6

Modellierung von Strukturen

In diesem Kapitel führen wir zwei grundlegende Kalküle ein, die sich besonders zur Modellierung struktureller Eigenschaften von Systemen eignen: Kontextfreie Grammatiken und das Entity-Relationship-Modell. Beide Kalküle werden *regelbasiert* angewandt, d. h. man formuliert Regeln mit den Konstrukten des Kalküls. Sie beschreiben, wie alle Ausprägungen der so modellierten Systeme aufgebaut sind. Die Regeln werden zur systematischen Konstruktion und auch zur Überprüfung von Systemen eingesetzt.

Kontextfreie Grammatiken eignen sich besonders zur Modellierung beliebig tief geschachtelter, rekursiver Strukturen. Einfache Anwendungsbeispiele sind etwa die Regeln zum Aufbau von strukturierten und markierten Texten oder von Menüstrukturen. Kontextfreie Grammatiken können gleichzeitig hierarchische Baumstrukturen und Sprachen und deren textuelle Notation spezifizieren. Letzteres ist z. B. für die Definition komplexer Protokollstrukturen nützlich. Für die formale Definition von Sprachen sind kontextfreie Grammatiken ein grundlegender Kalkül.

Mit dem Entity-Relationship-Modell formuliert man Regeln, nach denen Systeme in Mengen gleichartiger Objekte mit bestimmten Eigenschaften gegliedert sind und Relationen dazwischen bestehen. Charakteristisch ist das zugrunde liegende Objektmodell und die Möglichkeit, beliebige Relationen zu formulieren. Typische Anwendungsbeispiele sind Organisationsstrukturen von Firmen oder die Belegung von Räumen durch Kurse. Die grafische Notation macht solche Modellierungen besonders anschaulich. Das Entity-Relationship-Modell ist Grundlage sowohl für die Schemata objektorientierter Datenbanken als auch für die Spezifikationen von Strukturen und Beziehungen in Software-Systemen. Die Klassendiagramme der Spezifikationssprache UML basieren auf dem Entity-Relationship-Modell.

Abbildung 6.1: Die Struktur eines Menüs

Abb. 6.1 zeigt ein Menü, wie man es vielfach zur Bedienung von Software-Systemen verwendet. Es ist nach wenigen sehr einfachen Regeln aufgebaut:

1. Ein Menü besteht aus einem Menünamen und einer Folge von Einträgen.
2. ein Eintrag besteht aus
   a) einem Operationsnamen oder
   b) einem Menü.

Da die Regel (2b) *rekursiv* ist, kann man mit diesen Regeln beliebig tief geschachtelte Menüs beschreiben. Wir können die Strukturbeschreibung auch weiter verfeinern und z. B. gliedernde Linien als dritte Variante von Einträgen einführen oder dem Operationsnamen noch ein Tastenkürzel hinzufügen. Im Kalkül *kontextfreier Grammatiken*, den wir in Abschnitt 6.1 einführen, kann man solche Regeln formal angeben. Mit einer Grammatik für Menüstrukturen kann man definieren, wie alle Menü-Bäume aufgebaut sein sollen, spezielle davon erzeugen und prüfen. Mit kontextfreien Grammatiken kann man neben Baumstrukturen auch deren *textuelle Notation* als Sprache definieren. Wir zeigen das an einer Sprache für Terme.

Im Abschnitt 6.2 stellen wir Grundelemente und die Notation der Sprache XML (Extensible Markup Language) vor. Wir zeigen, wie man anwendungsspezifische XML-Sprachen für Baum-strukturierte Daten entwickelt und stellen den Bezug zur Modellierung mit kontextfreien Grammatiken her.

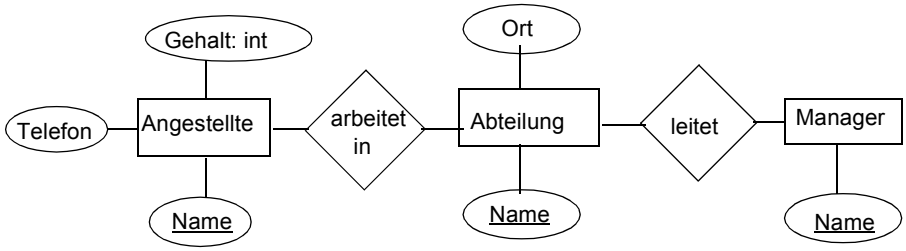


Abbildung 6.2: Ausschnitt aus dem Modell einer Firmenorganisation

Abb. 6.2 zeigt eine Spezifikation eines Ausschnittes einer Firmenorganisation im Entity-Relationship-Modell. Es definiert zwei Relationen zwischen den Objektmengen Angestellte, Abteilung und Manager und ordnet diesen Eigenschaften wie Name, Gehalt und Ort zu. Wir führen diesen Kalkül in Abschnitt 6.3 ein.

Im Abschnitt 6.4 zeigen wir, wie die Klassendiagramme der Spezifikationssprache UML (Unified Modeling Language) auf Konzepten und Notationen des Entity-Relationship-Modells aufgebaut sind.

6.1 Kontextfreie Grammatiken

Kontextfreie Grammatiken (KFGn) sind spezielle Ersetzungssysteme. Ihre Regeln geben an, wie man ein Symbol durch eine Folge von Symbolen ersetzen kann. Auf diese Weise definiert eine *KFG* eine *Sprache* als Menge von Sätzen, die mit den Regeln erzeugt werden können. Gleichzeitig definiert sie die *Baumstruktur* jedes Satzes, die sich durch die Anwendung der Regeln zu seiner Erzeugung ergibt. Der letzte Aspekt interessiert uns hier besonders für die Modellierung von Strukturen. Aber es ist häufig sehr nützlich, wenn derselbe Kalkül auch gleichzeitig eine textuelle Notation für die Strukturen liefern kann und Eigenschaften der Sprache untersucht werden können.

KFGn werden angewandt zur Definition von

- Programmen einer Programmiersprache und deren Struktur, z. B. Java, C, Pascal;
- Sprachen als Schnittstelle zwischen Software-Werkzeugen, Datenaustauschformate, z. B. HTML, XML;
- Bäumen zur Repräsentation strukturierter Daten, z. B. XML;
- Strukturen von Protokollen beim Austausch von Nachrichten zwischen Prozessen oder Geräten.

Definition 6.1: Kontextfreie Grammatik

*Eine **kontextfreie Grammatik G = (T, N, P, S)** besteht aus den endlichen Mengen*

T ***Terminalsymbole** (kurz:Terminale),*
N ***Nichtterminalsymbole** (kurz: Nichtterminale),*
P ***Produktionen***

*und dem **Startsymbol** $S \in N$. T und N sind disjunkt. $V = T \cup N$ heißt auch **Vokabular**; die Elemente von V nennt man auch **Symbole**. Die Produktionen haben die Form $P \subseteq N \times V^*$. Für eine Produktion $(A, x) \in P$ schreibt man auch $A ::= x$.* ■

Man sagt auch: „*In der Produktion A ::= x steht A auf der linken Seite und x auf der rechten Seite*.“ Häufig gibt man den Produktionen unterschiedliche Namen, wie p_1*: A ::= x*. In Symbolfolgen werden die Elemente nur durch Zwischenräume getrennt, z. B. *A ::= B c D*. Terminale, die Sonderzeichen sind oder enthalten, z. B. die Klammern *(* und *)*, kann man beim Auftreten in Produktionen in Apostrophe einschließen, wie ’(’und ’)’, um sie von Zeichen der KFG-Notation zu unterscheiden.

Als erstes Beispiel definieren wir eine kleine Grammatik für geschachtelte Klammerstrukturen.

