3 Logische Dokumentstruktur

In diesem Kapitel befassen wir uns genauer mit der (generischen und spezifischen) logischen Struktur von Dokumenten. In der Einführung wurde dieses Thema bereits angeschnitten. Außerdem werden wir die Umsetzung von logischen Strukturen in graphische behandeln. All dies geschieht erst von einem allgemeinen Standpunkt aus. Danach werden wir die Realisierung der logischen Struktur in verschiedenen konkreten dokumentverarbeitenden Systemen besprechen.

3.1 Elemente der logischen Dokumentstruktur

Die logische Struktur eines Dokuments bezieht sich auf die Bedeutung seiner Bestandteile oder auf die Absichten der Autoren. Allgemein kann man zwischen hierarchischer und nichthierarchischer logischer Struktur unterscheiden.

3.1.1 Hierarchische logische Struktur

In der hierarchischen logischen Struktur wird ein Dokument in verschiedene Komponenten aufgeteilt, die ihrerseits wieder aus Unterkomponenten bestehen usw. Diese Aufteilung induziert einen Baum, den *logischen Strukturbaum*. Die Unterkomponenten einer Komponente können geordnet oder ungeordnet sein.

Ein Teil der generischen logischen Struktur eines Buches könnte z. B. so beschrieben sein:

> Ein Buch besteht aus einem Titel, einem Vorwort, einem Hauptteil und einer Bibliographie.
>
> Der Hauptteil besteht aus einer Folge von Kapiteln.

Die Worte „Buch“, „Titel“ usw. bezeichnen *Typen* von logischen Komponenten. Die Definition der generischen logischen Struktur führt also eine Menge von Komponententypen ein und definiert ihren inneren Aufbau aus anderen Komponententypen (Relation „besteht aus“).

Eine generische logische Struktur entspricht etwa den Typdefinitionen in Programmiersprachen, während die Bestandteile einer spezifischen logischen Struktur den Objekten aus diesen Typen entsprechen. Zu einem bestimmten Dokument aus einer Klasse kann sowohl eine spezifische Beschreibung des individuellen Dokuments als auch eine generische Beschreibung der ganzen Klasse vorliegen. Ähnlich wie bei der Typüberprüfung in Programmiersprachen kann dann das Dokumentsystem prüfen, ob die spezifische und die generische Struktur zusammenpassen.

Im folgenden werden einige Möglichkeiten aufgezählt, wie sich Komponententypen aus ihren Bestandteilen aufbauen können. Dabei wird in kleinerer Schrift auf eventuelle Programmiersprachenanalogien eingegangen. Diese Absätze sind für das weitere Verständnis des Buches unerheblich.

Listen fester Länge mit fester Reihenfolge

Oft setzen sich die Elemente eines bestimmten Typs aus einer festen Zahl von Bestandteilen eventuell verschiedenen Typs zusammen, wobei die Reihenfolge der Bestandteile wesentlich ist.

Dies trifft im obigen Beispiel auf den Typ „Buch“ zu, der aus den Komponenten Titel, Vorwort, Hauptteil und Bibliographie besteht, die alle einen verschiedenen Typ haben. Feste Reihenfolge der Bestandteile bedeutet, daß in der zu dieser generischen Struktur gehörenden spezifischen Struktur die Teilkomponenten von den Autoren in genau dieser Reihenfolge angegeben werden müssen. Wegen ihrer festen Reihenfolge sind die einzelnen Komponenten in einer spezifischen logischen Struktur schon aufgrund ihrer Position identifizierbar. In einzelnen Dokumentsprachen müssen sie trotzdem zusätzlich durch geeignete Markierungen kenntlich gemacht werden.

Es ist zu erwarten (aber nicht notwendig), daß die Komponenten in einer schließlich entstehenden graphischen Struktur in der angegebenen Reihenfolge auftreten.

In Programmiersprachen werden die Parameter von Prozeduren meist als Liste von Objekten beliebigen Typs mit fester Reihenfolge angegeben (Pascal, C u.v.a.). Die Identifizierung erfolgt dabei allein aufgrund der relativen Position.

Listen fester Länge mit freier Reihenfolge

Diese Möglichkeit ist der eben genannten ziemlich ähnlich. Der Unterschied ist, daß in der zur generischen Struktur gehörenden spezifischen Struktur die Komponenten in beliebiger Reihenfolge auftreten können. Da die Komponenten dann aber nicht mehr anhand ihrer Reihenfolge zu identifizieren sind, müssen sie durch geeignete Markierungen kenntlich gemacht werden.

Die Anordnung der Komponenten in der schließlich entstehenden graphischen Struktur kann sich aus der Reihenfolge in der generischen oder der spezifischen logischen Struktur herleiten oder vom Formatierer nach festen Regeln oder den Umständen entsprechend flexibel hergeleitet werden.

Das Programmiersprachenanalogon zu diesem Konzept ist das der *Verbunde* („record“ in Pascal, „struct“ in C). Die einzelnen Verbundkomponenten können von verschiedenem Typ sein und erhalten eigene Namen zur Identifikation. Bei einem spezifischen Objekt vom Verbundtyp können die einzelnen Komponenten in beliebiger Reihenfolge angegeben werden; die Namen erlauben ihre Identifikation.

Listen beliebiger Länge

Die Elemente eines bestimmten Typs können auch Folgen aus einer beliebigen Zahl von Komponenten gleichen Typs sein. Zum Beispiel besteht der Hauptteil eines Buchs aus einer Folge von Kapiteln.

In der generischen logischen Beschreibung wird nur angegeben, daß es sich um eine Folge handelt und was der gemeinsame Typ aller Komponenten ist. In der spezifischen logischen Beschreibung werden dann die einzelnen Komponenten aufgezählt.

Von dem allgemeinen Konzept der Folge sind verschiedene Varianten denkbar: Folgen, die auch leer sein können; Folgen, die nicht leer sein können; Folgen mit einer gewissen Obergrenze für die Anzahl der Komponenten.

Bezüglich der graphischen Darstellung der Folge gibt es zwei Möglichkeiten: Die Reihenfolge der Folgenelemente in der graphischen Struktur wird unverändert aus der spezifischen logischen Darstellung übernommen, oder sie wird vom Formatierer nach irgendwelchen Kriterien neu festgesetzt.

Folgen aus einer beliebigen Zahl von Elementen gleichen Typs treten in imperativen Programmiersprachen wie Pascal oder C normalerweise nicht auf. Sie sind aber in funktionalen Sprachen wie SML, Miranda oder Haskell sehr populär und werden dort Listen genannt.

Optionale Komponenten

Bei einer Liste aus einer festen Anzahl von Komponenten beliebigen Typs könnte auch vereinbart werden, daß einzelne Komponenten fehlen dürfen. Diese wären dann optional, d. h. sie können vorhanden sein oder fehlen. Ein Beispiel ist:

> Ein Buch besteht aus einem Titel, einem Vorwort, einem Hauptteil und einer Bibliographie (die auch fehlen kann).

Dieses Konzept tritt in Programmiersprachen nur selten auf. In Ada ist es möglich, beim Prozeduraufruf einzelne Parameter wegzulassen. Diese erhalten dann in der Prozedurdeklaration vordefinierte Standardwerte.

Alternativen

Alternativen werden am besten durch das folgende Beispiel erklärt:

> Ein Absatz besteht aus einer nichtleeren Folge von Absatzbestandteilen.
>
> Ein Absatzbestandteil ist entweder „reiner Text“ (ohne weitergehende Struktur) oder ein Verweis (auf irgendetwas) oder eine Fußnote.

In der generischen logischen Struktur werden also verschiedene Möglichkeiten aufgezählt, wie ein Typ aufgebaut sein könnte. In der spezifischen logischen Struktur muß dann genau eine von diesen Möglichkeiten auftreten. Um sie erkennen zu können, muß normalerweise eine Markierung benutzt werden.

In Pascal tritt das Konzept der Alternative bei der Bildung varianter Verbunde auf, ist aber dann untrennbar mit der Verbundbildung verknüpft. In C gibt es „union“-Typen, die von der Verbundbildung unabhängig sind.

Vermischung von Konzepten

In der generischen logischen Struktur kann es auch Typen geben, die mit mehr als einem der oben definierten Konzepte gebildet sind. Sie ergeben sich durch das Ersetzen von Zwischentypen durch ihre Definitionen. Für diese Zwischentypen müssen dann keine eigenen Namen eingeführt werden. Aus

> Ein Buch besteht aus einem Titel, einem Vorwort, einem Hauptteil und einer Bibliographie.
>
> Der Hauptteil besteht aus einer Folge von Kapiteln.

ergibt sich z. B. durch Unterdrücken von „Hauptteil“

> Ein Buch besteht aus einem Titel, einem Vorwort, einer Folge von Kapiteln und einer Bibliographie.

Aus dem Beispiel für Alternativen

> Ein Absatz besteht aus einer nichtleeren Folge von Absatzbestandteilen.
>
> Ein Absatzbestandteil ist entweder „reiner Text“ oder ein Verweis oder eine Fußnote.

ergibt sich durch Unterdrücken von „Absatzbestandteil“

> Ein Absatz besteht aus einer nichtleeren Folge von Teilen, die entweder „reiner Text“ oder Verweise oder Fußnoten sind.

Strukturs prachen

Es gibt einige Sprachen zur Beschreibung logischer Strukturen, die es erlauben, eine generische logische Struktur zu definieren. In diesen Sprachen sind die oben genannten Konzepte in verschiedenem Grade realisiert. Auf jeden Fall bieten sie eine formale Teilsprache zur Definition des Aufbaus von Komponententypen, die es erlaubt, die Definitionen, die wir oben in deutschen Sätzen beschrieben haben, viel kürzer und präziser hinzuschreiben. Die wichtigste Strukturs prache ist SGML, das in Abschnitt 3.4.2 beschrieben wird.

Atome des hierarchischen logischen Aufbaus

Es stellt sich die Frage, bis zu welcher Ebene die hierarchische logische Zergliederung eines Dokuments gehen soll. Wenn sie bis zu einzelnen Zeichen im Text, in Formeln und Tabellen hinabgeht, dann wird die Hierarchie (bzw. die Datenstruktur zu ihrer Darstellung) viel zu groß. Wenn andererseits die Zerlegung bei ganzen Absätzen oder Tabellen endet, werden Verweise und Fußnoten sozusagen subatomar und sind nicht mehr über die logische Struktur zu erreichen.

Ein Mittelweg wäre, Wörter, Tabelleneinträge und Verweise als Atome zu benutzen. Im allgemeinen ist es aber nicht nötig, auf einzelne Wörter über die logische Struktur zugreifen zu können. Daher wird in real existierenden Systemen meist festgelegt, daß die Atome der logischen Struktur einerseits Folgen von Wörtern ohne Besonderheiten wie Verweise sind („reiner Text"), andererseits Verweise oder ähnliche Objekte.

3.1.2 Hierarchieunabhängige Beziehungen

Neben der baumartigen hierarchischen logischen Struktur gibt es auch andere logische Beziehungen, die den logischen Strukturbaum durch Querverweise zu einem Graphen machen. Im folgenden sollen die wichtigsten Arten von nichthierarchischen logischen Beziehungen aufgezählt werden.

Logische Verweise

Logische Verweise sind Verweise auf Komponenten der logischen Struktur, z. B. Abbildungen, Tabellen, Abschnitte, Kapitel, Formeln. Bei der Formatierung werden sie durch eine (automatisch) vergebene Bezeichnung ersetzt. Meist handelt es dabei um eine Zahl oder eine Folge von Zahlen (dieser Abschnitt hat z. B. die Bezeichnung 3.1.2, die sich aus der Nummer des Kapitels, des übergeordneten Abschnitts und einer eigenen Nummer zusammensetzt).

Ein Spezialfall dieses Konzepts sind Verweise auf Einträge im Literaturverzeichnis, die ersetzt werden durch

- eine laufende Nummer in der Reihenfolge des Auftretens, oder
- ein explizit angegebenes Symbol, oder
- ein schematisch aus den Autorennamen und dem Erscheinungsjahr berechnetes Kürzel.

Graphische Verweise

Graphische Verweise sind Verweise auf Elemente der graphischen Struktur, z. B. „auf Seite 7". Sie sind generell gefährlicher als logische Verweise, da z. B. bei wissenschaftlichen Artikeln, die in einen Tagungsband aufgenommen werden, die Seiten vom Verlag neu durchnumeriert werden.

Auf den ersten Blick führen graphische Verweise ein eigenartiges Zwitterdasein, da sie zwischen der logischen und der graphischen Struktur vermitteln. Sie können aber ganz auf die logische Ebene gehoben werden, wenn man sich vorstellt, daß die logische Struktur zwei Arten von atomaren Objekten enthält: Verweisobjekte und Bezugspunkte, auf die verwiesen wird. Bei der Formatierung werden die Bezugspunkte auf eine bestimmte Stelle in der graphischen Struktur abgebildet. Sie selbst sind an dieser Stelle unsichtbar, aber ihre Positionierungsinformation wird vom Formatierer in die betreffenden Verweisobjekte eingesetzt.

Erzeugen von Verweisen

Bei der Erzeugung von logischen oder graphischen Verweisen muß auf irgendeine Weise die logische Beziehung zwischen dem Verweis und dem Objekt, auf das verwiesen wird, etabliert werden. In interaktiven Systemen kann dies durch Zeigeaktionen mit der Maus geschehen, während die Autoren bei Benutzung von Batchsystemen das durch Einführen und Benutzen von symbolischen Namen tun müssen, die sie selbst erfinden.

Aktive Verweise

In neuartigen Systemen wie dem WWW (= World Wide Web) [DR95, Fis96] können Verweise auch „belebt“ werden, was allerdings nur bei der Betrachtung des Dokuments am Bildschirm funktioniert. Wenn die Leser einen im Dokument vorkommenden Verweis auf einen anderen Teil des Dokuments oder auf ein anderes Dokument mit der Maus selektieren, dann wird ihnen dieser andere Teil oder dieses andere Dokument direkt präsentiert. Andere Dokumente, die nicht auf dem eigenen Rechner liegen, werden dabei sogar von fremden Rechnern automatisch herbeigeholt (was allerdings zu Zeitverzögerungen führen kann).

Die Dokumentteile, auf die verwiesen wird, sind dabei nicht unbedingt nur Schriftdokumente, sondern können auch statische oder bewegte Bilder, akustisches Material oder Videos (bewegtes Bild und Ton zusammen) sein.

Eine schöne Anwendung dafür ist ein elektronisches Lexikon, wo die Einträge durch Querverweise untereinander vernetzt sind, so daß die Leser diesen Querverweisen ohne lästiges Suchen und Blättern folgen können. Bilder, Tonaufzeichnungen und Filme bieten ergänzende Informationen zu den klassischen textuellen Lexikoneinträgen.

Einbinden von Komponenten

Gewisse Komponenten der logischen Struktur können getrennt vom Dokument erstellt und ein- oder mehrfach in das Dokument eingebunden werden. Die eingebundene Komponente kann Standardtexte beinhalten wie z. B. Name und Anschrift der Autoren oder ihrer Firma, oder Dokumentteile (z. B. Graphiken), die von einem anderen Dokumentsystem als der Rest des Dokuments erzeugt worden sind.

Mehrfache Einbindung ist eine nichthierarchische Operation, da sie verschiedene Zweige des logischen Strukturbaums miteinander verbindet.

Visuell kann oft nicht unterschieden werden, ob eine Komponente mehrfach eingebunden wurde oder ob es sich um mehrere verschiedene, aber gleichaussehende Komponenten handelt. Logisch macht das einen Unterschied, da bei einer Modifikation einer mehrfach eingebundenen Komponente Veränderungen nur an einer Stelle durchgeführt werden müssen.

Bei der Umsetzung der logischen in die graphische Struktur gibt es mehrere Möglichkeiten, eine mehrfach eingebundene Komponente zu behandeln:

- Die Komponente wird nur einmal formatiert und dann mehrfach in die graphische Struktur eingehängt.
- Die Komponente wird unformatiert in mehreren Kopien in das Dokument eingesetzt, die dann unabhängig voneinander formatiert werden. Die einzelnen Kopien können sogar unterschiedliches Aussehen erhalten, wenn ihre Formatierung von ihrem Kontext abhängt.

