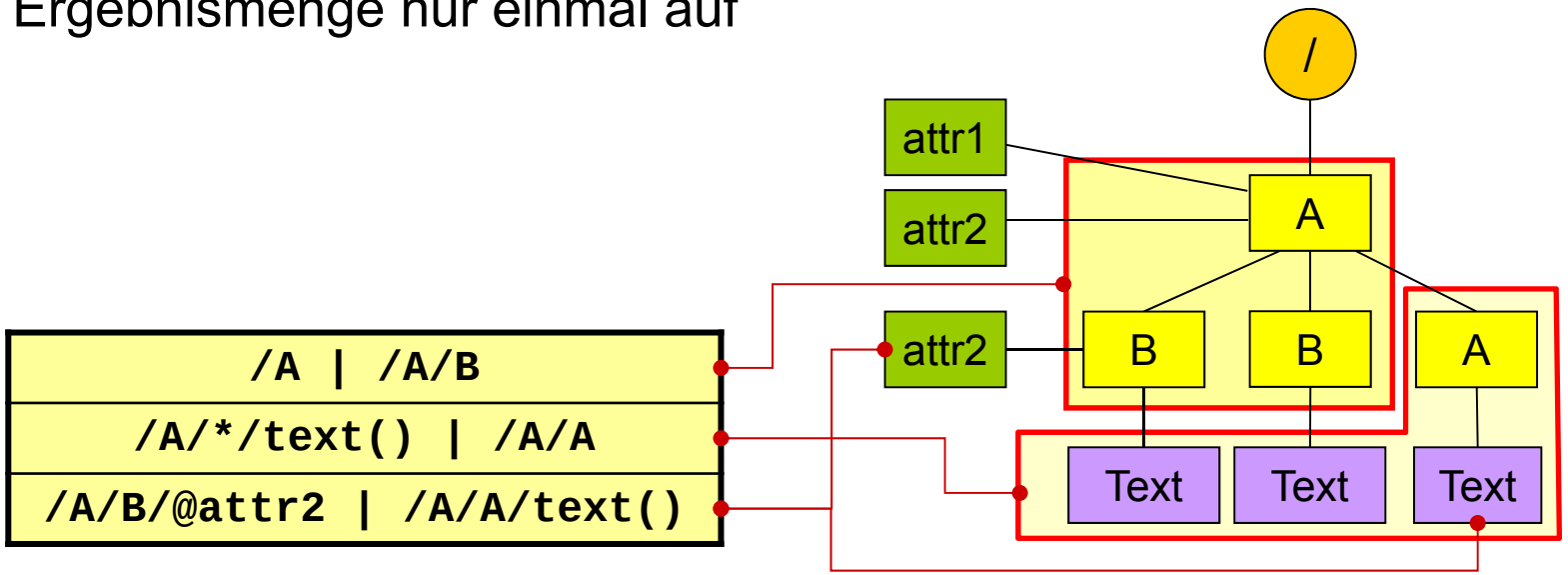
- da Elementknoten mehrere gleichnamige Knoten enthalten können, kann auch das Resultat eines Knotentests aus mehreren Knoten bestehen
  - das Resultat ist eine Knotenmenge
  - die Ergebnisknoten werden in Dokumentreihenfolge angeordnet

```
<?xml version="1.0"?>
<!- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B>
      dritter Textteil
    </B>
    <B>
      vierter Textteil
    </B>
  </B>
  <!- zweiter Kommentar -->
</A>
```



