Transportdateien und die SGML

Udo Kelter

14.05.2004

Zusammenfassung dieses Lehrmoduls

Transportdateien dienen dazu, Daten zwischen Applikationen, die ggf. auf verschieden Rechnern laufen, auszutauschen. Die Extensible Markup Language (XML) ist der Kern einer Gruppe von Standards, die es erlauben, Transportdateiformate zu definieren. XML ist eine vereinfachte Version der Standard Generalized Markup Language (SGML), die den Austausch textueller Dokumente zwischen Büroanwendungen behandelt. In diesem Lehrmodul motivieren wir zunächst generell den Einsatz von Transportdateien und beschreiben die Grundzüge und Randbedingungen der SGML.

obligatorisch: – Datenverwaltungssysteme

Stoffumfang in Vorlesungsdoppelstunden: 1.3

Inhaltsverzeichnis

1 Transportdateien

Transportdateien behandeln das Problem, Daten zwischen verschiedenen Applikationen, die ggf. sogar auf verschiedenen Rechnern laufen, auszutauschen.

Der Datenaustausch zwischen mehreren Applikationen ist einfach realisierbar, wenn diese auf dem gleichen Rechner laufen und in der gleichen Sprache geschrieben sind. Praktisch in jeder Programmiersprache sind Ein-/Ausgabekommandos (oder Bibliotheken) vorhanden, mit deren Hilfe die Werte von Variablen, die Integer-, Fließkomma- oder anderen Zahlen, Texte oder andere Arten von Werten enthalten, byte- oder satzweise in eine Datei geschrieben werden können. Der Inhalt solcher Dateien ist bei einer byteweisen Anzeige i.a. nicht lesbar, man spricht hier von **Binärformaten**. Durch Lesekommandos können die Dateiinhalte umgekehrt in Variablen eingelesen werden. Im Endeffekt kann eine Hauptspeicher-Datenstruktur durch Herausschreiben in eine Datei und späteres Wiedereinlesen komplett rekonstruiert werden. Das schreibende Programm und das lesende Programm müssen dabei vom exakt gleichen Dateiformat ausgehen.

Transportdateien zielen speziell auf den Fall, daß das schreibende und das lesende Programm

- auf verschiedenen Rechnern laufen, die möglicherweise andersartige Prozessoren und Datenformate haben,
- in verschiedenen Sprachen geschrieben sind, die unterschiedliche Konzepte zur Datenmodellierung bieten, und ggf.
- von verschiedenen, einander nicht kennenden Herstellern stammen.

Die wohl bekanntesten Transportdateien sind HTML-Dateien, die mit Hilfe unterschiedlicher Editoren erstellt und in diversen HTML-Browsern angezeigt werden. Transportdateien spielen beim e-Commerce eine ganz wesentliche Rolle, weil alle erdenklichen geschäftlichen Daten mit ihrer Hilfe zwischen Geschäftspartnern ausgetauscht werden.

Mit Transportdateien werden sowohl baumartig strukturierte Daten, die man meist als “Dokument” auffaßt, als auch tabellarische

Strukturen, also insb. Relationen, transportiert.

Transportdateien werden in allen möglichen Anwendungsgebieten benötigt; dementsprechend gibt es eine unüberschaubare Menge von Spezifikationen und teilweise sogar internationalen Standards (z.B. Edifact, EDIF, STEP/EXPRESS). Die grundlegenden Konzepte sind immer wieder gleich, die Annahmen über die Einsatzszenarien variieren teilweise, die technischen Details der zugehörigen Systeme sind oft sehr verschieden. Eine gewisse Vereinheitlichung der Technologien scheint inzwischen durch die XML einzutreten.

In diesem Lehrmodul skizzieren wir zunächst völlig unabhängig von der XML die grundlegenden Probleme und Herangehensweisen beim Transport von Daten. Wir werden dabei eine Reihe wichtiger Ähnlichkeiten zwischen Transportdateien und Datenbanken feststellen. Erst anschließend führen wir die XML ein, wobei wir uns auf die wichtigsten Konstrukte beschränken.

1.1 Formatkonversionen

Wir befassen uns hier nicht weiter mit Mechanismen, die eine Datei von einem Rechner auf einen anderen kopieren, sondern setzen diese i.f. voraus[1]. Binärformate (wie oben beschrieben) sind für den Datenaustausch zwischen Rechnern nicht brauchbar, denn die beteiligten Rechner können unterschiedliche Zahlendarstellungen (z.B. 32 vs. 64 Bit) oder unterschiedliche Zeichencodierungen verwenden.

Eine Alternative zu binären Formaten sind "lesbare" Formate, bei denen z.B. ein Zahlwert als Text mit Dezimalziffern und Komma dargestellt wird. Binäre Datenwerte im Hauptspeicher müssen nunmehr in diese textuelle Darstellung und zurück konvertiert werden.

Beim Austausch von Texten zwischen verschiedenen Rechnerplattformen bleibt das Problem unterschiedlicher Zeichensatzcodierungen zu lösen. Wegen der vielen nationalen Zeichensätze, Sonderzeichen

[1] Stattdessen können wir auch annehmen, daß das lesende und das schreibende Programm parallel laufen und über eine Kommunikationsverbindung wie z.B. Sockets einen Dateiinhalt austauschen, ohne daß überhaupt eine Datei benutzt würde.

und unterschiedlichen Codierungsmethoden ist dieses Thema unerwartet knifflig und Gegenstand mehrerer internationaler Standards. Bei sehr großen Zeichensätzen (z.B. Kanji) müssen mehrere Bytes zur Codierung eines Zeichens benutzt werden.

Sofern nun die beteiligten Systeme die gleiche Zeichencodierung und kompatible Konversionsroutinen benutzen, ist das Transportproblem gelöst[2], wenn auch nur auf einer elementaren syntaktischen Ebene.

Nicht gelöst bleibt das Problem, wie die verschiedenen Programme inhaltlich aufeinander abgestimmt werden können. Nehmen wir an, wir hätten mehrere Programme, mit denen Adreßbücher verwaltet werden können und zwischen denen Adressen ausgetauscht werden sollen, und daß jedes Programm sein eigenes, proprietäres Speicherungsformat benutzt. So könnten die Felder einer Adresse verschieden angeordnet sein, es könnte eine Sortierung vorausgesetzt sein oder nicht, es könnten diverse Hilfsdaten mit abgespeichert sein. Wir benötigen dann bei n Programmen n*(n-1) Konverter, um Daten zwischen allen Werkzeugen austauschen zu können (s. Bild 1).