3.1.3 Beziehung zwischen logischer und graphischer Struktur

Aus der logischen Dokumentstruktur wird durch den Formatierungsprozeß (Satz) die graphische Struktur des Dokuments berechnet. Die Beziehung zwischen logischer und graphischer Struktur kann dabei generisch oder spezifisch definiert werden.

In den Sprachen SGML/DSSSL, ODA und Interscript kann man verschiedene Strukturen für Dokumente einer Klasse spezifizieren und Dokumente aus einer Struktur in eine andere transformieren. Beispiele für verschiedene Dokumentstrukturen sind die logische und die graphische Dokumentstruktur in verschiedenen konkreten Realisierungen. Transformationen können beispielsweise erfolgen

- von linearer in hierarchische Darstellung (syntaktische Analyse),
- von einer hierarchischen Darstellung in eine andere (Baumtransformation),
- von einer hierarchischen Darstellung in eine lineare Zwischendarstellung und zurück,
- von einer hierarchischen Darstellung in eine Zielsprache, z. B. eine Seitenbeschreibungssprache.

Mitunter können Programme, die solche Transformationen durchführen, aus Spezifikationen der Dokumentstrukturen und der Transformationen automatisch generiert werden.

3.2 Logische Dokumentstruktur im *FrameMaker*

3.2.1 „Hierarchische“ logische Struktur

Abweichend von den obigen Erläuterungen gibt es im *FrameMaker* keine Möglichkeit, ein Dokument hierarchisch logisch zu strukturieren. Es ist also logisch gesehen nicht möglich, ein Dokument in Kapitel und diese in Abschnitte und dann in Unterabschnitte zu gliedern. Trotzdem kann, wie wir sehen werden, der graphische Anschein einer solchen Unterteilung simuliert werden.

Paragraphenarten

Das wichtigste Element der logischen Struktur in einem *FrameMaker*-Dokument sind *Paragraphen*. Das gesamte Dokument besteht aus einer Folge von Paragraphen verschiedener Art. Auch Überschriften, Listeneinträge und Formeln können Paragraphen sein. Wie bereits erwähnt, erschöpft sich die logische Strukturierung eines *FrameMaker*-Dokuments darin, daß es in eine lineare Folge von Paragraphen eingeteilt ist. Es gibt nicht die Möglichkeit, Paragraphen ineinanderzuschachteln.

Die Art eines Paragraphen legt fest, wie er formatiert wird. Die als Dokumentschablonen mitgelieferten Standardformate (Buch, Report usw.) beinhalten unterschiedliche Kataloge von Paragraphenarten mit verschiedenen Formatattributen.

Ähnlich wie bei den Zeichenformaten (Abschnitt 2.3.2) können die Benutzer des *FrameMaker* das zum gewählten Standardformat gehörende standardmäßige Layout einer vom System vorgegebenen Paragraphenart umdefinieren oder sich selbst neue Paragraphenarten hinzudefinieren. Die folgenden Paragraphenarten kommen (eventuell unter anderem Namen) in vielen Standardformaten vor:

- *Body:*
  Ein gewöhnlicher Textabsatz. Die erste Zeile ist in der Standardeinstellung eingerückt, die anderen nicht.
- *FirstBody:*
  Ein Textabsatz, der als erstes in einem Kapitel steht. Die Einrückung der ersten Zeile fehlt.
- *Bullet:*
  Ein Absatz, der zu einer nicht numerierten Liste gehört. Die erste Zeile beginnt mit einem „Bullet“ •, während die anderen Zeilen eingerückt sind.
- *CBullet:*
  Ein Folgeabsatz eines Eintrags in einer nicht numerierten Liste. Alle Zeilen sind eingerückt und zwar so viel wie die nicht-ersten Zeilen bei der Art „Bullet“.
- *1Step:*
  Der allererste Eintrag einer numerierten Liste. Die erste Zeile beginnt mit der Nummer „1.“, während die anderen Zeilen eingerückt sind.

- *Step:*
  Ein weiterer Eintrag einer numerierten Liste. Die erste Zeile beginnt mit einer Nummer, die um eins höher ist als die Nummer des letzten vorhergehenden Paragraphen der Art „Step“ oder „1Step“. Das gilt auch, wenn dazwischen Paragraphen anderer Art liegen, sogar Kapitelwechsel. Das Hochzählen und Ausgeben der Nummer wird dabei von *FrameMaker* selbst besorgt. Die Benutzer geben einfach nur die Paragraphenart an.

  Da es keine Schachtelung gibt, gibt es auch keine Möglichkeit, in einem Eintrag einer numerierten Liste wieder eine numerierte Liste unterzubringen. (Die entsprechende graphische Erscheinungsform kann allerdings simuliert werden, indem die Benutzer selbst für die korrekte Numerierung sorgen.)

- *CStep:*
  Ein Folgeabsatz in einer numerierten Liste. Er hat keine eigene Nummer und trägt nicht zur Numerierung bei. Alle Zeilen sind eingerückt und zwar so viel wie die nicht-ersten Zeilen bei der Art „1Step“ oder „Step“.

- *Chapter:*
  Eine Kapitelüberschrift. Sie beginnt mit dem Wort „Chapter“, gefolgt von der Kapitelnummer. Der Rest ist in einem besonders großen Font geschrieben.

- *Equation:*
  Eine Formel in einer eigenen Zeile. Sie wird normalerweise zentriert und rechts unten mit einer laufenden Nummer versehen. Man beachte aber, daß die Angabe der Art „Equation“ nicht genügt, um eine Formel einzugeben; es sind eigene Kommandos erforderlich, um den Formeleingabemodus zu starten und zu beenden.

Die Art eines existierenden Paragraphen kann durch Auswahl einer Art aus einer Liste von Arten geändert werden. Die Änderung bezieht sich auf denjenigen Paragraphen, in dem ein Einfügepunkt oder ein selektiertes Textstück liegt.

Während der Texteingabe wird der Paragraphenwechsel durch die Return-Taste angezeigt. Die Art des neuen Paragraphen wird nach Standardregeln bestimmt. Wenn man eine andere Art haben will, muß man sie anwählen, *nachdem man Return gedrückt hat*. Dann sind schon Teile des neuen Paragraphen sichtbar, z. B. die Nummer bei der Art „Step“. Es gibt keine Möglichkeit, durch *ein* Kommando anzuzeigen, daß man einen neuen Paragraphen beginnen will und welche Art er haben soll.

Die Standardregeln für die Bestimmung der Art des nächsten Paragraphen bei einigen der oben genannten Arten lauten wie folgt:

jetzige Art	Chapter	FirstBody	Body	1Step	Step	CStep	Equation
nächste Art	FirstBody	Body	Body	Step	Step	CStep	Equation

Definition von Paragraphenarten

Die Eigenschaften von Paragraphenarten können mit Hilfe eines Dialogfensters festgelegt (für neudefinierte Arten) oder modifiziert werden (für bereits existierende Arten). Um den Umfang des Dialogfensters zu begrenzen, gibt es ein übergeordnetes Menü „Properties“ zur Auswahl zwischen sechs Gruppen von Eigenschaften. Je nachdem, welche Gruppe angewählt wird, ändert sich das Aussehen des restlichen Dialogfensters.

Die Gruppe „Basic“ dient zur Festlegung der Grundeigenschaften einer Paragraphenart. Dazu gehören:

- Drei Längenangaben (Zahlen mit einer Längeneinheit) mit den Bezeichnungen „First“, „Left“ und „Right“. Dabei gibt „First“ die Einrücktiefe der ersten Zeile des Paragraphen von links an, „Left“ die der nicht-ersten Zeilen von links und „Right“ die aller Zeilen von rechts.
- Größe des freien vertikalen Abstands über und unter dem Paragraphen. Dieser Abstand entfällt am Seitenanfang oder -ende. Wenn zwei Paragraphen auf einer Seite hintereinanderstehen, dann ist der Abstand zwischen ihnen nicht die Summe, sondern das Maximum aus dem angegebenen Abstand hinter dem ersten Paragraphen und dem vor dem zweiten.
- Zeilenabstände im Innern des Paragraphen.
- Man kann ferner angeben, ob dieser Zeilenabstand konstant sein soll oder sich vergrößern soll, wenn einzelne Textteile im Paragraphen in einem sehr großen Font geschrieben werden.
- Die Ausrichtung des Paragraphen kann linksbündig, rechtsbündig, zentriert oder randausgeglichen sein.
- Tabulatorpositionen können gesetzt werden.
- Die Art des Folgeparagraphen kann angegeben werden. Wenn nichts angegeben wird, dann hat der Folgeparagraph dieselbe Art wie der gegenwärtige.

Die Gruppe „DefaultFont“ dient zur Angabe der Schriftart im Paragraphen. Die hierzu angebotene Auslegung des Dialogfensters entspricht derjenigen, die zur Neu- oder Umdefinition von Zeichenformaten benutzt wird (siehe Abschnitt 2.3.2).

Die Gruppe „Pagination“ erlaubt Festlegungen zum Seitenumbruch (siehe Abschnitt 4.4.1).

Die Gruppe „Numbering“ dient zur Definition der Art und Weise, wie gewisse Paragraphen, z. B. Einträge in numerierten Listen und Kapitelüberschriften, numeriert werden. Darunter fallen auch Festlegungen wie die, daß Einträge in einer nicht numerierten Liste mit • beginnen. Die Numerierung wird unten genauer besprochen.

Die Gruppe „Advanced“ beinhaltet unter anderem die Definition der minimalen, idealen und maximalen Wortabstände im Paragraphen, der Sprache des Paragraphen (wichtig für Rechtschreibprüfung und Worttrennung) und einiger Parameter, die die Worttrennung regulieren. Schließlich gibt es noch die Gruppe „TableCells“.

Numerierung, oder wie die Illusion einer Hierarchie erzeugt wird

Das vorliegende Buch (das in LaTeX geschrieben wurde) ist in Kapitel, Abschnitte und Unterabschnitte eingeteilt. Die Kapitelüberschriften sind numeriert mit „1“, „2“ usw., die Abschnitte von Kapitel 2 mit „2.1“, „2.2“ usw. und die Unterabschnitte von Abschnitt 2.2 mit „2.2.1“, „2.2.2“ usw. Dazu kommen noch Abbildungen, die in Kapitel 2 unabhängig von der Abschnittseinteilung mit „2.1“, „2.2“ usw. numeriert sind. Wie könnte eine solche Numerierung mit *FrameMaker* erzeugt werden, der ja keine Hierarchien kennt?

Jede Paragraphenart besitzt ein *Numerierungsformat*, das festlegt, wie diese Art von Paragraphen numeriert wird. (Viele Arten wie z. B. „Body“ haben ein leeres Numerierungsformat.) Ein Numerierungsformat besteht aus beliebigem Text, in den Referenzen auf Zähler eingebettet sind. Dazu kommt die Zuordnung zu einer *Zählerreihe*, das ist eine Gruppe von Zählern, die sich gegenseitig beeinflussen. Da in unserem Beispiel die Numerierung von Kapiteln, Abschnitten, Unterabschnitten und Abbildungen voneinander abhängt, gehören sie alle zur selben Zählerreihe, die wir „K“ wie Kapitel nennen wollen. Eine davon völlig unabhängige Numerierung wie z. B. die der Einträge in einer numerierten Liste würde zu einer anderen Zählerreihe gehören.

Eine Zählerreferenz wird in spitzen Klammern eingeschlossen. Sie kann die folgenden Formate haben:

`<n>`	Der Zähler wird angezeigt, ohne daß sich der Wert ändert.
`<n=0>`	Der Zähler wird auf Null gesetzt und dieser Wert wird angezeigt.
`<n+>`	Der Zähler wird um eins erhöht und der neue Wert wird angezeigt.
`< >`	Der Zähler wird weder verändert noch angezeigt.
`< =0>`	Der Zähler wird auf Null gesetzt, aber nicht angezeigt.

Der Buchstabe `n` steht für „numerisch“, d. h. der Zähler wird als 1, 2, 3 ... angezeigt. Ein `r` (römisch) würde Ausgabe als i, ii, ... bewirken. Es gibt noch einige andere solcher Buchstabencodes. Statt der Zahl 0 kann auch eine andere Zahl genommen werden. Der Leerraum in den beiden letzten Möglichkeiten ist zwingend. Zählerreferenzen können eingetippt oder aus einer Liste ausgewählt werden.

Insbesondere die Möglichkeit `< >` mutet seltsam an. Der Grund für ihre Existenz ist, daß Zähler keine Namen haben, sondern über ihre Reihenfolge identifiziert werden. Daher muß in Numerierungsformaten, die zur selben Zählerreihe gehören, die Anzahl der Zähler übereinstimmen.

Paragraphenart	*Format*	*Typische Ausgaben*	
Kapitel	`K:Kap. <n+>< =0>< =0>< =0>`	Kap. 1	Kap. 2
Abschnitt	`K:<n>.<n+>< =0>< >`	2.1	2.2
Unterabschnitt	`K:<n>.<n>.<n+>< >`	2.2.1	2.2.2
Abbildung	`K:Abb. <n>.< >< ><n+>`	Abb. 2.1	Abb. 2.2

Abbildung 3.1: Beispiele für Numerierungsformate

In unserer Beispielzählerreihe „K“ brauchen wir insgesamt vier Zähler: für Kapitel, Abschnitte, Unterabschnitte und Abbildungen. Sie müssen alle vier in allen zu „K“ gehörenden Numerierungsformaten in einer festen Reihenfolge erwähnt werden. Die Formate für die einzelnen Paragraphenarten sind in Abbildung 3.1 angegeben. Beachten Sie, daß die Art „Kapitel“ genau genommen „Kapitelüberschrift“ heißen müßte. Ähnliches gilt für die anderen Paragraphenarten in der Abbildung.

In den Numerierungsformaten gibt `K:` die Zuordnung zur Zählerreihe an. Von den vier Zählerreferenzen bezieht sich immer die erste auf den Kapitelzähler, die zweite auf den Abschnittszähler, die dritte auf den Unterabschnittszähler und die vierte auf den Abbildungszähler. Weil die Zähler nur über die Reihenfolge ihrer Referenzen erkennbar sind, muß der Abschnittszähler auch bei den Bildunterschriften referenziert werden, obwohl er weder ausgegeben noch verändert wird. Wenn es zusätzlich noch Sätze, Definitionen und Gleichungen geben soll, deren Nummern auch die Kapitelnummer enthalten, erhöht sich die Anzahl der Zählerreferenzen in *allen* Numerierungsformaten der Reihe „K“ entsprechend.

Wenn man nicht ein in einer Dokumentschablone vordefiniertes Numerierungssystem benutzen will, erfordert die Definition eines eigenen Systems also sorgfältiges und planvolles Arbeiten, bei dem die Interaktivität des *FrameMaker* keine große Hilfe ist.

3.2.2 Verweise im *FrameMaker*

Es gibt im *FrameMaker* zwei Arten von Verweisen: solche, die auf einen ganzen Paragraphen zeigen, und solche, die auf einen Punkt zeigen. Zunächst werden wir uns nur mit den Verweisen der ersten Art befassen. Das Verweisziel ist meist eine Überschrift, kann jedoch auch ein gewöhnlicher Textparagraph sein. Das Verweisziel kann sogar in einem anderen Dokument liegen, aber diese Möglichkeit werden wir hier nicht weiter betrachten, um die Darstellung nicht zu komplizieren.

Die graphische Erscheinungsform eines Verweises hängt von dem für ihn gewählten Format ab. Ähnlich wie es Kataloge von Zeichenformaten und Paragraphenarten gibt, gibt es auch welche von Verweisformaten. Eine Reihe von Verweisforma-

ten sind vordefiniert. Sie können umdefiniert und neue können hinzugefügt werden.

Erzeugen eines Paragraphverweises

Das Erzeugen eines Verweises auf einen Paragraphen geschieht in mehreren Schritten. In der folgenden Aufzählung sind Benutzeraktionen mit **B** markiert und *FrameMaker*-Reaktionen mit **F**.

B: Der Einfügepunkt wird an die Stelle gesetzt, wo der Verweis erscheinen soll. Bei linearer Eingabe des Dokuments befindet er sich bereits dort, so daß nichts getan werden muß.