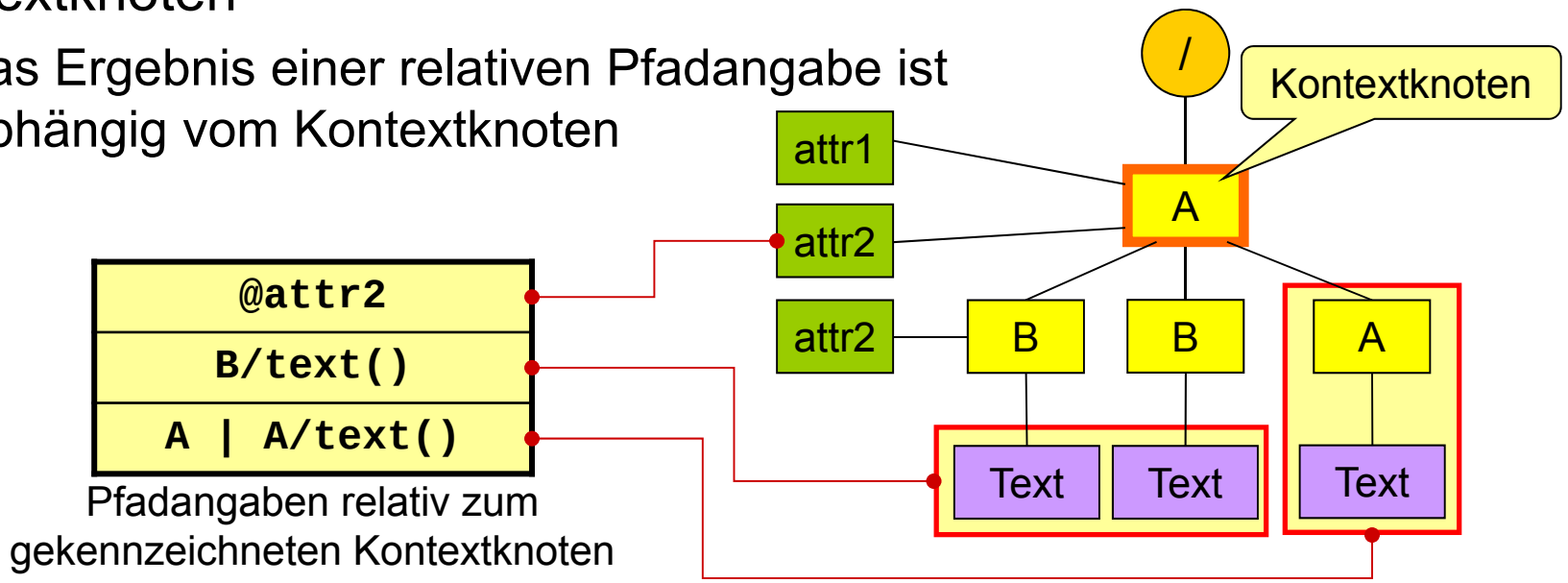
Vereinigung von Knotenmengen

- mit Hilfe des Kompositionsoperators | (senkrechter Strich) lassen sich zwei Knotenmengen vereinigen
  - auf beiden Seiten des Operators steht eine Pfadangabe
  - der Kompositionsoperator bindet schwächer als alle anderen Pfadoperatoren
  - die Knoten der Ergebnismenge sind immer in Dokumentreihenfolge angeordnet
  - Knoten, die von beiden Pfadangaben ausgewählt werden, tauchen in der Ergebnismenge nur einmal auf



Absolute und relative Pfadangaben

- XPath unterscheidet zwischen absoluten und relativen Pfadangaben
- **absolute Pfadangaben** beginnen immer bei der Dokumentwurzel
  - da die Dokumentwurzel mit einem Slash adressiert wird, beginnen absolute Pfadangaben immer mit einem Slash
  - das Ergebnis einer absoluten Pfadangabe ist für dasselbe XML-Dokument immer gleich
- **relative Pfadangaben** beziehen sich auf zuvor ausgewählte Kontextknoten
  - das Ergebnis einer relativen Pfadangabe ist abhängig vom Kontextknoten



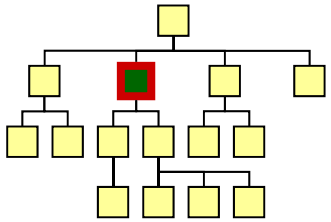
Pfadangaben relativ zum gekennzeichneten Kontextknoten

Achsen: Suchbereich von Knotentests angeben

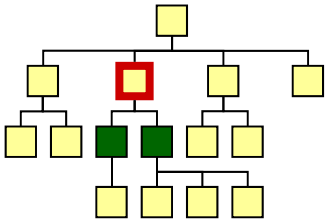
- wenn nicht anders angegeben, beziehen sich alle Kotentests immer auf die Kinder der aktuellen Kontextknoten
  - **`/A/B/C`** bedeutet:
    - suche alle *A*-Kindelemente des Dokumentknotens
    - suche in den gefundenen *A*-Elementen nach *B*-Kindelementen
    - suche in den gefundenen *B*-Elementen nach *C*-Kindelementen und liefere sie als Ergebnis zurück
  - jeder Location-Step bewirkt hier also einen Abstieg um eine Ebene im Baum
- mit Hilfe einer **Achsenangabe** kann der Bereich, auf den ein Knotentest angewendet werden soll, geändert werden, z.B. um
  - in Eltern- oder Geschwisterknoten zu suchen
  - rekursiv in allen Vorgänger- oder allen Folgeknoten zu suchen
  - Attribut- oder Namensraumknoten auszuwählen
- XPath 1.0 definiert 13 verschiedene Achsen

