Einführung in XML

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Einführung in XML

Teil I: Übersicht

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Inhalt

Zeitplan

- Session 1 (9:00 - 10:30h)
  - Geschichte
  - Grundlagen
- Session 2 (10:45 - 12:15h)
  - Beschreibung von XML-Strukturen
- Session 3 (13:15 - 14:45h)
  - Abfragesprachen
- Session 4 (15:00 - 16:30h)
  - Transformationen

Vortragende

Verteilte Systeme und Grid Computing (VSGC)

- Björn Hagemeier
- Ahmed Shiraz Memon
- Mohammad Shahbaz Memon

Inhalt

Übersicht

Heutige Themen

Themen

- Geschichte
- Motivation/Anwendungsfälle
- Einführung in XML
- Beschreibung von XML Dokumenten
- Abfragesprachen
- Verarbeitung

Einführung in XML

Teil II: XML

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Inhalt

Geschichte

- SGML
  - 1960er

Beispiel

```
<!DOCTYPE hallo SYSTEM "hallo.dtd">
<hallo>
  <welt>Dies ist der Text,
        der die Welt bedeutet
</HALLO>
```

Geschichte

- SGML
  - 1960er
- HTML
  - Anfang 1990er
  - SGML-Anwendung
  - wurde Missbraucht (Browserkrieg)

Beispiel

```
<html>
  <head>
    <title>Seitentitel</title>
  </head>
  <body>
    <h1>Überschrift</h1>
    <p>
     Dies ist ein Absatz mit
     einem <b>fetten</b> Wort.
    </p>
  </body>
</html>
```

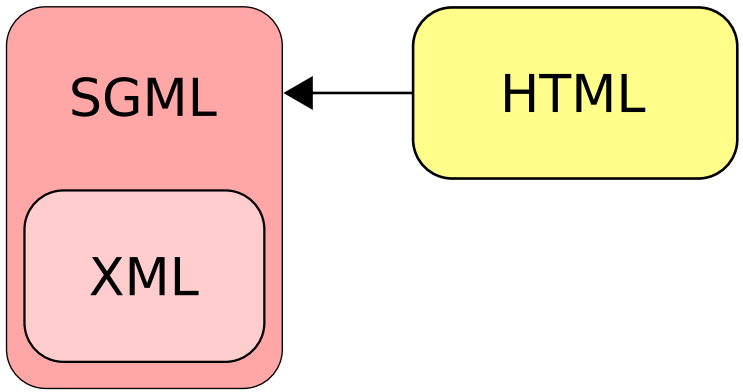
Geschichte

- SGML
  - 1960er
- HTML
  - Anfang 1990er
  - SGML-Anwendung
  - wurde Missbraucht (Browserkrieg)
- XML
  - Ende 1990er
  - Einschränkung von SGML
  - XHTML ist XML-Anwendung

Beispiel

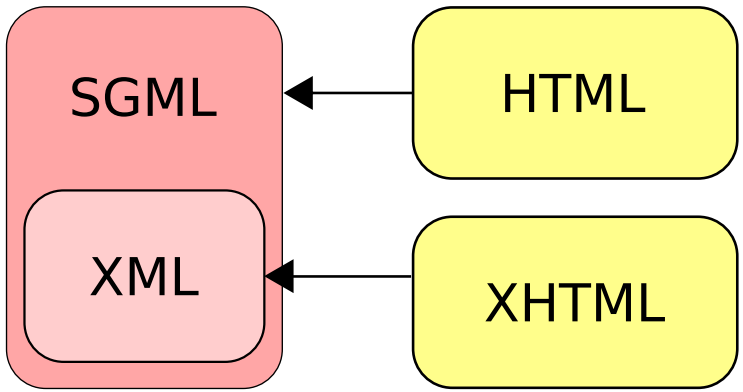
```
<?xml version="1.0"
      encoding="UTF-8"?>
<hallo>
  <anderesElement
      id="element1">
    Ein wenig Text
  </anderesElement>
</hallo>
```

Beziehungen



- HTML Anwendung von SGML
- XML Teilmenge von SGML

Beziehungen



- HTML Anwendung von SGML
- XML Teilmenge von SGML

Warum überhaupt XML

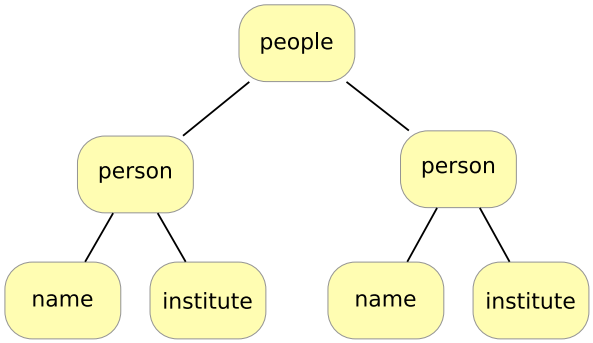
- SGML erfüllt seinen Zweck
  - Beschreibung von Sprachen zur Anreicherung von Text mit Auszeichnungselementen

Warum überhaupt XML

- SGML erfüllt seinen Zweck
  - Beschreibung von Sprachen zur Anreicherung von Text mit Auszeichnungselementen
- XML soll einen anderen Zweck erfüllen
  - Einfacher Austausch strukturierter Daten zwischen beliebigen Plattformen und Programmiersprachen
  - Weiterhin für Menschen lesbar
  - Beschreibung von Sprachen, die diesem Zweck dienen

Inhalt

Erster Eindruck




```
<people>
  <person>
    <name>Björn</name>
    <institute>ZAM</institute>
  </person>
  <person>
    <name>Shiraz</name>
    <institute>ZAM</institute>
  </person>
</people>
```

- Dokument kann als Baum aufgefasst werden
- Schachtelung der Elemente
- Keine Überlappungen
- Hierarchisch

Ziele von XML

1. XML soll sich im Internet auf einfache Weise nutzen lassen.
2. XML soll ein breites Spektrum von Anwendungen unterstützen.
3. XML soll zu SGML kompatibel sein.
4. Es soll einfach sein, Programme zu schreiben, die XML-Dokumente verarbeiten.
5. Die Zahl optionaler Merkmale in XML soll minimal sein, idealerweise Null.
6. XML-Dokumente sollten für Menschen lesbar und angemessen verständlich sein.
7. Der XML-Entwurf sollte zügig abgefasst sein.
8. Der Entwurf von XML soll formal und präzise sein.
9. XML-Dokumente sollen leicht zu erstellen sein.
10. Knappheit von XML-Markup ist von minimaler Bedeutung.

Quelle: `http://www.edition-w3c.de/TR/2004/REC-xml11-20040204/`

Terminologie

- Dokumente
- Elemente
- Attribute
- Kommentare
- Processing-Instructions

Dokumente

Definition – XML Dokument

Ein Datenobjekt ist ein XML Dokument wenn es *wohlgeformt* ist. Zusätzlich kann es *valide* sein, indem es weitere Einschränkungen erfüllt.

- Dokumenttyp
- Referenzen

Wohlgeformte Dokumente

Ein Dokument ist wohlgeformtes XML, wenn ...