B: Das Kommando „Cross-Reference“ wird gegeben (normalerweise durch Anwählen mit der Maus).

F: Es erscheint ein Hilfsfenster mit einer Liste der Paragraphenarten.

B: Die Paragraphenart des Verweisziels wird ausgewählt.

F: Das Hilfsfenster stellt dann alle (!) Paragraphen dieser Art durch Angabe ihres Inhalts dar.

B: Einer davon wird ausgewählt.

B: Des weiteren wird ein Verweisformat selektiert ...

B: ... und das Kommando „Insert“ gegeben.

F: Der Verweis erscheint am Einfügepunkt in der durch das Verweisformat beschriebenen Form.

F: Außerdem wird der Anfang des Paragraphen, auf den verwiesen wurde, speziell markiert. Die Markierung ist ein besonderes Symbol, das bei mehrfachen Verweisen auf dasselbe Ziel nicht wiederholt wird. Es erscheint nur auf dem Bildschirm, nicht aber im Ausdruck.

Erzeugen eines Punktverweises

Bevor ein Verweis auf eine bestimmte Stelle des Dokuments (nicht einen ganzen Paragraphen) erfolgen kann, muß diese Stelle irgendwie markiert werden. Dieser Markierungsschritt entfällt bei Paragraphverweisen, da dort alle Paragraphen einer bestimmten Art als potentielle Verweisziele angenommen werden. Das Markieren einer Stelle erfolgt so:

B: Der Einfügepunkt wird an die Stelle gesetzt, die markiert werden soll.

B: In einem Hilfsfenster, das durch das Kommando „Marker“ geöffnet werden kann, wird die Markierung als Verweisziel deklariert ...

B: ... und erhält einen symbolischen Namen. Dieser Name ist ein beliebiger Text mit bis zu 255 Zeichen, kann also auch ein oder mehrere Sätze umfassen.

B: Die Operation wird durch das Kommando „New Marker“ beendet.

F: Das Verweisziel wird markiert. Die Markierung ist ein besonderes Symbol, das nur auf dem Bildschirm, nicht aber im Ausdruck erscheint.

Um einen Punktverweis zu erzeugen, wird im Prinzip so vorgegangen wie bei der Erzeugung eines Paragraphenverweises. Statt einer Paragraphenart wird jedoch die zusätzliche Möglichkeit „Cross-Ref Markers“ selektiert. Dann erscheinen die symbolischen Namen aller existierenden Verweisziele, von denen einer ausgewählt werden kann.

Verweise und Dokumentänderungen

Im Gegensatz zu den meisten anderen von *FrameMaker* automatisch erstellten Informationen passen sich Verweise nicht sofort an auftretende Veränderungen an. Wenn ein Verweis die Seitennummer des Verweisziels enthält, dann ändert er sich nicht automatisch, wenn das Verweisziel aufgrund einer Dokumentveränderung auf eine andere Seite gerät. Erst nach einem besonderen Kommando des Benutzers berechnet der *FrameMaker* alle Verweise neu und bringt sie so auf den neuesten Stand.

Verweise können problemlos gelöscht, verschoben oder kopiert werden. Auch wenn alle Verweise auf einen bestimmten Paragraphen gelöscht werden, bleibt das Verweiszielsymbol am Anfang des Paragraphen stehen. Beim Herumwandern in einem Dokument kann von einem Verweis zu seinem Ziel gesprungen werden.

Die graphische Erscheinungsform eines Verweises kann durch Wechsel des Verweisformats geändert werden. Wenn kein adäquates Verweisformat existiert, kann der Verweis auch in gewöhnlichen Text verwandelt und danach beliebig verändert werden. Die Umwandlung in Text ist aber unwiderruflich und zerstört den Bezug zum Verweisziel. In Text verwandelte Verweise können also nicht mehr automatisch auf den neuesten Stand gebracht werden.

Verweisziele sind nicht so frei edierbar wie Verweise. *FrameMaker* achtet darauf, daß Verweisziele nicht dupliziert werden. Wenn Text kopiert wird, der ein Verweisziel enthält, dann wird das Verweisziel nicht mitkopiert, es sei denn, es wurde im Originaltext im Verlauf der Kopieraktion gelöscht, d. h. zwischen dem Kopieren des Originals in einen Zwischenspeicher und dem Entleeren des Zwischenspeichers an der Zielstelle.

Wird ein Verweisziel, auf das noch verwiesen wird, irrtümlich gelöscht, dann gibt es erst bei der nächsten Neuberechnung der Verweise eine Fehlermeldung. In diesem Falle bietet der *FrameMaker* Hilfestellung bei der Bereinigung der fehlerhaften Situation an.

Verweisformate

Die Verweisformate geben die graphische Erscheinungsform eines Verweises an. Sie haben einen Namen, unter dem sie ausgewählt werden können, und eine Definition. Die Definition eines Verweisformats besteht aus gewöhnlichem Text, in den Spezialelemente eingestreut sind. Einige der vielen möglichen Spezialelemente sind:

`<$paranum>`	Die Nummer des Zielparagraphen. Es handelt sich um die Nummer, die von *FrameMaker* automatisch gemäß dem Numerierungsformat erzeugt wird wie in Abschnitt 3.2.1 beschrieben.
`<$paratag>`	Die Art des Zielparagraphen.
`<$paratext>`	Der Inhalt des Zielparagraphen ohne die Nummer.
`<$pagenum>`	Die Nummer der Seite, auf der das Verweisziel liegt. Wenn das Ziel ein Paragraph ist, zählt der Anfang des Paragraphen.
`<$paratext[`*Art,...*`]>`	Der Inhalt des ersten Paragraphen mit einer der angegebenen Arten, den man findet, wenn man vom Verweisziel zum Anfang des Dokuments zurückläuft. Ähnliche Konstruktionen sind auch mit „paratag“, „paranum“ und „pagenum“ möglich.
<*Zeichenformat*>	Das Format tritt an dieser Stelle in Kraft und gilt so lange, bis ein anderes gewählt wird oder bis das Ende des Verweises erreicht wird.

Die Spezialelemente können entweder textuell eingegeben oder mit der Maus aus einer Liste selektiert werden. In Abbildung 3.2 werden einige Verweisformate und die von ihnen erzeugte graphische Erscheinungsform eines Verweises aufgezählt, dessen Ziel die Überschrift des aktuellen Unterabschnitts ist.

3.3 Logische Dokumentstruktur in LaTeX

Im Gegensatz zu *FrameMaker* erlaubt es LaTeX, Dokumente hierarchisch logisch zu strukturieren. Allerdings ist eine gewisse Vorsicht angebracht: es handelt sich nur teilweise um eine echte Hierarchie. Wir werden darauf am Ende dieses Abschnitts zurückkommen.

3.3.1 Die Abschnittseinteilung

Ein LaTeX-Dokument wie das vorliegende Buch kann in Kapitel, Abschnitte, Unterkapitel usw. eingeteilt werden. Die genaue Zahl und Natur der Gliederungsstufen

Verweisformat	erzeugter Verweis
`Abschnitt <$paranum>`	Abschnitt 3.2.2
`Seite <$pagenum>`	Seite 54
`siehe Abschnitt <Emphasis> <$paratext> <DefaultFont> auf Seite <$pagenum>`	siehe Abschnitt *Verweise im FrameMaker* auf Seite 54
`Abschnitt <Bold> <$paratext> <DefaultFont> im Kapitel <Bold> <$paratext[Kapitel]>`	Abschnitt **Verweise im** *FrameMaker* im Kapitel **Logische Dokumentstruktur**

Abbildung 3.2: Beispiele für Verweisformate

sowie ihre graphische Erscheinungsform werden durch den Dokumentstil festgelegt. Zur Vereinfachung nehmen wir hier einen festen Dokumentstil an.

Der Anfang von Gliederungsstrukturen wird durch die folgenden Befehle gekennzeichnet:

`\chapter{`*Überschrift*`}`	Anfang eines Kapitels
`\section{`*Überschrift*`}`	Anfang eines Abschnitts
`\subsection{`*Überschrift*`}`	Anfang eines Unterabschnitts

Jeder dieser Befehle hat mehrere Wirkungen: Der zu dem Befehl gehörende Zähler wird um eins hochgezählt. An der Stelle, an der der Befehl gegeben wird, erscheint eine eventuell mehrgliedrige Nummer, die sich aus den Zählern der übergeordneten Strukturen und dem eigenen Zähler zusammensetzt. Es folgt die Überschrift, die groß und fett gesetzt wird. Die Überschrift wird außerdem ins Inhaltsverzeichnis übernommen und bei manchen Dokumentstilen in der Kopfzeile der Seite angezeigt.

Die Numerierung erfolgt automatisch gemäß dem gewählten Dokumentstil. Der aktuelle Unterabschnitt z. B. beginnt mit dem Kommando

```
\subsection{Die Abschnittseinteilung}
```

Die Nummer „3.3.1“ wird automatisch erzeugt und braucht nicht mit eingegeben zu werden.

Die graphische Erscheinungsform der Überschriften und die Numerierung können umdefiniert werden. Das erfordert jedoch zum Teil tieferes TEX-Wissen. Eingriffe in die Numerierung werden dadurch erleichtert, daß die einzelnen Zähler im Gegensatz zu denen des *FrameMaker* Namen haben. Der Kapitelzähler z. B. heißt

`chapter`. Ein Zähler kann sich auf einen anderen beziehen, z. B. bezieht sich der Abschnittszähler `section` auf den Kapitelzähler `chapter`. Das bewirkt, daß `section` automatisch auf 0 gesetzt wird, wenn sich `chapter` um 1 erhöht.

Zähler können durch gewisse Kommandos gesetzt, erhöht oder abgefragt werden. Diese Kommandos werden durch die Einteilungsbefehle `\chapter` usw. automatisch ausgelöst und brauchen von LaTeX-Benutzern normalerweise nie selbst aufgerufen zu werden.

3.3.2 Umgebungen

Es gibt in LaTeX weitere Hilfsmittel zur logischen Gliederung auf kleinerer Ebene. Gewisse Stücke des Dokuments können zu einer *Umgebung* zusammengefaßt werden. Jede Umgebung hat einen Typ *T*. Der Anfang der Umgebung wird durch den Befehl `\begin{`*T*`}` und das Ende durch `\end{`*T*`}` bezeichnet. Das Material im Inneren der Umgebung wird abhängig vom Typ besonders behandelt. Manche Umgebungstypen haben zusätzliche Parameter, die nach dem `\begin`-Befehl angegeben werden.

Umgebungen können ineinandergeschachtelt auftreten. LaTeX achtet darauf, daß die Umgebungsanfänge und -enden eine korrekte hierarchische Klammerstruktur bilden: zu jedem `\begin` muß es ein `\end` geben und umgekehrt, beide müssen denselben Typ haben und Überlappungen der Form

```
\begin{A} ... \begin{B} ... \end{A} ... \end{B}
```

sind verboten.

Einige einfache Umgebungen

Es gibt eine große Zahl von vordefinierten Umgebungstypen und weitere können von LaTeX-Benutzern dazudefiniert werden. Einige vordefinierte Umgebungstypen sind:

`quote` Der Inhalt der Umgebung erscheint links und rechts eingerückt.

`center` Der Inhalt wird zentriert.

`tabular` Die Umgebung enthält die Beschreibung einer Tabelle. Eine Tabellenbeschreibung enthält besondere Befehle, die nur in einer `tabular`-Umgebung erlaubt sind.

`figure` Der Inhalt der Umgebung wird aus dem normalen Text herausgehoben, meist an den Anfang einer Seite gesetzt und als Abbildung bezeichnet.

Die Abbildung 3.2 z. B. kann wie in Abbildung 3.3 gezeigt definiert werden. Die Beschreibung des Tabelleninneren wurde dabei weggelassen. Der Befehl `\caption` erzeugt die Bildunterschrift. Das Wort „Abbildung“ und die Nummer entstehen automatisch. Der Befehl `\label` definiert ein Verweisziel (siehe Abschnitt 3.3.3).

```
\begin{figure}
\begin{center}
\begin{tabular} ...
...
\end{tabular}
\end{center}
\caption{Beispiele f"ur Verweisformate}%
\label{FM-Verweisformate}
\end{figure}
```

Abbildung 3.3: Definition von Abbildung 3.2

Listenumgebungen

Es gibt einige Umgebungstypen, die eine listenartige Aufzählung von Einträgen definieren. Jeder einzelne Eintrag wird durch den Befehl `\item` eingeleitet. Die wichtigsten Listenumgebungen sind:

`itemize`	Jeder Eintrag wird durch ein besonderes Zeichen eingeleitet.
`enumerate`	Die Einträge sind automatisch durchnumeriert.
`description`	Jeder Eintrag beginnt mit einer fettgedruckten Einleitung, die hinter dem Befehl `\item` in eckigen Klammern eingeschlossen angegeben wird.

Die Liste, die Sie gerade gelesen haben, hat übrigens keine der drei vordefinierten Arten, sondern eine selbstdefinierte Art `longlabellist`. Abbildung 3.4 enthält einige Beispiele für Listen. Die Art `longlabellist` hat einen Parameter, in diesem Fall „`Sommer:` “, dessen Breite die Tiefe der Einrückung der Listeneinträge definiert.

Wie bereits erwähnt, können Umgebungen geschachtelt werden:

1. Die äußere Liste ist eine `enumerate`-Umgebung.
   (a) Hier beginnt ein inneres `enumerate`.
   (b) Wie man sieht, hat es einen anderen Zählstil als das äußere.
   (c) Hier endet das innere `enumerate`.
2. Jetzt sind wir wieder in der äußeren Liste.
   - Hier beginnt ein eingeschachteltes `itemize`.
     - Es enthält wiederum ein `itemize`.
     - Der Stil der Markierung wird automatisch variiert.
   - Zurück zum äußeren `itemize`, das hier endet.
3. Jetzt geht auch die äußerste Liste zu Ende.

```
\begin{itemize}
\item Der Winter ist eine Jahreszeit,
      in der es kalt ist.
\item Im Sommer ist es dagegen hei"s.
\end{itemize}
```

- Der Winter ist eine Jahreszeit, in der es kalt ist.
- Im Sommer ist es dagegen heiß.

```
\begin{enumerate}
\item Der Winter ist eine Jahreszeit,
      in der es kalt ist.
\item Im Sommer ist es dagegen hei"s.
\end{enumerate}
```

1. Der Winter ist eine Jahreszeit, in der es kalt ist.
2. Im Sommer ist es dagegen heiß.

```
\begin{description}
\item[Winter: ]  eine Jahreszeit,
                 in der es kalt ist.
\item[Sommer: ]  hei"se Jahreszeit.
\end{description}
```

Winter: eine Jahreszeit, in der es kalt ist.

Sommer: heiße Jahreszeit.

```
\begin{longlabellist}{Sommer: }
\item[Winter:]  eine Jahreszeit,
                in der es kalt ist.
\item[Sommer:]  hei"se Jahreszeit.
\end{longlabellist}
```

Winter: eine Jahreszeit, in der es kalt ist.

Sommer: heiße Jahreszeit.

Abbildung 3.4: Listenumgebungen in LaTeX

Wie man sieht, wird die logische Struktur korrekt durch die graphische Struktur, insbesondere die Einrückungen, wiedergegeben. Bei geschachteltem `enumerate` benutzen die inneren Listen selbstverständlich einen anderen Zähler als die äußeren. Bei den Umgebungen `itemize` und `enumerate` wird außerdem die Gestalt der durch `\item` erzeugten Eintragsmarkierung variiert.

Definition von neuen Umgebungstypen

LaTeX bietet den Autoren mehrere Möglichkeiten an, sich neue Umgebungstypen selbst zu definieren.