Achsen

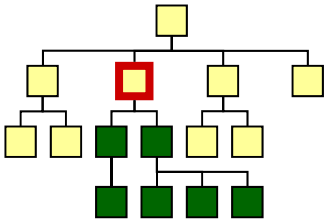
self



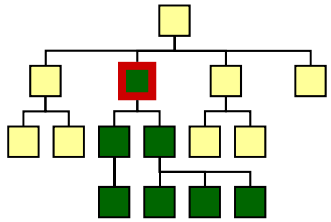
child

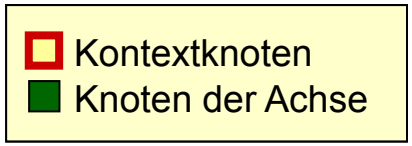


descendant

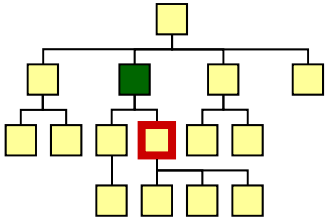


descendant-or-self

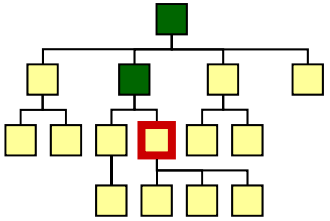




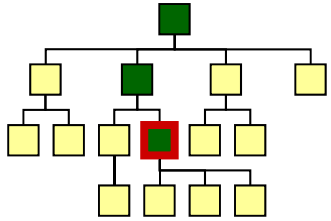
parent



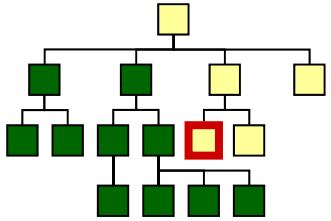
ancestor



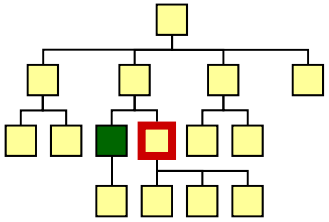
ancestor-or-self



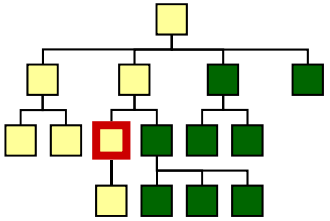
preceding



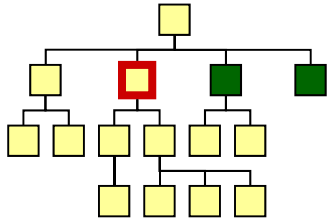
preceding-sibling



following



following-sibling



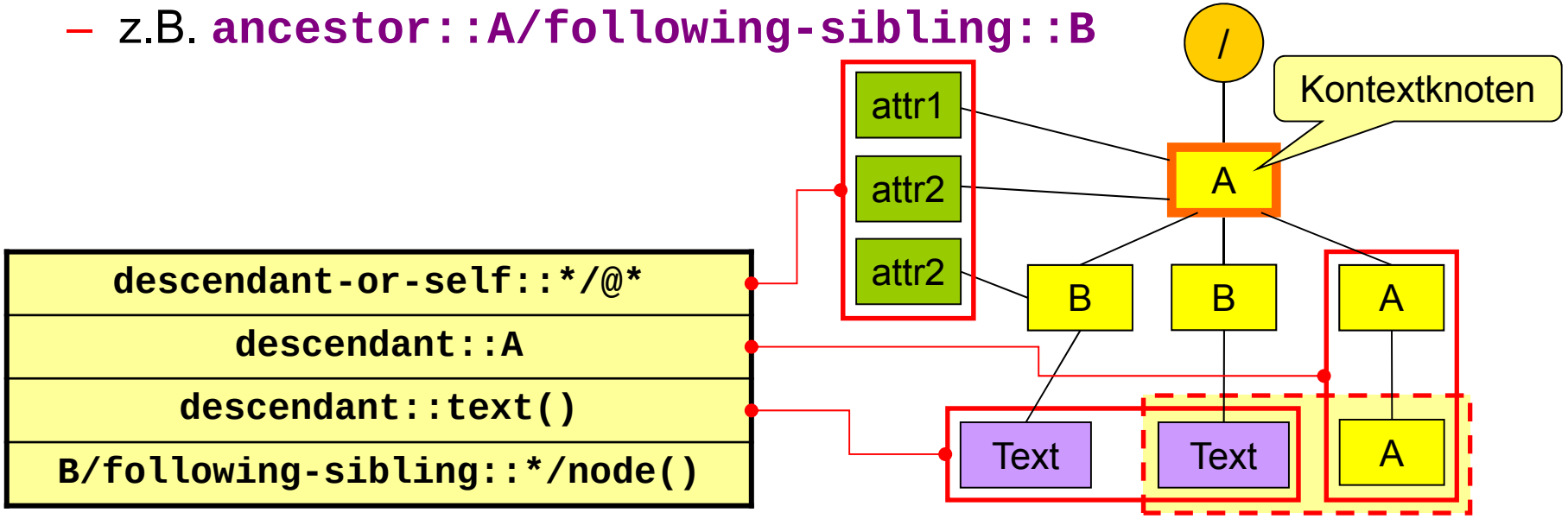
Achsen

Achse	Beschreibung
self	aktueller Knoten
child	alle unmittelbaren Kinder des aktuellen Knotens
descendant	alle Nachfahren des aktuellen Knotens
descendant-or-self	wie *descendant* aber zuzüglich des aktuellen Knotens
parent	Elternknoten des aktuellen Knotens
ancestor	alle Vorfahren des aktuellen Knotens
ancestor-or-self	wie *ancestor* aber zuzüglich des aktuellen Knotens
preceding	alle vorausgehenden Knoten ohne Vorfahren des aktuellen Knotens
preceding-sibling	alle vorausgehenden Geschwisterknoten
following	alle nachfolgenden Knoten ohne Nachfahren des aktuellen Knotens
following-sibling	alle nachfolgenden Geschwisterknoten
attribute	alle Attributknoten des aktuellen Elements

- die *preceding*-Achse umfasst alle Knoten, die sich im XML-Dokument vor dem Kontextknoten befinden und keine Vorfahren des Kontextknotens sind
- die *following*-Achse umfasst alle Knoten, die sich im XML-Datei hinter dem Kontextknoten befinden und keine Nachfahren des Kontextknotens sind

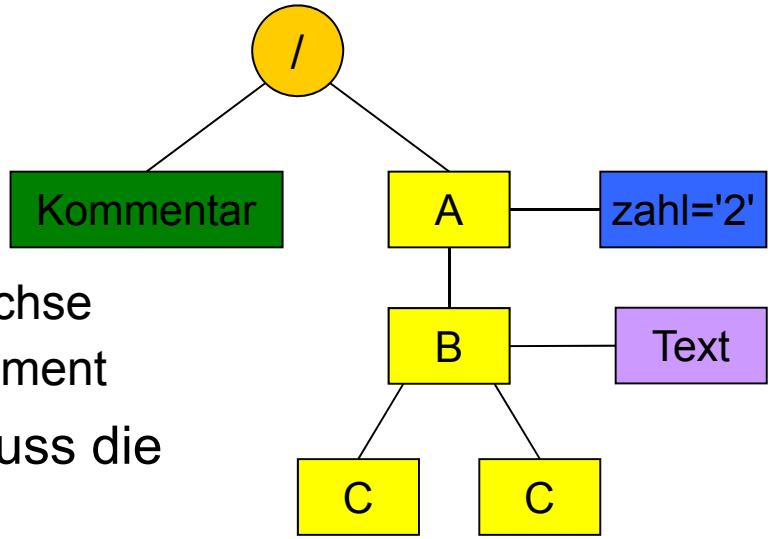
Auswahl einer Achse

- Achsenangaben werden einem Knotentest zusammen mit zwei Doppelpunkten vorangestellt
  - z.B. **`parent::A`**
- Achsenangaben gelten nur für den aktuellen Location-Step
  - in der Pfadangabe **`ancestor::A/B`** bezieht sich die *ancestor*-Achse nur auf die Auswahl der *A*-Knoten; die *B*-Knoten werden in den Kindelementen der gefundenen *A*-Knoten gesucht
    (bei fehlender Achsenangabe wird die *child*-Achse verwendet)
- jeder Location-Step kann eine Achsenangabe enthalten
  - z.B. **`ancestor::A/following-sibling::B`**



