XML

Udo Kelter

20.05.2005

Zusammenfassung dieses Lehrmoduls

Die Extensible Markup Language (XML) ist der Kern einer Gruppe von Standards, die es erlauben, Transportdateiformate zu definieren. Transportdateien dienen dazu, Daten zwischen Applikationen, die ggf. auf verschieden Rechnern laufen, auszutauschen. Dieses Lehrmodul stellt die wesentlichen Teile der Syntax und Semantik von XML vor, insbesondere die Datenmodellierung der Dokumente in Form von DTDs (Document Type Definitions).

obligatorisch: – Datenverwaltungssysteme

Stoffumfang in Vorlesungsdoppelstunden: 1.0

Inhaltsverzeichnis

1 Einleitung

Transportdateien dienen dazu, Daten zwischen Applikationen, die ggf. auf verschiedenen Rechnern laufen, auszutauschen. Die Extensible Markup Language (XML) ist der Kern einer Gruppe von Standards, die es erlauben, Transportdateiformate zu definieren. Die XML ist eine vereinfachte Version der Standard Generalized Markup Language (SGML), die den Austausch textueller Dokumente zwischen Büroanwendungen behandelt. Ausführliche Vorbemerkungen zum Anwendungskontext und zur Entstehungsgeschichte der XML und SGML finden sich in [TRD].

Dieses Lehrmodul stellt die wesentlichen Teile der Syntax und Semantik von XML vor. Basis ist die Spezifikation in [XML00].

Beispiel. Um einen ersten Eindruck von XML-Dateien zu bekommen, betrachten wir zuerst das Beispiel in Bild 1. Wir erkennen drei Hauptteile der Datei:

1. Die erste Zeile gibt u.a. die unterstellte XML-Version an.
2. Der Bereich von `<!DOCTYPE` bis `]>` gibt die hier vorliegende Grammatik an. Dieser Grammatik zufolge hat die Wurzel des Syntaxbaums den Typ `Adreßliste`. Eine `Adreßliste` enthält beliebig viele `Adressen`; jede `Adresse` hat 6 Felder.
3. Der Bereich von `<Adreßliste>` bis `</Adreßliste>` stellt eine konkrete Adreßliste dar; vollständig angegeben ist nur die erste Adresse.

Korrektheitsstufen. Der XML-Standard unterscheidet zwei Stufen der Korrektheit des Inhalts einer XML-Datei:

1. Der XML-Standard legt grundlegende syntaktische Merkmale fest. Diese Merkmale betreffen z.B. die Syntax der Tags; sie gelten einheitlich für
   - XML als Metasprache (also bei der Aufschreibung der Grammatiken)

```
<?xml version='1.0' standalone='yes' ?>

<!DOCTYPE Adreßliste [
   <!ELEMENT Adreßliste (Adresse)* >
   <!ELEMENT Adresse (Name, Vorname, Straße,
                      Hausnummer, PLZ, Ort) >
   <!ELEMENT Name (#PCDATA) >
   <!ELEMENT Vorname (#PCDATA) >
   <!ELEMENT Straße (#PCDATA) >
   <!ELEMENT Hausnummer (#PCDATA) >
   <!ELEMENT PLZ (#PCDATA) >
   <!ELEMENT Ort(#PCDATA) >
]>

<Adreßliste>
   <Adresse>
      <Name>Meier</Name>
      <Vorname>Hans</Vorname>
      <Straße>Hauptstr.</Straße>
      <Hausnummer>5</Hausnummer>
      <PLZ>57076</PLZ>
      <Ort>Siegen</Ort>
   </Adresse>
   <Adresse>
   ....
   </Adresse>
</Adreßliste>
```

Abbildung 1: Beispiel für eine XML-Datei

– für *alle* mit XML definierten Sprachen (also bei der Aufschreibung der Dokumente); in den konkreten Grammatiken für einzelne Dokumenttypen werden diese Merkmale daher nicht wie-

derholt

Ein Text, der diese grundlegenden syntaktischen Merkmale erfüllt, heißt **wohlgeformt**.

2. Ein wohlgeformter Text ist **gültig**, wenn er eine dazugehörige Dokumenttyp-Deklaration besitzt und wenn er die sich daraus ergebenden Beschränkungen einhält.