In mathematischen Texten gibt es häufig Strukturen wie „Axiom“, „Definition“, „Satz“ oder „Bemerkung“. Entsprechende Umgebungstypen könnten z. B. wie folgt eingeführt werden:

```
\newtheorem{axiom}{Axiom}
\newtheorem{def}{Definition}[chapter]
\newtheorem{satz}{Satz}[chapter]
\newtheorem{bem}[satz]{Bemerkung}
```

Die vier neu eingeführten Umgebungstypen heißen `axiom`, `def` usw. Ihre Instanzen werden mit `\begin{axiom}` – `\end{axiom}`, `\begin{def}` – `\end{def}` usw. abgegrenzt. Die Instanzen vom Typ `axiom` werden mit „Axiom 1“, „Axiom 2“ usw. numeriert. Wegen des Zusatzes `[chapter]` werden die Instanzen vom Typ `def` kapitelabhängig numeriert; die Definitionen in Kapitel 2 heißen daher z. B. „Definition 2.1“, „Definition 2.2“ usw. Dasselbe gilt für die Instanzen vom Typ `satz`. Sie benutzen einen anderen Zähler als die vom Typ `def`, d. h. nach „Definition 2.1“ kommt „Satz 2.1“. Der Typ `bem` dagegen benutzt denselben Zähler wie `satz`. Nach „Satz 2.1“ hat also die erste Bemerkung im Kapitel 2 die Bezeichnung „Bemerkung 2.2“.

Die durch `\newtheorem` definierten Umgebungstypen haben eine bestimmte einheitliche graphische Erscheinungsform. Jede einzelne Instanz kann durch die üblichen Befehle graphisch anders gestaltet werden. Eine generelle Änderung der Erscheinungsform ist auch möglich, erfordert aber tiefergehende Kenntnisse über die Implementierung von LaTeX in TeX.

Während die durch `\newtheorem` eingeführten Umgebungstypen ein einheitliches Aussehen besitzen, lassen sich durch `\newenvironment` Umgebungstypen mit nahezu beliebigem Aussehen erzeugen. Es würde hier zu weit führen, darauf näher einzugehen.

3.3.3 Verweise in LaTeX

Stellen im Dokument können durch das Kommando `\label{`*Name*`}` markiert werden. Der Formatierer bindet diesen Namen einerseits an die Nummer der Seite, auf der diese Stelle erscheint, und andererseits (in guter Näherung, siehe Abschnitt 3.3.4) an die Bezeichnung (Nummernfolge) des innersten logischen Dokumentteils, der besagte Stelle enthält und überhaupt eine eigene Bezeichnung hat. Verweise auf diese Stelle erfolgen durch die Kommandos `\ref{`*Name*`}`, was beim Formatieren durch die an *Name* gebundene Bezeichnung (Nummernfolge) ersetzt wird, oder `\pageref{`*Name*`}`, was durch die an *Name* gebundene Seitennummer ersetzt wird.

Der vorliegende Abschnitt beginnt z. B. in der LaTeX-Dokumentenbeschreibung mit

```
\subsection{Verweise in \LaTeX}\label{LaTeX-Verweise}
```

Die Zeile

```
Abschnitt \ref{LaTeX-Verweise} beginnt
auf Seite \pageref{LaTeX-Verweise}.
```

aus der Dokumentenbeschreibung wird also so formatiert:

> Abschnitt 3.3.3 beginnt auf Seite 62.

Der Leser möge nachprüfen, daß die angegebenen Werte für Abschnittsbezeichnung und Seitennummer tatsächlich stimmen.

Wegen des Problems der Vorwärtsverweise werden Verweise in LaTeX aus Effizienzgründen folgendermaßen bearbeitet: Der LaTeX-Formatierer sammelt während eines Laufs alle Verweisnamen mit ihren aktuellen Bindungen und schreibt diese Information in eine Hilfsdatei. Beim nächsten LaTeX-Lauf wird diese Hilfsdatei noch vor der eigentlichen Dokumentbeschreibung gelesen und die Verweise werden aufgrund der darin enthaltenen Information (die aber noch aus dem vorigen Lauf stammt) ersetzt. Daher ist erst nach zwei LaTeX-Läufen, zwischen denen die Autoren keine Änderungen vorgenommen haben, (fast) sicher, daß alle Verweise stimmen. (Das „fast“ bezieht sich auf die extrem unwahrscheinliche Möglichkeit, daß die sich verändernden Verweise die Seitenstruktur soweit ändern, daß sie sich selbst wieder verändern müßten.)

3.3.4 Echte und unechte Hierarchie

Die bisherige Beschreibung hat vermutlich den Eindruck erweckt, daß eine LaTeX-Dokumentbeschreibung eine ausgefeilte hierarchische logische Struktur besitzt. Bei genauerem Hinsehen ist dem jedoch nicht ganz so, wie man durch Experimentieren feststellen kann. Ein Teil der Hierarchie wird den Benutzern nämlich nur vorgegaukelt, was bei einigermaßen vernünftigen Dokumentgliederungen allerdings nicht auffällt.

Eine echte logische Hierarchie ist in LaTeX durch die Einteilung der Dokumentbeschreibung in ineinandergeschachtelte Gruppen gegeben.

Gruppen werden abgegrenzt durch die Umgebungsklammerung `\begin{...}` – `\end{...}`, durch geschweifte Klammern `{...}`, durch eckige Klammern `[...]` an bestimmten Positionen wie z. B. hinter `\item` sowie durch einige andere Mechanismen. LaTeX prüft, daß die Gruppeneinteilung korrekt ist, das heißt, daß zu jedem Anfang ein Ende gehört und umgekehrt, daß die Anfangs- und Endemarkierung von derselben Art ist und daß sich Gruppen nicht falsch überlappen wie z. B. in `\item[a{b]c}`.

Die Gruppeneinteilung kann unter anderem durch Schriftwechselbefehle wie z. B. `\bf` oder `\it` sichtbar gemacht werden. Ein solcher Befehl wirkt von der Stelle, an der er steht, bis er durch einen anderen Befehl widerrufen wird oder höchstens bis zum Ende der Gruppe, in der er vorkam. Am Ende der Gruppe tritt wieder die Schrift in Kraft, die am Anfang der Gruppe gegolten hat.

Während die Umgebungsbefehle `\begin{...}` und `\end{...}` also tatsächlich Gruppen bilden, tun dies *nicht* die Gliederungsbefehle `\chapter`, `\section` usw. und der Befehl `\item`, der Einträge in Listen kennzeichnet. Ein Kapitel, ein Abschnitt oder ein Listeneintrag bilden also *keine* Gruppe. Die genannten Befehle haben bis auf das Setzen globaler Variablen (Zähler) einen rein lokalen Charakter.

Weil ein `\section`-Kommando nichts mit einer Gruppeneinteilung zu tun hat, kann es also z. B. im Innern einer `enumerate`-Umgebung vorkommen, ohne daß das von LaTeX als Fehler bewertet wird. Es ist interessant zu beobachten, welche Wirkung eine solcherart „unlogische“ Struktur auf Verweise hat.

Offenbar gibt es eine „aktuelle Bezeichnung“, die sich ähnlich wie die aktuelle Schrift verhält. Befehle wie `\section` oder ein `\item` in einem `enumerate`, die automatisch eine Bezeichnung (Nummernfolge) erzeugen, setzen die aktuelle Bezeichnung auf die von ihnen erzeugte. Am Ende einer Gruppe nimmt die aktuelle Bezeichnung wieder den Wert an, den sie am Anfang der Gruppe hatte. Der Befehl `\label{`*Name*`}`, der ein Verweisziel definiert, bindet den Namen an die gerade gültige aktuelle Bezeichnung. Um Überraschungen zu vermeiden, sollte ein `\label`-Befehl, der mit `\ref` benutzt werden soll, direkt hinter dem Befehl stehen, der die Bezeichnung erzeugt, auf die man zu verweisen gedenkt. (Also z. B. `\label` direkt nach `\section` anstatt irgendwo im Innern des Abschnitts.)

3.4 Standard Generalized Markup Language (SGML)

3.4.1 Einführung

Ende der sechziger Jahre wurde von der Firma IBM unter der Leitung von Charles Goldfarb die Sprache GML als anwendungsunabhängiges Format zum Austausch von Daten entwickelt. Diese Sprache wurde unter führender Beteiligung von Goldfarb zu SGML [GR90, Bry88] weiterentwickelt, das schließlich im Oktober 1986 als ISO-Standard 8879 veröffentlicht wurde [ISO86]. Im Juli 1988 wurde dieser Standard leicht verändert und erweitert.

Die Sprache SGML dient dazu, logische Strukturen für Dokumente zu definieren und Dokumente mitsamt ihrer logischen Struktur aufzuschreiben. Eine SGML-Dokumentbeschreibung besteht also aus zwei Teilen: einer Beschreibung der generischen logischen Struktur (die auch aus einem Verweis auf die generische Struktur einer ganzen Dokumentklasse bestehen kann) und der spezifischen Dokumentbeschreibung, die aus dem Inhalt des Dokuments und Markierungen zur spezifischen logischen Struktur besteht.

In SGML wird nicht festgelegt, wie das Dokument formatiert werden soll. Im Prinzip kann ein in SGML beschriebenes Dokument auf viele verschiedene Weisen gesetzt werden. Es gibt allerdings, wie wir sehen werden, einige „Nischen“ in der SGML-Beschreibung, in denen Formatieranweisungen untergebracht werden können. Diese sind allerdings für SGML bedeutungslose Zeichenreihen, die nicht auf Korrektheit oder Konsistenz geprüft werden können.

Die spezifische Dokumentstruktur heißt in SGML *Dokumentinstanz*. Abstrakt gesehen handelt es sich um einen Baum, dessen Knoten mit Markierungen versehen sind, die angeben, was die logische Natur der betreffenden Unterbäume ist. In der

Beschreibung eines Buches könnte z. B. die Wurzel des Baumes mit „Buch“ markiert sein und gewisse Knoten in der Nähe der Wurzel mit „Vorwort“ oder „Kapitel“, während Knoten tief unten im Baum mit „Textabschnitt“ oder „Fußnote“ markiert sein könnten. Auf unterster Ebene beinhaltet der Baum Textstücke, die den eigentlichen Inhalt des Dokuments ausmachen. Die Knoten des Baumes können außerdem sogenannte *Attributwerte* enthalten. Dies sind Zusatzinformationen, die nicht zum eigentlichen Inhalt gehören, z. B. Namen für Verweise oder (für SGML bedeutungslose) Formatieranweisungen.

Die Definition der generischen logischen Struktur heißt in SGML *Dokumenttypdefinition, DTD*. Die in einer DTD definierten Arten von Dokumentbestandteilen werden in SGML *Elementtypen* genannt. Die Namen der Elementtypen dienen zur Markierung des die Dokumentinstanz darstellenden Baums. Die DTD legt auch fest, welche Attribute zu den einzelnen Elementtypen gehören und wie sich jeder Elementtyp aus anderen zusammensetzt.

Die DTD (Dokumenttypdefinition) legt also die folgenden Eigenschaften fest:

- die Menge der Elementtypen,
- für jeden Elementtyp den Aufbau aus anderen Elementtypen,
- für jeden Elementtyp die möglichen Attribute und deren Wertebereiche sowie Vorbelegungswerte.

Der markierte Strukturbaum, der die Dokumentinstanz ausmacht, muß natürlich auf irgendeine Weise in den Rechner eingegeben werden können. Dies erfolgt in linearisierter Form als eine *konkrete Dokumentbeschreibung*, die aus Text mit eingestreuten Marken besteht. Zu jedem Elementtyp gibt es eine Anfangs- und eine Endmarke, die in der konkreten Dokumentbeschreibung benutzt werden, um eine Instanz dieses Elementtyps abzugrenzen.

Dokumentbeschreibungen in SGML können von SGML-*Parsern* verarbeitet werden. Ein SGML-Parser liest zunächst die Dokumenttypdefinition. Dann prüft er, ob die folgende konkrete Dokumentbeschreibung (Text mit Marken) mit der Typdefinition konsistent ist, und übersetzt sie in die eigentlich gemeinte Baumstruktur, die *Elementstruktur* des Dokuments.

Vor der eigentlichen Dokumenttypdefinition wird festgelegt, wie die Anfangs- und Endmarken der Elemente konkret aussehen. In der Standardfestlegung haben die Anfangsmarken die Gestalt <*Name Attributsetzungen*>, während die Endmarken so aussehen: </*Name*>. Wenn z. B. der Typname von Kapiteln „chapter“ ist, dann werden Kapitel in der konkreten Dokumentbeschreibung mit `<chapter>` eingeleitet und mit `</chapter>` beendet. Diese Standardfestlegung kann geändert werden, d. h. statt der Begrenzer <, </ und > können auch irgendwelche anderen Zeichenfolgen benutzt werden. Dies wird jedoch nicht empfohlen, da es bei menschlichen Lesern von konkreten SGML-Dokumentbeschreibungen zu Verwirrung führen kann.

Es gibt auch eine Fülle von Regeln über mögliche Abkürzungen in der konkreten Dokumentbeschreibung. Endmarken etwa können weggelassen werden, wo sie redundant wären, z. B. am Ende von Listenelementen.

```
<list>
  <item>   ein Listenelement
  <item>   noch ein Listenelement
</list>
```

Auch Anfangsmarken können weggelassen werden, wenn sie aus dem Kontext rekonstruiert werden können. Auch wenn die Marken nicht ganz wegfallen können, dann kann doch oft der Elementname in der Marke weggelassen werden, was zu leeren Anfangsmarken `<>` und Endmarken `</>` führt. Es gibt noch weitere eigenartige Abkürzungsregeln, die allerdings den optischen Eindruck der Balancierung von spitzen Klammern zerstören: `<a><b>` darf zu `<a<b>` abgekürzt werden und `<a>bla</>` zu `<a/bla/`. Wir werden in diesem Buch nicht weiter auf solche Abkürzungsmöglichkeiten eingehen.

3.4.2 Elementtypdefinitionen

Eine Elementtypdefinition hat (etwas vereinfacht) folgende Syntax:

`<!ELEMENT` *Typname Aufbau* `>`

Sie führt einen neuen Elementtyp ein, indem sie den Namen des Typs und seinen inneren Aufbau aus anderen Typen spezifiziert. Die oben angegebene Syntax ist insoweit vereinfacht, als daß eine richtige SGML-Elementtypdefinition noch zusätzlich die Information enthält, ob die Anfangs- und Endmarken dieses Typs weggelassen werden dürfen.

Der Aufbau des Elementtyps wird entweder durch ein Schlüsselwort bezeichnet oder durch eine „Inhaltsmodellgruppe“ (englisch *content model group*) beschrieben. Dabei handelt es sich im wesentlichen um die in der Informatik bekannten *regulären Ausdrücke* (siehe Abschnitt 3.7.2). In SGML kann ein regulärer Ausdruck A eine der folgenden möglichen Strukturen haben, wobei B, B_1 usw. wieder reguläre Ausdrücke sind:

$(B_1$, … , $B_n)$	Liste von Bestandteilen, Reihenfolge fest
$(B_1$ & … & $B_n)$	Zusammenfassung von Bestandteilen, Reihenfolge beliebig
$(B_1$ | … | $B_n)$	Alternativen
B ?	Optionaler B-Bestandteil
B +	Nichtleere Liste von B-Bestandteilen
B *	Eventuell leere Liste von B-Bestandteilen
Name	ein vom Benutzer definierter Typname
#PCDATA	Text ohne Marken, d. h. ohne weitere logische Struktur

Die Zusammenfassung mit beliebiger Reihenfolge kann als Abkürzung angesehen werden:

$$(a \,\&\, b) \;=\; ((a,b) \mid (b,a))$$

Für #PCDATA gibt es keine Anfangs- und Endmarken. Jedes Stück reinen Textes zwischen zwei beliebigen Marken wird vom SGML-Parser als #PCDATA verstanden.

Beispiel: Die folgende Elementtypdefinition führt einen Dokumenttyp *article* ein.

```
<!ELEMENT  article  (title, abstract, section+, biblio?)>
<!ELEMENT  abstract (par) >
<!ELEMENT  section  (title, (par | fig)*) >
<!ELEMENT  fig      (graphics & caption) >
<!ELEMENT  caption  (par) >
<!ELEMENT  title    (par) >
<!ELEMENT  par      (#PCDATA | citation | figref)+ >
...
```

Ein Artikel besteht also aus einem Titel, einer Zusammenfassung (`abstract`), einer nichtleeren Folge von Abschnitten (`section`) sowie einer Bibliographie (einer Sammlung von Literaturhinweisen), die aber auch fehlen kann. Ein Abschnitt besteht aus einem Titel sowie einer eventuell leeren Folge von Absätzen (Paragraphen, `par`) oder Abbildungen (`fig`). Eine Abbildung besteht aus einem eigentlichen Bild (`graphics`) und der Bildunterschrift (`caption`), wobei der Autor in der konkreten Dokumentbeschreibung die Bildunterschrift vor oder nach dem eigentlichen Bild angeben kann. Ein Absatz besteht aus einer nichtleeren Folge von reinen Textstücken (#PCDATA), Zitaten (`citation`) und Verweisen (`figref`) auf Abbildungen. Die Typdefinition ist insoweit unvollständig, als daß die Definitionen von „biblio“, „graphics“, „citation“ und „figref“ fehlen.