- es genau ein Wurzelelement besitzt
- zu jedem öffnenden `<Element>` auch ein zugehöriges schließendes `</Element>` existiert
  - Beachte: `<element />` $\equiv$ `<element></element>`
- öffnende und schließende Elemente korrekt geschachtelt sind und keine Überlappungen haben
  - `<element1><element2></element1></element2>` funktioniert nicht

Struktur von Dokumenten

Ein XML-Dokument hat immer folgende Struktur

Struktur

```
<?xml version="1.1" encoding="UTF-8"?>
<!DOCTYPE addressbook SYSTEM "addressbook.dtd">
<addressbook>
  <address>Köln</address>
  <address>Aachen</address>
</addressbook>
```

Elemente

- Immer geschachtelt
  - Anfang und Ende zwingend vorhanden
  - Ausnahme: leere Elemente
- Namen
  - Keine Leerzeichen
  - dürfen nicht mit »xml« beginnen, weder groß noch klein geschrieben
  - Doppelpunkt reserviert, muss vermieden werden

Beispiel

```
<element1>
  <element2>foo</element2>
</element1>
```

Elemente – Mögliche Inhalte

- Weitere Kindelemente
- PCDATA (Parsed Character Data $\equiv$ Text)
- Mischung aus beidem

Elemente – Mögliche Inhalte

- Weitere Kindelemente
- PCDATA (Parsed Character Data $\equiv$ Text)
- Mischung aus beidem
- nichts

Definition – Leeres Element

Ein Element ohne Inhalt wird *leeres Element* genannt. Es kann in der Form `<element />` oder `<element></element>` geschrieben werden.

Markup vs. Text

Wie werden Markup-Zeichen im Text codiert?

- `<` und `&` dürfen nicht direkt als Zeichen verwendet werden
  - `<` leitet Elemente, Kommentare, Processing Instructions oder CDATA Abschnitte ein
  - `&` leitet Referenz auf Entitäten ein
- Diese Zeichen müssen anders kodiert werden
  - Als Referenz auf das zugehörige Zeichen in UTF (z. B. `&#60;` für < und `&#38;` für `&`
  - `&` wird als `&` geschrieben
  - < als `<` (less than)

CDATA

- Kann *Markup* enthalten
- Wird nicht interpretiert

Beispiel

Aus einem RSS-Feed

```
<description>
  <![CDATA[
    <h1>Hier kann auch der vollständige HTML-Inhalt
        des Artikels stehen
    </h1>
  ]]>
</description>
```

Vergleich von CDATA und PCDATA

- CDATA und PCDATA sind beide im Wesentlichen Text
- PCDATA wird geparsed (als XML interpretiert)
- CDATA wird lediglich als beliebiger Text betrachtet
- PCDATA wird als Inhalt von Elementen und Attributen deklariert
- CDATA kann nur explizit im Dokument verwendet werden, z. B. um XML innerhalb eines XML-Dokuments zu »verstecken«

Moment mal...

... da fehlt doch noch etwas.

Moment mal...

... da fehlt doch noch etwas.

```
<people>
 <person id="bjoernh">
   <name>
     <givenname>Björn</givenname>
     <surname>Hagemeier</surname>
   </name>
   <address type="company">
     <street>Leo-Brandt-Str. 1</street>
     <zip>52428</zip>
     <city>Jülich</city>
   </address>
 </person>
</people>
```

Attribute

- Sind einem Element zugeordnet
  - `<img src="logo.gif" width="320" />`
  - `<address type="company">...</address>`
- Werden im *öffnenden* Element-Tag angegeben
- »id« und »xml-*« reserviert (in Groß- und Kleinschreibung)
- Mehrere Attribute pro Element möglich
- Müssen immer in *doppelten* Anführungszeichen stehen, s. o.

Elemente vs. Attribute

Wann sollte was verwendet werden?

Attribute

- Zusatzinformationen
- »einfache« Daten

Elemente

- »eigentliche« Daten
- komplexe Kindelemente

Elemente vs. Attribute

Die Entscheidung ist nicht immer leicht zu treffen:

Beispiel

```
<person nickname="bjoernh" />
```

```
<person>
  <nickname>bjoernh</nickname>
</person>
```

Kommentare

- Gehören nicht zum eigentlichen Text des Dokuments
- Dürfen überall außerhalb des eigentlichen Markup erscheinen
- Werden mit `<!--` eingeleitet und mit `-->` beendet
  - `--` innerhalb des Kommentars nicht erlaubt
  - Kommentar kann nicht mit `--->` beendet werden
  - Schachtelung nicht möglich
- Parser erlauben Zugriff auf Kommentare

Beispiel

```
<!-- XML-Markup lässt sich durch Kommentare verstecken: -->
<!-- <hallo>Kommentar</hallo> -->
<!-- Ungültiger Kommentar -- enthält zwei Bindestriche -->
<!-- Ende dieses Kommentars ungültig --->
```

Processing Instructions – PI

- Anweisungen an Parser bzw. verarbeitendes Programm
  - `<?xml version="1.1" encoding="UTF-8"?>`
  - `<?xml-stylesheet type="text/xsl" href="company.xsl"?>`
- Können (fast) überall im XML-Dokument erscheinen
- Gehören (wie Kommentare) nicht zum eigentlichen Text

Processing Instructions – PI

- Anweisungen an Parser bzw. verarbeitendes Programm
  - `<?xml version="1.1" encoding="UTF-8"?>`
  - `<?xml-stylesheet type="text/xsl" href="company.xsl"?>`
- Können (fast) überall im XML-Dokument erscheinen
- Gehören (wie Kommentare) nicht zum eigentlichen Text

Übung

Übung zum Thema Wohlgeformtheit

Login

Am oberen Rand des Bildschirms steht

- Options → Remote Login → Enter Host Name ...
  - zam859
  - zam860
  - zam861
- Benutzername: xml<nnnn> (links oben)
- Passwort: ********

Namensräume – Motivation

- Bisher Einfache Elementnamen definiert
- Verschiedene Konzepte können gleiche Namen bekommen
  - Was ist eine »Adresse«?
  - Oder eine »Maschine«?

Beispiel

```
<machine>
  <address>192.168.10.1</address>
  <owner>
    <address>52425 Jülich</address>
  </owner>
</machine>
<machine>
  <horsePower>230</horsePower>
  <manufacturer>Honda</manufacturer>
</machine>
```

Namensräume – Motivation

- Im nächsten Abschnitt werden Konzepte vorgestellt, um die Struktur von Dokumenten vorzugeben
- Strukturbeschreibungen können von mehreren Anbietern bereitgestellt werden
- Dadurch können Konflikte in Element- und Attributnamen entstehen
- Elemente und Attribute werden daher in sogenannten Namensräumen deklariert, die eindeutig von jedem Anbieter vergeben werden

Namensräume

- Werden über URIs identifiziert
  - URI $\neq$ URL
  - Manchmal als URL verwendbar
  - Typischerweise eigener Domain-Name mit geeignetem Suffix
    - `http://www.fz-juelich.de/zam/kurse/xml`
  - `schemaLocation` Attribut
- Verhindern Namenskonflikte
- Vollständige Qualifizierung mit Namensräumen (QName)
- Präfixe als Abkürzungen mit lokaler Bedeutung