attribute-Achse

- außer *self* berücksichtigen die gerade erwähnten Achsen nur Dokument-, Element-, Text- und Kommentar-Knoten (sowie Processing-Instructions)
- Attribut-Knoten sind nur über die *attribute*-Achse erreichbar
  - für nebenstehenden XML-Baum liefert **`/A/node()`** nur den *B*-Knoten
    - **`node()`** sucht hier auf der *child*-Achse von Element *A*
    - Attributknoten gehören nicht zur *child*-Achse
    - → Knotentest **`node()`** findet nur das *B*-Element
  - zur Adressierung des Attributknotens muss die *attribute*-Achse gewählt werden, z.B.
    - **`/A/attribute::node()`**
    - **`/A/attribute::zahl`**
    - **`/A/attribute::*`**



Kurzformen für häufig verwendete Location-Steps

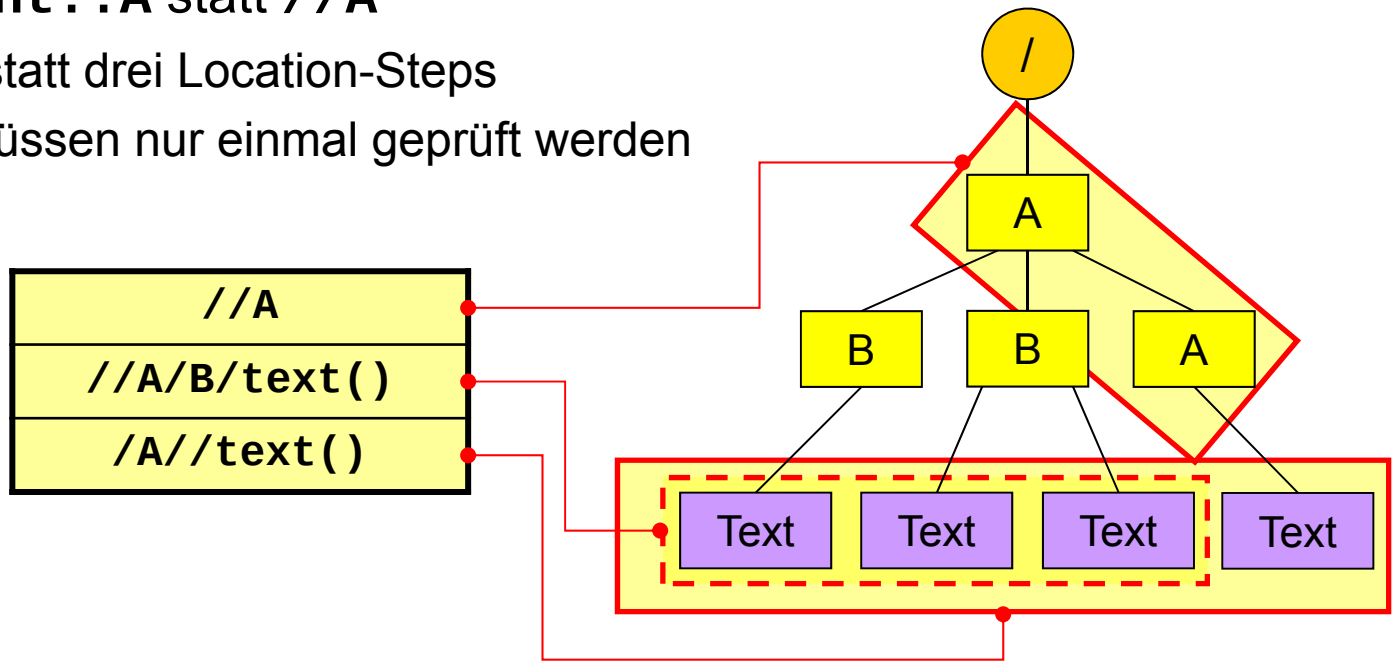
- in der Praxis werden einige Achsen zusammen mit bestimmten Knotentests besonders häufig benötigt
  - → für fünf ausgewählte Location-Steps stellt XPath Kurzschreibweisen zur Verfügung

Kurzform	ausführliche Form
`.`	`self::node()`
`..`	`parent::node()`
`//`	`/descendant-or-self::node()/`
`name`	`child::`*`name`*
`@name`	`attribute::`*`name`*

Beispiel:
`A//B/@attr` ist eine Kurzform von
`child::A/descendant-or-self::node()/child::B/attribute::attr`

Rekursiver Abstieg mit `//`

- die Zeichenfolge `//` ist eine Kurzschreibweise für **`/descendant-or-self::node()/`**
  - kann anstelle einfacher Slashes verwendet werden, um anschließende Knotentests auf alle Nachfahren anzuwenden
  - sollte sorgsam verwendet werden, da die Auswertung abhängig von der Größe des Teilbaums zeitaufwendig sein kann
  - abhängig vom XPath-Prozessor kann ein direkter Knotentest auf der *descendant*-Achse etwas effizienter sein, z.B. **`/descendant::A`** statt **`//A`**
    - nur zwei statt drei Location-Steps
    - Knoten müssen nur einmal geprüft werden



Datentypen in XPath 1.0

- XPath 1.0 unterstützt vier verschiedene Datentypen
  - **boolean** (Wahrheitswerte)
    - die beiden booleschen Konstanten lauten in XPath **`true()`** und **`false()`**
  - **number** (Gleitkommazahlen)
  - **string**
    - String-Konstanten können in einfachen oder doppelten Anführungszeichen angegeben werden, z.B. **`'Hallo'`** oder **`"Hallo"`**
  - **node-set** (Knotenmengen)
    - Knotenmengen werden üblicherweise durch Pfadangaben erzeugt