Zur Beschreibung der Syntax von XML benutzen wir i.f. die in Anhang A in [TRD] eingeführte Notation. Die Syntax von XML ist hier nicht vollständig wiedergegeben, einige für einen ersten Eindruck weniger wichtige Bereiche wurden komplett ausgelassen, ferner beschränken wir uns hier i.w. auf die kontextfreien Anteile der Syntax.

Standardkonforme Software. Der XML-Standard beschreibt in erster Linie, wann der Inhalt einer XML-Datei korrekt ist. Softwaretechnisch gesehen besteht der Hauptsinn einer XML-Datei darin, von einem generischen Parser in einen Syntaxbaum oder eine ähnliche Datenstruktur umgewandelt zu werden, die dann von einer Applikation weiterverarbeitet wird (s. [TRD]). Das Modul oder die Klasse, die solche Syntaxbäume einkapselt, muß ein geeignetes Interface haben. Die Spezifikation eines solchen Interfaces muß auch präzise festlegen, in welcher Form auf Fehler reagiert wird. Der XML-Standard definiert *kein* derartiges Interface. Dies ist auch schlecht möglich:

- Man müßte für jede relevante Programmiersprache ein eigenes Interface definieren. Selbst dann, wenn man den gemeinsamen funktionalen Kern in einer sprachunabhängigen Form nur einmal definiert, müssen einzelne Sprachanbindungen definiert werden. Der Arbeitsaufwand zur Erstellung aller Spezifikationen ist hoch.
- Je nach Anwendungskontext kann das Interface und die dahinterstehende Verarbeitungslogik durchaus verschieden gestaltet sein. Ein Beispiel hierfür ist die Frage, ob man den Syntaxbaum immer komplett aufbaut. Unterstellt man große Datenbestände, führt dies zu Performance-Problemen.

– XML-Dateien können auch durch Dienstprogramme weiterverarbeitet werden. Bei diesen kann z.B. die Reaktion auf Fehler ganz anders gestaltet sein als bei einem Syntaxbaummodul.

Der XML-Standard beschreibt daher Fehlerfälle nur umgangssprachlich und läßt bei weniger gravierenden Fehlern offen, wie "standardkonforme Software" damit verfahren kann. Die Definition von Interfaces zu Syntaxbäumen wird auf ergänzende Standards verschoben.

2 Grundlegende syntaktische Merkmale

Auf die "lexikalischen" Aspekte von XML bzw. die "physischen" Strukturen einer XML-Datei, insb. die schon oben erwähnten Entitäten, gehen wir hier nur sehr oberflächlich ein, obwohl dieser Bereich in der Spezifikation viel Raum beansprucht.

Sonderzeichen. Als Zeichen in den Dokumenten sind beliebige Zeichen gemäß einer vereinbarten Zeichensatzcodierung, z.B. ISO 10646, zulässig.

Die Zeichen stehen normalerweise für sich selbst mit Ausnahme weniger Sonderzeichen. Insb. das `<`, `>` und `&` werden normalerweise als Beginn und Ende eines *tags* bzw. Expansion eines Entity-Symbols interpretiert. Innerhalb von Attributwertangaben ist das Zeichen komplett `<` verboten, z.B. ist `<bedingung test="a<1" />` unzulässig.

Wenn Sonderzeichen in Attributwerten oder Elementinhalten auftreten sollen, kann man hierzu die vordefinierten Entitäten `gt` (*greater than*), `lt` (*less than*) und `amp` (*ampersand*) benutzen. Ein Beispiel für einen Aufruf ist `&lt;` . Unser obiges Beispiel kann wie folgt korrigiert werden: `<bedingung test="a&lt;1" />`

Mit einem CDATA-Abschnitt kann man in einem ganzen Textabschnitt die Interpretation der Sonderzeichen verhindern. CDATA-Abschnitte sind überall dort erlaubt, wo Texte stehen und haben die Form `<![CDATA[ .... uninterpretierter Text .... ]]>`. Die schließende Klammer `]]>` darf in dem uninterpretierten Text nicht auf-

treten. Die komplette Syntax ist (die Nummern der Regeln sind aus der Numerierung in [XML00] übernommen worden):

```
[18] CDSect ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20] CData ::= (Char* - (Char* ']]>' Char*))
[21] CDEnd ::= ']]>'
```

