Die Repräsentation komplexer Strukturen JSON, XML

Hans Jürgen Ohlbach

Keywords: JSON, XML, DTD, XML-Schema, RELAX-NG, XLink, XPath, XPointer, XSLT, XQuery, XML-Parsing

Inhaltsverzeichnis

1 Motivation

Die Eingabe eine Adresse, wie z.B.

Klaus Mustermann
Musterallee 1
D-123456 Musterstadt

in ein Computerprogramm bedeutet für das Programm eine Zeichenkette, der es zunächst keinerlei Bedeutung zuordnen kann. Woher soll es wissen, dass die erste Zeile ein Name ist, mit Vor- und Nachname, die nächste Zeile eine Straße und die dritte Zeile eine Stadt mit Postleitzahl? Selbst für einen Leser, der diese Konvention nicht gewohnt ist, könnte das schwierig zu erkennen sein.

Um schon bei der Formulierung des Textes deutlich zu machen, aus welchen Komponenten der Text besteht, und welche Bedeutung die einzelnen Komponenten haben, wurden verschiedene Systeme entwickelt. Die derzeit wohl gebräuchlichsten davon sind die *JavaScript Object Notation*, kurz **JSON** und die *Extensible Markup Language*, kurz **XML**.

2 JSON

JSON ist eine Sprache, die sich an die Programmiersprache JavaScript orientiert. In JavaScript kann man JSON Objekte unmittelbar einlesen und mit eval() in die internen Datenstrukturen integrieren. Viele andere Programmiersprachen bieten aber auch Möglichkeiten, JSON Objekte einzulesen und zu bearbeiten.

Die obige Adresse könnte in JSON folgendermaßen formuliert werden:

```
{
"Adresse":
  {
   "Name":
      {"Vorname": "Klaus",
       "Nachname": "Mustermann" },
   "Straße":
      {"Name": "Musterallee",
       "Hausnummer": 1},
   "Ort":
      {"Name": "Musterstadt",
       "Postleitzahl": "D-123456"}
   }
}
```

Die kleinsten Einheiten in dieser Sprache sind die *Eigenschaften* als Schlüssel:Wert-Paare (durch : getrennt). Der Schlüssel ist immer eine Zeichenkette in Anführungsstrichen, z.B. `"Name"`. Als Wert gibt es folgende Möglichkeiten:

- ebenfalls eine Zeichenkette in Anführungsstrichen, wie in `"Name": "Musterallee",`
- oder eine Zahl wie in `"Hausnummer": 1`.
  Zahlen können wie in den folgenden Beispielen formatiert werden:
  123, -123, 1.23, -12.3, 123e+45, -1.23e-22.
- Boolesche Werte `true` oder `false` wie z.B.: `"männlich": true`
- der Wert `null` (steht z.B. für *unbekannt*)
- ein Array eingeklammert mit [. . .]. Ein Beispiel könnte sein:
  `"Kinder": ["Karl", "Maria"]` oder auch
  `"Kinder": []` wenn es keine Kinder gibt.
- rekursiv eine Objektbeschreibung in Klammern { . . . } wie im Beispiel

```
"Name":
  {"Vorname": "Klaus",
   "Nachname": "Mustermann"}
```

Die Eigenschaften werden durch Komma getrennt.

Die JSON-Strings sind standardmäßig in UTF-8 kodiert. UTF-16 und UTF-32 sind auch möglich.

2.1 JSON in Java

Für Java gibt es Bibliotheken, mit denen man JSON-Files lesen und schreiben kann, und mit den gelesenen Objekten arbeiten kann. Wer damit arbeiten möchte, sollte sich das javax.json Packet ansehen.

Nach
`import javax.json.*;`
kann man z.B. einen Reader erzeugen, der von einem InputStream `stream` ließt:

```
JsonReader reader     = Json.createReader(stream);
JsonObject jsonObject = reader.readObject();
```

Über das jsonObject kann man alle Komponenten auslesen.