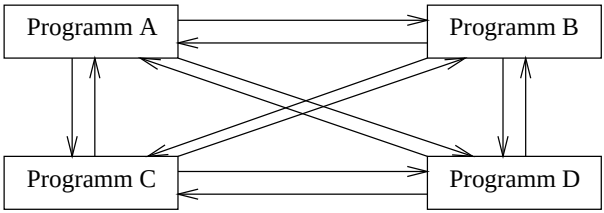

Abbildung 1: Paarweise Konverter zwischen Applikationen

[2]Daß eine lesbare Darstellung benutzt wird, veranschaulicht die Idee, ist aber eigentlich sekundär. Entscheidend sind die kompatiblen Konversionsroutinen auf beiden Seiten. Die XDR- (External Data Representation) Technologie, die für die Übertragung von Daten zwischen verteilten Anwendungen gedacht ist, aber auch als Dateiformat benutzt werden kann, setzt z.B. binäre Darstellungen ein. Der CDIF-Standard [Im91] sieht vor, daß zwischen lesbaren und binären Darstellungen (*encodings*) gewählt werden kann. Eine lesbare Darstellung hat nichtsdestotrotz den Vorteil, daß sie mit normalen Texteditoren lesbar und modifizierbar ist.

Bei einer größeren Zahl von Applikationen, die bei einem häufiger auftretender Dokumenttyp wie z.B. Adreßlisten, Terminkalender oder UML-Diagrammen leicht auftreten kann, ist der Realisierungsaufwand für die vielen Konverter untragbar. Lösbar ist dieses Problem, indem sich alle Applikationsanbieter auf ein gemeinsames *Austauschformat* einigen (s. Bild 2). Jede einzelne Applikation braucht jetzt nur noch einen Konverter zwischen diesem Austauschformat und seinem proprietären Speicherungsformat zu realisieren.

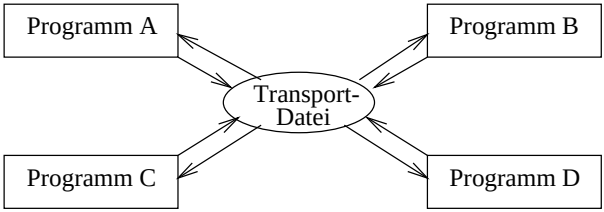

Abbildung 2: Sternförmige Konversion zwischen Applikationen mit Hilfe eines einheitlichen Formats für Transportdateien

Die einzelnen Programme werden häufig kleinere inhaltliche Abweichungen von der Durchschnittsfunktionalität haben. Als Beispiel nehmen wir an, daß der Benutzer beim Adreßverwaltungsprogramm A jeder Adresse eine Farbe zuordnen kann, in der die Adresse am Bildschirm dargestellt wird, während Adreßverwaltungsprogramm B dieses Merkmal nicht aufweist. Werden Daten von System A nach System B übernommen und dementsprechend konvertiert, geht in System B die Farbinformation verloren. Dies ist ein Beispiel dafür, daß die Konverter manchmal nicht ganz "verlustfrei" arbeiten und daß bei einem Transport von A nach B und wieder zurück nach A nicht mehr der ursprüngliche Zustand in A wiederhergestellt wird. Dies Problem läßt sich nur lösen, indem sich die verschiedenen Anbieter der Programme auch inhaltlich auf einen gemeinsamen Nenner einigen; dies ist kein technisches, sondern ein politisches oder wirtschaftliches Problem und soll hier nicht weiter vertieft werden.

1.2 Generische Parser

Wir erweitern unser Szenario jetzt um die Annahme, daß nicht nur ein einziger Dokumenttyp ausgetauscht werden soll, sondern viele verschiedene Dokumenttypen (wozu auch inhaltlich verschiedene Varianten des "gleichen" Dokumenttyps zählen). Für jeden Dokumenttyp benötigen wir einen passenden Scanner und Parser.

> **Exkurs über Compilerbau**: Für die Leser, die noch keine Vorlesung über Compilerbau besucht haben, seien diese Begriffe kurz erklärt: Ein **Scanner** übernimmt die erste Phase der Übersetzung eines Programms, die lexikalische Analyse; diese besteht darin, den Programmtext in eine Folge von sogenannten Token zu überführen. Man kann dies in etwa mit der Zerlegung eines Textes in Worte und Satzzeichen vergleichen; die einzelnen Worte können u.a. durch die dazwischenstehenden Leerzeichen erkannt werden.
>
> Ein **Parser** übernimmt die zweite Phase der Übersetzung, die syntaktische Analyse; Ergebnis dieser Phase ist der **Syntaxbaum**, der angibt, durch welche Produktionsschritte der vorliegende Text aus dem Startsymbol der Grammatik[3] konstruiert werden kann. Sofern lexikalische oder syntaktische Fehler auftreten, werden diese gemeldet, und die Übersetzung wird abgebrochen.
>
> Auf die syntaktische Analyse folgen noch die semantische Analyse, die Optimierung und die Codegenerierung, die uns hier nicht interessieren.
>
> Ein **Unparser** ist ein Programmteil, der einen Syntaxbaum wieder in eine textuelle Darstellung konvertiert.

Wir hatten bisher unterstellt, daß die Parser[4], die den Inhalt einer Transportdatei in einen Syntaxbaum im Hauptspeicher konvertieren, und die zugehörigen Unparser jeweils auf das Format der Transport-

[3] Die Leser dieses Textes sollten im Prinzip schon den Begriff Grammatik aus früheren Lehrveranstaltungen kennen; falls nicht, sollte zunächst Anhang A, der eine kurze Einführung in diesen Begriff gibt, gelesen werden.

[4] Die zugehörige Scanner erwähnen wir i.f. nicht immer extra.

datei zugeschnitten sind und dieses sozusagen hart verdrahtet in sich realisiert haben. Eine Alternative hierzu besteht darin, einen generischen Parser (bzw. Unparser) zu verwenden, der durch eine Grammatik gesteuert wird (s. Bild 3) und der diese Grammatik sozusagen interpretiert. Statt vieler einzelner Parser wird nun nur noch ein einziger generischer benötigt.