Namen. Namen werden wie in Programmiersprachen üblich gebildet; ihre Syntax ist:

```
[5]  Name ::= (Letter | '_' | ':') (NameChar)*
[4]  NameChar ::= Letter | Digit | '.' | '-' | '_' | ':'
                  | CombiningChar | Extender
```

Durch Anwendungen definierte Namen dürfen zusätzlich nicht mit dem Präfix 'xml' in Klein- oder Großbuchstaben (also ('X' | 'x') ('M' | 'm') ('L' | 'l')) beginnen; diese Namen sind für Standardisierungszwecke reserviert.

Der Doppelpunkt ist innerhalb von Namen erlaubt, hat aber im Zusammenhang mit Namensräumen eine besondere Bedeutung und sollte daher in "normalen" Namen nicht verwendet werden.

Groß- und Kleinschreibung innerhalb der *tags* werden (im Gegensatz zu HTML) unterschieden.

Attributinstanzen. Attribute eines Elements werden im Starttag in der Schreibweise `attrname="wert"` notiert; die Anführungszeichen sind obligatorisch.

Zu einem öffnenden *tag* muß immer das korrespondierende schließende vorhanden sein (im Gegensatz zu HTML, wo z.B. `<b><em> .... </b></em>` oder `<p> ... <p> ...` erlaubt ist). Als Sonderform ist bei Elementen, die keine inneren Elemente haben, die Notation `<tagname attribute />` erlaubt.

3 Die logische Struktur von XML-Dateien

3.1 `document`

Startsymbol der XML-Grammatik ist das Nichtterminalsymbol `document`. Für dieses gelten folgende Grammatikregeln:

```
[1]  document ::= prolog element Misc*
```

Unter einem Dokument wird hier also der gesamte Inhalt einer Transportdatei verstanden, während wir oben den Begriff enger gefaßt haben. Das `element` entspricht den Nutzdaten, `prolog` beinhaltet sowohl die Startzeile (`<?xml ....`) als auch die ggf. vorhandene Dokumenttypdeklaration.

Beginnen wir zunächst mit den einfacheren Konzepten:

```
[27] Misc ::= Comment | PI | S
[3]  S ::= (#x20 | #x9 | #xD | #xA)+
[15] Comment ::= '<!--' ((Char - '-') |
                 ('-' (Char - '-')))* '-->'
```

Ein `document` kann am Ende, aber auch an vielen anderen Stellen, beliebig viele "verschiedene Dinge" (`Misc`) enthalten.

`S` ist Leerraum (*White Space*), bestehend aus einem oder mehreren Leerzeichen, Wagenrückläufen, Zeilenvorschüben oder Tabulatoren.

In Regel [3] treten zum ersten Mal Terminalsymbole auf, und zwar einzelne Zeichen. Erlaubt in XML-Dateien sind außer den vorgenannten Zeichen die üblichen darstellbaren Zeichen und darüber hinaus alle Zeichen gemäß ISO/IEC 10646 (Universal Multiple-Octet Coded Character Set (UCS) [ISO10646-1993]) und gemäß dem Unicode Standard, Version 2.0 [Un96]. Für Zeichen aus diesen Zeichensätzen gibt es z.T. alternative Darstellungen; dieses Thema wollen wir hier nicht vertiefen.

Ein Beispiel für einen Kommentar ist `<!-- Dies ist ein Kommentar -->`. Regel [15] ist nicht ganz einfach zu lesen:

– `(Char - '-')` steht für ein beliebiges Zeichen außer dem Minuszeichen.

– `('-' (Char - '-'))` steht für ein Minuszeichen gefolgt von einem anderen Zeichen als dem Minuszeichen.
– `((Char - '-') | ('-' (Char - '-')))` ist somit ein Nicht-Minuszeichen, vor dem ggf. ein Minuszeichen steht. Ein Sequenz `(...)*` hiervon ist ein Text, in dem beliebig viele einzelne Minuszeichen, aber keine zwei Minuszeichen hintereinander und kein Minuszeichen als letztes Zeichen auftreten können.