XPath-Operatoren

a* = *b	*a* gleich *b*
a* != *b	*a* ungleich *b*
a* < *b	*a* kleiner *b*
a* > *b	*a* größer *b*
a* <= *b	*a kleiner oder gleich b*
a* >= *b	*a* größer oder gleich *b*

a* + *b	Addition
a* - *b	Subtraktion
a* * *b	Multiplikation
a* div *b	Gleitkomma-Division
a* mod *b	Gleitkomma-Modulo
a* or *b	logisches ODER
a* and *b	logisches UND
a* | *b	Vereinigung von Knotenmengen

Ausgewählte XPath-Funktionen

Funktion	Returntyp	Beschreibung
`concat(`*`s1`*`,...,`*`sn`*`)`	string	verkettet die Strings *s1* bis *sn* zu einem neuen String
`contains(`*`s1`*`, `*`s2`*`)`	boolean	prüft, ob String *s1* den String *s2* enthält
`count(`*`K`*`)`	number	Anzahl der Knoten in Menge *K*
`name(`*`K`*`)`	string	Name des ersten Knotens in Menge *K*
`normalize-space(`*`s`*`)`	string	entfernt Whitespace am Anfang und Ende von *s* und ersetzt die restlichen Whitespace-Folgen durch einfache Leerzeichen
`not(`*`b`*`)`	boolean	boolsche Negation
`round(`*`x`*`)`	number	rundet *x*
`starts-with(`*`s1`*`, `*`s2`*`)`	boolean	prüft, ob String *s1* mit *s2* beginnt
`string-length(`*`s`*`)`	number	Länge von String *s*
`substring(`*`s`*`, `*`n`*`, `*`l`*`)`	string	Teilstring von *s*, beginnend beim *n*-ten Zeichen und Länge *l*
`substring-after(`*`s1`*`, `*`s2`*`)`	string	sucht *s2* in *s1* und liefert alle Zeichen hinter dem ersten Treffer zurück (liefert Leerstring zurück falls *s2* in *s1* nicht vorkommt)
`substring-before(`*`s1`*`, `*`s2`*`)`	string	sucht *s2* in *s1* und liefert alle Zeichen vor dem ersten Treffer zurück (liefert Leerstring zurück falls *s2* in *s1* nicht vorkommt)
`sum(`*`K`*`)`	number	Addiert die Werte der Knoten in Menge *K*
`translate(`*`s1`*`, `*`s2`*`, `*`s3`*`)`	string	sucht in *s1* der Reihe nach die Zeichen von *s2* und ersetzt in *s1* das *i*-te Zeichen von *s2* durch das *i*-te Zeichen von *s3*

XPath-Ausdrücke

- neben Pfadangaben sind auch Funktionsaufrufe sowie Ausdrücke mit mathematischen und logischen Operatoren gültige XPath-Ausdrücke

- Beispiele:
  - `1+2*3`
  - `concat('XML', 'und', 'XSLT')`
  - `contains('Blumentopferde', 'pferd')`
  - `count(//A)`
  - `translate('Hello', 'eo', 'ae')`
  - `(1+2*3 > (1+2)*3) or starts-with('Hallo', 'ha')`
  - `not(count(ancestor::A) > 5)`

Automatische Typumwandlung

- Falls ein Operand oder ein Funktionsparameter nicht den erwarteten Typ besitzt, wird bei XPath 1.0 eine implizite Typ-konvertierung durchgeführt, z.B.
  `"5.5" + true() * 2` → 7.5

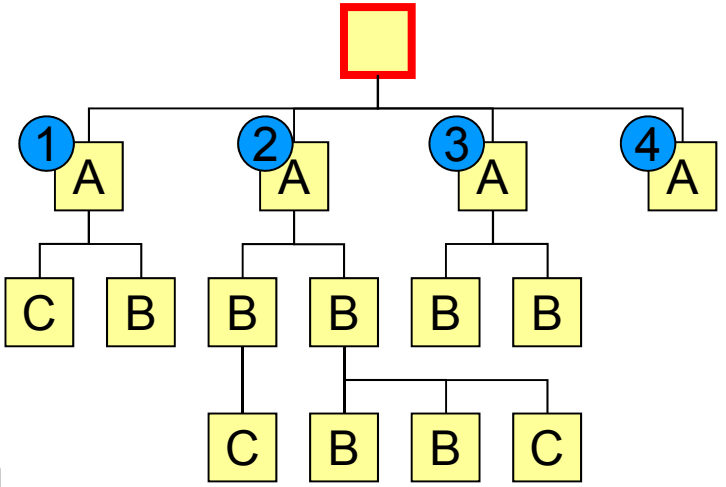
von \ nach	**boolean**	**number**	**string**	**node-set**
boolean		false → 0 true → 1	false → **`'false'`** true → **`'true'`**	*nicht erlaubt*
number	0 → false sonst true		Dezimalzahl als String	*nicht erlaubt*
string	**`''`** → false sonst true	interpretiert den String als Zahl		*nicht erlaubt*
node-set	**`{}`** → false sonst true	interpretiert den aus erstem Kno-ten gebildeten String als Zahl	hängt die Texte descendant::text() des ersten Kno-tens aneinander	

Prädikate

- **Prädikate** sind optionale Bestandteile eines Location-Steps und grenzen Knotenmengen durch Boolesche Ausdrücke weiter ein
  - nur Knoten, auf die der angegebene Ausdruck zutrifft, werden in die Ergebnismenge aufgenommen
- Prädikate werden in eckigen Klammern hinter einem Knotentest angegeben
  - `A[.='Hallo']`
    liefert alle *A*-Kindknoten, die den Text "Hallo" enthalten
  - `A[@attrib = 'Hallo']`
    liefert alle *A*-Kindknoten, die einen Attributknoten *attrib* mit Wert "Hallo" enthalten
  - `A[B]`
    liefert alle *A*-Kindknoten, die mindestens einen *B*-Kindknoten enthalten
- relative Pfadangaben in Prädikaten beziehen sich immer auf die durch den vorangehenden Pfadausdruck ausgewählte Knotenmenge