Namensräume – QName

- Qualifizierter Name
- Besteht aus zwei Teilen
  - Namensraumpräfix
  - Lokaler Teil
  - `<adressen:adresse>...</adressen:adresse>`
- Lokaler Teil entspricht dem bisherigen Elementnamen
- Präfix wird zuvor dem Namensraum zugeordnet
  - URIs nicht als Teile von Elementnamen geeignet
  - Kürzer als URI
  - Lokale Gültigkeit
  - Besser lesbar
  - `xmlns:adressen="http://www.fz-juelich.de/zam/adressen"`
- Leeres Präfix möglich
  - `xmlns="http://www.fz-juelich.de/zam/adressen"`

Namensräume – Beispiel

Der Anfang eines (typischen) Dokuments

Beispiel

```
<process name="test" targetNamespace="test.wsdl"
  xmlns=
    "http://schemas.xmlsoap.org/ws/2004/03/business-process/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:bpws=
    "http://schemas.xmlsoap.org/ws/2004/03/business-process/"
  xmlns:sam="http://samples.aware.org/EchoSchema"
  xmlns:typens="http://samples.aware.org/types"
  xmlns:test="test.wsdl">
    ...
</process>
```

Übung

Übung zum Thema Namensräume

JÜLICH
FORSCHUNGSZENTRUM

Einführung in XML

Teil III: Beschreibung von XML Strukturen

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Inhalt

In diesem Abschnitt

- Notwendigkeit der strukturellen Beschreibung
- Beschreibungssprachen
- DTD
- XML Schema

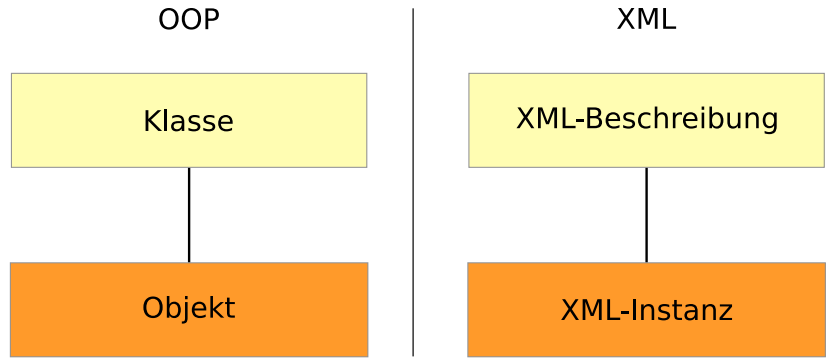
Strukturelle Beschreibung

- Schränkt mögliche Inhalte von Elementen ein
- Legt die möglichen Attribute von Elementen fest
- Fügt zu Dokumenten einen gewissen Sinn hinzu
- Ermöglicht Validierung *wohlgeformter* Dokumente

Modell

Analogie für Informatiker

Klassen (OOP) verhalten sich zu *Objekten* wie *XML-Instanzen* zur *XML-Beschreibung*



Beschreibungen ermöglichen Validierung

Definition – Valides Dokument

Ein XML-Dokument is valide, wenn es eine Document Type Definition (DTD) besitzt und seine Elemente den Regeln der DTD entsprechen.

Inhalt

DTD

Document Type Definition

- auch »Schemadefinition«
- Beschreibt die mögliche Struktur eines XML-Dokuments
- Grammatik einer formalen Sprache
  - Text gehört zu einer Sprache, wenn er aus den Produktionen seiner Grammatik erzeugt werden kann
- DTD ist Teil der XML-Spezifikation
  - Genaugenommen stellt XML damit nicht selbst eine Markup-Sprache, sondern ein System zur Beschreibung von Markup-Sprachen dar (Meta-Sprache)

DTD – Notation

Beispiele

```
adressbuch       ( adrbInfo? , personenListe )
adrbInfo         ( besitzer , letzteAenderung )
besitzer         ( #PCDATA )
letzteAenderung  ( #PCDATA )
personenListe    ( person* )
person           ( name , adresse+ )
name             ( #PCDATA )
adresse          ( textAdresse | ( straße, PLZ, Ort ) )
textAdresse      ( #PCDATA )
straße           ( #PCDATA )
PLZ              ( #PCDATA )
Ort              ( #PCDATA )
```

DTD

Beispiel

```
<!DOCTYPE people [

<!ELEMENT people ( person* ) >
<!ELEMENT person ( name , address+ ) >

<!ELEMENT name ( givenname , surname ) >
<!ELEMENT givenname (#PCDATA) >
<!ELEMENT surname (#PCDATA) >

<!ELEMENT address (street , zip , city ) >
<!ELEMENT street (#PCDATA) >
<!ELEMENT zip (#PCDATA) >
<!ELEMENT city (#PCDATA) >

<!ATTLIST person id ID #IMPLIED >
<!ATTLIST address type ( private | company | other ) #REQUIRED >

] >
```

Instanz des vorherigen Beispiels

```
<people>
 <person id="bjoernh">
   <name>
     <givenname>Björn</givenname>
     <surname>Hagemeier</surname>
   </name>
   <address type="private">
     <street>Mickey-Mouse-Weg. 326</street>
     <zip>55555</zip>
     <city>Entenhausen</city>
   </address>
   <address type="company">
     <street>Wilhelm-Jonen-Str. 1</street>
     <zip>52428</zip>
     <city>Jülich</city>
   </address>
 </person>
</people>
```

DTD – Notation

Kardinalitäten von Elementen

- *: das Element kann beliebig oft $(0..\infty)$ vorkommen
- +: das Element muss mintdestens einmal $(1..\infty)$ vorkommen
- ?: das element kann optional einmal vorhanden sein $(0..1)$
- ,: Elemente tauchen in der gegebenen Reihenfolge auf (Sequenz)
- |: Eines der Elemente taucht auf (Auswahl)
- (,): Gruppierung

DOCTYPE

- Einleitendes Element der DTD
- Können intern, extern oder kombiniert sein
  1. intern (s. o.)
     - `<!DOCTYPE people [ <!ELEMENT people (person*)>...]>`
  2. extern
     - `<!DOCTYPE people SYSTEM "external.dtd">`
     - Verweist auf URI `external.dtd`
     - Siehe früheres Beispiel
  3. Kombiniert
     - `<!DOCTYPE people SYSTEM "external.dtd" [!ELEMENT...]>`
- Wurzelelement des Dokuments ist `people`
- Wurzelelement muss ebenfalls definiert werden
  - Konvention: Definition als erstes Element

DOCTYPE – PUBLIC ID

Außer `SYSTEM` gibt es noch `PUBLIC` als Schlüsselwort

Beispiel

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

- `PUBLIC` IDs werden für bekannte Schemas verwendet
- Validierende Programme brauchen Schema nicht extra laden, da bereits vorhanden

ELEMENT

Definiert den erlaubten Inhalt eines Elementes

Beispiel

```
<!ELEMENT people   ( person* )          >
<!ELEMENT person   ( name, address* ) >
<!ELEMENT name     ( #PCDATA )          >
<!ELEMENT address  ( #PCDATA )          >
```

```
<!ELEMENT person   ( name ,
                   ( textAdress | structuredAddress )* >
<!ELEMENT par      ( #PCDATA | em | b | i )*            >
```

- `people` enthält eine beliebig lange Liste von `person`
- `person` besteht aus `name` und einer beliebig langen Liste von `address`
- `name` enthält `PCDATA`, also Text

ATTLIST

Funktion

Legt Attribute zu einem Element fest.