Nach obigen Typdefinitionen können innerhalb von Abbildungen (wegen `fig` $\supset$ `caption` $\supset$ `par` $\supset$ `figref`) Verweise auf Abbildungen vorkommen. Wenn dies aus irgendwelchen Gründen unerwünscht ist, dann kann man spezifizieren

```
<!ELEMENT   fig   (graphics & caption)   -(figref)>
```

Dadurch werden alle Vorkommen von „figref“ im Innern von „fig“ verboten; auch solche, die in der Hierarchie beliebig viele Stufen unterhalb von „fig“ liegen.

Wenn zwei oder mehrere Elementtypen dieselbe Struktur haben, dann kann man ihre Definitionen zu `<!ELEMENT` $(t_1|\ldots|t_n)$ *Aufbau*`>` zusammenfassen.

Der Aufbau eines Typs kann anstelle eines regulären Ausdrucks wie oben auch durch `EMPTY` oder `CDATA` beschrieben werden. Dies sind keine atomaren Ausdrücke wie z. B. #PCDATA, sondern spezielle Schlüsselwörter, die nicht in regulären Ausdrücken geschachtelt vorkommen können.

Im Verweis auf ein Bild („figref“) kann der Autor keinen Inhalt angeben, da der Verweis während des Formatierens automatisch erstellt wird. Daher wird spezifiziert:

```
<!ELEMENT  figref  EMPTY>
```

Ein solcher Typ hat keine Endmarke, da es keinen Inhalt gibt, der dadurch beendet werden könnte. Die Information, auf welches Bild verwiesen wird, wird über das Setzen eines Attributs mitgegeben. Ein Verweis auf ein Bild mit Fröschen sieht also z. B. so aus: `<figref to = Froschbild>`.

Wenn das eigentliche Bild aus Anweisungen an den Formatierer besteht, die vom Autor in die Dokumentbeschreibung aufgenommen werden, aber keine für SGML sinnvolle Struktur haben, dann kann man spezifizieren:

```
<!ELEMENT  graphics  CDATA>
```

Mit CDATA wird eine (fast) beliebige Folge von Zeichen beschrieben, die der SGML-Parser unverändert übernimmt. Er hält nur nach der Zeichenfolge `</graphics>` Ausschau. Daher darf diese Endmarke auch nicht wegfallen. (Bei #PCDATA wird auch nach beliebigen Anfangsmarken Ausschau gehalten.)

Wenn das eigentliche Bild mit Hilfe eines unabhängigen Graphikprogramms erzeugt wurde und in einer eigenen Datei abgespeichert ist, dann kann man spezifizieren:

```
<!ELEMENT  graphics  EMPTY>
```

und den Namen der Bilddatei als Attribut übergeben (siehe Ende des nächsten Abschnitts).

3.4.3 Attributlistendefinitionen

In SGML kann jedem Elementtyp eine Liste von *Attributen* zugeordnet werden. In der spezifischen logischen Struktur gehört dann zu jedem dieser Attribute ein Wert. Im Sinne der logischen Dokumentstruktur haben Attribute die folgenden drei Hauptanwendungen:

1. Dokumentteile können durch ein Attribut mit einem symbolischen Namen versehen werden, der benutzt werden kann, um auf diese Teile zu verweisen;
2. Verweisobjekte können mit dem symbolischen Namen derjenigen Dokumentteile versorgt werden, auf die sie verweisen sollen;
3. Dateieinbindungen können mit dem (symbolischen) Namen der Datei versorgt werden, die sie einbinden sollen.

Außerdem können beliebige Zusatzinformationen oder Anweisungen in Attributen gespeichert werden. Für SGML sind diese Informationen bedeutungslose Zeichenfolgen. Ein Formatierer, der SGML-Dokumentbeschreibungen bearbeiten kann, kann sie jedoch hoffentlich verstehen und benutzen.

Eine Attributlistendefinition hat die folgende Struktur:

<!ATTLIST Typname Attributname_1 Wertebereich_1 $\text{Vorbelegungswert}_1$
$\vdots$ $\vdots$ $\vdots$
Attributname_n Wertebereich_n $\text{Vorbelegungswert}_n$>

Der Typname gibt an, zu welchem Elementtyp die Attributliste gehört. Es folgt die Liste der Attributnamen mit Wertebereichen und Vorbelegungswerten. Innerhalb der Anfangsmarken des Elementtyps können Attributwerte angegeben werden. Die Attribute, für die keine Werte angegeben werden, nehmen die Vorbelegungswerte an.

Beispiel:

```
<!ATTLIST  article   status   (draft | final)  draft
                    version  NUMBER           #REQUIRED
                    date     NUMBERS          #IMPLIED  >
```

Der Wertebereich wird entweder durch Aufzählung angegeben (wie bei `status`) oder ist einer von mehreren vordefinierten SGML-Wertebereichen:

`ID`	Definition eines eindeutigen Namens (für ein Verweisziel)
`IDREF`	Benutzung eines eindeutigen Namens (in Verweisen)
`ENTITY`	Name eines externen Objekts (meist eine Datei)
`NUMBER`	eine Zahl (d. h. eine Folge von Ziffern)
`NUMBERS`	eine Liste von Zahlen
`NAME`	ein String, der lexikalisch einem SGML-Namen entspricht
`NAMES`	eine Liste von solchen Strings
`NUTOKEN`	ein String aus Ziffern und Buchstaben, der mit einer Ziffer beginnt
`NUTOKENS`	eine Liste von solchen Strings
`CDATA`	eine beliebige Zeichenfolge

Dabei entsprechen die ersten drei Wertebereiche den oben angegebenen drei „logischen“ Anwendungen von Attributen. Die restlichen Bereiche sind solche für Zusatzinformationen oder Anweisungen. Wie man sieht, sind diese Bereiche eher über die äußere Erscheinungsform als über die mögliche Bedeutung von Werten

definiert. Eine Dezimalzahl wie „3.14“ ist z. B. kein Element von NUMBER, da sie wegen des Dezimalpunktes nicht als Folge von Ziffern dargestellt wird. Wenn ein Attribut Dezimalzahlen als Wert annehmen können soll, dann muß sein Wertebereich als NUTOKEN deklariert werden. Ähnliches gilt für Attribute, die Längen oder Abstände wie z. B. „3mm“ oder „1.4inch“ bezeichnen sollen, denn es gibt keinen eigenen Wertebereich für Längenangaben. Der Nachteil ist, daß Attribute mit dem Wertebereich NUTOKEN auch Werte wie „17K3Xq“ annehmen dürfen. Man fragt sich, warum SGML überhaupt einen Versuch unternimmt, die für es irrelevanten Werte zu klassifizieren, und wieso das Ergebnis dieser Klassifikation dann so eigenartig ausgefallen ist.

Für den Vorbelegungswert gibt es – wie an dem obigen Beispiel zu sehen ist – unter anderem zwei spezielle Möglichkeiten: #REQUIRED bedeutet, daß auf jeden Fall ein Attributwert angegeben werden muß, und #IMPLIED bedeutet, daß ein fehlender Attributwert vom System bestimmt wird.

Attributwerte werden innerhalb der Anfangsmarke ihres Elements wie folgt gesetzt:

< Typname $\text{Attributname}_1 = \text{Wert}_1$ $\text{Attributname}_2 = \text{Wert}_2$ ... >

Damit kann der Anfang eines Artikels z. B. wie folgt aussehen:

```
<article version = 7>
```

Das System setzt dann für „status“ den Vorbelegungswert „draft“ ein und für „date“ einen berechneten Wert, z. B. das Datum von heute. Ein anderer möglicher Anfang ist

```
<article status = final version = 11 date = "11 11 2011">
```

Dabei ist zu beachten, daß eine Zahlenfolge in Anführungszeichen eingeschlossen werden muß. Es gibt in SGML komplizierte Regeln dafür, welche Attributwerte so eingeschlossen werden müssen, bei welchen es egal ist und welche auf keinen Fall eingeschlossen werden dürfen. Es gibt außerdem eine Zusatzregel, die besagt, daß Attributnamen weggelassen werden können, wenn der angegebene Attributwert einer aus einem aufgezählten Wertebereich ist, so daß der Attributname sich eindeutig aus dem Attributwert ergibt. Obiges Beispiel darf also abgekürzt werden zu

```
<article   final   version = 11   date = "11 11 2011">
```

Außerdem dürfen die Attributsetzungen beliebig vertauscht werden; es ist also auch möglich, die Versionsnummer ans Ende zu schieben.

Bilder und die Verweise darauf können mit Hilfe von Attributen wie folgt spezifiziert werden:

```
<!ELEMENT  fig       (graphics & caption) >
<!ATTLIST  fig       figname  ID       #IMPLIED >
<!ELEMENT  graphics  EMPTY >
<!ATTLIST  graphics  xsize    NUMBER   #IMPLIED
                     ysize    NUMBER   #IMPLIED
                     file     ENTITY   #REQUIRED >
<!ELEMENT  figref    EMPTY >
<!ATTLIST  figref    to       IDREF    #REQUIRED >
```

Statt `ID` und `IDREF` hätte man auch `NAME` angeben können, aber dann würde der SGML-Parser nicht testen, ob die definierten Namen eindeutig sind und jeder benutzte Name definiert ist. Bei `IDREF` sollte immer `#REQUIRED` angegeben werden, da Verweise ohne die Information, worauf verwiesen wird, ziemlich sinnlos sind. Bei `ID` kann `#REQUIRED` oder `#IMPLIED` angegeben werden. Letzteres erspart es dem Autor, Namen für Bilder zu erfinden, auf die im Text nie verwiesen wird.

Das Bild eines Froschs kann also wie folgt angegeben werden:

```
<fig  figname = Froschbild>
  <graphics  file = "bilder/frosch">
  <caption>Europ"aischer Laubfrosch (Hyla arborea)</caption>
</fig>
```

Das eigentliche Bild befindet sich in der Datei namens „bilder/frosch“. Aus den Angaben in dieser Datei berechnen sich die Werte von „xsize“ und „ysize“. Weil „graphics“ als `EMPTY` deklariert ist, gibt es keine Endmarke `</graphics>`. Ein Verweis auf dieses Bild kann mit `<figref to = Froschbild>` erfolgen, wie oben bereits erwähnt.

Das obige Beispiel ist allerdings in zweierlei Hinsicht vereinfacht. Erstens kann der Wert des Attributs „file“ nicht direkt ein Dateiname sein, sondern muß ein SGML-Name sein, der in einer eigenen „entity“-Deklaration an den wirklichen Dateinamen gebunden wurde. Zweitens gibt es in SGML nicht die einfache LaTeX-Schreibweise `"a` für den Umlaut „ä“. Stattdessen muß `&auml;` benutzt werden, so daß die Bildunterschrift im obigen Beispiel richtig

```
Europ&auml;ischer Laubfrosch (Hyla arborea)
```

heißt, was die Lesbarkeit der Dokumentbeschreibung nicht gerade fördert.

3.4.4 Weiterverarbeitung von SGML-Dokumenten

SGML erlaubt nur die Beschreibung der logischen Struktur von Dokumenten. Es macht keinerlei Aussagen über die gewünschte graphische Erscheinungsform der einzelnen Elementtypen. Selbst die Zählweise von Kapiteln, Abschnitten usw. sowie von Listeneinträgen bleibt undefiniert. Zwar können Formatieranweisungen

in Attributen untergebracht werden, doch sie sind für SGML nur bedeutungslose Zeichenfolgen.

Im Prinzip muß für jeden Dokumenttyp eigens die graphische Erscheinungsform jedes einzelnen Elementtyps definiert werden. Dazu könnte man sich folgendes Vorgehen vorstellen: Der Formatierer ist durch eine Menge von sogenannten *Prozeduren* realisiert, je eine für einen Elementtyp. Diese Prozeduren können zusätzlich durch die Attribute zu ihrem Elementtyp gesteuert werden. Die zu einem Elementtyp gehörende Prozedur beschreibt die Verarbeitung von Dokumentbestandteilen dieses Typs. Verschiedene Formatierer, z. B. für verschiedene Stile, sind durch verschiedene Mengen von Prozeduren realisiert. Außer Formatierern können auch andere Verarbeitungsprogramme auf diese Weise definiert werden, z. B. Übersetzer von einem Dokumenttyp in einen anderen. Je nach der gewählten Implementierungssprache und der Verfügbarkeit der Prozeduren gibt es beim prozeduralen Ansatz Probleme mit der Portierbarkeit von Dokumenten.

Um die Weiterverarbeitung von SGML-Dokumenten maschinenunabhängig beschreiben zu können, wurde die Sprache DSSSL entwickelt (siehe Abschnitt 3.6). Es handelt sich dabei um einen mächtigen und komplizierten Beschreibungsformalismus, der es erlaubt, graphische Strukturen generisch zu definieren und die Abbildung von logischer in graphische Struktur für beliebige Dokumenttypen zu spezifizieren.

Wie gesagt ist DSSSL sehr mächtig, aber auch sehr komplex. Eine viel einfachere Lösung wurde durch Beschränkung auf einen einzigen festen Dokumenttyp erzielt, dessen Elementtypen eine feste graphische Struktur erhielten. Das Ergebnis ist die Sprache HTML, die im nächsten Abschnitt beschrieben wird.

3.5 HyperText Markup Language (HTML)

Die Benutzung von HTML

Wie SGML ist HTML [Tol96] eine Sprache, die nicht von einem speziellen dokumentverarbeitenden System abhängig ist. HTML-Dokumentbeschreibungen können mit Hilfe beliebiger Texteditoren, spezialisierter Editoren oder interaktiver Systeme erzeugt werden. Es gibt mehrere verschiedene Dienstprogramme, sogenannte *Browser*, die HTML-Dokumente formatieren und am Bildschirm anzeigen sowie auf Wunsch ausdrucken.

Die Anzeige am Bildschirm ist der eigentliche Sinn eines HTML-Dokuments. Als besonderes Merkmal bietet HTML nämlich die Möglichkeit an, aktive Verweise zu definieren, die sich auf Dokumente irgendwo in der Welt beziehen. Daher bilden HTML-Dokumente das Rückgrat des *World Wide Web (*WWW*)* [DR95, Fis96]. Das WWW kann man sich als ein über die ganze Welt verteiltes Dokument vorstellen, das durch die aktiven Verweise in HTML-Dokumenten zusammengehalten wird. Außer Dokumentteilen, die in HTML geschrieben sind, enthält das WWW auch

anders dargestellte Schriftdokumente, Bilder, Animationen, akustische Dokumente und Videos. Diese wirken jedoch normalerweise als „Sackgassen“, da von ihnen aus mangels aktiver Verweise der Übergang zu weiteren Teilen des WWW nicht mehr möglich ist. Die meisten WWW-Browser erlauben es jedoch, einen einmal benutzten aktiven Verweis auch wieder in umgekehrter Richtung zu benutzen.

HTML **und** SGML

Die Sprache HTML ist aus SGML hervorgegangen, indem einerseits ein fester Dokumenttyp zu Grunde gelegt wurde und andererseits den Elementtypen dieses Dokumenttyps eine feste graphische Erscheinungsform gegeben wurde. Eine Dokumentbeschreibung in HTML sieht daher auf den ersten Blick so aus wie eine in SGML. Bei näherem Hinsehen fällt auf, daß die Dokumenttypdefinition DTD am Anfang fehlt, da ja von vorneherein eine feste DTD angenommen wurde. Diese beschreibt eine feste Menge von Elementtypen mit ihren Attributen. HTML ist immer noch in der Entwicklung: es werden ständig neue Elementtypen und neue Attribute hinzugefügt.