Etwas einfacher ausgedrückt darf in einem Kommentar die Zeichenfolge `--` nicht auftreten, der Kommentar darf auch nicht mit `--->` enden.

`PI` in Regel [27] sind Verarbeitungsanweisungen (*Processing Instructions*), die z.B. angeben können, mit welcher Anwendung ein Dokument verarbeitet werden soll; wir behandeln sie nicht näher.

3.2 `prolog`

Der Prolog zu Beginn einer Datei hat folgenden Aufbau:

```
[22] prolog ::= XMLDecl? Misc* (doctypedecl Misc*)?
```

Der Prolog enthält also optional eine Dokumenttypdeklaration (`doctypedecl`). Zwar optional, in der Regel aber vorhanden ist die "Startzeile" (`XMLDecl`), die insb. die XML-Version angibt. Ein Beispiel für die `XMLDecl`-Klausel ist:

```
<?xml version='1.0' standalone='yes' ?>
```

Die komplette Syntax von `XMLDecl` ist:

```
[23] XMLDecl ::= '<?xml' VersionInfo EncodingDecl?
                         SDDecl? S? '?>'
[24] VersionInfo ::= S 'version' Eq ("'" VersionNum "'"
                                   | '"' VersionNum '"')
[25] Eq ::= S? '=' S?
[26] VersionNum ::= ([a-zA-Z0-9_.:] | '-')+
[32] SDDecl ::= S 'standalone' Eq
```

```
(("'" ('yes' | 'no') "'") |
 ('"' ('yes' | 'no') '"'))
```

Die `XMLDecl`-Klausel gibt unter `VersionInfo` die unterstellte Version des XML-Standards an. Zur Zeit ist die einzig sinnvolle Versionsnummer 1.0. Unter `EncodingDecl` kann die Zeichensatzcodierung angegeben werden.

Wichtiger ist für uns die Angabe `SDDecl`. Wenn `standalone` den Wert yes hat (z.B. in `<?xml version='1.0' standalone='yes' ?>`), enthält diese Transportdatei auch die Grammatik. Wenn `SDDecl` fehlt oder `standalone` den Wert `no` hat, muß die Grammatik aus einer externen Quelle besorgt werden; diese Quelle muß in einer anschließenden Dokumenttypdeklaration angegeben werden.

3.3 `doctypedecl`

Die Syntax einer Dokumenttypdeklaration ist:

```
[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S?
                     ('[' (markupdecl | DeclSep)* ']' S?)? '>'
[75] ExternalID ::= 'SYSTEM' S SystemLiteral |
                    'PUBLIC' S PubidLiteral S SystemLiteral
```

Der nach `DOCTYPE` angegebene Name muß der des Wurzel-Elementtyps sein.

Ein Beispiel für eine Dokumenttypdeklaration haben wir schon in Abschnitt 1 gesehen. In diesem Beispiel waren innerhalb der Dokumenttypdeklaration einzelne Elementdeklarationen angegeben. Eine Elementdeklaration entspricht einer Grammatikregel (bzw. Produktion) für die Grammatik der Transportdatei.

Durch eine Elementdeklaration werden jeweils die öffnenden und schließenden *tags* mit diesem *tag*-Namen definiert.

Wenn wir im obigen Beispiel die zwischen `[...]` stehenden Elementdeklarationen in eine Datei namens `adressliste.dtd` geschrieben hätten, hätte die Dokumenttypdeklaration folgendermaßen aussehen können:

```
<!DOCTYPE Adreßliste SYSTEM "adressliste.dtd">
```

Markup-Deklarationen können sowohl innerhalb der Dokumenttypdeklaration als auch in einer externen Quelle angegeben sein; in diesem Fall besteht die Gesamtmenge der Dokumenttypdeklaration aus beiden Teilmengen. Im Falle von doppelten Definitionen hat die intern angegebene Menge Priorität; hierhinter steckt die Annahme, daß die grundlegenden Dokumentstrukturen konstant bleiben und die entsprechenden DTDs in externen Dateien stehen, während in einzelnen Dokumenten nur Abweichungen vom Normalfall explizit notiert werden.