Knotennummerierung und Positionstests

- die von einem Location-Step ausgewählten Knoten werden immer automatisch durchnummeriert
  - die Nummerierung beginnt bei 1
- die XPath-Funktion **`position()`** liefert die Positionsnummer des aktuellen Kontextknotens
  - kann in Prädikaten zur Auswahl einer der Ergebnisknoten verwendet werden
  - der Ausdruck **`A[position()=1]`** liefert den ersten A-Knoten der Ergebnismenge



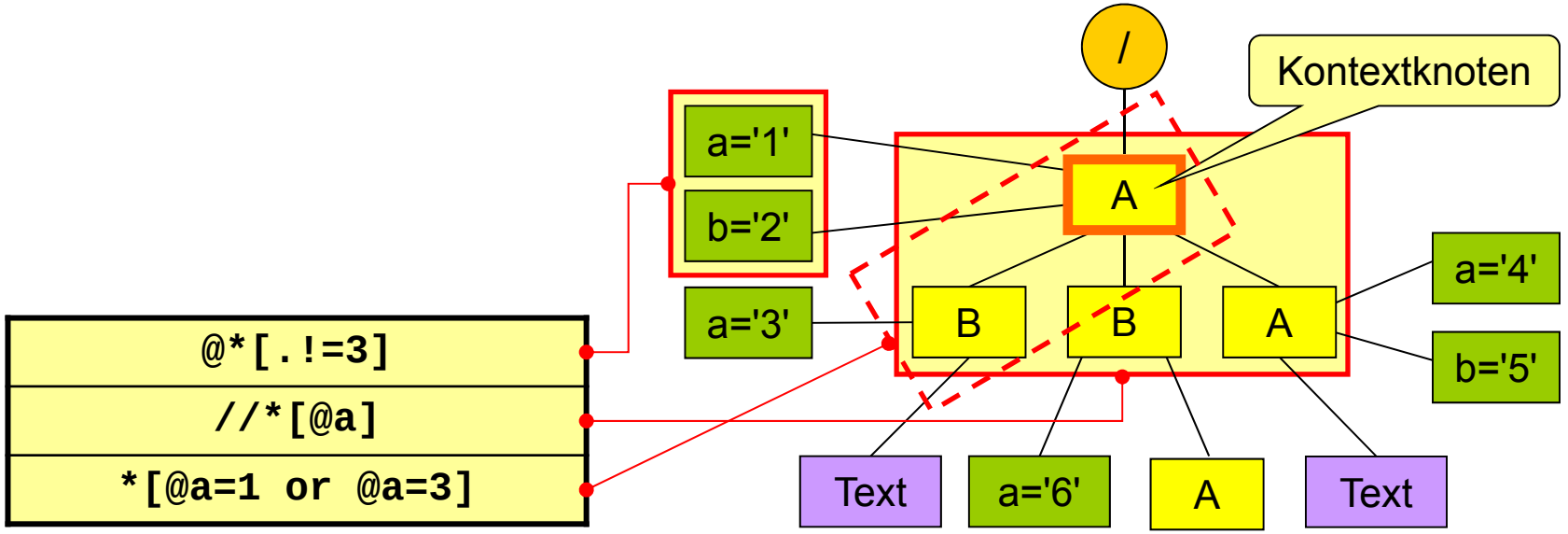
Nummerierung der Ergebnisknoten beim Knotentest **A** relativ zum rot markierten Kontextknoten

- Prädikate der Form **`position()=`*n***, wobei *n* eine Zahl (Typ *number*) ist, werden **Positionstests** genannt
- Prädikate, die nur aus einer Zahl bestehen, werden immer als Positionstests interpretiert
  - **`A[1]`** ist eine Kurzform von **`A[position()=1]`**

Prädikate: Beispiele

```
<?xml version="1.0"?>
<A a="5" b="8">
  <B a="20">Guten</B>
  <B>
    <A a="1">Morgen</A>
    <C>Heute</C>
  </B>
  <B a="50">Tag</B>
  <A>Abend</A>
</A>
```

```
//A[not(@a)]
/A/B[@a < 50]
//B[.='Tag']
```



XPath vs. XQuery

- mit XPath kann man:
  - in XML-Bäumen navigieren
  - einzelne Knoten und Knotenmengen aus XML-Bäumen auswählen
- mit XPath kann man nicht:
  - Daten sortieren
  - Daten neu gruppieren
  - Variablen und Funktionen definieren
  - neue XML-Knoten erzeugen
  - XML-Dokumente ändern
- XQuery ist eine funktionale Programmiersprache mit XPath 2.0 als Untermenge
  - ermöglicht komplexe Abfragen von Daten aus XML-Dokumenten
  - bietet Konstrukte zur Erzeugung neuer XML-Bestandteile
  - kann für bestimmte Aufgaben als Alternative zu XSLT dienen
  - Ändern von XML-Daten nur mit *XQuery Update Facility* möglich

XQuery: Datentypen und -strukturen

- XQuery (und XPath 2.0/3.0) verwendet das Typensystem von XML Schema
  - **atomare (einfache elementare) Typen:**
    xs:boolean, xs:integer, xs:double, xs:string, xs:date, …
  - **Knotentypen:**
    Element, Attribut, Text, Kommentar, Verarbeitungsanweisung, Namensraum
- als einzige eingebaute, komplexe Datenstruktur gibt es die **Sequenz**
  - geordnete Liste von Objekten beliebigen Typs
  - Sequenzen dürfen aus beliebig vielen Komponenten bestehen
  - jede Komponente darf einen anderen Typ haben
  - die Reihenfolge der Komponenten ist signifikant
- Sequenzen können u.a. durch Auflistung der Komponenten oder als Ergebnis von Pfadausdrücken erzeugt werden
  - `(1, 2, 'Hallo', @wert)`
  - `//person/vorname`