Beispiel 6.1: Klammergrammatik G_1

T = { (,)}

```
N = { Klammerung, Liste}
S = Klammerung,
P = { Klammerung ::= '(' Liste ')',
      Liste ::= Klammerung Liste,
      Liste ::=
    }
```

Jede Produktion einer KFG, etwa *A ::= x,* kann man als *Strukturregel* mit der Bedeutung „*Ein A besteht aus x*“ auffassen oder als *Ersetzungsregel* mit der Bedeutung „*A kann man durch x ersetzen*“. Ersteres betont die *Definition von Strukturen*, Letzteres die *Definition von Sätzen einer Sprache*. In diesem Sinne können wir einige Produktionen aus verschiedenen Grammatiken als Beispiele betrachten:

```
DeutscherSatz ::= Subjekt Prädikat Objekt
```

kann man verstehen als:

Ein DeutscherSatz besteht aus Subjekt Prädikat Objekt.

Entsprechende Bedeutung haben die Produktionen

```
Klammerung  ::= '(' Liste ')'
Liste       ::= Klammerung Liste
Liste       ::=
```

Das Grundkonzept für die Anwendung von Produktionen einer KFG ist die Ableitung:

Definition 6.2: Ableitung

Sei G = (T, N, P, S) eine KFG, dann kann man mit der Produktion A ::= x ∈ P das Nichtterminal A ∈ N in der Symbolfolge u A v ∈ V^+ durch die rechte Seite x der Produktion zu u x v ersetzen. Das ist ein ***Ableitungsschritt****. Er wird notiert als u A v ⇒ u x v. Einige Ableitungsschritte, nacheinander angewandt, heißen* ***Ableitung*** *und werden notiert als* $w_o \Rightarrow w_1 \Rightarrow ... \Rightarrow w_n$ *oder, falls die Einzelschritte nicht angegeben werden,* $w_o \Rightarrow^* w_n$. ■

Mit der Klammergrammatik aus Beispiel 6.1 können wir z. B. folgende einzelne Ableitungsschritte angeben:

```
( Liste ) ⇒ ( Klammerung Liste )
( ( Liste ) Liste ) ⇒ ( ( ) Liste )
Klammerung ⇒ ( Liste )
```

Ein Beispiel für eine Folge von Ableitungsschritten ist

```
Klammerung
⇒ ( Liste )
⇒ ( Klammerung Liste )
⇒ ( Klammerung Klammerung Liste )
⇒ Klammerung ( Liste ) Liste )
⇒ ( ( Liste ) ( Liste ) Liste )
⇒ ( ( ) ( Liste ) Liste )
⇒ ( ( ) ( ) Liste )
```

$\Rightarrow$ (( ) ( ) ( ))

In jedem Schritt wird jeweils eine *Produktion* auf ein *Nichtterminal* der vorangehenden Symbolfolge angewandt. Wir können diese Ableitung auch kurz schreiben:

Klammerung $\Rightarrow^*$ (( ) ( ) ( )))

Es ist eine spezielle Eigenschaft dieser Ableitung, dass sie vom Startsymbol ausgeht und eine Folge von Terminalsymbolen erreicht. Letztere ist dann ein Satz der Sprache der KFG:

Definition 6.3: Sprache der KFG

*Sei G= (T, N, P, S) eine KFG, dann definiert G eine **Sprache L (G)**. Das ist eine Menge von Sätzen, jeder Satz ist eine Folge von Terminalen, die aus S ableitbar ist, also L(G) := {w | w $\in$ T^* und S $\Rightarrow^*$ w}.* ■

Die Klammergrammatik aus Beispiel 6.1 definiert so geschachtelte Folgen paariger Klammern als Sprache. Einige davon sind z. B.

{( ), (( )), (( ) ( ) ( )), ((( ) ( )) (( )))} $\in$ L(G_1)

Die folgende Grammatik definiert eine noch einfachere Sprache:

Beispiel 6.2: Grammatik G_2

N = {A},

T = {a},

S = A,

P = {A ::= A a, A ::= a}

$L(G_2) = \{a^n \mid n \geq 1\}$, wobei a^n eine Folge von *a* der Länge *n* bedeutet.

Die Ableitung eines Satzes definiert auch dessen Struktur durch einen Baum. Wir nennen ihn den *Ableitungsbaum* des Satzes.

Definition 6.4: Ableitungsbaum

*Eine **Ableitung** zu einer KFG kann man als **gewurzelten Baum** darstellen: Seine Knoten mit ihren Marken repräsentieren Vorkommen von Symbolen. Ein Knoten mit seinen Kanten zu den Nachbarn in Richtung der Blätter repräsentiert die Anwendung einer Produktion. Die Wurzel ist mit dem Startsymbol markiert. Terminale kommen nur an den Blättern vor.* ■

Um solche Ableitungsbäume zu konstruieren, notiert man jede Produktion als einen Baum mit der linken Seite als Wurzel und Kanten zu jedem Symbol der rechten Seite.

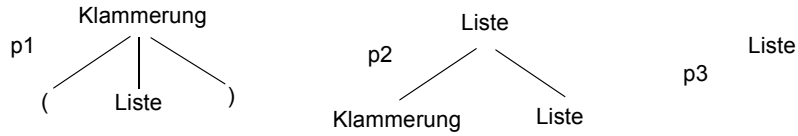


Abbildung 6.3: Drei Produktionen, als Bäume dargestellt

Abb. 6.3 zeigt die Bäume für die drei Produktionen der Klammergrammatik. Dort haben wir auch Namen *p1, p2, p3* für die Produktionen eingeführt. Einen Ableitungsbaum stellt man her, indem man für jeden Ableitungsschritt eine Kopie des Baumes der angewandten Produktion an den Knoten der Anwendungsstelle heftet.

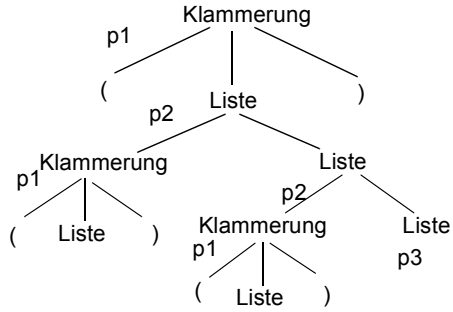


Abbildung 6.4: Ableitungsbaum für die Ableitung des Satzes (( ) ( ))

Abb. 6.4 zeigt einen Ableitungsbaum für die Ableitung des Satzes (( ) ( )) , die wir oben angegeben haben.

Zu jedem Ableitungsbaum erhält man den zugehörigen Satz der Sprache, wenn man den Baum links-abwärts durchläuft und die Terminale in der Reihenfolge ausgibt, in der ihre Knoten besucht werden.

Wenn wir Ableitungen und zugehörige Ableitungsbäume vergleichen, stellen wir fest, dass ein Ableitungsbaum durchaus verschiedene Ableitungen desselben Satzes repräsentieren kann. Sie unterscheiden sich in der Reihenfolge, in der die Ableitungsschritte durchgeführt werden. So hätte man in der oben angegebenen Ableitung von (( ) ( )) schon im vierten Schritt die Produktion *p3: Liste::=* auf die ganz rechts stehende Liste anwenden können, statt erst im letzten Schritt.

Im Ableitungsbaum wird von der Reihenfolge, in der er konstruiert wurde, abstrahiert.

Diese Beobachtung führt uns zu einer wichtigen Eigenschaft kontextfreier Grammatiken. Häufig ist es erwünscht, dass die Struktur jedes Satzes der Sprache eindeutig bestimmt ist, d. h. dass es zu jedem Satz der Sprache nur einen Ableitungsbaum gibt.

Definition 6.5: Mehrdeutigkeit

*Eine kontextfreie Grammatik ist **mehrdeutig**, wenn es einen Satz ihrer Sprache gibt, der zwei verschiedene Ableitungsbäume hat.* ■

Für die oben angegebene Klammergrammatik kann man durch strukturelle Induktion über die Höhe der Bäume nachweisen, dass sie nicht mehrdeutig ist. Den Nachweis der Mehrdeutigkeit führt man, indem man einen Satz angibt, der mehrere verschiedene Ableitungsbäume hat. Als Beispiel konstruieren wir eine mehrdeutige Klammergrammatik, indem wir zu den Produktionen von G_1 noch

p4: Liste ::= Klammerung

hinzunehmen. Nun können wir für den Satz (( ) ( )) einen weiteren Ableitungsbaum angeben (Abb. 6.5), der sich von dem in Abb. 6.3 unterscheidet. Es gibt allerdings auch KFGn, für die es nicht entscheidbar ist, ob sie mehrdeutig sind. Für die Praxis ist das jedoch kein Problem.