Überraschenderweise können auch beide Teilmengen fehlen. Eine solche XML-Datei ist dennoch meist verarbeitbar, weil sich die Struktur des Dokuments auch weitgehendst durch die Tags rekonstruieren läßt.

Auf einige weitere Details, z.B. wie die Namen der externen Ressourcen aufgebaut sind, gehen wir hier nicht ein.

3.4 `markupdecl`

Im Beispiel in Abschnitt 1 haben wir bereits mehrere Elementdeklarationen gesehen. Elementdeklarationen sind ein Spezialfall von Markup-Deklarationen:

```
[29] markupdecl ::= elementdecl | AttlistDecl | EntityDecl
                  | NotationDecl | PI | Comment
[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'
[46] contentspec ::= 'EMPTY' | 'ANY' | Mixed | children
[51] Mixed ::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
             | '(' S? '#PCDATA' S? ')'
```

Die Tags von Markup-Deklarationen beginnen alle mit `<!` und sind daher leicht von anderen Tags unterscheidbar.

Wir gehen hier nur auf die Deklaration von Elementtypen und deren Attributlisten ein; die anderen Alternativen für `markupdecl` betreffen inhaltlich andere Themen.

Elemente haben einen Inhalt und ggf. Attribute (im Sinne von Attributinstanzen). Die Attribute werden in separaten `AttlistDecl`-Klauseln angegeben; der Name in einer solchen Klausel muß der Name eines Elementtyps sein.

Mit `contentspec` können folgende Arten von Inhalten unterschieden werden:

`EMPTY` Elemente dieses Typs haben einen leeren Inhalt.

`ANY` Elemente dieses Typs können einen beliebigen Inhalt haben, also normalen Text und darin eingestreut *tags*. Die *tags* werden als solche interpretiert, d.h. sie müssen ebenfalls den Korrektheitsregeln entsprechen.

`Mixed` Elemente dieses Typs können entweder nur normalen Text als Inhalt haben oder zusätzlich darin eingestreut Elemente vorgegebener Typen. Ein Beispiel für den ersten Fall ist:

```
<!ELEMENT e (#PCDATA)>
```

Die Abkürzung PCDATA bedeutet *parsed character data*. PCDATA-Inhalte dürfen *keine tags* enthalten. Wenn man in einem solchen Text die Zeichen '<' oder '>' verwenden will, muß man sie in der Form `<` bzw. `>` angeben.

Ein Beispiel für den Fall zulässiger eingestreuter Elemente ist:

```
<!ELEMENT e (#PCDATA | a | b | c)*>
```

Man beachte, daß in diesem Fall – aus welchen Gründen auch immer – am Ende ein '`*`' stehen muß. In unserem Beispiel können neben einfachem Text Elemente der Typen `a`, `b` und `c` im Inhalt von `e` auftreten.

`children` Elemente dieses Typs können nur Elemente enthalten, die auch als Kind-Elemente bezeichnet werden; einfacher Text ist nicht erlaubt. Dies ist der Normalfall, wenn man mit XML strukturierte Daten übertragen möchte. Die Syntax von `children` ist wie folgt:

```
[47] children ::= (choice | seq) ('?' | '*' | '+')?
[49] choice ::= '(' S? cp ( S? '|' S? cp )+ S? ')'
[50] seq ::= '(' S? cp ( S? ',' S? cp )* S? ')'
[48] cp ::= (Name | choice | seq) ('?' | '*' | '+')?
```

In [48] ist `Name` (vgl. Grammatikregel [5]) der Name eines Elementtyps; Elemente dieses Typs sind als Kindelemente an der betreffenden Stelle erlaubt. Die Zeichen `?`, `*` und `+` geben an, wie oft das davorstehende syntaktische Konstrukt auftreten muß:

`?` gar nicht oder einmal
`*` beliebig oft (incl. keinmal)
`+` wenigstens einmal

`choice` stellt eine Auswahl dar; `(a | b | c)` bedeutet, daß entweder `a` oder `b` oder `c` an dieser Stelle steht.