XQuery: neues Typensystem

- mit der Einführung des neuen Typensystems findet eine strengere Prüfung der Typen statt
  - Objekte mit atomarem Typ werden nicht mehr automatisch in andere Typen konvertiert
    - der Ausdruck **`1+'2'+true()`** ist in XQuery nicht erlaubt
  - Typkonvertierungen müssen explizit notiert werden
    - z.B. **`1+xs:integer('2')+xs:integer(true())`**
  - Knoten ohne zugewiesenen Schema-Typ bekommen den Typ *xs:untyped* und werden bei der Auswertung wie bei XPath 1.0 ggf. automatisch konvertiert
    - im Ausdruck **`1+@wert`** wird das *wert*-Attribut des Kontextknotens in eine Zahl (xs:decimal) konvertiert
    - im Ausdruck **`concat('Hallo', @wert)`** wird das *wert*-Attribut des Kontextknotens in einen String (xs:string) konvertiert

Sequenzen

- in XPath 1.0 erzeugen Pfadausdrücke Knotenmengen, die keinen, einen oder mehrere Knoten enthalten können
- in XQuery sind Sequenzen endliche Listen, die beliebige Objekte mit beliebigen Typen enthalten dürfen
- Sequenzen können explizit durch Aufzählung der Komponenten in runden Klammern erzeugt werden
  - `(1, 2, 'Hallo', 3.0)`
  - `(1 to 10)` ist eine Kurzform für `(1,2,3,4,5,6,7,8,9,10)`
- Sequenzen dürfen auch Sequenzen enthalten
  - geschachtelte Sequenzen werden immer aufgelöst, so dass eine eindimensionale Liste entsteht („innere Klammern werden entfernt“)
  - `(1,(2,(3,4)),('Hallo', 'Welt'))` wird zu `(1,2,3,4,'Hallo','Welt')`
  - `(//bundesland[@typ='stadtstaat'], //name)` kombiniert die Ergebnissequenzen der beiden Pfadausdrücke

XQuery: Variablen definieren mit **let**

- in XPath kann zwar auf Variablen zugegriffen werden, es gibt aber keine Möglichkeit, Variablen zu definieren
- XQuery stellt die Anweisung **let** zur Variablendefinition bereit
  - Variablen dürfen beliebige Objekte oder Sequenzen zugewiesen werden
  - Variablen können nachträglich nicht mehr geändert werden
  - Variablen können optional typisiert werden
- **`let`** **`$`***var* **`:=`** *ausdruck*
  - `let $zahl := 5`
  - `let $namen := //person/name`
  - `let $seq := (1,2,6,3,'Hallo')`
- **`let`** **`$`***var* **`as`** *typ* **`:=`** *ausdruck*
  - `let $zahl as xs:double := 5`
  - `let $namen as xs:string* := //person/name`

XQuery: Ergebnis angeben mit **return**

- jeder XPath-Ausdruck produziert ein Ergebnis, das anschließend weiterverarbeitet werden kann
  - das Ergebnis kann aus einem Leerstring oder einer leeren Sequenz bestehen
  - es gibt keinen XPath-Ausdruck vom Typ „void“
- in XQuery gilt im Prinzip das gleiche:
  jeder XQuery-Ausdruck muss ein Ergebnis produzieren
- da eine Variablendefinition kein Ergebnis erzeugt, darf sie nicht isoliert verwendet werden
- das gewünschte Query-Ergebnis muss mit einer folgenden **return**-Anweisung festgelegt werden

```
let $personen := //personen[vorname='Maria']
return $personen/nachname
```

XQuery: Definition mehrerer Variablen mit gleichem Namen

- in XQuery gibt es keinen Zuweisungsoperator, so dass Variablen nach ihrer Definition nicht mehr geändert werden können
- trotzdem ist folgendes Query erlaubt:

```
let $n := 1
let $n := $n+1
return $n
```

  - Ergebnis des Querys ist 2
  - wie kann das sein, wenn Variablen unveränderbar sind?

- jede Variablendefinition erzeugt eine neue Variable
  - gleichnamige Variablen verdecken die jeweils vorher definierten
  - auf der rechten Seite von `:=` ist die vorangehende Definition noch sichtbar, deshalb wird dem zweiten *n* hier der Wert 2 zugewiesen
  - in der *return*-Anweisung ist nur noch die zweite Variable sichtbar, die erste existiert aber noch
- das Query ist identisch zu folgendem:

```
let $n := 1
let $m := $n+1
return $m
```

  - hier kann im *return*-Statement sowohl auf *n* als auch auf *m* zugegriffen werden

XQuery: Bedingte Ausdrücke mit **if-then-else**

- XQuery erlaubt fallunterscheidende Ausdrücke mit Hilfe einer **if-then-else**-Konstruktion:
- Syntax: **`if (`**<bedingung>**`) then`** <ausdruck1>
  `else` <ausdruck2>
  - da XQuery-Audrücke immer ein Ergebnis produzieren müssen, ist der *else*-Teil verpflichtend, kann also nicht weggelassen werden
  - ist vergleichbar mit dem ternären Operator **`?:`** in C/C++/Java
    - Java: **`(b > c) ? b : c;`**
    - XPath: **`if ($b > $c) then $b else $c`**
  - die Datentypen vom *then*- und *else*-Teil müssen nicht identisch sein
    - **`if (adressen) then adressen/adresse else "Hallo Welt"`** ist erlaubt
  - der *then*-Ausdruck wird nur ausgewertet, wenn die Bedingung wahr ist und der *else*-Ausdruck nur, wenn die Bedingung falsch ist
    - **`if ($a > 0) then 1 div $a else "Division durch 0"`** produziert also keinen Fehler falls **`$a=0`**