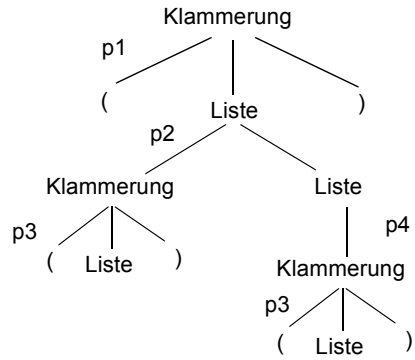


Abbildung 6.5: Weiterer Ableitungsbaum für den Satz (( ) ( ))

Wir wenden uns nun wieder dem Beispiel der Menü-Strukturen vom Anfang dieses Abschnittes zu. Wir können die verbale Beschreibung unmittelbar in die Produktionen einer KFG umsetzen:

Beispiel 6.3: Menü-Grammatik

Die oben skizzierte Menü-Struktur wird durch folgende KFG spezifiziert:
T = {MenüName, OperationsName},
N = {Menü, EintragsFolge, Eintrag},
S = Menü,
P = { Menü ::= MenüName EintragsFolge,
EintragsFolge ::= Eintrag,
Eintrag ::= OperationsName,
Eintrag ::= Menü
}

Der Zweck dieser Grammatik ist es, die Struktur von Menüs zu spezifizieren. Deshalb interessieren wir uns besonders für die Ableitungsbäume, die man bilden kann, und weniger für die Sprache zur Grammatik.

Abb. 6.6 zeigt einen Ableitungsbaum, der die Struktur des Menüs aus Abb. 6.1 repräsentiert. Natürlich können wir auch Sätze zu den Ableitungsbäumen der Menügrammatik angeben. Man erhält einen Satz der Sprache, wenn man – wie oben – die Terminale des Ableitungsbaumes in einem Links-abwärts-Durchgang ausgibt. Solch ein Satz ist eine Folge von Menü- und Operationsnamen, angeordnet in *Präfixform*.

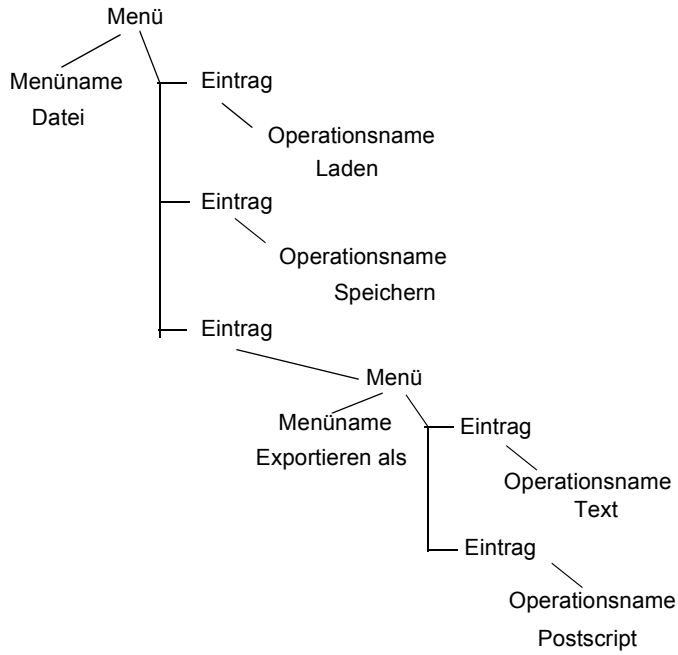


Abbildung 6.6: Ableitungsbaum für das Menü in Abb. 6.1

In Kapitel 3 haben wir den Begriff der *Signatur* eingeführt: Sie definiert *Operatoren* und deren *Operanden*- und *Ergebnissorten*. Damit ist auch die Menge der korrekten Terme zur Signatur definiert. Wir können jede Signatur systematisch in eine KFG umschreiben, sodass deren Ableitungsbäume die korrekten Terme repräsentieren. Wir zeigen das am Beispiel der abstrakten Booleschen Algebra aus Kapitel 3 (Beispiel 6.4):

Beispiel 6.4: Boolesche Term-Bäume

T = { true, false, ∧, ∨, ¬, Variable}
N = { BOOL}
S = BOOL
P = { BOOL::= Variable,
BOOL::= true,

BOOL::= false,
BOOL::= '∧' BOOL BOOL,
BOOL::= '∨' BOOL BOOL,
BOOL::= '¬' BOOL
}

Wir haben die Grammatik nach folgenden Regeln aus der Signatur erzeugt:

1. Die Operatoren werden Terminale.
2. Die Sorten werden Nichtterminale.
3. Die definierte Sorte wird das Startsymbol.
4. Jede Operatorbeschreibung der Form

   $opr: S_1 \times S_2 \times ... \times S_n \rightarrow S_o$

   wird zu einer Produktion der Form

   S_o ::= opr S_1 S_2 ... S_n
5. Wir fügen eine Produktion *S ::= Variable*
   hinzu, wobei *S* die *definierte Sorte* und *Variable* ein *weiteres Terminal* ist.

Ein Ableitungsbaum zu dieser Grammatik ist z. B. der in Abb. 6.7:

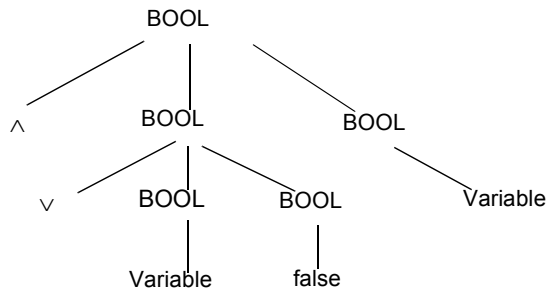


Abbildung 6.7: Ein Boolescher Term-Baum

Die Sätze der Sprache dieser KFG sind die Terme, notiert in *Präfixform,* da in den Produktionen die Operatoren den Nichtterminalen für ihre Operanden vorangestellt sind. Der Satz zu obigem Ableitungsbaum lautet:

∧ ∨ Variable false Variable

Oder mit zwei Variablennamen eingesetzt:

∧ ∨ a false b

Wir können durch Variation der Stellung der Operatoren in den Produktionen auch *Postfix-* und *Infix-Form* erzeugen. Ersetzen wir die Produktionen aus Beispiel 6.4 durch die folgenden, so erhalten wir *Termnotationen* in *Infixform* als Sätze:

BOOL ::= Variable

```
BOOL    ::= true
BOOL    ::= false
BOOL    ::= BOOL '∧' BOOL
BOOL    ::= BOOL '∨' BOOL
BOOL    ::= '¬' BOOL
```

Die Menge der Ableitungsbäume in dieser Grammatik repräsentiert die Menge der korrekten Terme zur ursprünglichen Signatur – ebenso wie die Bäume der KFG des Beispiels 6.4. Betrachten wir aber die textuellen Sätze dazu, erkennen wir, dass es Sätze gibt, die man jeweils aus verschiedenen Ableitungsbäumen erzeugen kann. Die Grammatik ist mehrdeutig. Z. B. zu dem Satz

a ∨ false ∧ b

kann man zwei verschiedene Ableitungsbäume angeben, Abb. 6.8.:

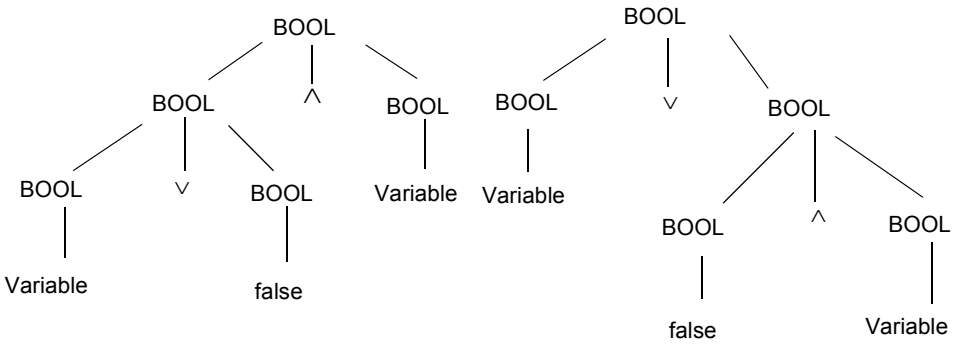


Abbildung 6.8: Zwei Ableitungsbäume zu demselben Satz

In der Grammatik für die Präfixform ist das nicht möglich, sie ist eindeutig. Wir können die Mehrdeutigkeit der Infixform beseitigen, indem wir den Operatoren unterschiedliche Präzedenzen zuordnen.

In der folgenden Produktionsmenge haben wir die Mehrdeutigkeit beseitigt:

```
BOOL    ::= BOOL '∨' BOOL1
BOOL    ::= BOOL1
BOOL1   ::= BOOL1 '∧' BOOL2
BOOL1   ::= BOOL2
BOOL2   ::= '¬' BOOL2
BOOL2   ::= Variable
BOOL2   ::= true
BOOL2   ::= false
BOOL2   ::= '(' BOOL ')'
```

Zu jedem Satz gibt es genau einen Ableitungsbaum. Zu dem Satz

a ∨ false ∧ b

gehört der Ableitungsbaum in Abb. 6.9.

Seine Struktur entspricht dem rechten der beiden Ableitungsbäume, die wir für den Satz oben angegeben haben. Diese Grammatik ordnet dem Operator $\neg$ höchste, $\wedge$ geringere und $\vee$ geringste Präzedenz zu. Beide zweistelligen Operatoren sind durch die Produktionen *linksassoziativ* definiert. Schließlich mussten noch geklammerte Terme durch die letzte Produktion eingeführt werden. Die Struktur des linken der beiden Ableitungsbäume zu dem mehrdeutigen Satz erhält man nun durch Klammerung:

$(a \vee \text{false}) \wedge b$

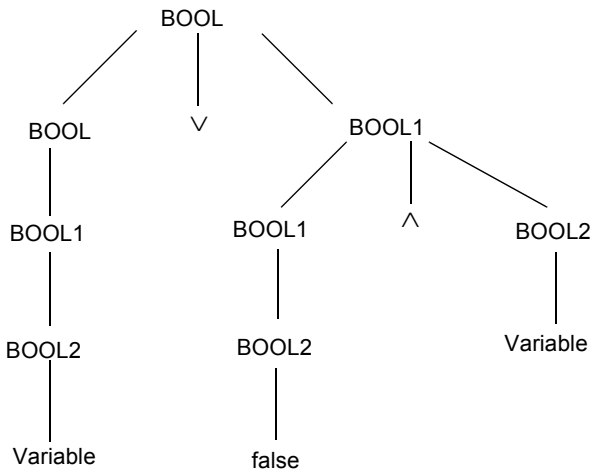


Abbildung 6.9: Ableitungsbaum zur Grammatik mit Operatoren unterschiedlicher Präzedenzen

Für die Repräsentation von verzeigerten Dokumenten, insbesondere im World Wide Web, haben Sprachen wie HTML und XML grundlegende Bedeutung. Die Sätze dieser Sprachen repräsentieren Bäume durch Texte, die durch spezielle Klammern strukturiert sind. Solch ein Paar öffnender und schließender Klammern hat die Form

<x> ... </x>

wobei *x* für eine Marke mit bestimmter Bedeutung steht, z. B. *table* für eine Tabelle, *tr* für eine Tabellenzeile, *td* für ein Datenelement einer Tabelle. Wir geben nun eine KFG für Tabellen in HTML-Notation an:

Beispiel 6.5: HTML Tabellen

Die folgenden Produktionen definieren eine vereinfachte Form von geschachtelten Tabellen in der Notation von HTML:

Table ::= '<table>' Rows '</table>'
Rows ::= Row*
Row ::= '<tr>' Cells '</tr>'
Cells ::= Cell*
Cell ::= '<td>' Text '</td>'

```
Cell    ::=     '<td>' Table '</td>'
```

Die Angabe von *Row** auf der rechten Seite der zweiten Produktion steht für eine Folge von beliebig vielen (auch keinem) Auftreten von *Row*. Der Stern hat dieselbe Bedeutung wie bei der Angabe von Folgen im Kapitel 2. Er vereinfacht eine ausführlichere Schreibweise, die wir auch in der Menü-Grammatik verwendet haben. Die Produktion

```
Rows  ::= Row*
```

ist gleichbedeutend zu

```
Rows  ::= Y
Y     ::= Y Row
Y     ::=
```

unter der Annahme, dass *Y* sonst nicht in der Grammatik vorkommt. Allerdings drückt das Konstrukt *Row** in der Baumstruktur deutlicher aus, dass es sich um eine Folge von *Row* auf gleicher Ebene handelt, während die Produktionen für *Y* eine tiefe, rekursive Struktur erzeugen, siehe Abb. 6.10.

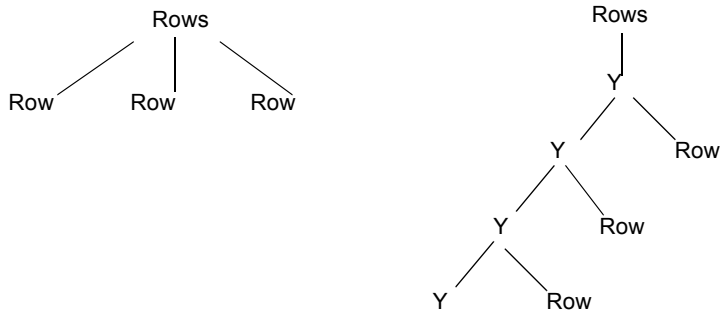


Abbildung 6.10: Eine Folge in flacher und tiefer Baumstruktur

```
<table>
        <tr>    <td>Tag</td>
                <td>Zeit</td>
                <td>Raum</td></tr>
        <tr>    <td>Mo</td>
                <td>11:00-12:30</td>
                <td>AM</td></tr>
        <tr>    <td>Fr</td>
                <td>9:15-10:45</td>
                <td>AM</td></tr>
</table>
```

Abbildung 6.11: Satz der Tabellen-Grammatik

Abb. 6.11 zeigt einen Satz zur Tabellen-Grammatik, Abb. 6.12 den Ableitungsbaum dazu und Abb. 6.13 die formatierte Darstellung der dreizeiligen Tabelle.