Eine SGML-DTD beschreibt nicht nur, welche Elementtypen es gibt, sondern auch, wie sich jeder Typ aus anderen aufbaut. Darauf wird jedoch in HTML weniger Wert gelegt; die vorhandenen Elementtypen können ziemlich frei ineinander geschachtelt werden. Sogar Hierarchieverletzungen wie

```
<B> Dieser Text <I> ist nicht </B> korrekt geklammert. </I>
```

führen nicht zu einem Fehler. Sie sollten allerdings vermieden werden, da die zugehörige graphische Struktur nicht wohldefiniert ist.

Es gibt ferner in HTML viele Gebilde, die wie SGML-Anfangsmarken aussehen, aber kaum noch zur logischen Strukturierung beitragen, sondern eine überwiegend graphische Bedeutung haben. Wir werden im folgenden solche Gebilde meist als Kommandos bezeichnen.

Einige einfache HTML**-Kommandos**

Die folgende Liste ist nicht vollständig. Zu den meisten Gruppen von Kommandos gibt es weitere verwandte.

- `<H1> Text </H1>`
  Der Text wird als Überschrift betrachtet und groß und fett gesetzt. Die Überschrift wird nicht automatisch numeriert.
- `<H2> Text </H1>`
  Auch hier wird der Text als Überschrift gesetzt, allerdings weniger groß und fett als bei `<H1>`. Ähnlich wie bei *FrameMaker* und LaTeX wird durch diese Überschriften *keine* hierarchische logische Einteilung in Kapitel und Unterkapitel erzeugt. Es entsteht nur der graphische Anschein einer solchen Einteilung.

- `<EM> Text </EM>`
  Der Text wird hervorgehoben (Emphasis = Hervorhebung). Die meisten Browser benutzen *kursive Schrift* für eine Hervorhebung.
- `<I> Text </I>`
  Der Text wird „*italic*“, d. h. kursiv geschrieben. Während `<EM>` einen logischen Stil bezeichnet, ist `<I>` eine rein graphische Auszeichnung.

`<P>` bezeichnet einen Paragraphenwechsel. Er wird graphisch durch einen Zeilenwechsel mit einem vertikalen Zwischenraum angezeigt.

`<BR>` bezeichnet einen Zeilenwechsel ohne vertikalen Zwischenraum.

`<HR>` erzeugt einen horizontalen Trennstrich quer über die Seite.

`<` erzeugt das Zeichen „<“.

`&auml;` erzeugt ein „ä“.

`&szlig;` erzeugt ein „ß“.

An den letzten beiden Beispielen ist zu sehen, daß HTML von SGML die umständliche Notation für Sonderzeichen geerbt hat.

Listenumgebungen

In HTML gibt es drei Arten von Listenumgebungen, die den drei Arten von LaTeX entsprechen: die ungeordnete Liste `UL` entspricht `itemize`, die geordnete Liste `OL` entspricht `enumerate` und die Definitionsliste `DL` ähnelt `description`. In Abbildung 3.5 werden Beispiele für HTML-Listen gezeigt, die den LaTeX-Listen in Abbildung 3.4 entsprechen. Wie in LaTeX können die Listenumgebungen ineinander geschachtelt werden. Bei den `UL`-Listen wird dabei das Einleitungszeichen variiert. Bei `OL`-Listen zählen die inneren Listen korrekt von 1 an, variieren aber nicht wie in LaTeX die graphische Erscheinungsform des Zählers.

Die speziellen Eigenschaften von HTML

Die im folgenden geschilderten HTML-Kommandos entfalten ihre Wirkung nur bei Betrachtung am Bildschirm.

Text, der zwischen `<title>` und `</title>` eingeschlossen ist, ist nicht als Teil des HTML-Dokuments im Anzeigefenster sichtbar, sondern bildet die Überschrift des ganzen Fensters.

Die wichtigste Besonderheit von HTML sind jedoch die aktiven Verweise. Sie werden wie folgt definiert:

`<A HREF = "`*Verweisziel*`">` *Verweistext* `</A>`

```
<UL>
<LI> Der Winter ist eine Jahreszeit,
     in der es kalt ist.
<LI> Im Sommer ist es
     dagegen hei&szlig;.
</UL>
```

- Der Winter ist eine Jahreszeit, in der es kalt ist.
- Im Sommer ist es dagegen heiß.

```
<OL>
<LI> Der Winter ist eine Jahreszeit,
     in der es kalt ist.
<LI> Im Sommer ist es
     dagegen hei&szlig;.
</OL>
```

1. Der Winter ist eine Jahreszeit, in der es kalt ist.
2. Im Sommer ist es dagegen heiß.

```
<DL>
<DT> Winter: <DD> eine Jahreszeit,
                  in der es kalt ist.
<DT> Sommer: <DD> hei&szlig;e
                  Jahreszeit.
</DL>
```

Winter:
eine Jahreszeit, in der es kalt ist.

Sommer:
heiße Jahreszeit.

Abbildung 3.5: Listenumgebungen in HTML

Das Verweisziel ist meist eine ganze Datei. Der Wert des Attributs HREF muß exakt beschreiben, wie diese Datei zu finden ist. Im Gegensatz zu den Verweisen in *FrameMaker* und LaTeX wird die graphische Erscheinungsform des Verweises nicht automatisch aus dem Verweisziel hergeleitet, sondern als Verweistext explizit angegeben. Eine HTML-Dokumentbeschreibung könnte z. B. den folgenden Satz beinhalten:

```
Die Sprache <A HREF =
  "http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html">
HTML </A> erlaubt die Definition aktiver Verweise.
```

Ein typischer Browser stellt diesen Satz am Bildschirm so dar:

> Die Sprache HTML erlaubt die Definition aktiver Verweise.

Die Unterstreichung zeigt an, daß sich hinter dem Wort „HTML“ ein aktiver Verweis verbirgt. Wenn dieses Wort mit der Maus selektiert wird, dann wird die als Verweisziel angegebene Datei herbeikopiert und am Bildschirm angezeigt. Sie enthält eine HTML-Kurzbeschreibung. Die angegebene Dateiadresse zeigt, daß sich die Datei in einem Rechner des Nationalen Zentrums für Supercomputer-Anwendungen (ncsa) der Universität von Illinois in Urbana-Champaign (uiuc) in den USA (edu) befindet.

Statt einer ganzen Datei kann das Verweisziel auch nur ein bestimmter Teil einer Datei sein. Ein Teil, der als Verweisziel benutzt werden soll, muß zunächst markiert

werden. Eigenartigerweise erfolgt die Markierung ebenfalls durch das Kommando A, aber mit einem anderen Attribut:

<A NAME = "*Bezeichnung*"> *Dateiteil* </A>

Ein aktiver Verweis auf das so markierte Ziel wird gesetzt, indem an den Dateinamen die Bezeichnung des Ziels, durch ein # getrennt, gehängt wird. Wenn das Ziel in derselben Datei liegt wie der Verweis, kann der Dateiname entfallen.

Schlußbemerkungen

Die Sprache HTML ist dafür gedacht, Dokumente sehr schnell in einer einigermaßen ansprechenden graphischen Form am Bildschirm anzuzeigen. Sie ist daher wesentlich einfacher und schwächer als die von den Programmen *FrameMaker* und LaTeX definierten Sprachen. Von HTML-Formatierern wird nicht erwartet, daß sie automatisch Kapitel numerieren, Inhaltsverzeichnisse erzeugen, Wörter am Zeilenende trennen oder für einen optimalen Zeilenumbruch sorgen.

Die Sprache HTML unterstützt keine Erweiterungen; HTML-Benutzer können keine neuen Listenumgebungen, Paragraphenarten, Zeichenformate oder dergleichen definieren und benutzen. Andererseits wird HTML selbst ständig durch die Einführung weiterer Kommandos und Attribute erweitert.

3.6 Document Style Semantics and Specification Language (DSSSL)

3.6.1 Einführung

Die Sprache DSSSL [ISO96] wurde entwickelt, um die Weiterverarbeitung von Dokumenten zu beschreiben, die in hierarchischer logischer Struktur vorliegen. Dabei wurde hauptsächlich daran gedacht, daß diese Struktur in SGML spezifiziert ist. DSSSL kann allerdings auch an ODA angepaßt werden.

Eine in SGML beschriebene logische Struktur sagt nichts darüber aus, wie diese Struktur weiterverarbeitet werden soll und wie die spätere graphische Erscheinung des Dokuments sein wird. Diese Lücke wird durch DSSSL geschlossen.

In DSSSL beschreibt man einerseits die graphische Dokumentstruktur, d. h. den Aufbau von Seiten aus Bestandteilen wie Spalten, Kopf- und Fußzeilen (Definition der virtuellen Oberfläche, Virtual Surface Definition „VSD“), und andererseits die Zuordnung von logischen zu graphischen Elementen (Zuordnung von Quelle zu Darstellung, Source-to-Presentation Association Specification „STPAS“). Dazu kommen noch Stil- und Formatierungsregeln für den Zeilen- und Seitenumbruch.

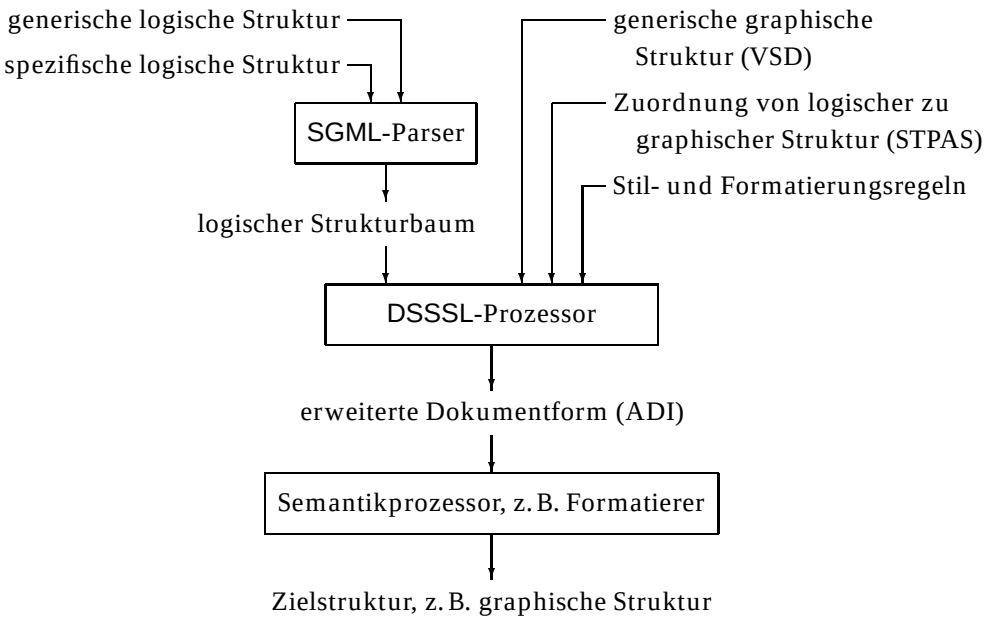

Abbildung 3.6: Das DSSSL-*Verarbeitungsmodell*

3.6.2 Das Verarbeitungsmodell von DSSSL

Die Verarbeitung eines SGML-Dokuments soll in drei Schritten erfolgen (siehe Abbildung 3.6). Im ersten Schritt liest ein SGML-*Parser* das aus einer Dokumenttyp-Definition und einem gemäß dieser Definition strukturierten Inhaltsteil bestehende *Quelldokument* und erzeugt daraus die entsprechende hierarchische Struktur. Im zweiten Schritt verwandelt der DSSSL-*Prozessor* unter Benutzung der Definition der graphischen Dokumentstruktur (VSD) zusammen mit der Zuordnung Quelle–Darstellung (STPAS) das hierarchisch strukturierte Dokument in eine *erweiterte Dokumentform* (Augmented Document Instance „ADI“). Diese ADI wird im dritten Schritt einem sogenannten *Semantikprozessor* übergeben, der die Anweisungen in der ADI ausführt und das Ergebnisdokument erzeugt. Die DSSSL-Spezifikation gibt dabei nur den Rahmen an. Der Semantikprozessor gehört nicht zum DSSSL-System, sondern ist ein unabhängiges Programm, z. B. ein schon existierender Formatierer wie TEX.

Das Ergebnis der Verarbeitung (Ergebnisdokument) kann tatsächlich in graphischer Struktur vorliegen, die in einer Seitenbeschreibungssprache spezifiziert ist. Es kann sich allerdings auch wieder um ein Dokument in logischer SGML-Struktur handeln, die sich irgendwie von der Ausgangsstruktur unterscheidet.

Die Schnittstelle zwischen dem SGML-Parser und dem DSSSL-Prozessor ist im Gegensatz zu der Schnittstelle zwischen DSSSL-Prozessor und Semantikprozessor standardisiert. Formatierer und DSSSL-Prozessor können Unterprogramme des-

selben Programms bilden und miteinander verzahnt ablaufen, oder sie können unabhängige Programme sein, die nacheinander ablaufen und über eine Datei kommunizieren.

3.6.3 Die Spezifikation der graphischen Dokumentstruktur

Die Elemente der graphischen Dokumentstruktur, genannt *virtuelle Oberflächen*, dienen als Empfänger des Inhalts der Elemente der logischen Struktur. Virtuelle Oberflächen sind Rechtecke, deren Kanten parallel zum Rand der Seite sind. Diese Rechtecke sind wieder hierarchisch angeordnet, z. B. Blatt ⊃ Satzspiegel ⊃ Spalte. Die Spezifikation der virtuellen Oberflächen legt nicht nur die Schachtelung fest, sondern auch eine hierarchisch gesehen horizontale Ordnung auf den Instanzen der Oberflächen. Diese wird benutzt, um festzulegen, in welcher Reihenfolge solche Instanzen mit Inhalt gefüllt werden.

Die Hierarchie der virtuellen Oberflächen wird durch eine Folge von Definitionen eingeführt, die den Definitionen der SGML-Elementtypen ähneln. Die Instanzen von virtuellen Oberflächen haben fünf *Flächenattribute*:

1. ihre *Position* in der umfassenden Oberfläche (Elternoberfläche);
2. ihre *Ausdehnung* in horizontaler und vertikaler Richtung;
3. ihr inneres *Koordinatensystem*;
4. ihren *Inhaltsteil*, das ist eine beliebig geformte Teilfläche, die den Inhalt der Oberfläche aufnimmt;
5. eine *Ausblendmaske*. Der Teil des Inhalts einer Oberfläche, der unter die Maske derselben oder einer umfassenden Oberfläche gerät, wird ausgeblendet und damit unsichtbar.

Flächenattribute können durch exakte Zahlenwerte spezifiziert werden, durch Angabe von kleinen Intervallen, um dem Formatierer etwas Spielraum zu lassen, oder völlig unspezifiziert sein. Im letzten Fall ergeben sie sich erst während des Formatierungsprozesses. Das ist z. B. der Fall, wenn eine virtuelle Oberfläche eine Folge von Instanzen einer anderen virtuellen Oberfläche enthält und die Größe dieser Instanzen und damit ihre Positionierung sich erst durch die Formatierung ergibt.

3.6.4 Die Zuordnung von Quelle zu Darstellung

Diese Zuordnung verknüpft Knoten in der logischen Hierarchie (Elementtypen) mit Knoten in der graphischen Hierarchie (virtuelle Oberflächen). Sie gibt an, in welche Oberfläche ein bestimmter logischer Teil des Dokuments plaziert werden soll, und ermöglicht es, Stilinformationen sowohl an logische Teile als auch an Oberflächen zu binden.

Der *Stil* beschreibt die Plazierung und das Aussehen von Inhalten. Stilattribute können als Teil der Definition der graphischen Dokumentstruktur spezifiziert werden, z. B. die Flächenattribute, oder als Teil der Definition der Quelle-Darstellungs-Zuordnung, z. B. Schriftschnitt, -grad, Farbe und Ausrichtung. Teilweise kann man Attribute sowohl auf der logischen Struktur als auch auf der graphischen Struktur festlegen.

Da der gleiche logische Elementtyp in verschiedenen Kontexten eventuell verschieden zugeordnet werden soll, muß die Möglichkeit bestehen, Elementnamen durch Angabe von Kontextinformation genauer zu qualifizieren. Auch virtuelle Oberflächen können in verschiedenen Kontexten vorkommen und müssen eventuell dann mit verschiedenem Inhalt gefüllt werden. Daher können auch Oberflächennamen mit Kontextinformationen versehen werden.