Mit `JsonBuilderFactory factory = Json.createBuilderFactory(...);` kann man eine factory erzeugen, mit der man wiederum ArrayBuilder und ObjectBuilder erzeugen kann. Damit kann dann im Speicher JSON-Objekte aufbauen.

Mit `createWriter(OutputStream out)` kann man dann einen JsonWriter erzeugen, der ein JsonObject in einen OutputStream schreiben kann.

3 XML (Extensible Markup Language)

3.1 Die Sprache XML

Die Sprache XML sieht etwas anders aus als JSON, und bietet auch mehr Strukturierungsmöglichkeiten als JSON. Die Ideen und Anwendungsmöglichkeiten sind aber sehr ähnlich: Man möchte Text oder Daten zusammen mit ihrer Bedeutung speichern, und zwar völlig unabhängig von einer möglichen textuellen Darstellung in einem bestimmten Layout.

XML kann man als für Computeranwendungen besser geeigneten Nachfolger der *Standard Generalized Markup Language (SGML)* bezeichnen. SGML wurde entwickelt, um Strukturierungen von Dokumenttypen zu spezifizieren. Als Beispiel: die in Web-Browsern verwendete Sprache HTML wurde ursprünglich mit SGML spezifiziert. Die Anfänge von SGML gehen auf die 1960er Jahre zurück. SGML wurde aber erst 1986 als internationaler Standard festgeschrieben. Viele Komponenten von XML, welches erstmals 1998 veröffentlicht wurde, gehen auf SGML zurück.

XML wurde vom World Wide Web-Konsortium (W3C) entwickelt und wird permanent weiterentwickelt. Alle Spezifikationen findet man daher auf dessen Webseiten: https://www.w3.org/. Zu XML gibt es auch umfangreiche Sekundärliteratur und Tutorials.

Es würde den Rahmen dieses Dokuments sprengen, alle Details zu XML und seinen Erweiterungen aufzulisten. Es soll daher nur ein kurzer Überblick über die Möglichkeiten, die XML bietet, gegeben werden.

Die im ersten Abschnitt benutzte Adresse würde man in einer XML-Datei z.B. folgendermaßen darstellen.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Adressliste>
  <Adresse>
    <Name>
       <Vorname>Klaus </Vorname>
       <Nachname>Mustermann</Nachname>
    </Name>
    <Straße>
       <Name> Musterallee</Name>
       <Hausnummer>1 </Hausnummer>
    </Straße>
    <Ort>
       <Name>Musterstadt</Name>
       <Postleitzahl>D-123456</Postleitzahl>
    </Ort>
  </Adresse>
</Adressliste>
```

Eine etwas kürzere alternative Darstellung wäre:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Adressliste>
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

Die erste Zeile der Datei `<?xml version="1.0" encoding="UTF-8" standalone="yes"?>` beschreibt die XML-Version, die Kodierung (XML-Dokumente sollten immer die in Unicode kodierten Zeichen benutzen), sowie weitere Hinweise für die jeweilige Anwendung.

Danach kommt das *Wurzelelement*, im Beispiel `<Adressliste> ... </Adressliste>`.

Das Wurzelelement enthält weitere *Elemente*. Ein Element startet mit einem *Starttag* und endet meist mit einem *Endtag*. Im obigen Beispiel sind Start- und Endtags:
`<Adressliste> ... </Adressliste>`,
`<Adresse> ... </Adresse>`,
`<Name>... </Name>` usw.

Falls das Element nur aus Attributen besteht, wie bei
`<Name Vorname = "Klaus" Nachname = "Mustermann"/>`
dann ersetzt `/>` das Endtag.

Attribute sind Paare: `Name = "Wert",` durch Leerzeichen getrennt. Nach dem Starttag können beliebig viele Attribute kommen.

Jedes Element kann folgende Strukturen enthalten:

- einfachen Text (CDATA, Character Data)
- verschachtelt weitere Elemente, wie bei