XQuery: Iterieren über Sequenzen mit **for-return**

- um einen Ausdruck auf jede Komponente einer Sequenz anzuwenden wird **for-return** verwendet
  - das Resultat ist eine neue Sequenz mit den Ergebnissen der einzelnen Iterationsschritte
- Syntax: `for $var in` <sequenz> `return` <ausdruck>
- Beispiele:
  - `for $i in (1 to 10) return $i*0.1`
    erzeugt die Sequenz (0.1, 0.2, .0.3, ..., 1.0)
  - `for $i in //name return concat('Hallo ', $i)`
    iteriert über alle *name*-Elemente des aktuellen Dokuments und erzeugt daraus eine String-Sequenz der Form ('Hallo Jim', 'Hallo Anna', …)
- **for-return** darf überall dort verwendet werden, wo Sequenzausdrücke erlaubt sind, z.B.

```
let $grüße := if (//name) then
  for $i in //name return concat('Hallo ', $i)
else
  'niemand zum Begrüßen da'
return $grüße
```

XQuery: FLWOR – erweiterte **for-return**-Anweisung

- XQuery erlaubt zwischen *for* und *return* zusätzlich die folgenden optionalen Angaben:
  - beliebig viele **let**-Anweisungen zur Variablendefinition
  - eine **where**-Anweisung der Form **`where`** *bedingung*
    zum Filtern von Sequenzkomponenten
  - eine **order by**-Anweisung der Form **`order by`** *ausdruck richtung*
    zum Ändern der Iterationsreihenfolge

XQuery: FLWOR – erweiterte **for-return**-Anweisung

- **let:** Definition von Variablen bei jedem Iterationsschritt

```
for $n in (4,7,1,1)
let $m := 2*$n
return $m
```

  - die Variable *m* wird bei jedem Iterationsschritt neu definiert
  - das Query liefert die Sequenz **(8,14,2,2)**

- auch hier gilt, dass eine Variable mit gleichem Namen die vorangehende verdeckt

```
let $m := 5
for $n in (4,7,1,1)
let $m := $n*$m
return $m
```

  - im Ausdruck **$m*$n** bezeichnet **$m** die erste Variable *m* (mit Wert 5), im *return*-Statement wird die zweite verwendet
  - liefert die Sequenz **(20,35,5,5)**

XQuery: FLWOR-Ausdrücke

- **where**: Iteration beschränken
  - es werden nur Sequenzelemente berücksichtigt, die die angegebene Bedingung erfüllen

```
for $person in //person
let $vname := $person/name/vorname
where starts-with($vname, 'M')
return $person
```

  - hat den gleichen Effekt wie ein Prädikat
  - im *where*-Ausdruck können zusätzlich innere Variablen verwendet werden
- **order by**: Verarbeitungsreihenfolge ändern
  - vor der Iteration wird für jede Komponente der Sortierausdruck ausgewertet und die Sequenz anhand dieser Werte sortiert
  - anschließend wird über die sortierte Sequenz iteriert

```
for $person in //person
order by $person/name/nachname, $person/name/vorname
return $person
```

XQuery: FLWOR-Ausdrücke

- die optionalen Bestandteile von *for-return* müssen immer in der Reihenfolge **let – where – order by** angegeben werden
- ein kompletter *for-return*-Ausdruck hat also die Form **for – let – where – order by – return**
- die Anweisung wird deshalb auch **FLWOR-Ausdruck** genannt, unabhängig davon, ob alle Bestandteile verwendet werden
  - FLWOR wird wie engl. *flower* ausgesprochen

```
for $person in //person
let $vorname := $person/name/vorname
let $nachname := $person/name/nachname
where $person/wohnort = 'Osnabrück'
order by $nachname, $vorname
return concat('Hallo ', $vorname, ' ', $nachname, '&#10;')
```

  - trotz der verschiedenen Schlüsselwörter handelt es sich hier um nur einen Ausdruck
  - liefert hier als Resultat eine Sequenz mit String-Objekten zurück

XQuery: Element-Konstruktoren

- XQuery-Skripte können literale XML-Elemente enthalten
- der Inhalt von Attributen und Elementrümpfen wird als literaler Text interpretiert und nicht weiter ausgewertet
- XQuery-Anweisungen innerhalb von Attributen oder Elementen müssen mit **{…}** geklammert werden

```
let $personen :=
  <personen>
    <person geschlecht="m">Jim Panse</person>
    <person geschlecht="w">Anna Konda</person>
  </personen>
return
  <personen-neu>{
    for $person at $pos in $personen/person
    return
      <person pos="{$pos}">
        {$person/@geschlecht}
        <vname>{substring-before($person, ' ')}</vname>
        <nname>{substring-after($person, ' ')}</nname>
      </person>
  }</personen-neu>
```

Ausblick

- mit den vorgestellten XQuery-Konstrukten können bereits komplexe Abfragen formuliert werden
- XQuery bietet aber noch viele weitere Features, u.a.
  - mehrfache *for*-Anweisungen in FLWOR-Ausdrücken zur Realisierung von „Joins“
  - Positionsvariablen u.a. zur Nummerierung von Sequenzkomponenten
  - *XQuery Update Facility* (XUF) zum gezielten Ändern von Daten in XML-Dokumenten
  - Definition von Funktionen u.a. zur Strukturierung von XQuery-Programmen und zur Formulierung rekursiver Algorithmen
- die Konvertierung von XML-Dokumenten in andere Formate ist mit XQuery möglich, aber relativ umständlich
- mit XSLT geht das sehr viel einfacher
  - XSLT ist eine XML-basierte Sprache zur Beschreibung von XML-Transformationen
  - Konzepte ähneln denen von XQuery
  - bietet einen mächtigen Template-Mechanismus