Beispiel

```
<!ATTLIST termdef
          id       ID       #REQUIRED
          name     CDATA    #IMPLIED>
<!ATTLIST list
          type     (bullets|ordered|glossary)  "ordered">
<!ATTLIST form     method   CDATA    #FIXED "POST">
```

Aus vorherigem Beispiel

```
<!ATTLIST address type (private | company | other) "company">
```

ATTLIST
Erklärung

- `#REQUIRED`
  - Attribut muss angegeben werden
- `#IMPLIED`
  - Attribut kann weggelassen werden
- Wertangabe
  - Standardwert, der überschrieben werden kann
  - `#FIXED`
    - Attribut nimmt immer den gegebenen Wert an

ENTITY

Funktion

- Führt eine Abkürzung ein
- Aus HTML bekannt in Form von
  - `&ouml;`
  - `&uuml;`
- Sonderzeichen und Zeichen anderer Sprachen

ENTITY

Funktion

- Führt eine Abkürzung ein
- Aus HTML bekannt in Form von
  - `&ouml;`
  - `&uuml;`
- Sonderzeichen und Zeichen anderer Sprachen

Vordefiniert

- `amp` ≡ &
- `lt` ≡ <
- `gt` ≡ >
- `quot` ≡ "
- `apos` ≡ '

ENTITY

Beispiel

`<!ENTITY bjoernh "Bj&#xF6;rn Hagemeier">` kann im Dokument als `&bjoernh;` referenziert werden. Außerdem

- `<!ENTITY ouml "&#xF6;">` — ö
- `<!ENTITY uuml "&#xFC;">` — ü

Übung

Übung zum Thema DTD

Inhalt

Strukturelle Beschreibung

DTD

XML Schema
- Struktur
- Typen
- Erweiterte Funktionen

Grenzen von DTDs

- Eingeschränktes Typsystem
  - Inhalte von Elementen sind Text oder weitere Elemente
  - Attribute immer (untypisiertes) CDATA oder wenige Spezialtypen
- Namensräume unzureichend unterstützt
  - werden Simuliert
- XML Schema verbessert die Situation

XML Schema

- Logische Obermenge von DTD
  - Größere Ausdrucksstärke
- Definiert Struktur und Inhalte von XML Dokumenten
- Beschreibt XML Dokumente in Form von XML-Dokumenten
- Besitzt ein Typsystem
- XMLSchema ist selbst in XMLSchema normativ beschrieben

XML Schema

Struktur

- Namensraum: `http://www.w3.org/2001/XMLSchema`
- `targetNamespace`: definiert den Namensraum der definierten Typen und Elemente
- Andere Namensräume werden wie üblich importiert

Beispiel

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetnamespace="http://www.fz-juelich.de/zam/kurse/xml"
    xmlns:tns="http://www.fz-juelich.de/zam/kurse/xml">
   <!-- Schema Definitionen -->
</xsd:schema>
```

xsd:element

Funktion

Definiert ein Element

Beispiel

```
<element name="people"
   type="PeopleType" />
```

Attribute

Für den Anfang

- `name`: Name des Elements
- `type?`: Typ des Elements
- `ref?`: Referenz auf ein definiertes Element (QName)
- `minOccurs?`: Element muss mind. so oft vorkommen
- `maxOccurs?`: Element darf höchstens so oft vorkommen
- `fixed?`: Fester Wert
- `default?`: Vorgegebener Wert

xsd:attribute

- Definiert die möglichen Attribute zu einem Element
- Attribute gehören zum Typ des Elementes

Beispiel

```
<xsd:element name="element1" type="xsd:string" />
```

```
<xsd:element name="element1">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="attribut1" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

xsd:sequence

Funktion

- Beschreibt eine Sequenz von Elementen
- Reihenfolge ist relevant
- Entspricht , in DTD

Beispiel

```
<sequence>
  <element name="name" type="xsd:string" />
  <element name="telefonnummer"
           type="tns:TelefonnummernTyp"
           minOccurs="0" maxOccurs="unbounded" />
  <element name="adresse" type="tns:AdressTyp"
           maxOccurs="unbounded" />
</sequence>
```

xsd:all

Funktion

- Jedes der Elemente innerhalb `xsd:all` kann höchstens einmal in beliebiger Reihenfolge vorkommen
- Definition von Listentypen üblich

Beispiel

```
<xsd:all>
  <element name="name" type="xsd:string" />
  <element name="adresse" type="AdressTyp" minOccurs="0" />
</xsd:all>
```

```
<person>
  <adresse>...</adresse>
  <name>Tanja Tester</name>
</person>
```

xsd:choice

Funktion

- Beschreibt eine Auswahl von Elementen
- Genau eines der Elemente
- Entspricht | in DTD

Beispiel

```
<choice>
  <group ref="versandUndRechnung" />
  <element name="eineAdresse" type="AdressTyp" />
</choice>
```

```
<!ELEMENT auftrag ( versandUndRechnung |
                  ( versand , rechnung ) ) >
<!ELEMENT versandUndRechnung (adresse) >
<!ELEMENT versand            (adresse) >
<!ELEMENT rechnung           (adresse) >
```

xsd:group

Funktion

- Gruppiert mehrere Elemente
- Gruppe kann über einen Namen referenziert werden
- Wiederverwendung
- xsd:choice (s. o.)

Beispiel

```
<group id="versandUndRechnung">
  <element name="versand" type="AdressTyp" />
  <element name="rechnung" type="AdressTyp" />
</group>
```

xsd:annotation

- Ermöglicht Zusatzinformationen
  - Dokumentation (`xsd:documentation`)
  - Programminformationen
- Kann immer am Anfang eines Blocks stehen

Beispiel

```
<element name="name" type="xsd:string">
  <annotation>
    <documentation>Der Name der Person,
      zu der dieser Adressbucheintrag gehört.
    </documentation>
  </annotation>
</element>
```

XML Schema Struktur

Zusammenfassung

- Behandelte Elemente
  - xsd:schema
  - xsd:element
  - xsd:attribute
  - xsd:sequence
  - xsd:all
  - xsd:choice
  - xsd:group
  - xsd:annotation

XML Schema Typsystem

- Einfache Typen vordefiniert
- Basistypen
- Atomar, unstrukturiert
- Typen können für Elemente und Attribute verwendet werden
  - Ausnahmen: ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKESNS sollten nur in Attributen verwendet werden
  - Kompatibilität mit entsprechenden Typen aus XML 1.0 DTD

Datentypen

Vollständige Liste

- string
- normalizedString
- token
- base64Binary
- hexBinary
- integer
- positiveInteger
- negativeInteger
- nonNegativeInteger
- nonPositiveInteger
- long
- unsignedLong
- int
- unsignedInt
- short
- unsignedShort
- byte
- unsignedByte
- decimal
- float
- double
- boolean
- duration
- dateTime
- date
- time
- gYear
- gYearMonth
- gMonth
- gMonthDay
- gDay
- Name
- QName
- NCName
- anyURI
- language

Datentypen
Kategorisierung

- Zeichenketten (Strings)
- Ganzzahlen (Integer)
- Dezimalzahlen
- Datums- und Zeittypen
- Spezielle Typen