`seq` stellt eine Sequenz dar; `(a, b, c)` bedeutet, daß an dieser Stelle `a` gefolgt von `b` und `c` steht.

Als Beispiel betrachten wir noch einmal die folgende Definition aus dem Beispiel in Abschnitt 1:

```
<!ELEMENT Adreßliste (Adresse)* >
```

`Adresse` ist der Name eines Elementtyps. `(Adresse)` ist als Sequenz der Länge 1 nach Regel [50] zu verstehen. Der * wird mit Regel [47] hinzugefügt.

Die "Typkonstruktoren" Sequenz, Auswahl, ?, * und + können beliebig geschachtelt werden. Das folgende Beispiel definiert eine Person, die einen oder mehrere Vornamen, einen optionalen Titel, einen oder mehrere Wohnsitze sowie beliebig viele Telefone hat:

```
<!ELEMENT Person (Titel?, Name, Vorname+,
                  (Straße, Hausnummer, PLZ, Ort)+,
                  Telefonnr* )>
```

3.5 AttlistDecl

Die Struktur der Kindelemente spannt eine baumartige Gesamtstruktur auf; an jedem einzelnen Knoten in dieser Gesamtstruktur können

Attribute plaziert werden. Notiert werden die Attributwerte eines Elements stets im öffnenden *tag*. Die Attribute eines Elementtyps werden in einer separaten Klausel deklariert:

```
[52] AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
[53] AttDef ::= S Name S AttType S DefaultDecl
```

Die Deklaration einer Attributliste ordnet dem Elementtyp mit Namen `Name` beliebig viele Attributdefinitionen (`AttDef`) zu.

Zu einem Elementtyp kann es mehrere Attributlistendeklarationen geben; sofern hierbei das gleiche Attribut mehrfach definiert wird, ist das kein Fehler; es gilt dann die erste verarbeitete Definition, alle späteren werden ignoriert. Standardkonforme Software, die XML-Dateien verarbeitet, muß nicht unbedingt, darf aber in solchen Fällen eine Warnung ausgeben.

Eine Attributdefinition ordnet dem Namen des Attributs (`Name` in [53]) einen Attributtyp (`AttType`) und einen Vorgabewert bzw. eine Optionalitätsangabe (`DefaultDecl`) zu.

Vorgabewerte. Die Bedeutung der meisten XML-Sprachkonstrukte liegt darin, Aufbau und Korrektheit des Inhalts einer XML-Datei zu beschreiben; bei Vorgabewerten ist dies *nicht* der Fall. Die Bedeutung von Vorgabewerten liegt darin, das Verhalten standardkonformer Software zu definieren: Wenn an einem bestimmten Element ein Attribut gelesen wird und die XML-Datei keine explizite Wertangabe für dieses Attribut in diesem Element enthält, liefert die Software den Vorgabewert.

3.5.1 Optionalität von Attributwertangaben

`DefaultDecl` gibt ab, ob für ein Attribut Werte angegeben werden müssen, und hat folgende Syntax:

```
[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED' |
     (('#FIXED' S)? AttValue)
```

`#REQUIRED` bedeutet, daß ein Wert angegeben werden muß. `#IMPLIED` bedeutet, daß die Wertangabe weggelassen werden kann[1] und daß kein Wert vorgegeben ist. Wenn die beiden ersten Alternativen nicht zutreffen, muß in Klammern ein Vorgabewert angegeben werden.

Durch den Zusatz `#FIXED` wird bestimmt, daß das Attribut immer diesen Vorgabewert haben muß, es liegt also eine Konstante vor. Wenn das Attribut in den Nutzdaten auftritt, muß der angegebene Wert gleich dem Vorgabewert sein.

Die redundante Speicherung des immer gleichen Vorgabewerts in vielen Elementen ist natürlich unsinnig (sofern nicht die XML-Datei ohne DTD verarbeitet werden muß), man kann sich diese Attributwertangaben auch sparen.