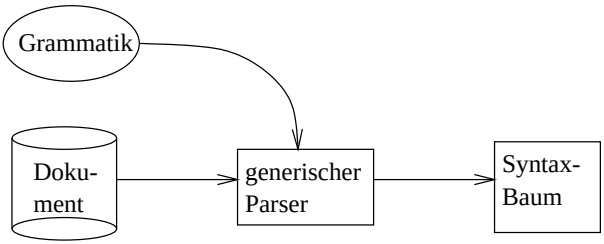

Abbildung 3: Generischer Parser

Woher die Grammatik kommt, ist in Bild 3 bewußt offengelassen worden. Möglichkeiten sind:

1. Die Grammatik ist als “Konstante” fest in das Programm eincompiliert. Eine Änderung der Grammatik macht eine Neucompilierung des Programms notwendig.
2. Die Grammatik ist in einer lokalen Ressourcen-Datei in einem geeigneten Format gespeichert. Sie wird beim Programmstart durch einen Parser in eine Hauptspeicherdarstellung konvertiert, also i.w. die “Konstante” aus der ersten Alternative. Eine Änderung der Grammatik macht nur noch einen Neustart des Programms notwendig.
3. Die Grammatik ist in der Transportdatei enthalten, z.B. im vorderen Teil der Datei vor dem eigentlich interessierenden Inhalt, dem übertragenen Dokument (s. Bild 4).

   Wenn man viele Dokumente des gleichen Typs verwendet, wird man lieber die Grammatik nur einmal an einer definierten Stelle speichern und in der Transportdatei statt der kompletten Grammatik nur noch eine Referenz auf diese Stelle vorsehen, z.B. in Form

eines URL. Diese Referenz kann man auch als include-Anweisung interpretieren.

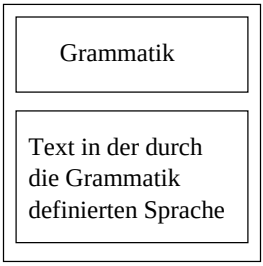

Abbildung 4: Denkbare Anordnung der Grammatik in einer Transportdatei

Statt von einer Grammatik spricht man bei Transportdateien oft von einer **Dokumenttypdefinition**.

In allen drei Fällen muß natürlich dafür gesorgt werden, daß in einem Kontext, in dem mit mehreren Dokumenttypen gleichzeitig gearbeitet wird, jeder Transportdatei die passende Grammatik zugeordnet wird. Am einfachsten wird hierzu im Anfangsteil der Transportdatei vermerkt, welche Version welcher Grammatik unterstellt wird. Dies kann man als reinen Konsistenztest interpretieren - wenn die im Programm verwendete Grammatik nicht kompatibel mit der in der Transportdatei unterstellten ist, wird abgebrochen - oder aber als Anweisung, die angegebene Grammatik zu beschaffen und zu verwenden, was auf die dritte oben genannte Alternative hinausläuft.

1.3 Meta-Sprachen

Wir nehmen i.f. an, daß die Grammatik direkt in der Transportdatei enthalten ist oder durch eine Anweisung eingefügt wird. Es stellt sich nun das Problem, daß auch die Grammatik in irgendeiner Form notiert werden muß und daß man einen Parser benötigt, der den Text, der die Grammatik repräsentiert, in eine interne Darstellung konvertiert. Wir benötigen also eine *Sprache, insb. eine Syntax, zur Notation*

von Grammatiken. Ein sehr bekanntes Beispiel hierfür ist die Backus-Naur-Form (vgl. auch Anhang A): mit ihrer Hilfe wird die Syntax von (Programmier-) Sprachen aufgeschrieben.

Als Beispiel betrachten wir eine sehr einfache Sprache, in der Adreßlisten aufgeschrieben werden können. Jede Adresse steht in einer Zeile, die einzelnen Angaben werden durch ein Komma und ein Leerzeichen getrennt. Bild 5 zeigt ein Beispiel für eine (kurze) Adreßliste.

```
Schmitz, Rudolf, Landstr.  142, 50123, Köln
Müller, Lisa, Talweg 121, 53132, Bonn
Grünspecht, Gunter, Feldweg 1, 57489, Drolshagen
```

Abbildung 5: Beispiel für Nutzdaten

Bild 6 zeigt die zugehörige Grammatik notiert in der BNF. Auf die Definition eines `Zeichen` und die Behandlung der Zeilenenden verzichten wir aus Platzgründen.

```
Adreßliste ::= Adresse*
Adresse ::= Name ', ' Vorname ', ' Straße ', '
            Hausnummer ', ' PLZ ', ' Ort
Name ::= Zeichen*
Straße ::= Zeichen*
Hausnummer ::= Ziffer*
PLZ ::= Ziffer*
Ort ::= Zeichen*
Ziffer ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

Abbildung 6: Beispiel für eine Grammatik

Genaugenommen enthält Bild 6 einen Text, den wir als Grammatik für die Nutzdaten in Bild 5 interpretieren können. Dieser Text hat eine Syntax und eine Semantik, die durch die BNF festgelegt ist. Jeder

korrekte Text in der BNF stellt also eine Grammatik und damit einen wesentlichen Teil einer Sprache dar (die Semantik wird offensichtlich nicht dargestellt). Leicht übertreibend ist die BNF also eine *Sprache, in der Sprachen notiert werden können*; solche Sprachen nennt man auch **Meta-Sprachen**. Weitere Beispiele für Meta-Sprachen sind die Extensible Markup Language (XML) [XML00] und die Standard Generalized Markup Language (SGML) [ISO8879-1986]: mit ihrer Hilfe kann man die Syntax von sogenannten *tag*-Sprachen aufschreiben.

Eine Meta-Sprache hat offensichtlich auch eine Syntax; diese können wir zumindest teilweise ebenfalls in BNF darstellen; einen Auszug aus einer möglichen Definition zeigt Bild 7.