In Beispiel 6.5 haben wir nur die Produktionen der KFG angegeben. Die drei übrigen Komponenten einer KFG kann man daraus entnehmen:

1. Alle Symbole, die auf der linken Seite einer Produktion vorkommen, sind Nichtterminale.

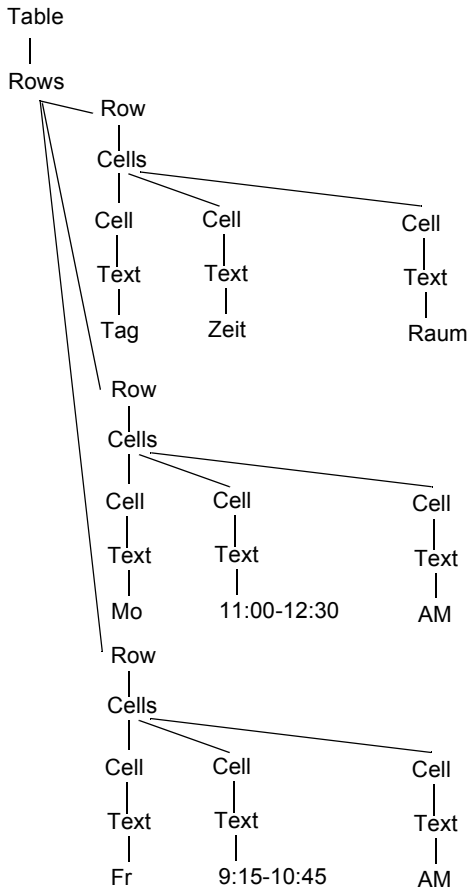


Abbildung 6.12: Ableitungsbaum zum Satz aus Abb. 6.11

Tag	Zeit	Raum
Mo	11:00-12:30	AM
Fr	9:15-10:45	AM

Abbildung 6.13: Formatierte Darstellung des Satzes aus Abb. 6.11

2. Alle übrigen Symbole sind Terminale.
3. Ein Nichtterminal, das in keiner Produktion auf der rechten Seite vorkommt, ist das Startsymbol. (Wenn alle Nichtterminale auch auf rechten Seiten vorkommen, muss das Startsymbol explizit bestimmt werden.)

KFGn werden daher meist nur durch eine Folge von Produktionen angegeben.

6.2 Baumstrukturen in XML

XML ist zurzeit die wichtigste Sprache, um strukturierte Dateien Werkzeug-unabhängig zu repräsentieren. Besonders durch seinen Einsatz im *World Wide Web (WWW)* hat XML enorme Verbreitung erlangt. Das *World Wide Web Consortium (W3C)*, das die Entwicklung der Techniken des WWW koordiniert, hat XML 1996 definiert und seitdem weiterentwickelt. XML steht für *Extensible Markup Language*, also Erweiterbare Auszeichnungssprache. Die Texte von Markup- oder Auszeichnungssprachen enthalten Markierungen, die bestimmte Eigenschaften der Textelemente kennzeichnen, z. B. geschachtelte Strukturen in XML oder Layoutangaben in HTML. Vorbild für das Prinzip ist die Auszeichnung von Texten für den Drucksatz.

XML ist orientiert an der *Standardized Generalized Markup Language (SGML*) von 1986. XML ist, wie auch schon SGML, eine Meta-Sprache, d. h. ein Schema, aus dem man für prinzipiell beliebige Zwecke jeweils eine spezielle XML-Sprache definieren kann. Das *Extensible* im Namen von XML deutet auf diese Eigenschaft hin. Für alle XML-Sprachen ist die Notation, insbesondere der Auszeichnungen, festgelegt. Die syntaktische Struktur wird dann für jede XML-Sprache individuell definiert. Auf diese Weise kann man mit XML für spezielle Zwecke modellieren. Es gibt viele XML-Sprachen, die recht breit eingesetzt werden, wie *SVG* zur Notation skalierbarer Vektorgrafiken, *MathML* zur Notation mathematischer Formeln, *SMIL* zur Synchronisation und Integration von multimedialen Daten und *RDF* zur Bereitstellung von Daten im WWW. Man kann aber auch eine XML-Sprache entwerfen, um die Speicherung von Daten eines einzelnen Software-Werkzeuges zu realisieren. Es können dann allgemeine XML-Werkzeuge verwendet werden, um solche Daten z. B. zu lesen, zu schreiben und als Datenstrukturen aufzubauen. Davon wird häufig in Web-Anwendungen Gebrauch gemacht.

Der Definition von XML liegen drei sehr einfache und wirksame Prinzipien zu Grunde:

a) Auszeichnungen mit darin enthaltenen Namen werden als Klammern verwendet, um den Text zu strukturieren.
b) Jede Klammerstruktur repräsentiert einen gewurzelten Baum.
c) Eine kontextfreie Grammatik definiert spezielle Baumstrukturen.

Im Folgenden zeigen wir, dass sich XML mit diesen Prinzipien leicht verstehen und wirkungsvoll zum Modellieren einsetzen lässt.

6.2.1 XML Notation

Ein Satz in einer XML-Sprache ist ein Text, der durch Kennzeichen, sog. *Tags*, strukturiert wird. Im Folgenden verwenden wir das englische Wort *Tag*, da es als technischer Begriff im Zusammenhang mit XML besser eingeführt ist als eine deutsche Übersetzung.

In ihrer einfachen Form werden Tags als Namen in spitzen Klammern notiert, wie <*ort*> und </*ort*>. Die erste Form ist ein Anfangs-*Tag* die zweite ein *End-Tag*. Grundsätzlich treten Tags immer in Paaren von Anfangs- und End-Tag auf, z. B.

```
<ort>Paderborn</ort>
```

Sie klammern dann den dazwischen stehenden Text, der auch komplex strukturiert oder leer sein kann.

```
<adressBuch>
    <adresse>
        <name>
            <nachname>Kastens<nachname>
            <vorname>Uwe</vorname>
        </name>
        <anschrift>
            <strasse>Abtsbrede 40a</strasse>
            <ort>Paderborn</ort>
            <plz>33098</plz>
        </anschrift>
    </adresse>
</adressBuch>
```

Abbildung 6.14: Eine Adresse einer XML-Sprache für Adressbücher

Abb. 6.14 zeigt einen Satz in einer XML-Sprache, in dem Anfangs- und End-Tags mit verschiedenen Namen vorkommen. Die Namen werden beim Entwurf einer XML-Sprache frei erfunden. Der Name eines Tag-Paares soll das von ihm begrenzte Element charakterisieren. Das Beispiel in Abb. 6.14 beschreibt den Inhalt eines Adressbuches als geschachtelte Struktur: Es enthält eine einzige Adresse; sie besteht aus einem Namen und einer Anschrift, die beide aus mehreren Elementen zusammengesetzt sind. Jedes Strukturelement wird durch ein Paar aus Anfangs- und End-Tag begrenzt. Deren Tag-Name benennt die Rolle, die das Element in der Struktur hat.

```
<msup>
    <mfenced>
        <mrow>
            <mi>a</mi>
            <mo>+</mo>
            <mi>b</mi>
        </mrow>
    </mfenced>
    <mn>2</mn>
</msup>
```

Abbildung 6.15: Der Satz $(a+b)^2$ in der XML-Sprache MathMl

Die in Abb. 6.14 verwendete XML-Sprache ist speziell für diese Vorstellung von XML entwickelt worden. Dagegen wird in Abb. 6.15 die weiter verbreitete XML-Sprache *MathML* verwendet, um die Formel $(a + b)^2$ darzustellen. Hier ist die Bedeutung der Tag-Namen nicht ganz selbsterklärend: So kennzeichnet z. B. <msup> eine Formel mit hochgestelltem Wert und <mo> einen Operator.

In XML-Sprachen kann man Strukturelemente nicht nur durch den Tag-Namen kennzeichnen, sondern ihnen auch Eigenschaften zuordnen. Dazu erweitert man das Anfangs-Tag eines Elementes um Attribute, z. B.

```
<telefon typ = "dienst">05251606686</telefon>
```

Ein Attribut wird durch ein Paar aus dem Namen und dem Wert des Attributes angegeben, hier *typ="dienst"*. Anfangs-Tags können beliebig lange Folgen solcher Name-Wert-Paare enthalten. Die Attributnamen und -werte können (wie die Tag-Namen) beim Entwurf der XML-Sprache prinzipiell frei gewählt werden. So könnten wir zum Beispiel die in Abb. 6.14 verwendete XML-Sprache für Adressbücher um *Telefonangaben* erweitern und die Telefonnummern wie oben durch ein Attribut namens *typ* mit Werten wie *"dienst"* oder *"privat"* kennzeichnen. Im Folgenden konzentrieren wir uns auf die Strukturen von XML-Texten und betrachten Attribute nicht weiter.