  ```
  <Adressliste>
    <Adresse>
      ...
    </Adresse>
  </Adressliste>
  ```

- Kommentare `<!-- Kommentar -->`,
- Verarbeitungsanweisungen `<?Zielname Daten ?>`. Diese sind anwendungspezifisch. Ein Beispiel wäre eine Anweisung zum Seitenumbruch, wenn aus der XML-Datei ein druckfähiger Text erzeugt werden sollte.

Unicode: XML basiert auf Unicode. Jedes Unicode-Zeichen ist daher in XML-Dokumenten erlaubt. Keine Tastatur ermöglicht es allerdings, die riesige Menge an Unicode-Zeichen direkt einzugeben. Deshalb gibt es einen Abkürzungsmechanismus, mit dem man Unicode-Zeichen über ihre Nummern (Code Points) eingeben kann. Die Nummern können entweder dezimal oder hexadezimal eingegeben werden. Z.B. kann das englische Pfund-Zeichen £ entweder als `&163;` (dezimal) oder als `&x00A3;` (hexadezimal) eingegeben werden.

Da sich die Nummern schwer zu merken sind, wurden in vielen Anwendungen Abkürzungen definiert, die man mit `&name;` nutzt. Insbesondere wenn man in Texten Zeichen verwenden, will, die in XML eine Sonderbedeutung haben, muss man die Unicode Abkürzungen verwenden.

Das sind für <: <, für >: >, für &: &, für ”: ", für ’: '.

Entities: Die Möglichkeit, Abkürzungen für Unicode-Zeichen zu definieren ist nicht auf einzelne Zeichen beschränkt. Mit einer Deklaration `<!ENTITY autor "Klaus Musterautor" >` definiert man `&autor;` als Abkürzung für den String `Klaus Musterautor`.

Namensräume (Namespaces): Die Elementnamen eines XML-Dokuments können im Prinzip beliebig gewählt werden. Für viele Anwendungen ist es aber sinnvoll, eine Dokumentstruktur fest vorzugeben, und damit auch die Elementnamen festzulegen. Wenn man z.B. ein Adressverwaltungsprogramm schreiben will, und die Adressen als XML-Dokumente speichern will, dann erleichtert es die Programmierung ungemein, wenn man von fest vorgegebenen Elementnamen in dem XML-Dokument ausgehen kann.

Bei komplexeren Anwendungen hat man jedoch auch komplexere Dokumente, die sich aus unterschiedlichen Teilen zusammensetzen können, deren Struktur unabhängig voneinander festgelegt wurde. Als Beispiel, eine Anwendung, die „Gelbe Seiten“ verwaltet braucht nicht nur Adressen, sondern auch Tätigkeitsbeschreibungen. Die XML-Struktur von Tätigkeitsbeschreibungen sollte auch festgelegt werden, hat aber zunächst nichts mit Adressen zu tun.

Falls beide Strukturbeschreibungen zufällig oder wegen ähnlicher Bedeutung die gleichen Elementnamen benutzen, kann es bei der Vereinigung der beiden XML-Strukturen zu Konflikten kommen. Um mit diesen Konflikten umzugehen, wurde das Konzept der *Namensräume* eingeführt.

Ein Namensraum ist ein Präfix, der einem Elementnamen, durch : getrennt, vorangestellt werden kann. Dadurch kann man gleiche Elementnamen in verschiedenen Namensräumen voneinander unterscheiden. In unserem Addressverwaltungsbeispiel würde man einen Namensraum z.B. folgendermaßen als Attribut definieren:

```
<Adressliste xmlns:adr="http://www.addressverwaltung.de/version1">
  <adr:Adresse>
    <adr:Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <adr:Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <adr:Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

`http://www.addressverwaltung/version1` ist der eigentliche Namensraum, `adr` wäre eine Abkürzung dafür. Darin, dass der Namensraum als URI (Universal Resource Identifier) angegeben wird, steckt die Idee, dass hinter dieser URI genauere Informationen über den Namensraum zu finden ist. Dies muss aber nicht sein, die URI kann frei erfunden sein. Sie sollte aber Hinweise liefern auf den Autor / die Organisation, die für die Definition des Namensraumes verantwortlich ist.