```
Grammatik ::= Produktion*
Produktion ::= Nonterminal ’::=’ Ausdruck
Ausdruck ::= ’(’ Ausdruck ’)’ | Ausdruck ’*’ |
             Ausdruck ’+’ | Ausdruck ’?’  | Sequenz
             | Alternative | Terminal
Sequenz ::= Ausdruck ’ ’ Ausdruck
Alternative ::= Ausdruck ’|’ Ausdruck
....
```

Abbildung 7: Teile der Syntax der Meta-Sprache BNF

Die Texte in den Bildern 5, 6 und 7 liegen offensichtlich in einer Hierarchie von Sprachebenen jeweils eine Ebene höher (vgl. Tabelle in Bild 8): Die unterste Ebene enthält die Nutzdaten, die erste Ebene Grammatiken für Nutzdaten, die zweite Grammatiken für Grammatiken für Nutzdaten; theoretisch kann man dies noch weiterführen.

Die Texte in Ebene 1 und höher stellen alle Grammatiken dar; es sollte aber klar sein, daß es sich auf jeder Ebene um eine ganz eigene Kategorie von Sprachen handelt, die sich vor allem hinsichtlich ihrer Semantik – die in den Texten gar nicht dargestellt wird – grundlegend unterscheiden[5]. Das Erkennen dieses Unterschieds wird dadurch er-

[5] Auch die Grammatiken werden nur unvollständig dargestellt, es fehlen z.B. bei

Ebene	Text repräsentiert ...	Syntax und Semantik des Textes ist bestimmt durch ...	Beispiel in Bild
2	Grammatik der Meta-Sprache	Meta-Meta-Sprache	7
1	Grammatik der Nutzdatensprache	Meta-Sprache	6
0	Fakten / Nutzdaten	Nutzdatensprache	5

Abbildung 8: Sprachebenen

schwert, daß die grundlegenden syntaktischen Strukturen (d.h. die kontextfreie Syntax) der Meta-Sprache und der höheren Sprachen völlig gleich gestaltet sein können.

Bei Transportdateien werden oft auch die grundlegenden syntaktischen Strukturen der Grammatik der Nutzdatensprache und der Meta-Sprache identisch (odersehr ähnlich) gestaltet. Dies hat den Vorteil, daß man bei dem Parser, der die Grammatik in der Transportdatei einliest, Implementierungsaufwand einspart. Diese syntaktische Ähnlichkeit darf aber auch hier *nicht* zum Fehlschluß führen, daß es sich um eine einzige Sprache handele oder daß die Meta-Sprache nur eine "Erweiterung" der Dokumentbeschreibungssprache sei[6]. Die Meta-Sprache liegt sozusagen in einer anderen Dimension. Im Gegensatz zu den mit ihr definierten Dokumentbeschreibungssprachen ist sie nicht veränderbar, ihre vollständige Syntax und ihre Semantik sind in dem generischen Parser hart verdrahtet.

der BNF alle kontextsensitiven Anteile.

[6]Dieser Irrtum liegt der öfter gehörten Behauptung zugrunde, die XML sei eine Erweiterung von HTML.

1.4 Transportdateien im Vergleich mit Datenbanken

1.4.1 Grammatiken als Schemata

Es fallen einige verblüffende Ähnlichkeiten zwischen Transportdateien und Datenbanken ins Auge; die wichtigste besteht darin, daß bestimmte Teile der Grammatik dem konzeptuellen Schema einer Datenbank entsprechen und der Inhalt einer Transportdatei in etwa einer Datenbank entspricht.

An dieser Stelle muß man sich zunächst vergegenwärtigen, daß die bei Datenbanken übliche Trennung zwischen konzeptionellen und physischen Aspekten der Datenverwaltung bei Grammatiken nicht vorhanden und auch nicht möglich ist. Eine Transportdatei entspricht einer Datenbank gesehen auf der *physischen* Ebene. Bei Datenbanken ist diese Ebene für Normalbenutzer nicht einsehbar, das DBMS verkapselt diese Darstellung der Datenbank und macht die Inhalte nur auf einer viel höheren Ebene, nämlich in der Denkwelt des Datenbankmodells, zugreifbar. Während in Grammatiken Zahlen- und Zeichendarstellungen, Klammerstrukturen, Schlüsselworte zur Strukturdarstellung etc. notwendigerweise behandelt werden müssen, entfallen diese Aspekte bei der Definition von Datenbankschemata weitgehend.

Wenn wir diese eher lexikalischen bzw. physischen Aspekte außer Acht lassen, dann kann man jede kontextfreie Produktion der Grammatik als Aggregation interpretieren, die festlegt, aus welchen Teilen sich eine komplexe Entität zusammensetzt. Diese Aggregationsstruktur wird ja auch im Syntaxbaum dargestellt.

In diesem Sinne kann man jede kontextfreie Grammatik in ein ER- oder OOA/OOD-Modell umformen, wobei jedes Nichtterminalsymbol einer Klasse entspricht und wobei für eine Produktion der Form