Es gibt mehrere Möglichkeiten, Kontextinformationen zu spezifizieren:

- durch Angabe eines oder mehrerer umfassender Strukturelemente, z. B. Titel in einem Kapitel, Paragraph in einer Auflistung oder Kopfzeile auf einer linken Seite;
- durch Angabe der Position relativ zu „Geschwistern“ der gleichen Art, z. B. der erste oder letzte Paragraph in einem Kapitel oder die zweite Spalte auf einer Seite;
- durch Angabe der Position relativ zu „Geschwistern“ anderer Art, z. B. Paragraphen direkt vor Auflistungen.
- Die Verarbeitung eines logischen Elements kann auch von dessen Attributwerten abhängen. Wenn ein Abschnitt z. B. ein Attribut `status` mit zwei möglichen Werten `draft` und `final` hat, dann kann die Zuordnung Quelle zu Darstellung zwei Einträge haben, abhängig vom Wert des Attributs `status`.

3.7 Grammatiken

3.7.1 Beschreibung von Textklassen

Nicht nur in der Dokumentverarbeitung, sondern auch in vielen anderen Gebieten der Informatik möchte man für Texte bestimmter Art ihre innere Struktur, man spricht von ihrer *Syntax*, beschreiben. Die Syntax von Programmen einer Programmiersprache wird z. B. sehr viel detaillierter beschrieben als die Syntax von Dokumenten, die uns in diesem Buch interessieren.

Beginnen wir mit einem einfachen Beispiel. Jeder kennt die Syntax von deutschen postalischen *Adressen*. Sie bestehen aus mehreren Teilen, dem *Namen*, dem *Postfach* oder einer Kombination von *Straße* und *Hausnummer*, einer *Postleitzahl* und einem *Ortsnamen*. Gemäß der aktuellen Konvention sollen diese Teile auch in der angegebenen Reihenfolge aufeinanderfolgen. Die einzelnen Bestandteile haben

wieder einen mehr oder weniger festgelegten Aufbau. *Postleitzahlen* bestehen aus genau fünf *Ziffern*, *Hausnummern* aus mindestens einer. Der Aufbau von *Namen* kann variieren; der *Vorname* kann zuerst kommen oder, getrennt durch ein Komma, hinter dem *Nachnamen* stehen. *Nachnamen* können sehr viele Bestandteile haben; man denke etwa an Nölle-Neumann-Maier-Leibnitz. Nach weiterer Zerlegung landet man schließlich und endlich bei den elementaren Bestandteilen; das sind *Buchstaben*, *Ziffern*, Bindestriche, Punkte usw.

Kindern wird irgendwann, wenn sie lesen und schreiben lernen, durch Beispiele und verbale Beschreibung der Aufbau von Adressen beigebracht. Informatiker jedoch müssen häufig den Aufbau von Texten beschreiben, die von einem Informatiksystem verarbeitet werden sollen. Da muß die Beschreibung unmißverständlich sein. Für eine solche Beschreibung benötigt man geeignete Hilfsmittel. In der Informatik werden *Grammatiken* zur Beschreibung von Syntax benutzt.

Wie soll man sich die Beschreibung der syntaktischen Struktur einer Klasse von Texten vorstellen? Oder mit anderen Worten, was muß man in einem solchen Beschreibungsformalismus sagen können? Ein Vorrat an Ausdrucksmitteln könnte der folgende sein:

1. „Die elementaren Bestandteile sind …“
   Solch eine Auflistung der elementaren Bestandteile wurde oben für Adressen gegeben.

2. „Es gibt die folgenden Arten von Textbestandteilen: …“
   In dem Adressenbeispiel gibt es als Arten von Textbestandteilen *Adresse*, *Name*, *Postfach*, *Straße*, *Hausnummer* usw.

   Hinter der Einführung von Namen für Arten von Textbestandteilen steckt ein wichtiger Abkürzungsmechanismus. Man kann nämlich festlegen, daß ein X eine angegebene Struktur hat, und dann überall in der Beschreibung, wo ein Text dieser Struktur auftreten kann, abkürzend ein X hinschreiben.

3. „Hier muß einem X ein Y folgen.“
   Dies wird üblicherweise als XY geschrieben. In Adressen kommen etwa die Kombinationen *Straße* und *Hausnummer* sowie *Postleitzahl* und *Ortsname* vor.

4. „Hier muß ein X stehen; es kann aber auch fehlen.“
   Eine gängige Schreibweise dafür ist $X?$.

5. „Hier kann eine (eventuell leere / nichtleere) Folge von Xen stehen.“
   Wenn die Folge leer sein kann, schreibt man das als X^*, sonst als X^+. *Hausnummern* sind nichtleere Folgen von *Ziffern*, eventuell gefolgt von einem *Buchstaben*; also beschreibt man sie durch $Ziffer^+\ Buchstabe?$.

6. „Hier kann ein X oder ein Y stehen.“
   In Zeichen schreibt man das als $(X \mid Y)$, oder bei mehr als zwei Möglichkeiten allgemeiner als $(X_1 \mid \cdots \mid X_n)$. Wenn man *Ziffer* als abkürzende Bezeichnung für

eines der Zeichen 0, 1, ..., 9 einführen möchte, kann man das durch

Ziffer → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

tun. Vorhin haben wir *Hausnummer* als *Ziffer*$^+$ *Buchstabe*? dargestellt. Möchte man aber ausdrücken, daß Hausnummern nicht mit 0 beginnen dürfen, so kann man ihren Aufbau beschreiben durch

Hausnummer → (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) *Ziffer** *Buchstabe*? .

Beachten Sie, daß jetzt auf die führende Ziffer eine leere Folge von *Ziffern* folgen kann, damit auch einstellige *Hausnummern* beschrieben werden können.

Wir werden später sehen, daß ein Teil dieser Ausdrucksmittel entbehrlich ist.

Beispiel

Elementtypdefinitionen in SGML sind Grammatiken. Kommen wir auf die SGML-Elementtypdefinition für `article` in Abschnitt 3.4.2 zurück. Wir könnten sie verbal so formulieren: Ein `article` besteht aus einem `title`, gefolgt von einem `abstract`, gefolgt von einer Folge von `section`s, eventuell gefolgt von einer `biblio`. Die Namen `title`, `abstract`, `section` und `biblio` sind wieder Abkürzungen für Dokumentteile, deren Aufbau wir angeben müssen. Eine `section` z. B. besteht aus einem `title` gefolgt von einer Folge von Elementen, die jeweils entweder ein `par` oder ein `fig` sind. In einer vollständigen Elementtypdefinition besteht letztlich alles aus atomaren Bestandteilen der Art #PCDATA, das ist reiner Text.

3.7.2 Übliche Terminologie und Notation

Jetzt wollen wir die in der Informatik übliche Terminologie einführen. Die Elemente, aus denen der Text letztlich besteht, bilden das Alphabet der *Terminalsymbole*. Der Name ist jetzt noch wenig intuitiv; er wird später klar werden. Die Festlegung der Terminalsymbole entspricht dem Punkt 1 der Aufzählung von Ausdrucksmitteln in Unterabschnitt 3.7.1.

Die Terminalsymbole in Adressen sind die oben aufgelisteten elementaren Bestandteile, also Buchstaben, Ziffern, Bindestriche und Punkte. In SGML gibt es nur ein einziges Terminalsymbol, nämlich #PCDATA. Man sieht an den beiden Beispielen, daß die Beschreiber eine gewisse Freiheit haben, wie weit herunter sie mit der Beschreibung der Struktur gehen wollen.

Die für gewisse Arten von Textbestandteilen eingeführten Abkürzungen (Punkt 2 der Aufzählung) heißen *Nichtterminalsymbole*. Das sind im Beispiel der Adressen z. B. *Ziffer* und *Hausnummer*, im SGML-Beispiel unter anderem `article`, `title`, `abstract`, `section`, `biblio`, `citation` und `figref`. Eines von den Nichtterminalsymbolen zeichnet man aus und nennt es das *Startsymbol*. Es steht gewissermaßen für die ganze Textsorte, während die anderen Nichtterminale für Teile von

Texten stehen. Bei den Adressen ist das Startsymbol *Adresse* und in dem SGML-Beispiel ist es `article`.

Das Einführen eines Abkürzungssymbols zusammen mit der Beschreibung der Struktur der von ihm bezeichneten Textklasse nennt man *Produktion*. Die Produktion aus dem SGML-Beispiel von oben, die verbal hingeschrieben worden war, hätte üblicherweise die Form

`article` → `title abstract section`* `biblio`?.

Diese Schreibweise deutet an, daß man überall, wo ein Vorkommen von `article` steht, dieses durch die *rechte Seite* `title abstract section`* `biblio`? ersetzen kann.

Ausdrücke, die mit Hilfe der in Punkt 3 bis 6 aufgezählten Operatoren Hintereinanderschreibung, ?, *, + und | aus Terminal- und Nichtterminalsymbolen aufgebaut sind, heißen in der Informatik *reguläre Ausdrücke*. Die oben eingeführten Grammatiken bestehen also aus Produktionen, deren rechte Seiten reguläre Ausdrücke sind; es handelt sich um *Grammatiken mit regulären rechten Seiten* oder kurz *rrS-Grammatiken*. In solchen Grammatiken gibt es zu jedem Nichtterminal genau eine Produktion.

Statt der rrS-Grammatiken findet man oft sogenannte *kontextfreie Grammatiken*. In diesen dürfen die rechten Seiten der Produktionen nur noch aus einer Folge von Terminal- und Nichtterminalsymbolen bestehen; die Operatoren ?, *, + und | sind verboten. Zum Ausgleich darf es zu einem Nichtterminal X hier mehrere verschiedene Produktionen $X \to \alpha_1, \ldots, X \to \alpha_n$ mit verschiedenen rechten Seiten geben. Diese geben verschiedene Alternativen für den Aufbau der von dem Nichtterminal bezeichneten Textsorte an. Statt $X \to \alpha_1, \ldots, X \to \alpha_n$ wird allerdings oft $X \to \alpha_1 \mid \cdots \mid \alpha_n$ geschrieben. Diese Schreibweise hat jedoch nur abkürzenden Charakter; es handelt sich nach wie vor um n Produktionen, deren rechte Seiten keine Alternativzeichen | enthalten.

3.7.3 Die von einer Grammatik erzeugte Sprache

Als nächstes stellt sich die Frage, wie man mit einer Grammatik arbeiten kann. Die Antwort ist, daß man damit Texte aus der beschriebenen Textsorte erzeugen kann und daß man gegebene Texte daraufhin überprüfen kann, ob sie zu dieser Textsorte gehören.

Uns interessiert hier die erste Fähigkeit. Dazu stellt man sich einen von der speziellen Grammatik unabhängigen Ersetzungsmechanismus vor. Gibt man ihm eine Grammatik, die den syntaktischen Aufbau von Texten einer Textsorte beschreibt, so kann er alle Texte dieser Sorte erzeugen. Die Gesamtheit dieser Texte heißt die von der Grammatik erzeugte *Sprache*.

Die Sprache einer kontextfreien Grammatik

Für kontextfreie Grammatiken arbeitet der Ersetzungsmechanismus wie folgt: Er beginnt mit dem Startsymbol und ersetzt es durch eine seiner rechten Seiten. Darin können neben Terminalen auch wieder Nichtterminale stehen. Diese ersetzt er wieder durch eine ihrer rechten Seiten usw. Das alles treibt man so lange, bis kein Nichtterminal mehr irgendwo in dem Text auftritt. Dann hat man einen Text der von der Grammatik beschriebenen Sprache *produziert* (*abgeleitet*).

Dabei gibt es natürlich im allgemeinen Möglichkeiten, verschiedene Texte zu erzeugen; sonst wäre die Textsorte nicht besonders interessant. Sie ergeben sich dadurch, daß die oben aufgelisteten Beschreibungselemente den Ersetzungsprozeß nicht eindeutig festlegen. Immer wenn man ein Vorkommen eines Nichtterminals X ersetzen möchte, kann man zwischen den verschiedenen Produktionen von X wählen. Wenn es mehrere Vorkommen von X gibt, können für sie verschiedene Produktionen gewählt werden.

Jetzt werden übrigens auch die Bezeichnungen Terminal und Nichtterminal klar. Sie stammen aus diesem Ersetzungsprozeß. Wurde einmal ein Terminalsymbol erzeugt, so ist hier keine weitere Ersetzung mehr möglich; der Ersetzungsprozeß hat an dieser Stelle terminiert. Ein Nichtterminal dagegen verschwindet im weiteren Verlauf des Prozesses wieder, da es irgendwann durch die rechte Seite einer seiner Produktionen ersetzt wird. Der Name „kontextfrei“ rührt daher, daß die möglichen Ersetzungen für ein Nichtterminal X nicht von dem Kontext abhängen, in dem X vorkommt.

Wir wollen jetzt eine etwas formalere Notation einführen. Angenommen, α ist eine Folge von Terminalen und Nichtterminalen, die ein Nichtterminal X enthält, also $\alpha = \beta X \gamma$ für geeignete Folgen β und γ. Weiter angenommen, X habe eine Produktion $X \to \delta$. Dann kann ein von α ausgehender *Ableitungsschritt* aus der Ersetzung von X durch δ bestehen. Das Ergebnis ist also $\alpha' = \beta\delta\gamma$. Die Tatsache, daß α' aus α durch einen Ableitungsschritt entstanden ist, wird in Formeln als $\alpha \Rightarrow \alpha'$ ausgedrückt. Die gesamte Ableitung eines Terminalwortes w (einer Folge von Terminalsymbolen) aus dem Startsymbol S kann dann als $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow w$ geschrieben werden.

Beispiel

Unsere Beispielgrammatik hat zwei Terminale a und b und zwei Nichtterminale S und B, von denen S das Startsymbol ist. Die Produktionen sind $S \to SB$, $S \to \varepsilon$, $B \to aB$ und $B \to b$. Das Zeichen ε steht für die leere Folge von Terminalen und Nichtterminalen. Für diese Grammatik gibt es unter anderem die folgenden Ableitungen:

$S \Rightarrow \varepsilon$
$S \Rightarrow SB \Rightarrow B \Rightarrow b$
$S \Rightarrow SB \Rightarrow SBB \Rightarrow BB \Rightarrow BaB \Rightarrow BaaB \Rightarrow Baab \Rightarrow baab$

Man kann zeigen, daß die von dieser Grammatik beschriebene Sprache, d. h. die Menge der von S aus ableitbaren Terminalfolgen, aus allen Worten besteht, die nicht mit a enden, also aus dem leeren Wort ε und allen Worten, die mit b enden.

Die Sprache einer Grammatik mit regulären rechten Seiten

Für eine rrS-Grammatik muß der Ersetzungprozeß modifiziert werden. Ausgangspunkt ist wieder das Startsymbol. Zwischenschritte sind nicht Folgen von Terminalen und Nichtterminalen, sondern reguläre Ausdrücke. Mögliche Ersetzungen in einem regulären Ausdruck sind die folgenden:

1. Ein Nichtterminal kann durch die rechte Seite seiner Produktion ersetzt werden.
2. Ein Teilausdruck der Form $r?$ kann durch r oder durch ε ersetzt werden. Als Teil einer größeren Folge wird dieses ε unsichtbar.
3. Ein Teilausdruck der Form r^+ kann durch eine nichtleere Folge $r \ldots r$ von Kopien von r ersetzt werden. Ein r^* kann zusätzlich durch die leere Folge ε ersetzt werden.
4. Ein Teilausdruck der Form $(r_1 \mid \cdots \mid r_n)$ kann durch einen der Teile r_i ersetzt werden.

Bei diesen Ersetzungen müssen je nach den logischen Erfordernissen Klammern eingefügt oder weggelassen werden. Wir wollen das hier nicht genauer erläutern, da die Einzelheiten zu weit führen würden.