Selbst dann sind solche Konstanten unsinnig, wenn man von der Denkwelt von Datenbanken ausgeht und die Daten einer XML-Datei als Darstellung von Realweltobjekten versteht. XML-Dateien werden aber nicht nur für diesen Zweck eingesetzt, sondern auch intensiv für die Übertragung nichtpersistenter Daten, z.B. von Formularen. Ein bestimmtes Formular kann nur für ein bestimmtes Release eines Softwaresystems gedacht oder nur für bestimmte Verarbeitungsmodi geeignet sein; entsprechende Daten können sinnvoll als Konstanten modelliert werden, z.B.:

```
<!ATTLIST FormularX
          Uebertragungsmethode CDATA #FIXED "POST">
```

3.5.2 Attributtypen

Attributtypen können wie folgt definiert werden:

```
[54] AttType ::= StringType | TokenizedType |
                 EnumeratedType
[55] StringType ::= 'CDATA'
[56] TokenizedType ::= 'ID' | 'IDREF' | 'IDREFS' | 'ENTITY'
                     | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'
```

[1] Die Bezeichnung "impliziert" ist alles andere als suggestiv.

```
[57] EnumeratedType ::= NotationType | Enumeration
[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S?
                      Name)* S? ')'
[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)*
                     S? ')'
[7]  Nmtoken ::= (NameChar)+
[8]  Nmtokens ::= Nmtoken (S Nmtoken)*
```

Man kann grob 3 Gruppen von Attributtypen unterscheiden; diese Arten unterscheiden sich darin, welche Attributwerte zulässig sind und inwieweit die gespeicherten Attributwerte vorverarbeitet werden, bevor sie an die Applikation weitergeleitet werden.

1. beliebige Texte: Hier ist in der Attributdefinition `CDATA` als `AttType` anzugeben (s. Nonterminal `StringType` und Regel [55]). Abgesehen davon, daß das Zeichen `<` nicht erlaubt ist, kann der Wert eines `CDATA` ein beliebiger Text sein. Es sind auch Tabulatoren und Zeilenvorschübe erlaubt, diese werden aber bei der Vorverarbeitung durch Leerzeichen ersetzt, gehen also aus Sicht von Applikationen verloren.
2. Wortlisten (s. Nonterminal `TokenizedType` und Regel [56]): über die bei `CDATA`-Attributen angewandten Vorverarbeitung hinaus wird hier am Anfang und am Ende stehender Leerraum entfernt, und mehrere aufeinanderfolgende Leerzeichen werden durch ein einziges ersetzt.
3. Aufzählungstypen (s. Nonterminal `EnumeratedType` und Regel [57])[2]: Der Attrbutwert muß genau ein Wort aus einer vorgegebenen Namensliste sein.

Für die Wortlisten existieren u.a. folgende Alternativen für den Attributinhalt:

`ID` Es muß genau ein Wort vorhanden sein, das der Syntax von `Name`

[2]Auf die hierunter fallenden `NotationType`s gehen wir nicht näher ein; mit ihrer Hilfe kann man u.a. sog. *unparsed entities* verarbeiten und Applikationen verwalten, die zur Anzeige derselben genutzt werden sollen.

(s.o. Regel [5]) entspricht. Dieses Wort wird als Identifizierung des Elements interpretiert.

In einem gültigen XML-Dokument darf in zwei `ID`-Attributen nicht der gleiche (normalisierte) Wert auftreten, auch nicht, wenn die Attribute zu verschiedenen Elementtypen gehören und/oder verschiedene Namen haben.

Pro Elementtyp darf maximal ein `ID`-Attribut definiert werden.

ID-Attribute müssen entweder `#IMPLIED` oder `#REQUIRED` sein. Im Fall von `#IMPLIED` kann es also Elemente geben, die keine Identifizierung haben.

`IDREF` Es muß ein Name gemäß der Syntax von `Name` (s.o. Regel [5]) vorhanden sein; dieser wird als Referenz auf ein anderes Element interpretiert. In einem bzgl. einer DTD gültigen XML-Dokument muß es zu jedem Namen, der ein einem `IDREF`-Attribut auftaucht, ein Element mit einem `ID`-Attribut geben, dessen Wert dieser Name ist.

`IDREFS` analog zu `IDREF`, aber mehrere, durch Leerraum getrennte Namen

`NMTOKEN` Es muß genau ein Namenstoken in der Syntax von `Nmtoken` vorhanden sein. Während Namen mit einem Buchstaben, Tiefstrich oder Doppelpunkt anfangen müssen, können Nmtokens auch nur aus Ziffern bestehen.