Es ist auch erlaubt, die Abkürzung wegzulassen.

```
<Adressliste xmlns="http://www.addressverwaltung.de/version1">
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

Jetzt werden alle Elementnamen in der Adressliste in diesem Namensraum interpretiert.

Das Namensraumkonzept wirkt nur auf Elementnamen, *nicht auf Attributnamen.* Da Attributnamen nur innerhalb der zugehörigen Elemente vorkommen, kann es i.A. keine Verwechslungen geben, falls gleiche Attributnamen an verschiedenen Stellen verwendet werden.

Das `xmlns`-Attribut darf in jedem XML-Element vorkommen und wirkt dann auf die Elemente innerhalb dieses Elements. Es kann auch mehrfach vorkommen, um gleichzeitig verschiedene Namensräume mit verschiedenen Abkürzungen zu definieren, die dann gemischt benutzt werden dürfen.

Als Abkürzung für einen Namensraum darf man beliebige Namen benutzen, außer dem Namen `xml`. Dieser wird für spezielle Attributnamen benutzt, wie z.B. `xml:lang = "de"`.

Das Konzept von Namensräumen ist sehr ähnlich zum Konzept von Packages oder Modulen in Programmiersprachen. Dort kann man allerdings Namensräume hierarchisch verschachteln. Dies ist in XML nicht vorgesehen.

3.2 XML-Strukturdefinitionen

XML kann man verwenden, indem man für die Elemente und Attribute beliebige Namen vergibt, und die auch beliebig verschachtelt strukturiert. In vielen Anwendungen möchte man aber die Struktur der XML-Dokumente fest vorgeben, und Namen benutzen, die eine Bedeutung haben.

In unserem Einführungsbeispiel

```
<Adressliste>
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

haben die Element- und Attributnamen ja eine feste Bedeutung. Wenn man diese willkürlich verändert, dann bekommt die Anwendungssoftware, die mit den Daten arbeiten soll, Probleme.

Es wurden daher verschiedene Mechanismen entwickelt, mit denen man die Struktur von XML-Dokumenten festlegen kann.

3.2.1 Document Type Definition (DTD)

Dies ist die älteste Form, mit der man eine SGML- und daher auch eine XML-Dokumentstruktur definieren kann. Eine DTD kann entweder direkt in dem Dokument stehen, oder in einem externen File. Die obige Adressliste würde in einer DTD folgendermaßen spezifiziert:

```
<!DOCTYPE Adressliste [
<!ELEMENT Adressliste Adresse*>
<!ELEMENT Adresse (Name,Straße,Ort)>
<!Element Name EMPTY>
<!Element Straße EMPTY>
<!Element Ort EMPTY>
<!ATTLIST Name
  Vorname  CDATA #IMPLIED
  Nachname CDATA #REQUIRED>
<!ATTLIST Straße
  Name       CDATA #REQUIRED
  Hausnummer CDATA #REQUIRED>
<!ATTLIST Ort
  Name         CDATA #REQUIRED
  Postleitzahl CDATA #REQUIRED>
]>
```

Die Element-Deklarationen könnten auch in einem externen File `Adressen.dtd` stehen, worauf mit `<!DOCTYPE Adressliste SYSTEM Adressen.dtd>` referenziert wird.

`CDATA (Character Data)` steht für beliebige Zeichenketten, die einfach so, wie sie sind eingelesen werden müssen. im Gegensatz zu `PCDATA (Parsable Character Data)`. Dies steht auch für beliebige Zeichenketten, worin aber entities und andere Elemente vorkommen dürfen, die entsprechend ihrer Bedeutung verarbeitet werden.

Ein als `#REQUIRED` gekennzeichnetes Attribut muss vorkommen, während ein `#IMPLIED` gekennzeichnetes Attribut optional ist.

Die DTD-Syntax bietet einige weitere Möglichkeiten, z.B. kann man die Inhalte von Elementen ähnlich regulären Ausdrücken spezifizieren.