Stringtypen

- xsd:string
  - Wird so Behandelt, wie er ist
- xsd:normalizedString
  - Zeilenumbruch, Tabulator und Wagenrücklauf werden in Leerzeichen umgewandelt
- xsd:token
  - Wie normalizedString, zusätzlich werden aufeinander folgende Leerzeichen zusammengefasst

Integertypen

- integer: ..., −1, 0, 1, ...
- positiveInteger: 1, 2, ...
- negativeInteger: ..., -2, -1
- nonNegativeInteger: 0, 1, 2, ...
- nonPositiveInteger: ..., -2, -1, 0
- long: ca. -9.0E18, ..., -1, 0, 1, ..., 9.0E18
- unsignedLong: ca. 0, 1, ..., ca. 18.0E18
- int: -2.0E9, ..., -1, 0, 1, ..., 2.0E9
- unsignedInt: 0, 1, ..., 4.0E9
- short: -32 768, ..., -1, 0, 1, ..., 32 767
- unsignedShort: 0, 1, ..., 65 535
- byte: -128, ..., -1, 0, 1, ..., 127
- unsignedByte: 0, 1, ..., 255

Dezimaltypen

- decimal: beliebige Dezimalzahl
- float: äquivalent zu 32-Bit einfach genau
- double: äquivalent zu 64-Bit doppelt genau

xsd:simpleType

Funktion

Definiert einen *einfachen* Typ, keine Struktur

Attribute

- name: Name des neuen Typs
- final: Erlaubt/verbietet weitere Einschränkung

Beispiel

```
<simpleType name="PLZ">
  <restriction base="nonNegaiveInteger">
    <xsd:pattern value="[0-9]{5}" />
  </restriction>
</simpleType>
```

- Elemente eines einfachen Typs können keine Attribute haben

xsd:simpleType

Erläuterungen

- Können auf drei Arten definiert werden
  - Einschränkung bestehender einfacher Typen (restriction)
  - Listen-Typen
  - Union-Typen
- Kann als Typ für Elemente und Attribute verwendet werden
- Kann keine Attribute definieren
- Elemente mit Attributen und einfachem Typ als Inhalt
  - Erweiterung des einfachen Typs zu komplexem Typ mit einfachem Inhalt und dem gewünschten Attribut

Ableitung durch Einschränkung

xsd:restriction

- Einschränkung bestehender Typen mittels Aspekten (Facet)
  - `length`, `minLength` und `maxLength` für Zeichenketten und ähnliche Typen
  - `pattern` für alle Grundtypen
  - `enumeration` für alle außer Boolean
  - `maxInclusive`, `maxExclusive`, `minInclusive` und `minExclusive` für Zahlentypen
  - `totalDigits` für IntegerTypen
  - `fractionDigits` für Grundtyp `decimal` (Integertypen nur mit Wert 0)
- nicht alle Aspekte auf jeden Grundtyp anwendbar
- Grundsätzlich Einschränkung auch für abgeleitete Typen möglich

Facets
Allgemein

- Facets (Aspekte/Facetten) dienen der Einschränkung von Datentypen
- Beschreiben verschiedene Aspekte eines Typs
- Einschränkende Facets sind immer nach folgendem Schema aufgebaut

Beispiel

```
<xsd:restriction base="BASISTYP">
  <FACET value="VALUE" />
</xsd:restriction>
```

- Beispiele aller 12 möglichen Facets folgen

Facets

length, minLength und maxLength

- Anwendbar auf Zeichenketten und Binärdaten
- Legt Grenzen für die Länge der Werte fest
  - `length`: Werte müssen genau die angegebene Länge haben
  - `minLength`: Werte müssen mindestens so lang sein wie angegeben
  - `maxLength`: Werte dürfen höchstens so lang sein wie angegeben

Beispiel

```
<restriction base="xsd:string">
  <length value="5" />
</restriction>
```

Facets
pattern

- Anwendbar auf alle Grundtypen
- Werte müssen dem angegebenen Muster entsprechen
- Beschreibung der Muster als reguläre Ausdrücke ähnlich wie in Perl

Beispiel

```
<!-- Deutsche PLZ -->
<restriction base="xsd:nonNegativeInteger">
  <pattern value="[0-9]{5}" />
</restriction>
```

Pattern

Reguläre Ausdrücke

- Wird in Anhang F zu »XML Schema Teil 2« beschrieben
- Muster wird immer auf dem ganzen Wert überprüft
  - Zeilenanfang und -ende spielen keine Rolle
  - Die Zeichen ^ und $ haben andere bzw. keine Bedeutung
  - ^ schließt Zeichen aus Mengen aus

Beispiel

```
[^0-9].*
A.*Z
```

- Vordefinierte Klassen von Zeichen existieren
  - Liste in Spezifikation

Facets

enumeration

- Schränkt den Wertebereich eines Typs auf die gegebene Menge ein
- Werte müssen zum Grundtyp kompatibel sein

Beispiel

```
<xsd:simpleType name="EUStaat">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Deutschland" />
    <xsd:enumeration value="Italien" />
    <xsd:enumeration value="Frankreich" />
    <xsd:enumeration value="Polen" />
    <!-- Weitere folgen ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Facets

maxInclusive, maxExclusive, minInclusive und minExclusive

- Definiert obere und untere Grenzen, die die gegebenen Werte einschließen (inklusive) oder nicht einschließen (exclusive)
- Offene/abgeschlossene Intervalle
- Anwendbar auf geordnete Wertebereiche
  - alle Zahlentypen
  - Datums- und Zeittypen

Beispiel

```
<restriction base="nonNegativeInteger">
  <maxExclusive value="100">
    <annotation>
      <documentation>Werte von 0 - 99</documentation>
    </annotation>
  </maxExclusive>
</restriction>
```

Facets
totalDigits

- Legt die maximale Anzahl Ziffern im Dezimalsystem fest
- Führende Nullen und im Nachkommateil am Ende folgende Nullen werden nicht gezählt

Facets
fractionDigits

- Legt die maximale Anzahl relevanter Nachkommastellen fest

Beispiel

```
<restriction base="decimal">
  <fractionDigits value="2">
    <annotation>
      <documentation>
        Für die meisten Währungen interessant
      </documentation>
    </annotation>
  </fractionDigits>
</restriction>
```

xsd:complexType

Funktion

Komplexe Typen ...

- können Kindelemente haben
- können Attribute haben

Beispiel

```
<complexType name="AdressTyp">
  <choice>
    <element name="textuell" type="string" />
    <group ref="AdressElemente" />
  </choice>
</complexType>

<group name="AdressElemente">
  <element name="Straße" type="string" />
  <element name="PLZ" type="PLZType" />
  <element name="Ort" type="string" />
</group>
```

xsd:include

- Einfügen einer anderen XMLSchema Datei
- Namensräume müssen übereinstimmen
- Im Ergebnis dasselbe, als wenn Typen etc. direkt im einbindenden Schema definiert wären

xsd:import

Einbinden fremder Namensräume

- Macht andere Namensräume (z. B. für Adresstypen) im aktuell definierten Schema verfügbar (z. B. Adressbuch)

Beispiel

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace=
         "http://www.fz-juelich.de/zam/kurse/xml/adressbuch"
    xmlns:adressen=
         "http://www.fz-juelich.de/zam/kurse/xml/adressen">
<xsd:import namespace=
           "http://www.fz-juelich.de/zam/kurse/xml/adressen"
    schemaLocation="adressen.xsd" />
```