`NMTOKENS` Es müssen mehrere, durch Leerraum getrennte Worte in der Syntax von `Nmtokens` vorhanden sein.

3.5.3 Schlüsselattribute in XML-Dateien

`ID`-Attribute wirken auf den ersten Blick wie UNIQUE-Attribute bzw. -Spalten in relationalen Datenbanken. Ein gravierender Unterschied besteht aber darin, daß die Werte eines UNIQUE-Attributs nur innerhalb einer Tabelle eindeutig sind, nicht über alle Tabellen hinweg. `ID`-Attribute sind eher mit Surrogaten in objektorientierten Datenbankmodellen vergleichbar.

IDREF-Attribute ähneln Fremdschlüsselattributen in relationalen Datenbanken insofern, als referentielle Inkonsistenzen ("hängende" Referenzen) vermieden werden sollen. Analog gilt dies für IDREFS-Attribute. Die Elemente, die durch die Namen in den Instanzen eines bestimmten IDREF-Attributs referenziert werden, können im Gegensatz zu relationalen Datenbanken heterogen sein, d.h. die Typen dieser Elemente sind nicht notwendig alle gleich.

ID- und IDREF(S)-Attribute vermischen zwei eigentlich unabhängige Themen: Sie geben erstens den Typ, also den zulässigen Wertebereich, des Attributs an (Namenslisten), zweitens ein dokumentweites Konsistenzkriterium.

Konsistenzkriterien werden in XML-Dateien anders behandelt als in Datenbanken: In Datenbanken werden Änderungen, die zu einer Inkonsistenz führen würden, abgelehnt, d.h. die Datenbank ist *immer korrekt* hinsichtlich aller definierten Konsistenzkriterien. In XML-Dateien dies nicht der Fall, da sie von Hand bzw. beliebigen Softwaresystemen erzeugt worden sein können. Es kann höchstens beim Laden einer XML-Datei ein entsprechender Validitätstest durchgeführt werden; dieser Test weist eine Datei bei einer Inkonsistenz als nicht gültig aus, und die Applikation kann dann die Verarbeitung der Datei ablehnen. Das hilft einem nur begrenzt weiter, denn der Datenbestand ist und bleibt defekt. ID- oder IDREF(S)-Attribute bewirken also keinerlei Schutz der Daten.

Ein Problem bei den Validitätstest besteht darin, daß man nur dann, wenn eine DTD vorhanden ist, überhaupt weiß, welche Konsistenzkriterien gelten sollen. Sehr häufig haben XML-Dateien aber keine DTD. Zur Zeit wird ein ergänzender Standard vorbereitet, wonach der reservierte Attributname xml:id stets ein ID-Attribut bezeichnet[3]. Die Eindeutigkeit von xml:id-Attributwerten kann auch ohne DTD geprüft werden.

[3] s. http://www.w3.org/TR/2005/CR-xml-id-20050208/. Gemäß diesem kommenden Ergänzungsstandard kann ein Attribut mit Namen kann nur als

```
<!ATTLIST irgendeinElementMitIdAttribut
          xml:id ID #IMPLIED >
```

bzw. als #REQUIRED definiert werden.

Literatur

[ISO8879-1986] ISO/IEC 8879-1986. 10646-1993. Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML), First edition; International Organization for Standardization; 1986

[ISO10646-1993] ISO/IEC 10646-1993. Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane (plus amendments AM 1 through AM 7); International Organization for Standardization; 1993

[Un96] The Unicode Consortium: The Unicode Standard, Version 2.0; Addison-Wesley Developers Press; 1996.

[RFC2396] Berners-Lee, T.; Fielding, R.; Masinter, L. (eds.): IETF RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax; IETF (Internet Engineering Task Force); 1998-08

[XML00] Extensible Markup Language (XML) 1.0 (Second Edition); World Wide Web Consortium; `http://www.w3.org/TR/2000/REC-xml-20001006.html`; 2000-10-06

[TRD] Kelter, U.: Lehrmodul "Transportdateien und die SGML"; 2004

Index