Die Attributwerte kann man noch genauer spezifizieren, z.B. ein Attribut als ID kennzeichnen, worauf man mit IDREF referenzieren kann, oder man kann feste Werte aus einer vorgegeben Liste festlegen usw.

Des weiteren kann man in einer DTD auch Entities definieren.

DTDs wurden schon für SGML entwickelt. Da es in SGML kein Konzept von Namensräumen gibt, unterstützen DTDs nicht die Definition von Namensräumen.

3.2.2 XML-Schema

Ein Nachteil einer DTD ist, dass sie nicht selbst als XML-Dokument definiert ist. Daher braucht man für die DTDs nochmal einen eigenen Parser.

XML-Schema ist dagegen in XML selbst definiert. Konkrete Instanzen von XML-Schema werden auch als *XML-Schema Definitionen* bezeichnet, und in Files mit der Endung `.xsd` abgelegt.

Außer der XML-Syntax bietet XML-Schema gegenüber DTDs noch folgende Möglichkeiten

- vordefinierte *atomare Datentypen*: `xs:string, xs:decimal, xs:integer, xs:float xs:boolean, xs:date, xs:time` plus einige XML-spezifische Typen,
- die Möglichkeit, komplexe Datentypen zu definieren,
- neue Datentypen können durch Einschränkung alter Datentypen definiert werden.
- ähnlich wie in relationalen Datenbanken lassen sich eindeutige Schlüssel definieren.
- Fremde Schemata lassen sich durch include- und import-Tags wiederverwenden.

Obwohl die original XML-Schema Definition mehrere hundert Seiten umfasst, und schwierig zu lesen ist, und darüber hinaus etliche praktische Defizite aufweist, wird XML-Schema in vielen Anwendungen benutzt.

3.2.3 RELAX-NG (REgular LAnguage for XML Next Generation)

RELAX-NG wurde etwas zur selben Zeit als XML-Schema entwickelt, und hat auch ähnliche Ausdrucksmöglichkeiten. Neben einer XML-Syntax für RELAX-NG Schema Definitionen gibt auch noch eine *Compact Syntax*.

Zur Illustration betrachten wir wieder in unsere Addressliste:

```
<Adressliste>
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

RELAX-NG Definition in XML-Syntax Diese könnte folgendermaßen aussehen:

```
<element name = "Adressliste">
  <zeroOrMore>
    <element name = "Adresse">
      <element name = "Name">
          <attribut name = "Vorname"><text/></attribut>
          <attribut name = "Nachname"><text/></attribut>
      </element>
      <element name = "Straße">
          <attribut name = "Name"><text/></attribut>
          <attribut name = "Hausnummer"><text/></attribut>
      </element>
      <element name = "Ort">
          <attribut name = "Name"><text/></attribut>
          <attribut name = "Postleitzahl"><text/></attribut>
      </element>
    </element>
  </zeroOrMore>
</element>
```

RELAX-NG Definition in Compact Syntax Diese könnte folgendermaßen aussehen:

```
element Adressliste {
  element Adresse {
    element Name   {attribut Vorname {text}, attribut Nachname {text}}
    element Straße {attribut Name {text}, attribut Hausnummer {text}}
    element Ort    {attribut Name {text}, attribut Postleitzahl {text}}}*}
```

RELAX-NG hat natürlich noch viele weitere Strukturierungsmöglichkeiten. Es kann auch auf extern definierte Datentypen zurückgegriffen werden, insbesondere auf die Datentypen von XML-Schema.

3.3 Spezielle Sprachen zur Unterstützung von XML-Anwendungen

XML hat die Entwicklung einer Reihe von Sprachen getriggert, mit denen man spezielle Anwendungen unterstützen kann.

3.3.1 URIs, IRIs und URLs

URIs (Uniform Resource Identifiers) dienen zur Identifikation von abstrakten oder physischen Ressourcen, z.B. XML-Dokumenten, die irgendwo im Internet gespeichert sind. Für URIs werden nur ASCII-Zeichen benutzt. Die Erweiterung auf Unicode Zeichen bezeichnet man als *International Resource Identifier (IRI)*.

URIs bestehen aus bis zu 5 Komponenten, z.B.:

$$\underbrace{\text{http}}_{\text{Schema}} : // \underbrace{\text{www.personen.de:8070}}_{\text{authority}} / \underbrace{\text{Addressen.txt}}_{\text{path}} ? \underbrace{\text{name=Mustermann}}_{\text{query}} \# \underbrace{\text{Vorname}}_{\text{fragment}}$$

Die konkrete Syntax von path, query und fragment richtet sich danach, was das Programm, welches auf die Ressource zugreifen muss, verarbeiten kann. In dem Beispiel oben wäre es ein eher fiktiver Zugriff auf ein Text Dokument.

Als Schema sind neben `http` eine ganze Reihe anderer Schemata definiert, u.a. `file` für File-Zugriff, `ssh` für verschlüsselter shell Zugriff usw.

Ein wichtiges Schema ist auch *Uniform Resource Name (URN)*, mit dem man ortsunabhängige Identifikatoren für Ressourcen vergeben kann, z.B. `urn:isbn:3827370191` für ein bestimmtes Buch.

URL (Uniform Resource Locators) sind URIs, die speziell für Schemata wie http, mailto, news, file oder ftp definiert sind. Dabei werden die 5 Komponenten von URIs weiter verfeinert, wie in folgendem Beispiel:

$$\underbrace{\text{http}}_{\text{Schema}} : // \underbrace{\text{MaxUser}}_{\text{Benutzer}} : \underbrace{\text{PWD}}_{\text{Passwort}} @ \underbrace{\text{www.personen.de}}_{\text{host}} : \underbrace{\text{8070}}_{\text{port}} / \underbrace{\text{Addressen.txt}}_{\text{path}} ? \underbrace{\text{name=Mustermann}}_{\text{query}} \# \underbrace{\text{Vorname}}_{\text{fragment}}$$

Für die anderen Schemata gelten entsprechend modifizierte Syntaxregeln.

3.3.2 XLink

Die XLink-Sprache definiert eine Reihe von XML-Attributen, mit denen man die Verbindung zwischen zwei Dokumenten beschreiben kann, ähnlich zum Anchor-Element `<a...` in HTML-Dokumenten. XLink erlaubt aber genauere Angaben über die Art der Verbindung.

Ein Beispiel:

```
<user xmlns:xlink="http://www.w3.org/1999/xlink"
       xlink:href="user.xml"
       xlink:type="simple"
       xlink:role="http://www.example.com/list/userlist.xml"
       xlink:title="User List">Current List of Users>
</user>
```

Zusätzlich zu den normalen Hyperlinks kann man mit XLink auch multidirektionale Verbindungen beschreiben, d.h. unterschiedliche Wege zwischen beliebig vielen Dokumenten (welche nicht unbedingt XML-Dokumente sein müssen).

3.3.3 XPath

Mit der XPath-Sprache kann man ganz bestimmte Komponenten eines XML-Dokuments adressieren. Das kann eine einzelne Komponente sein (Element, Attribute,...), oder auch eine Menge von Komponenten mit bestimmten Eigenschaften. Dafür betrachtet XPath ein XML-Dokument als Baum, in dem man nach unten, seitwärts, und nach oben navigieren kann, und das abhängig von den Werten der jeweiligen Knoten.

Betrachten wir wieder unser Standardbeispiel:

```
<Adressliste>
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

XPath-Ausdrücke hierfür könnten sein:

XPath-Ausdruck	Bedeutung
/Adressliste	das erste Element Adressliste
/*	das äußerste Element, unabhängig vom Namen
//Adresse	alle Adresse-Elemente
//Adresse/Name	alle Namen-Elemente innerhalb der Adresse-Elemente
//Name[@Vorname = 'Klaus']	alle Namen mit Vorname 'Klaus'
//Name[@Vorname = 'Klaus']/parent	alle Adressen von 'Klaus'en.

XPath ist Teil von weiteren Sprachen, wie XPointer, XSLT und XQuery.