```
A ::= B C
```

Teil-von-Beziehungen definiert werden müssen, die die Klassen `B` und `C` zu Komponenten der Klasse `A` machen. Mit diesem Schema würde ein korrekter Text gemäß der Grammatik in der Datenbank durch seinen Syntaxbaum repräsentiert werden.

Umgekehrt kann man aus den Teil-von-Strukturen in einem ER-

oder OOA/OOD-Modell eine entsprechende Grammatik ableiten. Problematisch hierbei sind Typhierarchien. Wenn man die Grammatiken in einer Sprache ähnlich der BNF notiert, muß eine Klasse mit Unterklassen in der Grammatik als oder-Verknüpfung dieser Klasse (bzw. des entsprechenden Nichtterminalsymbols) und aller ihrer direkten und indirekten Unterklassen nachgebildet werden.

Weitere Merkmale, in denen sich Datenbanken und Transportdateien ähneln, sind:

- *Sichtenintegration:* Verschiedene Applikationen können nur dann Daten erfolgreich austauschen, wenn sie konzeptionell konsistent sind bzw. konsistent gemacht worden sind. Dieses Problem ist exakt das gleiche, wie wenn die Applikationen ihre Daten in einer Datenbank speichern würden; es wird im Datenbankkontext als Sichtenintegration bezeichnet. Die Vorgehensweise, insb. das Bilden eines konzeptuellen Modells mit Hilfe von ER- oder OOA-Modellen, ist bei Transportdateien völlig analog anwendbar.

- *Sichten:* Wenn der Inhalt einer Transportdatei beim Einlesen anhand gewisser Kriterien gefiltert wird, entsteht eine ähnliche Wirkung wie durch den Sichtenmechanismus eines Datenbanksystems.

- *Lesbarkeit:* Eine grundlegende Annahme beim Entwurf vieler Sprachen ist, daß die Texte für ein bestimmtes Leserpublikum gut lesbar sein sollen – die Grammatiken können deshalb sehr komplex werden – und mit jedem beliebigen Editor erstellbar sein sollen[7]. Inhalte von Datenbanken werden den Benutzern dagegen nicht auf der physischen Ebene präsentiert, sondern durch Applikationen in Formularen und anderen Darstellungen und auf dem gleichen Wege erzeugt bzw. modifiziert.

  Transportdateien enthalten zwar meist darstellbaren Text, aber er ist bei komplexen Dokumentstrukturen nicht wirklich gut lesbar.

[7] Dies schließt nicht aus, daß diese Texte in speziellen Editoren anders dargestellt werden, um sie besser lesbar zu machen, z.B. durch Färbung von Schlüsselworten oder andere Formatierung.

Ihr Inhalt wird deshalb auch nur selten mit normalen Zeicheneditoren erstellt oder modifiziert. Die Inhalte von Transportdateien werden normalerweise nur von Applikationen erzeugt und wieder gelesen – das war ja auch die ursprüngliche Motivation –, was aber typisch für Datenbanken ist.

1.4.2 Unterschiede

Den vorstehend aufgezählten Ähnlichkeiten stehen andere Aspekte gegenüber, bei denen sich Transportdateien wesentlich von Datenbanken unterscheiden:

- Transportdateien sind nur für vergleichsweise kleine Datenvolumina gedacht. Der Begriff "klein" ist relativ, entscheidend ist, daß die komplette Datei in einem Zeitraum, der im Anwendungskontext vertretbar ist, geladen werden kann und komplett in den Hauptspeicher paßt.
- Transportdateien sind sequentielle Dateien, ein direkter Zugriff zu einzelnen Datenelementen wird nicht unterstützt. Transportdateien können nicht punktuell geändert, sondern nur komplett neu geschrieben werden, sind also bei kleinen Änderungen sehr ineffizient.
- Transportdateien werden typischerweise komplett in einer Applikation verarbeitet, während Datenbankapplikationen oft nur kleine Teile der Datenbank lesen oder verändern. Mehrere Datenbankapplikationen können parallel in verschiedenen Teilen der Datenbank arbeiten, eine Transportdatei kann immer nur von einer einzigen Applikation benutzt werden.
- Feingranulare Zugriffskontrollen sind bei Transportdateien nicht möglich. Rechte können nur für ganze Transportdateien über das unterliegende Betriebssystem vergeben werden.
- Umgekehrt sind natürlich (physische) Datenbanken für den Austausch von Daten zwischen verschiedenen Rechnern nicht geeignet.

Offensichtlich sind also Transportdateien und Datenbanken trotz einiger Ähnlichkeiten in ganz verschiedenen Kontexten sinnvoll anwendbar.

2 Einführung in die SGML

2.1 Historie

Unter den Meta-Sprachen für Transportdateien ist die Extensible Markup Language (XML) die bekannteste und in Zukunft die wichtigste.

Die XML ist eine vereinfachte Variante der Standard Generalized Markup Language (SGML) [ISO8879-1986]. Die SGML ist seit Anfang der 80er Jahre durch die ISO standardisiert. Die SGML ist eine sehr komplexe Meta-Sprache und hat sich deshalb nicht weit verbreiten können. Die XML ist eine vereinfachte Version von SGML, bei der viele kompliziertere und selten benötigte (insb. im Kontext des WWW nicht benötigte) Merkmale weggelassen oder überarbeitet wurden und die leicht implementierbar ist. Die Spezifikation der SGML hat rund 500 Seiten Umfang, die der XML nur ca. 50. Verantwortlich für die XML-Spezifikationen ist das Word Wide Web Consortium (W3C).

Die SGML adressiert den Austausch von Daten, allerdings in einem speziellen Anwendungsgebiet, nämlich dem Austausch textueller Dokumente in Büroinformations- und ähnlichen Systemen (man beachte das "Text and Office Systems" im Namen des SGML-Standards); in etwa kann man auch das WWW hierzu zählen, da dieses, wenn man die multimedialen Anteile einmal außer acht läßt, ein verteiltes Hypertextsystem darstellt.

Viele grundlegende Merkmale von XML gehen direkt auf SGML zurück und sind nur verständlich, wenn man den bei SGML unterstellten Anwendungskontext in groben Zügen kennt. Dieser Anwendungskontext unterscheidet sich in einigen Punkten signifikant vom Anwendungskontext, der bei Datenbanken gegeben ist und den wir auch in Abschnitt 1 unterstellt haben. Trotz der schon erwähnten prinzipiellen Ähnlichkeiten zwischen Transportdateien wirken auf Datenbänkler daher manche Begriffe der SGML bzw. XML etwas seltsam.

Im folgenden Abschnitt geben wir daher zunächst eine kompakte Einführung in die grundlegenden Konzepte der SGML (eine ausführlichere, sehr gut lesbare Einführung ist [SGML]), die bis auf wenige Ausnahmen auch für XML gelten.

2.2 Strukturen in textuellen Dokumenten

Die Struktur von Texten ist wesentlich anders als bei Tabellen in relationalen Systemen oder hierarchisch aufgebauten Objekten in objektorientierten Datenbanken: Strukturmerkmale in Texten sind Hervorhebungen durch Fett- oder Kursivschrift, Fußnoten, Literaturverweise, Absätze, numerierte Kapitel und Abschnitte, eingelagerte Bilder oder Tabellen, der Seitenaufbau usw.

Notiert wurden derartige Strukturen früher durch sogenannte Textauszeichnungen (*markup*), die ursprünglich – handschriftlich auf einem Manuskript eingetragen – Formatierungsanweisungen für den Setzer waren. SGML und XML (aber auch HTML) definieren ebenfalls Textauszeichnungen und werden daher als **Markup-Sprachen** bezeichnet.

An dieser Stelle können wir bereits zwei Punkte notieren, in denen sich die Denkwelt von SGML ganz grundlegend von der Denkwelt von Datenbanken unterscheidet:

- Ein Text ist auch ohne Markup lesbar und durchaus sinnvoll verwendbar. In der Praxis ist es oft so, daß zunächst der Text ohne Markup existiert und das Markup erst später hinzugefügt wird.
- Ein Text kann aus verschiedenen Sichtweisen verschieden strukturiert sein, z.B. könnte man eine Kapitelstruktur und eine Seitenstruktur unterscheiden. Eine Strukturierung eines Textes wird als eine in gewissem Maße willkürliche Interpretation des Textes verstanden.

Wenn man das Markup als Darstellung der konzeptuellen Strukturen eines Textes ansieht, stehen die beiden ersten Punkte in krassem Gegensatz zur Denkwelt von Datenbanken, wonach es immer genau ein konzeptionelles Schema in einer Datenbank gibt und Nutzdaten erst nach Definition des Schemas eingetragen werden können.

Markup-Anweisungen können Bedeutungen haben, die in zwei ganz verschiedenen Bereichen liegen[8]:

[8] Beide Bereiche sind aber je nach Betrachtungsweise bei Texten nicht strikt trennbar.

- Layout (Formatierungsanweisungen): z.B. soll ein Textstück kursiv angezeigt werden
- inhaltliche Struktur: z.B. soll ein Textstück eine Fußnote, eine Randnotiz oder eine Bildunterschrift sein

Beim klassischen Markup gibt es eine feste Anzahl von Formatierungsanweisungen, die eine bestimmte Bedeutung haben. Die Formatierungsanweisungen beziehen sich sich primär auf das Layout und weniger auf die inhaltliche Struktur. Inhaltliche Strukturen werden oft durch Layout ausgedrückt. Ein Text, der z.B. eine Rechnung darstellt, könnte einzelne Rechnungsposten in je einer Zeile darstellen und die Rechnungssumme darunter in Fettschrift.

Auch für textuelle Dokumente gibt es das Austauschproblem, bzw. allgemeiner das Problem, solche Dokumente in Textprozessoren, Büroinformationssystemen usw. zu verarbeiten. Beim klassischen Markup entstehen hier einige Probleme:

- Vielfach will man die Struktur des Textes ausnutzen, z.B. beim Suchen innerhalb von Texten. Man kann die Struktur aber i.a. nicht aus der Formatierung ableiten.
- Vielfach will man den Text inhaltlich gleich, aber anders formatiert anzeigen, u.a. wegen verschiedener Hardware-Eigenschaften der Ausgabemedien (Bildschirm, Drucker).
- Auf unterschiedlichen Plattformen (Hardware, Betriebssystem) sind verschiedene Zeichensatzcodierungen vorhanden, hinzu kommen landesspezifische Zeichensätze.

2.3 Entitäten

Die Bearbeitung eines SGML- (oder XML-) Textes kann man sich grob in zwei Stufen gegliedert vorstellen, die der lexikalischen und syntaktischen Analyse entsprechen.

Ziel der lexikalischen Analyse ist es bekanntlich, einen Quelltext in eine Sequenz von Tokens umzuwandeln. Dieser Vorgang wird in SGML

durch das Konzept der **Entitäten** gesteuert[9]. Entitäten adressieren folgende Probleme:

- Für nicht direkt darstellbare Zeichen (z.B. aus Fremdsprachen) müssen Ersatzdarstellungen definiert werden können.
- Wiederholt auftretende Textteile sollen analog zu Makros nur einmal definiert und durch Aufruf wiederholt einsetzbar sein.
- Ein Gesamttext soll auf verschiedene Dateien verteilbar sein, die an den gewünschten Stellen eingefügt werden.
- Es soll möglich sein, aus einem einzigen Quelltext verschiedene Ausgabevarianten zu generieren (z.B. Preislisten für verschiedene Länder mit unterschiedlichen Steuersystemen und Steuersätzen).

Die meisten der vorstehenden Probleme unterstellen offensichtlich, daß die SGML-Texte weitgehend von Hand ediert werden. Technisch gelöst werden die vorstehenden Probleme wie nachfolgend beschrieben. Deklariert werden Entitäten in einer der beiden folgenden Formen:

```
<!ENTITY entityname "ersetzungzeichenkette">
<!ENTITY entityname SYSTEM "Dateiname">
```

Aufgerufen werden Entitäten in der Form `&entityname;`. Bei der ersten Variante wird der Aufruf durch den angegebenen Text ersetzt, bei der zweiten Variante durch den Inhalt der Datei mit dem angegebenen Namen.

Entitäten in der Dokumenttypdefinition und im eigentlichen Dokument werden verschieden gehandhabt. Hierauf und auf andere Details gehen wir hier nicht ein.

[9] SGML-Entitäten sind konzeptionell etwas völlig anderes als Entitäten im ER-Ansatz und sind eher mit Makros vergleichbar. Diese "falsche" Benutzung der Bezeichnung Entität ist für Personen, die mit der Begriffswelt der Softwaretechnik vertraut sind, sehr störend; vermutlich waren solche Personen an der Definition des Standards nicht beteiligt.

2.4 *Tags*

Das Markup besteht bei SGML und XML aus *tags*[10]. Syntaktisch hat ein *tag*, das zur Darstellung der Dokumentstruktur benutzt wird, eine der beiden folgenden Formen:

```
<tagname attributliste>
</tagname>
```

Die erste Form gilt für öffnende, die zweite für schließende *tags*. *Tags* kann man als benannte Klammern auffassen, die analog zur korrekten Klammerung arithmetischer Ausdrücke immer nur passend paarweise auftreten dürfen und auf diese Weise eine Baumstruktur repräsentieren. Solche Texte heißen **wohlgeformt**.

In SGML kann man zulassen, daß einzelne *tags* weggelassen werden dürfen[11]. Motiviert wird derartiger "syntaktischer Zucker" durch die Annahme, die Tags müßten mühsam von Hand eingetragen werden.

In XML dürfen keine *tags* fehlen, selbst dann, wenn der dazwischenstehende Inhalt leer ist; in diesem Fall ist aber die abkürzende Schreibweise `<tagname attributliste />` erlaubt.

2.5 Elemente

Ein Paar zusammengehöriger *tags* und der dazwischenstehende Inhalt bilden ein **Element**.

Elementtypen werden in einer **Dokumenttypdefinition** definiert. Diese legt zu jedem Dokumenttypen fest:

- eine Menge von Attributen; diese werden bei Instanzen des Elementtyps im öffnenden *tag* notiert.
- die zulässigen inneren Elemente sowie ggf. deren Anzahl und Anordnung. Da Texte neben einer baumartigen Grundstruktur oft

[10] Eine passende deutsche Übersetzung von *tag* ist eigentlich Etikett. Wir verwenden hier dennoch den englischen Ausdruck.

[11] Dies gilt auch in HTML. Beispielsweise wird ein Paragraph mit `<p> .... </p>` geklammert; das schließende `</p>` kann aber weggelassen werden, weil mit Beginn des nächsten Paragraphen implizit der vorherige beendet wird.

Bestandteile haben, die auf beliebigen Hierarchiestufen auftreten oder in bestimmten Bereichen wiederum nicht auftreten können, kann man diverse Ausnahmen von der baumartigen Grundstruktur definieren.

Elementtypen entsprechen den Nichtterminalsymbolen einer Grammatik wie in Abschnitt 1.4.1 erläutert.

In der SGML (nicht hingegen in der XML) kann man mehrere parallel vorhandene Grundstrukturen definieren.

Während HTML und ähnliche *tag*-Sprachen einen fest definierten Satz von *tags* haben und die *tags* eine bestimmte Bedeutung (in bezug auf das Layout) haben, können in SGML und XML beliebige *tags* definiert werden; diese Tags haben über die Schachtelung der enthaltenen Elemente hinaus keinerlei Bedeutung, d.h. für ein Dokument, das in einer XML-Datei enthalten ist, ist keine externe Darstellung vorgegeben. Dies führt zu der generellen Fragestellung, wie aus dem Inhalt einer Transportdatei eine externe Darstellung des Dokuments gewonnen werden kann.

3 Visualisierung strukturierter Daten

3.1 Grundsätzliche Alternativen

An dieser Stelle ist es sinnvoll, generelle Ansätze zur Visualisierung strukturierter Daten zu identifizieren.

API vs. Seitenbeschreibung. Zunächst einmal müssen wir voraussetzen, daß Betriebssystem oder ein Graphikpaket Ausgabefunktionen zur Verfügung stellt, durch die z.B. eine Ausgabefläche auf dem Bildschirm, die wir i.f. als *Seite* bezeichnen, in Bereiche strukturiert werden kann, durch die Texte oder graphische Elemente an bestimmten Stellen angezeigt werden können und durch die ferner Eingaben entgegengenommen werden können. Hier können zwei Ansätze unterschieden werden:

- Der angezeigte Seiteninhalt wird *schrittweise (über ein API) konstruiert*; für jeden einzelnen Schritt, z.B. das Zeichnen eines Rahmens oder Anzeigen eines Stück Textes, ist eine Funktion verfügbar (in einer Bibliothek), die von der Applikation *einzeln* mit entsprechenden Argumenten aufgerufen wird.
- Die Applikation konstruiert eine *Seitenbeschreibung*, also einen Text, der den anzuzeigenden Seiteninhalt in einer hierzu geeigneten Sprache beschreibt[12]. Eine Seitenbeschreibung wird als ganze an das unterliegende Basissystem übergeben und dort von einem Interpreter abgearbeitet, der dann die Ausgabe erzeugt. Eine Seitenbeschreibung kann daher auch als "Anzeigeprogramm" angesehen werden. HTML ist ein Beispiel für eine Sprache, in der man solche Anzeigeprogramme formulieren kann.

Die zweite Alternative hat den Nachteil, daß man einen zusätzlichen Interpreter für die Seitenbeschreibungssprache braucht bzw. von diesem abhängig wird und Fehler in der Seitenbeschreibung schlechter erkannt bzw. durch Tests gefunden werden können. Ihr großer Vorteil ist aber, daß nur *eine einzige Kommunikation* zwischen Anwendung und Anzeigesystem zur Anzeige einer kompletten Seite erforderlich ist; dieser Vorteil ist in verteilten Systemen ausschlaggebend.

Abstraktionsebene. Wenn nun strukturierte Daten visualisiert werden sollen, muß eine Applikation unter Benutzung der verfügbaren Anzeigetechnologie eine Darstellung der Daten auf dem Bildschirm oder auf Papier prozedural konstruieren. Dies gilt unabhängig davon, welche der beiden vorstehenden Alternativen vorliegt.

Viele Applikationen sind an die Plattform, worunter wir hier das Betriebssystem, ggf. separate Graphikpakete und relevante Hardware-Eigenschaften verstehen, gebunden und nicht auf andere Plattformen portabel, weil eine andere Darstellung auf dem Ausgabemedium und