XMLSchema einbinden

Beispiel

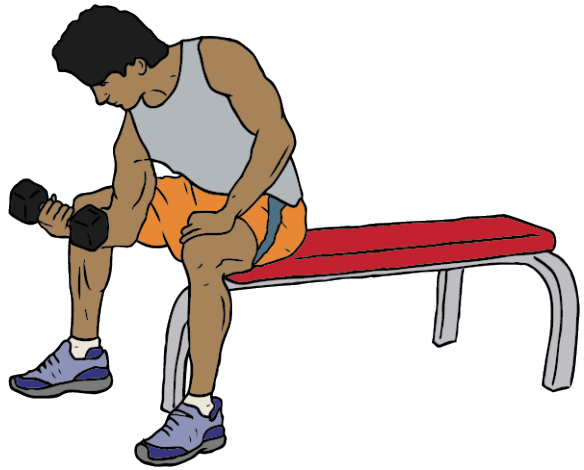
```
<ns1:name xmlns:ns1="http://www.fz-juelich.de/jsc/kurse/xml/2009
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.fz-juelich.de/jsc/kurse
<!-- ... -->
</ns1:name>
```

XML Schema – Weitergehende Informationen

- Nicht-normative Einführung
  - `http://www.w3.org/TR/xmlschema-0/`
  - Leicht zu verstehen
- Normative Texte
  - Strukturen: `http://www.w3.org/TR/xmlschema-1/`
  - Datentypen: `http://www.w3.org/TR/xmlschema-2/`
- Tutorial auf w3schools.com
  - `http://www.w3schools.com/schema/`

Übung

Übung zum Thema XML Schema



Zusammenfassung

Beschreibung von XML

- Beschreibung erlaubt Validierung
- DTD
  - kein Typsystem
  - rudimentär im Vergleich zu XML Schema
  - aus SGML in vereinfachter Form übernommen
- XML Schema
  - Typsystem
  - Erweiterte Funktionen bzgl. Kardinalitäten von Elementen
  - Vollständige Unterstützung von Namensräumen

JÜLICH
FORSCHUNGSZENTRUM

Einführung in XML

Teil IV: Abfragesprachen

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Abfragesprachen

- Bieten allgemein verständliche Syntax, um Inhalte von XML-Dokumenten zu untersuchen
- Können zusammen mit anderen Zusatzsprachen verwendet werden, um bestimmte Ziele zu erreichen
  - Umwandlung von einer Dokumentstruktur in eine andere
    - XSLT
    - z. B. XML-Dokument $\rightarrow$ XHTML

Inhalt

XPath

XML Path Language

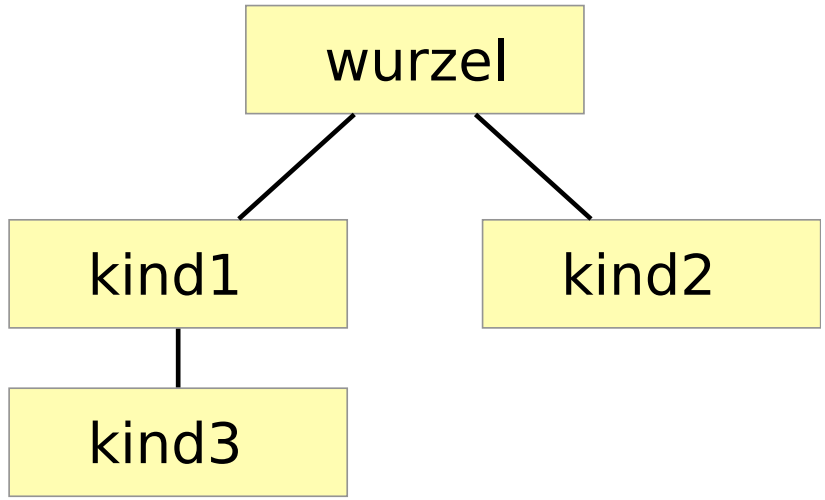
- Zwei Versionen
  - XPath 1.0 (1999)
  - XPath 2.0 (2007)
- pfadorientierte Lokatorsprache
- nur lesen, keine Manipulation, daher keine echte Anfragesprache
- Wird zusammen mit anderen Sprachen verwendet
  - XSLT
  - XPointer
  - XML Schema
  - XQuery
- Wir behandeln XPath 1.0

XPath – Prinzip

- Navigation durch hierarchische Dokumentstruktur
- Ausnutzung der Eltern/Kind-Beziehungen zwischen Knoten

XPath – Elementschritte

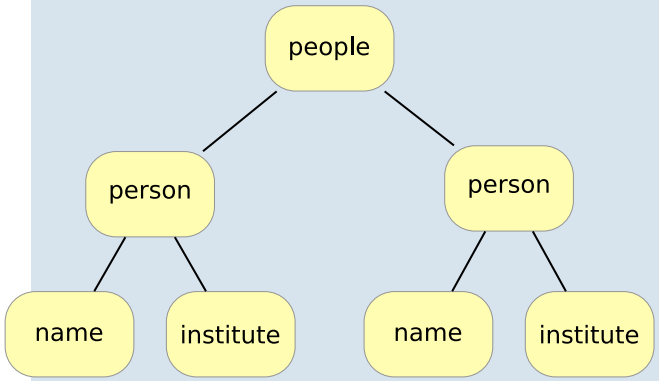
- Wurzel: Start des Ausdrucks mit »/«
- Kinder: /wurzel/kind1
- Eltern: /wurzel/kind1/../kind2
- Beliebig: //kind2
- Selbstreferenz: .
- Attribut: @attrName



XPath – Pfade

- XML Dokument kann als Baum aufgefasst werden
- Eindeutiger Pfad von Wurzel zu jedem Element

Beispiel



- `/people/person/institute` gibt Liste von `institute`-Knoten zurück
- `/people/person[1]/institute`: Liste mit genau einem `institute`-Knoten

- XPath 1.0
  - Rückgabe von Knoten*mengen* (ungeordnet)
- XPath 2.0
  - Rückgabe von Knoten*sequenzen* (geordnet)

XPath
Beispiele

```
/adressen//name[@id="bjoernh"]/../adresse[@type="company"]
/adressbuch[1]/person[1]
```

- Jeder Schritt generiert einen neuen Kontext
- Sequenz von Knoten
- Von dort aus geht es weiter

XPath – Attribute

- Werden mit @ adressiert
- Vergleiche einfach möglich

Beispiel

```
//name[@id="bjoernh"]/../adresse[@type="company"]
```

XPath – Bedingungen

- An einen Pfad können Bedingungen geknüpft werden
  - `//adresse[@type="company"]`
  - `/bookstore//title[contains(., "Potter")]/../price`