Es soll hier noch erwähnt werden, dass in XML zwei gleichnamige Anfangs- und End-Tags, die unmittelbar aufeinander folgen, zu einem einzigen Tag zusammengefasst werden können, z. B.

```
<bild name = "Hans.jpg"></bild> zu
<bild name = "Hans.jpg" />
```

Der Schrägstrich vor der schließenden Klammer kennzeichnet diese Zusammenfassung zweier Tags. Wir betrachten sie im Folgenden nicht weiter, sondern nehmen an, dass Tags immer paarig auftreten.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE adressBuch SYSTEM "adressBuch.dtd">
<?xml-stylesheet type="text/xsl" href="adressBuch.xsl" ?>
<adressBuch>
    <adresse>
        <name>
            <nachname>Kastens</nachname>
            <vorname>Uwe</vorname>
        </name>
        <anschrift>
            <strasse>Abtsbrede 40a</strasse>
            <ort>Paderborn</ort>
            <plz>33098</plz>
        </anschrift>
    </adresse>
</adressBuch>
```

Abbildung 6.16: Eine vollständige XML-Datei

Abb. 6.16 zeigt eine vollständige XML-Datei für die Adressbuch-Sprache aus Abb. 6.14. Sie wird durch drei spezielle technische Angaben eingeleitet: Die erste Zeile kennzeichnet das Dokument als XML-Datei, gibt die Versionsnummer der XML-Definition und den verwendeten Zeichen-Code an. Diese Informationen können von Leseprogrammen für XML benutzt werden. Die zweite Zeile verweist auf die Definition der XML-Sprache, in welcher der folgende Text formuliert ist: *adressBuch* ist der Name des äußersten Tag-Paares der folgenden Struktur; *adressBuch.dtd* ist der Name einer Datei, die die Strukturdefinition für diese XML-Sprache enthält. Die dritte Zeile dient der Formatierung der folgenden Struktur: Sie gibt an, dass die XSL-Technik angewandt wird. Dabei wird die XML-Struktur in einen HTML-Text transformiert, der Angaben zum Layout der Elemente enthält. Die Datei *adressBuch.xsl* enthält die Vorschriften zur Transformation der verschiedenen Arten von Elementen.

6.2.2 XML-Texte als Bäume

XML-Texte beschreiben geschachtelte Strukturen, die nach folgenden Regeln aufgebaut sind:

a) Ein Element beginnt mit einem Anfangs-Tag und endet mit einem gleichnamigen End-Tag. Dazwischen steht eine eventuell leere Folge von Elementen und elementaren Texten.

b) Elementare Texte können beliebige Zeichen, aber keine Tags enthalten.

c) Ein XML-Text ist ein Element.

Es leuchtet unmittelbar ein, dass auf diese Weise XML-Texte rekursiv als gewurzelte Bäume definiert werden: Regel (c) definiert den Wurzelknoten, Regel (a) innere Knoten und Regel (b) Blattknoten. In der XML-Terminologie heißen XML-Texte *wohlgeformt* (engl. well-formed), wenn sie nach obigen Regeln gebildet sind. Abb. 6.17 zeigt drei

Texte, von denen (1) und (2) die obigen Regeln erfüllen; (3) verletzt sie, weil die Klammern nicht paarig geschachtelt sind, sondern sich überlappen. Im Beispiel (1) treten elementare Texte nur allein in einer Folge zwischen den Tag-Klammern auf; in (2) gibt es auch Folgen, die aus elementaren Texten und Elementen bestehen.

```
<a>                    <a>                    <a>
    <b>                    1                      <b>
        <c>1</c>           <b>                        <c>1</b>
        <d>2</d>               2                  </c>
    </b>                       <c>3</c>       </a>
    <e>3</e>                   4
<a>                            <d>5</d>
                           </b>
                           <e>6</e>
                       <a>
        (1)                    (2)                    (3)
```

Abbildung 6.17: (1), (2): wohlgeformte XML-Texte, (3) nicht-wohlgeformter XML-Text

Man kann die obigen Regeln auch verwenden, um aus einem XML-Text systematisch einen gewurzelten Baum zu erzeugen: Die inneren Knoten und den Wurzelknoten beschriftet man mit dem Namen des Tag-Paares, dessen Element er repräsentiert. Die Blattknoten beschriftet man mit den elementaren Texten. Abb. 6.18 zeigt die Bäume zu Abb. 6.17, Abb. 6.19 den Baum zum XML-Text aus Abb. 6.14.

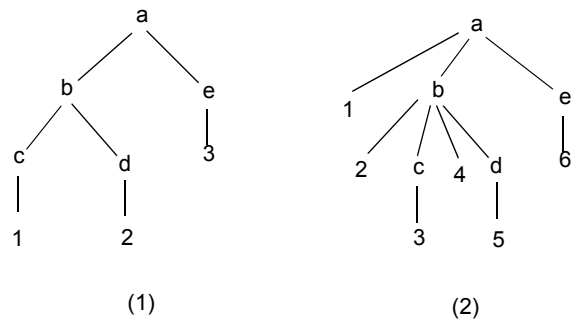


Abbildung 6.18: Bäume zu Abb. 6.17 (1) und (2)

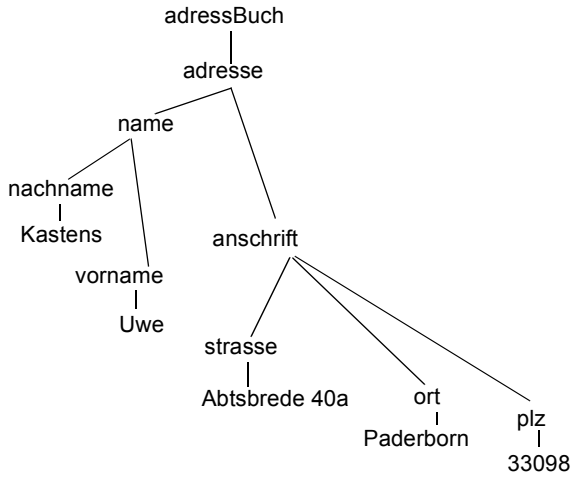


Abbildung 6.19: Baum zu Abb. 6.14

XML-Werkzeuge, die XML-Dateien lesen, können die Klammerstruktur wohlgeformter XML-Texte erkennen und als Bäume darstellen, ohne die XML-Sprache zu kennen, der die Texte angehören. Natürlich kann man auch jeden gewurzelten Baum systematisch und eindeutig in einen XML-Text transformieren, wenn seine inneren Knoten und sein Wurzelknoten mit Namen markiert sind, welche die Art des Teilbaumes charakterisieren: Beginnend mit der Wurzel wird zu einem Knoten ein Tag-Paar mit dem Namen der Knotenmarke erzeugt und das Ergebnis der Transformation der Unterbäume zwischen das Anfangs- und das End-Tag eingefügt. Dabei muss die Reihenfolge der Unterbäume erhalten bleiben.

Die eindeutige Zuordnung zwischen gewurzelten Bäumen und geklammerten Texten gilt natürlich für jede Art von Klammerung. Abb. 6.20 zeigt einen Baum für den geklammerten Text (1 (2 (3) 4 (5)) (6)). Er hat dieselbe Struktur wie der Baum in Abb. 6.17(2); allerdings hat er keine Namen an Nicht-Blattknoten, da die Klammern in diesem Text unbenannt sind.

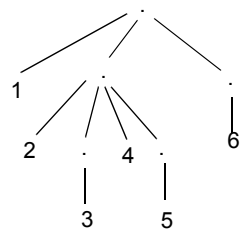


Abbildung 6.20: Baum zur Klammerstruktur (1 (2 (3) 4 (5)) (6))

6.2.3 Strukturdefinition für XML-Bäume

Im vorigen Abschnitt haben wir die Baumstruktur von XML-Texten nur hinsichtlich der Schachtelung bzw. Klammerung betrachtet. Dabei war es unerheblich, welche Art von Elementen in welcher Reihenfolge ineinander geschachtelt sind. Wann wir eine XML-Sprache entwerfen, legen wir für jedes Element fest, welche Art von Elementen in welcher Reihenfolge zwischen den Tags stehen können und an welchen Stellen elementare Texte auftreten können. Für solche Strukturdefinitionen sind kontextfreie Grammatiken (Abschnitt 6.1) der geeignete Kalkül.

adressBuch	::= adresse*
adresse	::= name anschrift
name	::= nachname vorname
anschrift	::= strasse ort plz
nachname	::= PCDATA
vorname	::= PCDATA
strasse	::= PCDATA
ort	::= PCDATA
plz	::= PCDATA

Abbildung 6.21: Eine KFG für die Adressbuch-Sprache

Abb. 6.21 enthält eine kontextfreie Grammatik, welche die Struktur von Bäumen zu unserer XML-Sprache für Adressbücher beschreibt. Die Produktionen legen fest, dass ein *adressBuch* eine evtl. leere Folge von *adresse* ist, eine *adresse* aus *name* gefolgt von *anschrift* besteht, usw. Das einzige Terminalsymbol dieser Grammatik ist *PCDATA*. Es steht für einen elementaren Text ohne Tags. Die Namen der Nichtterminalsymbole sind so gewählt, dass sie als Marken der inneren Knoten und des Wurzelknotens zu verwenden sind. Bildet man einen Ableitungsbaum zu der Grammatik, markiert man den Wurzelknoten und die inneren Knoten mit den Namen der Nichtterminale, die sie repräsentieren. Die Blätter werden wieder mit den elementaren Texten markiert, die *PCDATA* klassifiziert. Solch ein Baum lässt sich wie in Abschnitt 6.2.2 beschrieben in einen XML-Text transformieren.

Grundsätzlich kann man auf diese Weise mit einer kontextfreien Grammatik und der Wahl der Nichtterminal-Namen die syntaktisch korrekten Sätze einer XML-Sprache und die zugehörigen Bäume definieren. Leider hat man bei der Entwicklung von SGML und XML zur Strukturdefinition kontextfreie Grammatiken nicht in der üblichen Notation und mit den üblichen Begriffen verwendet. Stattdessen wurde eine sogenannte *Document Type Definiton (DTD)* eingeführt. Ihre Notation kann aber direkt der für kontextfreie Grammatiken üblichen zugeordnet werden.

Abb. 6.22 enthält eine DTD für unsere Adressbuch-Sprache, die der KFG in Abb. 6.21 entspricht: Jede Produktion wird in *<!ELEMENT* und > eingeschlossen. Sie besteht aus dem Nichtterminal der linken Seite der Produktion, gefolgt von der rechten Seite, die in

Klammern eingeschlossen ist. Die Elemente der rechten Seite sind durch Kommata getrennt.