[12] Begrifflich ist dies eng verwandt mit Seitenbeschreibungensprachen für Papierausgaben wie PostScript, die von Druckern interpretiert werden. Die Unterschiede liegen in den anderen technischen Randbedingungen und dem Fehlen jeglicher interaktiver Elemente auf Papier.

andere Interaktionsformen nötig sein können. In heterogenen Umgebungen ist diese fehlende Portabilität äußerst störend. Der naheliegende Ansatz besteht darin, von den Besonderheiten der einzelnen Plattformen zu abstrahieren und eine "abstrakte Ausgabeplattform" als Zwischenebene einzuziehen; eine Dokumentdarstellung auf dieser Zwischenebene soll vollautomatisch in Darstellungen in den konkreten Plattformen umsetzbar sein.

HTML und die zugehörigen WWW-Browser kann man als eine derartige abstrakte Ausgabeplattform ansehen; wegen der Verteilung wird hier der o.g. Ansatz eines Anzeigeprogramms verfolgt.

Die erzeugte Anzeige wird mittels entsprechender **Layout-Daten** gesteuert. Beispiele für Layout-Daten sind die diversen *tags* in HTML. Tags sind in den anzuzeigenden Text eingestreut und erkennbar durch eine öffnende und schließende spitze Klammer. Beispielsweise bedeutet in einer HTML-Seite der Text "`Kontostand <b> 1200.00 </b> DM`", daß "Kontostand **1200.00** DM" angezeigt werden soll; `<b>` schaltet die Fettschrift ein, `</b>` schaltet sie wieder aus. Ein anderes Beispiel ist "`...<h2> 1.1 Einführung </h2>...`"; hier soll der Text "1.1 Einführung" als Überschrift zweiter Ordnung angezeigt werden.

Extraktion von "Inhalten" aus HTML-Texten. Bei vielen Gelegenheiten würde man gerne die in einem HTML-Text übermittelten strukturierten Daten auch anderweitig verwenden. Man kann diese Daten jedoch aus einem HTML-Text i.a. nicht mehr oder nur unsicher extrahieren. Zunächst enthält der HTML-Text keine Hinweise darauf, daß er überhaupt strukturierte Daten enthält und, falls ja, welche. Selbst dann, wenn man von einem HTML-Text weiß, daß darin irgendwo ein Kontostand steht und fett gedruckt ist, ist die Suche danach schwierig und fehleranfällig. Selbst Überschriften-Tags wie `<h2> ... </h2>` garantieren nicht, daß durch sie tatsächlich die Schachtelungsstruktur der Abschnitte dargestellt wird, diese Tags könnten auch für andere Formatierungsaufgaben ge- (oder miß-) braucht werden.

3.2 Layout-Spezifikationen

Der grundlegende Ansatz, (strukturierte) "Inhalte" erkennbar zu machen, besteht darin, *Inhalt und Darstellung strikt zu trennen*:

- Der "Inhalt" wird in strukturierter Form wie oben bei Transportdateien beschrieben übertragen, also insb. durch eine Grammatik und einen dazu passenden Text. Bei der XML spricht man nicht von einer Grammatik, sondern von einer **Dokumenttypdefinition** (**DTD**) und vom in der Transportdatei enthaltenen Dokument. Die Nichtterminalsymbole der Grammatik, die ja den Strukturelementen des Dokuments entsprechen, werden Elementtypen genannt; sie entsprechen in der Denkwelt von Datenbanken den Entitätstypen.
- Um das Dokument nun anzeigen zu können, muß der Inhalt der Transportdatei in einen Text mit Darstellungsanweisungen übersetzt werden. Ein entsprechender Übersetzer baut zunächst mit Hilfe eines Parsers einen Syntaxbaum auf und erzeugt aus diesem anhand von **Darstellungsregeln** einen neuen Text (s. Bild 9).

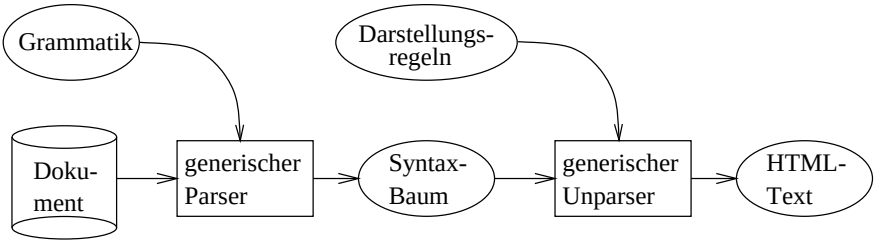

Abbildung 9: Erzeugung von HTML-Text

Die Übersetzung läuft i.w. so ab, daß der Syntaxbaum von der Wurzel aus durchlaufen wird und abhängig vom jeweils vorgefundenen Nichtterminalsymbol eine dazu passende Darstellungsregel ausgenutzt wird. Formuliert werden können die Darstellungsregeln in der **Extensible Stylesheet Language** (**XSL**).

Ob hierbei HTML-Text, LaTeX-Quellcode oder anderes erzeugt

wird, hängt einzig von den Darstellungsregeln ab.

Im WWW stellt sich noch die Frage, ob die in Bild 9 gezeigten Transformationen serverseitig oder im WWW-Browser stattfinden sollen, worauf wir hier nicht weiter eingehen wollen.

A Grammatiken

Eine **Grammatik** wird definiert durch

1. eine Menge von Terminalsymbolen
2. eine Menge von Nichtterminalsymbolen
3. eine Menge von Grammatikregeln (oder Produktionen)
4. ein Startsymbol

Eine Grammatik spezifiziert letztlich eine Menge von Texten (oder Worten oder Sätzen); jeder einzelne Text ist eine Folge von Terminalsymbolen, die aus dem Startsymbol durch Anwendung von Grammatikregeln "produziert" werden kann. Solche Texte nennen wir korrekt bzgl. der Grammatik.

Terminalsymbole können einzelne Zeichen oder Zeichenfolgen sein. Beispielsweise kann das deutsche, französische oder japanische Alphabet als Menge von Terminalsymbole sinnvoll sein (jeweils incl. aller Varianten bzgl. Groß-/Kleinschreibung und Akzenten). In Programmiersprachen werden die reservierten Worte (wie `do`, `while`, `return` usw.) als einzelne Terminalsymbole betrachtet.

Nichtterminalsymbole sind einzelne Zeichen oder Zeichenfolgen, die normalerweise bestimmte Teile eines korrekten Textes repräsentieren. Das Startsymbol ist auch ein Nichtterminalsymbol; aus ihm kann man alle korrekten Texte ableiten. Beispiele für Nichtterminalsymbole in Programmiersprachen sind *Programm*, *Typdeklaration*, *Anweisung*, *arithmetischer Ausdruck* usw. Um Nichtterminalsymbole von Terminalsymbolen unterscheiden zu können, notieren wir Terminalsymbole in Apostrophen.

Grammatikregeln (bzw. **Produktionen**) legen fest, wie aus einem Text, der noch Nichtterminalsymbole enthält, der also noch nicht "entwickelt" ist, ein anderer Text abgeleitet, also produziert werden kann. Notiert werden Grammatikregeln häufig in der BNF (Backus-Naur-Form) oder einer Erweiterung derselben. Wir verwenden hier die erweiterte BNF, die auch in der XML-Definition [XML00] verwendet wird. Eine Grammatikregel wird notiert in der Form