XPath – Funktionen

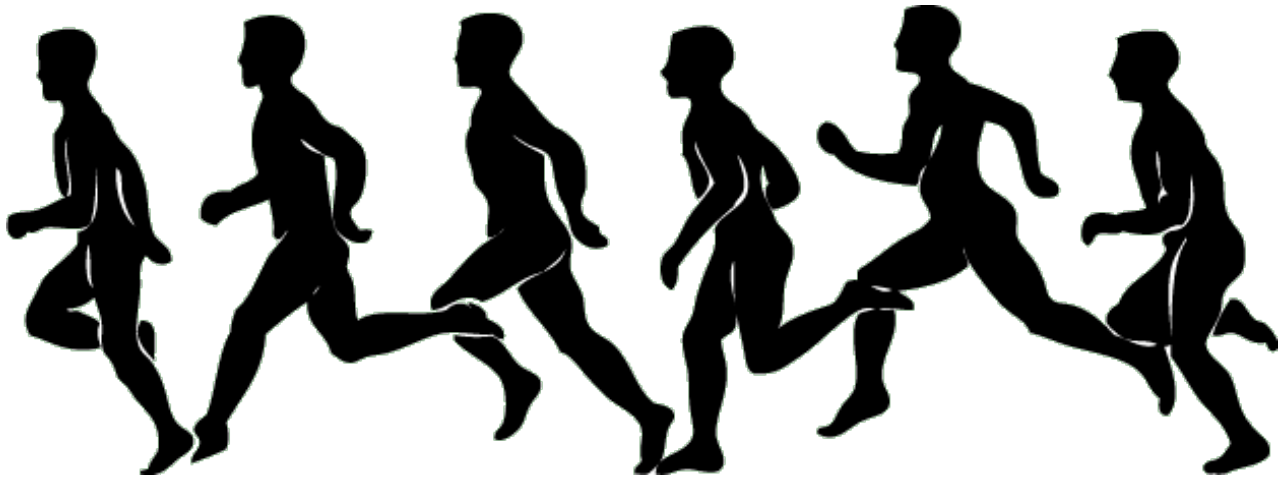
- In XPath 1.0 gab es einen festen Satz von Kernfunktionen
- XPath 2.0 definiert Funktionen gemeinsam mit XQuery 1.0
- XPath 2.0 und XQuery 1.0 Functions and Operators
- `http://www.w3.org/TR/xpath-functions/`
- Funktionen verschiedener Kategorien
  - Numerische Funktionen
  - Funktionen von Zeichenketten
  - Bool'sche Funktionen
  - Funktionen von Zeit- und Zeitraum-Werten
  - Funktionen von QNames
  - Funktionen auf Sequenzen
  - weitere

Funktionen

- Einige Beispiele bereits gesehen
  - `contains("abc", "c")`
    - Gibt Wahrheitswert zurück
  - `count(.)`
    - Zählt die Anzahl der zurückgegebenen Elemente
  - `lower-case(.)`
    - Wandelt einen String in Kleinbuchstaben um
    - `lower-case(//name)`
- Funktionen können geschachtelt werden
  - `round(avg(//book/price))`

Übung

Übung zum Thema XPath



Inhalt

XPath

Funktionen

XQuery

XQuery
XML Query

- »... hat für XML die gleiche Bedeutung wie SQL für Datenbanken«
- ... baut auf XPath Ausdrücken auf
  - Jeder XPath-Ausdruck ist ein gültiger XQuery-Ausdruck
  - XQuery Obermenge von XPath
- Suche und Extraktion von XML Elementen in Dokumenten
- Anwendungen
  - »Suche alle CDs mit einem Preis unter 10€ aus dem XML-Dokument cdsammlung.xml«
  - »Suche alle Mitarbeiter, deren Wohnort im PLZ-Bereich 52000-52999 liegt«
- W3C Recommendation seit 23. Januar 2007

Öffnen eines Dokuments

doc("books.xml")

- Verknüpfung von Daten mehrerer Dokumente möglich
- Daher explizite Angabe der Dokumente erforderlich
- `doc("books.xml")` öffnet das Dokument
- Macht die Inhalte von books.xml verfügbar
- Anschließend weiter mit Pfadausdrücken wie in XPath
  - `doc("books.xml")/bookstore/book/title`

Einschränkung der Ergebnisliste

Prädikate

- Prädikate schränken die Ergebnisliste ein
  - `doc("books.xml")/bookstore/book[price<30]`
- Fortsetzung wieder mit Schritten
  - `doc("books.xml")/bookstore/book[price<30]/title`

FLWOR

- Bietet erweiterte Möglichkeiten gegenüber XPath-Ausdrücken
- Gesprochen »Flower«
- Ähnlich der SELECT Anweisung in SQL
- Einfache Beispiele

FLWOR

- Bietet erweiterte Möglichkeiten gegenüber XPath-Ausdrücken
- Gesprochen »Flower«
- Ähnlich der SELECT Anweisung in SQL
- Einfache Beispiele

Beispiel

```
for $v in $doc//video return $v
```

oder

```
for $v in $doc//video where $v/year = 1999 return
$v/title
```

oder in XPath (und natürlich auch XQuery)

```
$doc//video[year=1999]/title
```

FLWOR – Bedeutung

- 5 Klauseln
  - **F**or
  - **L**et
  - **W**here
  - **O**rder by
  - **R**eturn

XQuery Klauseln
For

- Bindet Variable für jedes Element einer Sequenz
- Führt weitere Operationen darauf aus

Beispiel

```
for $i in (1 to 10) return $i * $i

for $v in //video return count($v/actorRef)
```

Schachteln von Ausdrücken

- Ausdrücke lassen sich schachteln

Beispiel

```
avg(
    for $v in //video return count($v/actorRef)
)

round-half-to-even(
  avg(for $v in //video return count($v/actorRef))
, 2)
```

Oder mittels Pfadausdruck

```
round-half-to-even(avg(//video/count(actorRef)), 2)
```

Unterschied

Pfadausdruck vs. For

```
for $actorId in //video/actorRef
return //actors/actor[@id=$actorId]
```

ist im Wesentlichen dasselbe wie

```
//actors/actor[@id=//video/actorRef]
```

aber der Pfadausdruck liefert weniger Elemente als der `for`-Ausdruck

- Pfadausdrücke eliminieren doppeltes Vorkommen
- `for`-Ausdrücke erlauben mehrfaches Vorkommen von Elementen

XQuery Klauseln

Let

- Deklariert eine Variable
- Zuweisung eines Wertes

Beispiel

```
let $maxCredit := 3000
let $overdrawnCustomers :=
      //customer[overdraft > $maxCredit]
return count($overdrawnCustomers)
```

ist identisch mit

Beispiel

```
count(//customer[overdraft > 3000])
```

XQuery Klauseln
for vs. let

For

- Variable nimmt einen Wert für jedes Element der Sequenz an

Let

- Variable wird einmal an einen Wert gebunden
- Wert kann Sequenz oder einzelnes Element sein
- Einfachheit
- Ausdruck wird nur einmal ausgewertet
  - Kann Effizienz steigern

XQuery Klauseln

`for` und `let`

Beispiel

```
for $genre in //genre/choice
let $genreVideos := //video[genre = $genre]
let $genreActorRefs := $genreVideos/actorRef
for $actor in //actor[@id = $genreActorRefs]
return concat($genre, ": ", $actor)
```

- Gibt eine zusammengesetzte Liste von Genres und Schauspielern zurück
  - `action: Bonet, Lisa action: Fisher, Carrie action: Ford, Harrison action: ... comedy: Anderson, Jeff comedy: Ford, Harrison ...`

XQuery Klauseln

Where

- Spezifiziert Bedingungen für die auszugebenden Daten
- Optional in einem FLWOR-Ausdruck
- Darf höchstens einmal nach allen `for` und `let` Anweisungen vorkommen