Beispiel

Unsere Beispielgrammatik hat jetzt drei Terminalsymbole a, b und c und ein Nichtterminal S, das dann natürlich das Startsymbol sein muß. Die Produktion von S ist $S \to a\,(S \mid b)^*\,c?$.

$$
\begin{aligned}
S &\Rightarrow a\,(S \mid b)^*\,c? \Rightarrow a\,c? \Rightarrow a \\
S &\Rightarrow a\,(S \mid b)^*\,c? \Rightarrow a\,(S \mid b)\,c? \Rightarrow a\,b\,c? \Rightarrow abc \\
S &\Rightarrow a\,(S \mid b)^*\,c? \Rightarrow a\,(S \mid b)\,c? \Rightarrow a\,S\,c? \Rightarrow \\
&\quad a\,a\,(S \mid b)^*\,c?\,c? \Rightarrow a\,a\,(S \mid b)\,(S \mid b)\,c?\,c? \Rightarrow \cdots \Rightarrow aabb
\end{aligned}
$$

3.7.4 Beseitigung der regulären Operatoren

Grammatiken mit regulären rechten Seiten bieten mehr Beschreibungsmöglichkeiten als kontextfreie Grammatiken. Daher mag man sich fragen, ob sich mit ihnen mehr Sprachen beschreiben lassen. Die Antwort ist nein; zu jeder rrS-Grammatik gibt es eine äquivalente kontextfreie Grammatik und umgekehrt. Dabei heißen zwei Grammatiken *äquivalent*, wenn sie dieselbe Sprache erzeugen.

Aus einer gegebenen kontextfreien Grammatik läßt sich sehr leicht eine äquivalente rrS-Grammatik erzeugen. Man muß nur die verschiedenen Produktionen $A \to \alpha_1$, ..., $A \to \alpha_n$ aus einem Nichtterminal A zu einer rrS-Produktion $A \to (\alpha_1 \mid \cdots \mid \alpha_n)$ zusammenfassen.

Der umgekehrte Weg ist etwas komplizierter. Eine rrS-Grammatik wird dabei schrittweise in eine äquivalente kontextfreie Grammatik umgeformt. (Die Zwischenstufen sind Grammatiken, die noch reguläre rechte Seiten haben, aber schon mehrere Produktionen pro Nichtterminal.) Es gibt dabei die folgenden Umformungsschritte:

1. Ersetze einen regulären Teilausdruck der Form $(r_1 \mid \cdots \mid r_n)$ durch ein neues Nichtterminal A (eines, das bisher noch nicht vorgekommen war). Füge die Produktionen $A \to r_1, \ldots, A \to r_n$ zur Grammatik hinzu.
2. Ersetze einen regulären Teilausdruck der Form $r?$ durch ein neues Nichtterminal A. Füge die Produktionen $A \to r$ und $A \to \varepsilon$ zur Grammatik hinzu.
3. Ersetze einen regulären Teilausdruck der Form r^* durch ein neues Nichtterminal A. Füge die Produktionen $A \to rA$ und $A \to \varepsilon$ zur Grammatik hinzu. Dadurch lassen sich aus A eventuell leere Folgen von Kopien von r erzeugen, und zwar durch $A \Rightarrow \varepsilon$, $A \Rightarrow rA \Rightarrow r$, $A \Rightarrow rA \Rightarrow rrA \Rightarrow rr$ usw. Allgemein kann man durch n-malige Anwendung der ersten Produktion und anschließende einmalige Anwendung der zweiten Produktion eine Folge aus n Kopien von r erzeugen.
4. Bei r^+ sind die neuen Produktionen $A \to rA$ und $A \to r$. Durch diese beiden Produktionen lassen sich aus A nichtleere Folgen von Kopien von r erzeugen, und zwar durch die Ersetzungen $A \Rightarrow r$, $A \Rightarrow rA \Rightarrow rr$, $A \Rightarrow rA \Rightarrow rrA \Rightarrow rrr$ usw.

Bei keinem dieser Transformationsschritte verändert sich die von der Grammatik erzeugte Sprache. Bei r^+ und r^* haben wir versucht, das durch Ableitungen zu begründen. Es wird so lange transformiert, bis keiner der Transformationsschritte mehr anwendbar ist. Dann ist das Ergebnis eine kontextfreie Grammatik.

Beispiel

Wir wollen die rrS-Grammatik mit der einen Produktion $S \to a(S \mid b)^* c?$ aus dem vorigen Beispiel in eine äquivalente kontextfreie Grammatik transformieren.

Zuerst ersetzen wir gemäß Ziffer 3 den Teilausdruck $(S \mid b)^*$ durch das neue Nichtterminal A. Die Ergebnisgrammatik ist

$$S \to aAc?, \quad A \to (S \mid b)A, \quad A \to \varepsilon$$

Nun ersetzen wir gemäß Ziffer 1 den Teilausdruck $(S \mid b)$ durch das neue Nichtterminal B. (A ist jetzt nicht mehr „neu“!) Das Ergebnis ist

$$S \to aAc?, \quad A \to BA, \quad A \to \varepsilon, \quad B \to S, \quad B \to b$$

Schließlich ersetzen wir $c?$ durch C. Damit ergibt sich die kontextfreie Grammatik

$$S \to aAC, \quad A \to BA, \quad A \to \varepsilon, \quad B \to S, \quad B \to b, \quad C \to c, \quad C \to \varepsilon$$

und wir sind fertig.

3.7.5 Reguläre Grammatiken

Wir haben gesehen, daß rrS-Grammatiken und kontextfreie Grammatiken gleichmächtig sind, d. h. mit diesen Grammatikklassen lassen sich dieselben Sprachen definieren. Jetzt wollen wir eine weniger mächtige Grammatikklasse einführen.

Ein Nichtterminal in einer kontextfreien Grammatik heißt *rekursiv*, wenn es sich selbst direkt oder indirekt, d. h. über mehrere Zwischenschritte, ableiten kann. Solch ein rekursives Nichtterminal X hat also entweder eine Produktion $X \to \ldots X \ldots$ oder eine Produktion $X \to \ldots Y \ldots$, so daß X wiederum direkt oder indirekt von Y erzeugt werden kann. Etwas formaler gesagt ist X rekursiv, wenn es eine Ableitung $X \Rightarrow \cdots \Rightarrow \alpha X \beta$ mit mindestens einem Ableitungsschritt gibt.

X heißt *eingebettet rekursiv*, wenn dabei sowohl α als auch β nicht leer sind. Zum Beispiel ist A durch die Produktion $A \to aA$ zwar rekursiv, aber nicht eingebettet rekursiv. Durch $B \to aBc$ wird B eingebettet rekursiv. Ebenso werden durch die Produktionen $C \to cD$ und $D \to Cd$ die Nichtterminale C und D eingebettet rekursiv wegen $C \Rightarrow cD \Rightarrow cCd$ und $D \Rightarrow Cd \Rightarrow cDd$.

Eine kontextfreie Grammatik heißt *regulär*, wenn sie keine eingebettet rekursiven Nichtterminale enthält. Der Grund für diese Benennung ist, daß sich beweisen läßt, daß eine Sprache sich genau dann von einer regulären Grammatik definieren läßt, wenn sie durch einen einzelnen regulären Ausdruck, der keine Nichtterminale enthält, beschreibbar ist. Die von einem regulären Ausdruck r beschriebene Sprache ist dabei die Sprache der rrS-Grammatik mit Startsymbol S und der Produktion $S \to r$.

Um aus einem regulären Ausdruck r, der keine Nichtterminale enthält, eine äquivalente reguläre Grammatik zu erhalten, wende man das in Unterabschnitt 3.7.4 vorgestellte Verfahren auf die rrS-Grammatik $S \to r$ an. Man kann sich leicht davon überzeugen, daß die neuen Nichtterminale zwar mitunter rekursiv, aber nie eingebettet rekursiv sind. Der umgekehrte Weg, der aus einer regulären Grammatik einen äquivalenten nichtterminal-freien regulären Ausdruck erzeugt, ist viel komplizierter und kann hier nicht betrachtet werden.

Reguläre Grammatiken und nichtterminal-freie reguläre Ausdrücke haben also dieselbe Mächtigkeit. Kontextfreie Grammatiken sind dagegen mächtiger als reguläre. Es gibt nämlich Sprachen, die sich zwar von einer kontextfreien Grammatik, nicht aber von einer regulären beschreiben lassen. Das Standardbeispiel dafür ist die Sprache $\{a^n b^n \mid n \geq 0\}$, wobei a und b Terminalsymbole sind. Dabei ist a^n das Wort, welches aus n hintereinandergeschriebenen a's besteht. Die Sprache besteht also

aus allen Worten gerader Länge, deren vordere Hälfte nur *a*'s und deren hintere Hälfte nur *b*'s enthält. Eine kontextfreie Grammatik für diese Sprache hat man schnell hingeschrieben. Sie besteht aus den Produktionen $A \to aAb$ und $A \to \varepsilon$, wobei A das Startsymbol ist. Diese Grammatik ist nicht regulär, da A eingebettet rekursiv ist; es steht auf der rechten Seite der ersten Produktion zwischen *a* und *b*.

Schauen wir, wie man mit dieser Grammatik z. B. das Wort *aabb* ableitet. Ausgehend von A wird erst zweimal die erste und dann einmal die zweite Produktion angewendet.

$$A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aabb$$

Wichtig ist, daß mittels der ersten Produktion immer die gleiche Zahl von *a*'s und *b*'s erzeugt wird. Gerade dies gelingt mit keiner regulären Grammatik. Hier kann man zwar auch rekursive Nichtterminale benutzen, allerdings ohne eingebettete Rekursion. Die folgende Grammatik wäre ein Versuch:

$$S \to AB \qquad A \to aA \qquad A \to \varepsilon \qquad B \to Bb \qquad B \to \varepsilon$$

Sie erzeugt durchaus die Worte *ab*, *aabb*, *aaabbb* usw., aber leider auch die nicht erwünschten Worte *a*, *b*, *aaaab* usw. Ihre Sprache ist $\{a^n b^m \mid n \geq 0,\ m \geq 0\}$. Der entsprechende nichtterminal-freie reguläre Ausdruck ist $a^* b^*$. Es gibt auch keine andere reguläre Grammatik, welche die gewünschte Sprache erzeugt.

Nachdem wir jetzt eine Sprache kennengelernt haben, die zwar von einer kontextfreien Grammatik, aber nicht von einer regulären erzeugt werden kann, stellt sich die Frage, ob es auch Sprachen gibt, die von keiner kontextfreien Grammatik erzeugt werden können.

Die Antwort ist ja. Das einfachste Beispiel ist $\{a^n b^n c^n \mid n \geq 0\}$, also nur wenig komplizierter als die Beispielsprache von eben. Um diese Sprache grammatikalisch beschreiben zu können, muß das Konzept der kontextfreien Grammatik verallgemeinert werden. Es gibt verschiedene solche allgemeinere Grammatikformalismen, auf die wir hier jedoch nicht näher eingehen werden.

3.7.6 Ausschnitt aus einer Pascal-Grammatik

Nach diesem Ausflug in abstrakte Regionen wollen wir noch ein paar konkretere Beispiele betrachten.

Ein Buch über das Programmieren in einer speziellen Programmiersprache enthält immer eine Beschreibung der Syntax der Sprache, meist in der Form einer kontextfreien oder rrS-Grammatik. Es folgt ein Ausschnitt aus der Grammatik für die Programmiersprache Pascal.

Anweisung	→	*AnwGruppe* | *if-Anw* | *while-Anw* | *repeat-Anw* | *Zuweisung*
AnwGruppe	→	**begin** *AnwFolge* **end**
if-Anw	→	**if** *Bedingung* **then** *Anweisung* **else** *Anweisung*
while-Anw	→	**while** *Bedingung* **do** *Anweisung*
repeat-Anw	→	**repeat** *AnwFolge* **until** *Bedingung*
Zuweisung	→	*Variable* := *Ausdruck*
AnwFolge	→	*AnwFolge* ; *Anweisung* | *Anweisung*
Bedingung	→	*Ausdruck*

Es fehlen noch die Produktionen für die Nichtterminale *Variable* und *Ausdruck*. Die *Schlüsselwörter*, das sind Terminalsymbole, die verschiedene Bestandteile in Anweisungen trennen, sind hier fett gedruckt. Damit deutet man an, daß sie *reservierte Symbole* sind, also nicht z. B. als Benennung von Programmvariablen benutzt werden können. In Pascal-Programmen sind sie natürlich mit dem gleichen Zeichensatz geschrieben wie der Rest.

Diese Grammatik ist noch einmal dazu geeignet, den Unterschied zwischen regulären und kontextfreien Grammatiken zu verdeutlichen. In unserer Beispielsprache $\{a^n b^n \mid n \geq 0\}$ könnte man *a* als eine öffnende Klammer und *b* als eine dazu gehörende schließende Klammer betrachten. Dann kann man in einem Wort dieser Sprache, z. B. a^3b^3, von innen nach außen ein *a* einem *b* zuordnen, $\overbrace{a\ \overbrace{a\ \underbrace{a\ b}\ b}\ b}$.

Die Produktion $A \to aAb$ erzeugt also jeweils zueinandergehörige *a*'s und *b*'s. Es können auch nur Sätze erzeugt werden, in denen die Klammerpaare so schön korrespondieren.

Ein ähnliches Phänomen gibt es in der Pascal-Grammatik. Betrachten Sie die Produktionen

Anweisung	→	*AnwGruppe*
AnwGruppe	→	**begin** *AnwFolge* **end**
AnwFolge	→	*Anweisung*

Mit Hilfe dieser drei Produktionen können nur Worte der Form **begin**^{n} *Anweisung* **end**^{n} aus dem Nichtterminal *Anweisung* erzeugt werden, also wieder Worte mit korrekter Schachtelung von Klammerpaaren. Die Eigenschaft, daß in Pascal-Programmen **begin**-**end**-Klammern auf diese Art geschachtelt sein müssen, läßt sich durch eine reguläre Grammatik nicht ausdrücken.

3.7.7 Grammatiken in SGML

In Pascal-Programmen werden Anweisungen, Ausdrücke usw. nicht eigens gekennzeichnet. Der Pascal-Compiler muß anhand der Schlüsselwörter selbständig

die syntaktische Struktur des Programms, d. h. die Einteilung in die verschiedenen „Textsorten“, erkennen.

In SGML gibt es dagegen das Prinzip, daß die Texte einer Teilkomponentensorte X mit der Anfangsmarke <X> beginnen und mit der Endmarke </X> enden. (Die Endmarke kann, wie bereits erwähnt, unter gewissen Bedingungen entfallen, aber das tut hier nichts zur Sache.) Die Anfangs- und Endmarken ähneln den Schlüsselwörtern von Pascal, treten aber in der Dokumenttypdefinition nicht explizit auf; man muß sie sich dazudenken.

Die SGML-Dokumenttypdefinitionen entsprechen den Grammatiken mit regulären rechten Seiten, die wir am Anfang betrachtet haben. Da aber jedes Nichtterminal *X* in einer Dokumenttypdefinition eine Komponentensorte mit Marken einführt, sollte man nicht leichtfertig neue Nichtterminale einführen und Produktionen transformieren, wie wir das in Unterabschnitt 3.7.4 getan haben. Es entstünde dann ein neuer Dokumenttyp, dessen konkrete Instanzen möglicherweise ganz anders markiert werden müßten.

3.7.8 Zusammenfassung

Fassen wir noch einmal zusammen, was in diesem Abschnitt über Grammatiken zu lernen war.

1. Zur Definition von Sprachen (Textsorten) benutzen Informatiker Grammatiken. Mittels eines grammatikunabhängigen Ersetzungsmechanismus können für eine gegebene Grammatik Sätze der Sprache erzeugt werden.
2. Produktionen einer kontextfreien Grammatik haben in der üblichen Schreibweise auf ihrer linken Seite ein Nichtterminal, auf der rechten eine Folge von Terminalen und / oder Nichtterminalen. Man kann aber den Beschreibungskomfort erhöhen, ohne die Mächtigkeit des Formalismus zu erweitern, indem man auf der rechten Seite reguläre Ausdrücke, d. h. Alternativen, Iterationen und optionale Elemente, zuläßt.
3. Es ist wesentlich, ob eine Grammatik eingebettete Rekursion enthält. Grammatiken ohne eingebettete Rekursion sind zu einem nichtterminal-freien regulären Ausdruck äquivalent. Sie sind weniger mächtig als allgemeine Grammatiken.