3.3.4 XPointer

XPointer erweitert XPath um Möglichkeiten, die in XPath nicht vorgesehen sind, u.a.:

- `start-point()`,`end-point()` bezeichnet Punkte vor und nach XML-Strukturelementen
- `range()`,`range-to()` bezeichnet Bereiche zwischen XML.Strukturelementen
- mit `string-range()` kann man Bereiche in Freitexten adressieren.

XPointer-Ausdrücke kann man insbesondere mit XLink-Ausdrücken verbinden, um auf bestimmte Bereiche eines XML-Dokuments zuzugreifen.

Beispiel:
`xlink:href="Adressen.xml#xpointer(//Adresse/Name[@Nachname='Maier']/range-to(//Adresse/Name[@Nachname='Mueller'])"`

würde im Dokument `Adressen.xml` alle Adressen zwischen `Maier` und `Mueller` selektieren.

3.3.5 XSLT (Extensible Stylesheet Language Transformations)

XSLT ist eine Sprache, mit der man ein XML-Dokument in ein anderes XML-Dokument transformieren kann. XSLT ist Turing-vollständig, d.h. so mächtig wie jeder moderne Programmiersprache. Entsprechend komplex sind seine Bestandteile.

Um einen ersten Eindruck über diese Sprache zu bekommen, erweitern wir das Standardbeispiel etwas:

```
<Adressliste>
  <Adresse>
    <Name Vorname = "Klaus"           Nachname = "Mustermann"/>
    <Straße  Name = "Musterallee"   Hausnummer = "1"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
  <Adresse>
    <Name Vorname = "Maria"           Nachname = "Musterfrau"/>
    <Straße  Name = "Musterallee"   Hausnummer = "2"/>
    <Ort     Name = "Musterstadt" Postleitzahl = "D-123456/>
  </Adresse>
</Adressliste>
```

Die folgende XSLT-Spezifikation transformiert die Namens-Anteile in Englisch:

```
<?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"/>

  <xsl:template match="/Adressliste">
    <names>
      <xsl:apply-templates select="Adresse"/>
    </names>

  <xsl:template match="Name">
    <name firstname="{@Vorname}" surname="{@Nachname}"/>
   </xsl:template>
</xsl:stylesheet>
```

Angwandt auf das obige Beispiel ergibt es folgendes Resultat.

```
<?xml version="1.0" encoding="UTF-8"?>
<names>
  <name firstname="Klaus" surname="Mustermann"/>
  <name firstname="Maria" surname="Musterfrau"/>
</names>
```

Dabei benutzt XSLT die Sprache XPath, um bestimmte Komponenten des Dokuments zu selektieren, auf die dann die Transformations-Templates angewandt werden. Jedes Template legt die Struktur des Ausgabeelements fest, und benutzt Attributnamen und weitere spezielle Funktionen, um die Werte aus dem Ausgangsdokument in das Zieldokument zu übertragen.

3.3.6 XQuery (XML Query Language)

XQuery hat eine ganz ähnliche Funktion wie XSLT. Während XSLT urspünglich entwickelt wurde, um z.B. XML-Dokumente als HTML-Dokumente anzeigbar zu machen, ist XQuery eher in Analogie zu SQL für Datenbankabfragen gedacht. Beide erzeugen als Ausgaben jedoch wiederum XML-Dokumente.

Zur Illustration implementieren wir das Adressenbeispiel von oben in XQUERY.

```
<names>
{
 for $name in doc("Adressen.xml")//Name
   return <name firstname = "{$name[@Vorname]}"
                surname   = "{$name[@Nachname]}" />
}
</names>
```

Es erzeugt genau die gleiche Ausgabe wie oben.

3.4 XML-Parsing

Um XML-Dokumente von Files in den Computer einzulesen, gibt es zwei Möglichkeiten:

SAX-Parser: (Simple API for XML) Diese Parser lesen ein XML-Dokument Element-für Element in den Hauptspeicher. Dabei können für bestimmte XML-Konstrukte sog. *Eventhandler* definiert werden. Das sind Programme, die dann aufgerufen werden, wenn der Parser dieses Konstrukt erkannt hat. Das komplette Dokument selbst wird nicht im Hauptspeicher gespeichert. Sobald das Dokument eingelesen wurde, und alle Events abgearbeitet wurden, sind auch alle Daten des Dokuments im Hauptspeicher wieder gelöscht.