Beispiel

```
for $genre in //genre/choice
for $video in //video
for $actorRefs in $video/actorRef
for $actor in //actor
where $video/genre = $genre
   and $actor/@id = $actorRefs
return concat($genre, ": ", $actor)
```

XQuery Klauseln

Order by

- Normale Ordnung
  - als ob die `for` Klauseln geschachtelt ausgeführt würden
  - ungeordneter Modus der Ausführung wird unterstützt
- `order by` ändert explizit diese Ordnung

Beispiel

```
for $x in //video
order by $x/year ascending,
         number($x/user-rating) descending
return $x/title
```

- Wird für mehrere Sortierkriterien sehr kompliziert
- Ordnung wie angegeben, nicht mehr so wie geschachtelte `for` Klauseln implizieren

XQuery Klauseln

Return

- Jeder FLWOR Ausdruck hat eine `return` Klausel
- Definiert die Elemente, die ausgegeben werden sollen

Beispiel

```
for $v in //video[genre="comedy"]
return //actor[@id = $v/actorRef]
```

XQuery Klauseln

Return

- Jeder FLWOR Ausdruck hat eine `return` Klausel
- Definiert die Elemente, die ausgegeben werden sollen

Beispiel

```
for $v in //video[genre="comedy"]
return //actor[@id = $v/actorRef]
```

Beispiel

besser:

```
for $v in //video[genre="comedy"]
return
   <actors video="{$v/title}">
      {//actor[@id = $v/actorRef]}
   </actors>
```

Übung

Übung zum Thema XQuery

Zusammenfassung
Abfragesprachen

- XPath
  - Pfadorientierte Lokatorsprache
  - Liefert Sequenzen von Knoten
- XQuery
  - Erweiterte Funktionen gegenüber XPath
  - Erlaubt Verknüpfungen ähnlich SQL

Einführung in XML

Teil V: Verarbeitung von XML

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Inhalt

Transformation
XSLT

XSLT 2.0

- Bedeutung
  - XSL $\equiv$ eXtensible Stylesheet Language
  - XSLT $\equiv$ XSL Transformations
  - XML-basierte Stylesheet-Sprache
- Anwendungen
  - XML-Dokumente (sinnvoll) im Browser betrachten
  - Umwandlung von einem XML-Format in ein anderes
    - auch reine Textformate
- Trennung von Daten und Präsentation
- Moderne Browser unterstützen XSLT
- XSLT verwendet XPath

XSLT

Funktionsweise

- Ein XSLT-Dokument definiert Templates
- Templates definieren einen XPath-Ausdruck, auf dessen Übereinstimmungen das jeweilige Template angewendet wird

XML

```
<people>
<person>
  <name>Björn</name>
  <surname>Hagemeier
  </surname>
</person>
<person>
  <name>Oliver</name>
  <surname>Pocher
  </surname>
</person>
</people>
```

XSLT

```
<xsl:template match=
    "surname">
  <b><xsl:value-of
    select="."/></b>
</xsl:template>
```

Ergebnis

```
Björn
<b>Hagemeier</b>
Oliver
<b>Pocher</b>
```

xsl:stylesheet oder xsl:transform

- Wurzelelement eines XSL Stylesheets
- Mehrere `xsl:template` Elemente möglich

Beispiel

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
...
</xsl:stylesheet>
```

oder

```
<xsl:transform
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
...
</xsl:transform>
```

XSLT im Browser

- Das Stylesheet muss dem Browser bekannt gemacht werden
- spezielles Tag

Beispiel

Angenommen, die Datei mit dem Stylesheet ist stylesheet.xslt

```
<?xml-stylesheet type="text/xsl" href="stylesheet.xslt"?>
```

- Wird von allen gängigen Browsern interpretiert
  - Firefox
  - Netscape
  - Internet Explorer
  - Opera

xsl:template

- XPath-Ausdruck des `match`-Attributes assoziiert das Template mit einem Element

Beispiel

```
<xsl:template match="/">
  <html>
    <body>
      <h1>foo</h1>
    </body>
  </html>
</xsl:template>
```

xsl:value-of

- Wählt einen Wert aus dem aktuellen Knoten aus

Beispiel

```
<xsl:value-of select="../name"/>
<xsl:value-of select="."/>
<xsl:value-of select=".[@type]"/>
```

xsl:for-each

- Wendet den Block auf jedes Element im Kontext an

Beispiel

```
<table>
  <xsl:for-each select="person/address[@type='company']">
    <tr><td>
      <xsl:value-of select="./street"/>
    </td><td>
      <xsl:value-of select="./city"/>
    </td></tr>
  </xsl:for-each>
</table>
```

xsl:sort

- Kann in einer `for-each` Umgebung angegeben werden
- Sortiert nach dem gegebenen Element im Kontext
- `select` gibt in diesem Fall das Element an, nach dem sortiert wird

Beispiel

```
<xsl:for-each select="catalog/cd">
  <xsl:sort select="artist"/>
  <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
  </tr>
</xsl:for-each>
```

xsl:if

- Wendet Block nur Bedingt auf die Eingabe an
- `test`-Attribut muss boolean zurückgeben

Beispiel

```
<xsl:for-each select="catalog/cd">
  <xsl:if test="price &gt; 10">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
  </xsl:if>
</xsl:for-each>
```

xsl:choose

- Ermöglicht mehrfache bedingte Auswahl
- Nur einer der Blöcke wird angewendet
- Analog zu `if ... else if ... else ...`

Struktur

```
<xsl:choose>
  <xsl:when test="...">...</xsl:when>
  <xsl:when test="...">...</xsl:when>
  <xsl:otherwise>...</xsl:otherwise>
</xsl:choose>
```

xsl:apply-templates

- Wendet Templates an
- `select` wählt Knoten aus dem aktuellen Kontext

Beispiel

```
<message>Proceed
  <emph>at once</emph>
to the exit!</message>
```

```
<xsl:template match="message">
  <p>
    <xsl:apply-templates select="child::node()"/>
  </p>
</xsl:template>
<xsl:template match="emph">
  <b>
    <xsl:apply-templates select="child::node()"/>
  </b>
</xsl:template>
```

Eingebaute Templates

- Werden verwendet, wenn kein explizites Template für einen Konten vorhanden ist

Vollständiges Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">
  <xsl:output method="xml" indent="yes" version="1.0"
      encoding="UTF-8"/>
  <xsl:template match="//person">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head><title>Private Adressen</title></head>
      <body>
        <table border="1">
  <xsl:for-each select="./adresse[@type='company']">
    <tr>
      <td><xsl:value-of select="../name"/></td>
      <td><xsl:value-of select="."/></td>
    </tr>
  </xsl:for-each>
</table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Übung

Übung zum Thema XSLT

JÜLICH
FORSCHUNGSZENTRUM

Einführung in XML

Teil VI: Zusammenfassung

21. Oktober 2009 | Björn Hagemeier, Ahmed Shiraz Memon,
Mohammad Shahbaz Memon

Zusammenfassung

- Einführung
- XML
  - Terminologie
  - Struktur
- Grammatik
  - DTD
  - XML Schema
- Abfragesprachen
  - XPath
  - XQuery
- Transformation
  - XSLT