```
muster ::= ausdruck
```

`muster` ist eine Sequenz von Terminal- und Nichtterminalsymbolen, in der wenigstens ein Nichtterminalsymbol enthalten ist. Wenn das `muster` in einem Text auftritt, ist die Regel anwendbar; wird sie angewandt (i.a. können mehrere Regeln gleichzeitig anwendbar sein), so wird das `muster` im Text ersetzt durch den `ausdruck`.

In der XML-Definition besteht das `muster` immer nur aus einem einzigen Nichtterminalsymbol; solche Grammatiken nennt man auch **kontextfrei**, weil die Nichtterminalsymbole unabhängig von ihrem Kontext ersetzt werden. Allerdings werden in der XML-Definition in umgangssprachlicher Form Restriktionen für korrekt geformte Texte angegeben, infolgedessen sind die Grammatiken nicht kontextfrei.

Ein Beispiel für eine Produktion ist:

```
whileStatement ::= 'while' '(' Bedingung ')' Statement
```

Das Nichtterminalsymbol `whileStatement` wird gemäß dieser Regel durch die angegebene Folge von 5 Symbolen ersetzt. Die Hintereinanderschreibung der Symbole in der Produktion bedeutet, daß die Symbole in dieser Reihenfolge konkateniert werden müssen (wobei üblicherweise zwischen den Symbolen beliebig viele Leerzeichen und Zeilenvorschübe eingefügt werden dürfen). Neben der Konkatenation sind folgende Operatoren verfügbar (`A` und `B` sind beliebige Folgen von Symbolen):

`(A)`	wird durch `A` ersetzt; die Klammerung können dazu eingesetzt werden, den Wirkungsbereich der anderen Operatoren zu definieren
`A?`	kann durch `A` oder nichts ersetzt werden
`A | B`	kann durch `A` oder `B` ersetzt werden
`A+`	kann durch ein oder mehrere Auftreten von `A` ersetzt werden
`A*`	kann durch nichts oder ein oder mehrere Auftreten von `A` ersetzt werden (m.a.W. `A*` = `(A+)?`)

Literatur

[SGML] A Gentle Introduction to SGML; `www.lib.virginia.edu/etext/sgml.html`; [kein Publikationsjahr bekannt]

[Im91] Imber, M.: CASE Data interchange format standards; Information and Software Technology 33:9, p.647-655; 1991/11

[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

[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

Index