SAX-Parsing eignet sich insbesondere für sehr große Dokumente, die einmalig nach bestimmten Informationen durchsucht werden müssen, und mehr wird davon nicht gebraucht.

In Java gibt es dazu das *Java API for XML Processing (JAXP)*.

DOM Parser: DOM ist das *Document Object Model*. Es ist eine standardisierte Darstellung des kompletten XML-Dokuments im Hauptspeicher, zusammen mit Zugriffsfunktionen auf die einzelnen Komponenten.

Ein DOM-Parser liest das komplette Dokument ein und erzeugt das DOM im Hauptspeicher. Dies kann dann von entsprechenden Programmteilen beliebig (und beliebig oft) bearbeitet werden.

In Java gibt es dazu die Klassen `DocumentBuildFactory` und `DocumentBuilder`, die ein XML-Dokument in ein DOM-Objekt parsen können.

Weiterhin unterscheidet man *validierendes* und *nicht validierendes* Parsing. Zum validierenden Parsing benötigt man eine Spezifikation der Dokumentstruktur, z.B. als DTD, XML-Schema oder RELAX-NG. Ein validierender Parser überprüft das Dokument, ob es dem Schema genügt, und meldet eventuell Fehler.

3.5 XML und SGML

SGML wurde ursprünglich in erster Linie entwickelt, um große Textdokumente zu strukturieren. Dies wurde meist von den Autoren selbst gemacht. Daher sollte SGML *Menschen* das Editieren von SGML-Dokumenten erleichtern. Dies führte dazu dass die Syntaxregeln teilweise gelockert sind. Z.B. muss nicht immer zu einem Starttag das entsprechende Endtag hingeschrieben werden. Wenn das Endtag aus dem Kontext eindeutig ist, kann man es weglassen.

Solche Lockerheiten erschweren natürlich die Implementierung von Parsern. In XML wurden daher die Syntaxregeln von SGML verschärft, um effizientes Parsing zu ermöglichen. Jedes XML-Dokument sollte aber weiterhin von einem SGML-Parser gelesen werden können. Viele Weiterentwicklungen wurden aber auf der Basis von XML gemacht, so dass die XML-Welt inzwischen weit umfangreicher ist als die SGML-Welt.

3.6 XML-Sprachen

Viele weit verbreitet Sprachen wurden inzwischen als Anwendungen von XML definiert.

Einige der vielen Beispiele sind:

xhtml (XML-konformes HTML: HTML selbst wurde ursprünglich auf der Basis von SGML definiert, und das gilt auch noch für HTML 5. XHTML hält sich dagegen an den XML-Standard.

XSL-FO (Extensible Stylesheet Language – Formatting Objects) ist eine XML-Anwendung, die beschreibt, wie Text, Bilder, Linien und andere grafische Elemente auf einer Seite angeordnet werden. XSL-FO ist daher eine Alternative zu Latex. Allerdings kann XSL-FO mittels XSLT aus XML-Dokumente erzeugt werden, was für Latex nicht so einfach ist.

DocBook ist ein Dokumentenformat, das sich besonders eignet zur Erstellung von Büchern, Artikeln und Dokumentationen im technischen Umfeld (Hardware oder Software).

SVG (Scalable Vector Graphic) zur Spezifikation von 2D-Vektorgraphiken.

X3D für 3D-Graphiken

GML (Geographic Markup Language) für Landkarten

GPX (GPS Exchange Format): für GPS-Daten

OSM (OpenStreeMap): Kartendarstellungen, als Konkurrenz zu Google Maps.

MusicXML um Musik (Noten) aufzuschreiben

SMIL für zeitsynchroniserierte Multmedia Inhalte

MPEG-7 MPEG-7 Metadaten