```
<!ELEMENT  adressBuch   (adresse)*>
<!ELEMENT  adresse      (name, anschrift)>
<!ELEMENT  name         (nachname, vorname)>
<!ELEMENT  anschrift    (strasse, ort, plz)>

<!ELEMENT  nachname     (#PCDATA)>
<!ELEMENT  vorname      (#PCDATA)>
<!ELEMENT  strasse      (#PCDATA)>
<!ELEMENT  ort          (#PCDATA)>
<!ELEMENT  plz          (#PCDATA)
```

Abbildung 6.22: DTD zur KFG in Abb. 6.21

Auf der rechten Seite von Produktionen einer DTD können auch Operatoren verwendet werden, die der Notation regulärer Ausdrücke entstammen, z. B. für Folgen, Alternativen und optionale Teile. Solche Erweiterungen sind auch für die Notation kontextfreier Grammatiken gebräuchlich. Dies sind in DTD-Notation:

(Y)+	eine nicht-leere Folge von Y
(Y)*	eine evtl. leere Folge von Y
(A | B)	alternativ A oder B
A?	optional A
EMPTY	zur Definition eines leeren Elementes verwendet

Der *-Operator wird in unserem Beispiel der DTD und KFG für die Adressbuch-Sprache für die Folge von Adressen verwendet. Folgende DTD-Produktion verwendet den Options- und den Alternativ-Operator:

```
<!Element adr (anrede?, name, (postfach | anschrift) )
```

In einer gebräuchlichen erweiterten Notation für KFGn würde die Produktion lauten:

```
adr ::= [anrede] name (postfach / anschrift)
```

Mit dem Konzept der kontextfreien Grammatiken in der Notation von DTD definiert man für beliebige XML-Sprachen die Menge der syntaktisch korrekten Sätze, d. h. XML-Dokumente und deren Baumstrukturen. Im Beispiel der Abb. 6.16 haben wir gezeigt, wie in einem XML-Dokument die DTD angegeben wird, die dessen Struktur definiert. Mit dieser Information sind XML-Werkzeuge, sog. XML-Parser, in der Lage, das XML-Dokument auf syntaktische Korrektheit zu prüfen und einen Baum als Datenstruktur aufzubauen. Für diese Datenstruktur werden Zugriffsfunktionen bereitgestellt, mit denen auf Elemente des Baumes unter Verwendung der in der DTD definierten Namen und Strukturen zugegriffen werden kann. Diese Funktionen werden im sogenannten *Document Object Model (DOM)* definiert. Abb. 6.23 skizziert ein Szenario, in dem ein Programm eine XML-Datei als Datenspeicher benutzt. Sie wird von einem XML-Parser gelesen und daraus anhand der DTD ein Baum als Datenstrukur aufgebaut. Das Programm greift darauf lesend und ggf. ändernd mit Hilfe der DOM-Funktionen zu. Schließlich schreibt ein

XML-Werkzeug den geänderten Baum wieder als XML-Text in die Datei. Solche Techniken werden häufig in der Web-Programmierung eingesetzt oder um die Datenspeicherung von Software-Werkzeugen in einem weitverbreiteten Format zu realisieren.

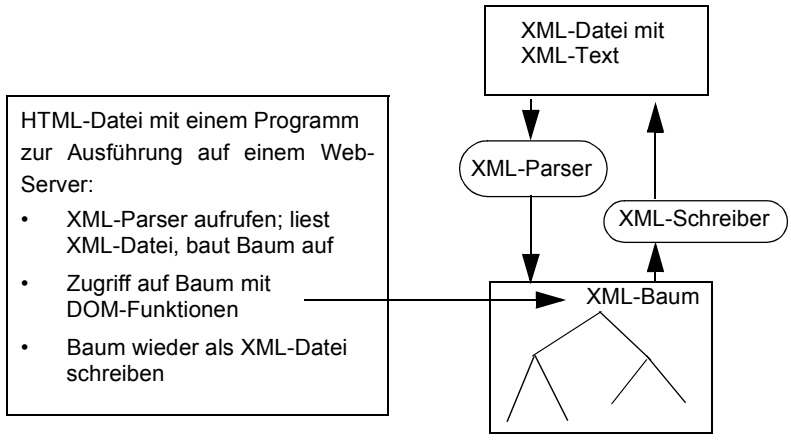


Abbildung 6.23: XML-Datei als Speicher für strukturierte Daten

6.3 Entity-Relationship-Modell

Das *Entity-Relationship-Modell* (*ER-Modell*) ist ein formaler Kalkül zur Modellierung von Themenbereichen mit ihren Objekten, deren Eigenschaften und Beziehungen zwischen ihnen. Charakteristisch für den Kalkül ist die Zusammenfassung gleichartiger Objekte zu Mengen. Im Unterschied zu den Werten von Wertemengen (Kapitel 2) hat jedes Objekt seine individuelle Identität.

Das ER-Modell ist auch Grundlage für weitergehende Beschreibungsmittel: Die Strukturen und Beziehungen der Daten, die eine Datenbank enthalten soll, das sog. *konzeptionelle Schema*, beschreibt man mit Begriffen des ER-Modells. Im Gebiet des Software-Entwurfs ist die *Unified Modeling Language* (*UML*) zu einer bedeutenden, viele Modellierungsaspekte umfassenden Spezifikationssprache für Software-Strukturen geworden. Das ER-Modell bildet die Grundlage für die Klassendiagramme, eine der Teilsprachen von UML.

Für Spezifikationen im ER-Modell kann man grafische oder textuelle Notationen verwenden. Wegen der besseren Anschaulichkeit benutzen wir hier nur die grafische Notation. Das ER-Modell basiert auf drei Grundbegriffen:

- *Entity*: Ein Objekt des Themenbereiches.
- *Relation*: Eine Beziehung zwischen Objekten.
- *Attribut*: Eine Eigenschaft von Objekten, beschrieben durch einen Wert.

In Abb. 6.2 ist ein Ausschnitt einer Firmenorganisation spezifiziert. Es sind die Entity-Mengen Angestellte, Abteilung und Manager, die zweistelligen Relationen arbeitet in und leitet und Attribute wie Name, Gehalt und Ort angegeben. Wir können dann die Daten von konkreten Firmenorganisationen so strukturieren, dass sie dem spezifizierten Schema genügen.

In Abb. 6.24 ist eine konkrete Ausprägung zu dem Schema angegeben: Die Entity-Menge Angestellte enthält drei Objekte, die in Beziehung stehen zu den vier Abteilungsobjekten. Die Attributwerte der Objekte sind mit ihnen durch Pfeile verbunden.

Wir können Abb. 6.24 auch als den Inhalt einer Datenbank auffassen, deren Schema durch das ER-Diagramm in Abb. 6.2 spezifiziert ist.

6.3.1 Entity-Mengen

Nach diesem einführenden Beispiel definieren wir nun die Grundbegriffe präziser und geben Beispiele für typische Anwendungen. Der zentrale Begriff des ER-Modells ist die Entity-Menge:

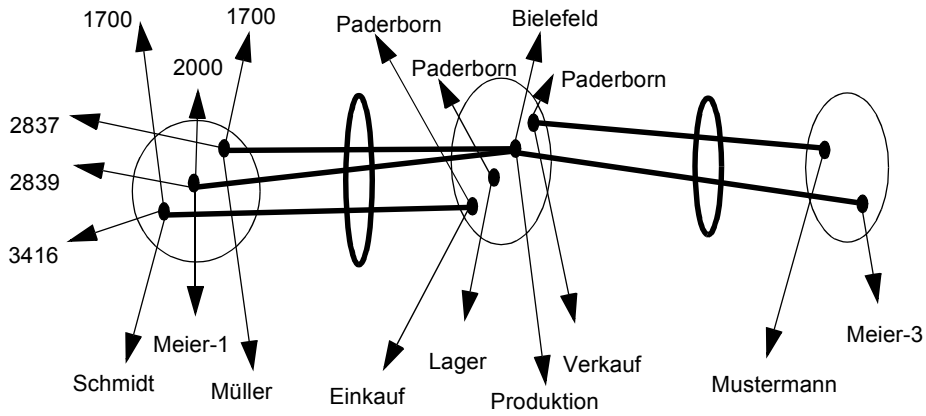


Abbildung 6.24: Konkrete Ausprägung zum Modell der Firmenorganisation in Abb. 6.2

Definition 6.6: Entity-Menge

*Eine **Entity-Menge** repräsentiert eine Zusammenfassung von Objekten, die im Modell als gleichartig angesehen werden, d. h. sie werden durch die gleichen Attribute charakterisiert und können an den gleichen Relationen beteiligt sein. In einem Modell steht eine Entity-Menge für die ggf. nicht-endliche Menge aller in Frage kommenden Objekte dieser Art. Eine **konkrete Ausprägung zu einer Entity-Menge** ist eine endliche Teilmenge der Objekte dieser Art. In der grafischen Notation wird eine Entity-Menge durch ein Rechteck mit einem Namen darin angegeben. In einer konkreten Ausprägung geben wir die Objekt-*

Menge als Ellipse und die Objekte als Punkte ggf. mit identifizierenden Namen an. ■

Die Entity-Menge Abteilung aus unserem einführenden Beispiel steht also für die Menge aller Abteilungen, die wir in konkreten Ausprägungen zu unserem Modell zulassen wollen. In Abb. 6.25 ist eine konkrete Ausprägung für eine Firma in vier Abteilungen dargestellt.

Objekte im ER-Modell (auch *Entities* genannt) repräsentieren Gegenstände des modellierten Themenbereiches. Jedes Objekt hat eine eindeutige Identität, die es von allen anderen Objekten unterscheidet. Zwei Objekte sind also immer verschieden, auch wenn alle ihre Eigenschaften übereinstimmen. Dieser Objektbegriff stimmt mit dem in objektorientierten Datenbanken und in objektorientierten Programmiersprachen überein. In einer Sprache wie Java entsprechen Klassen den Entity-Mengen. Bei der Ausführung von Java-Programmen werden zu einer Klasse Objekte gebildet, die alle eine eindeutige Identität und gleichartige Attribute haben.

Dieser Objektbegriff unterscheidet sich grundsätzlich von dem Begriff der typisierten Werte: Wenn zwei zusammengesetzte Werte in allen ihren Komponenten übereinstimmen, sind sie nicht unterscheidbar, da sie nur vergleichbar, aber nicht identifizierbar sind. Hierin unterscheidet sich auch die Modellierung mit dem ER-Modell grundsätzlich von der mit Wertemengen, die wir in Kapitel 2 beschrieben haben: Die Elemente von Wertemengen werden nur durch ihre Werte unterschieden, die Elemente von Entity-Mengen durch ihre Identität.

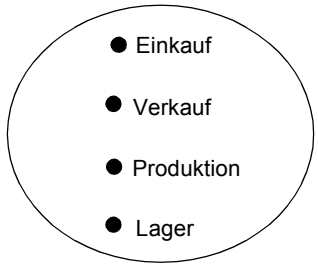


Abbildung 6.25: Konkrete Ausprägung der Entity-Menge Abteilung

6.3.2 Attribute

Attribute werden eingeführt, um Eigenschaften von Entities zu beschreiben. Einer Entity-Menge im Modell können einige Attribute zugeordnet werden. Sie werden durch Ellipsen notiert, die mit dem Rechteck ihrer Entity-Menge verbunden sind und den Attributnamen als Inschrift enthalten.

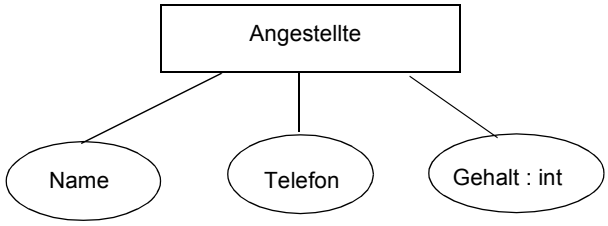


Abbildung 6.26: Entity-Menge mit Attributen

In Abb. 6.26 werden der Entity-Menge Angestellte die Attribute Name, Telefon und Gehalt zugeordnet.

Definition 6.7: Attribut

*Ein **Attribut** ist eine Funktion, die jeder Entity aus der konkreten Ausprägung einer Entity-Menge einen Wert zuordnet.* ■

Alle Entities einer Entity-Menge haben also die gleichen Attribute mit möglicherweise unterschiedlichen Werten.

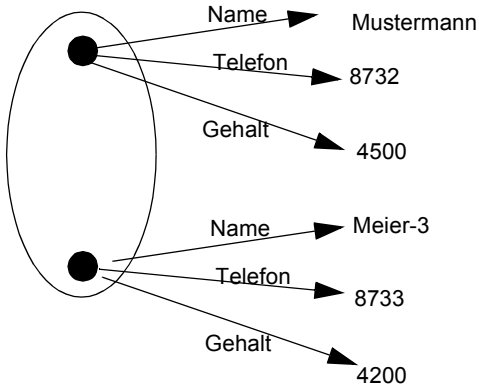


Abbildung 6.27: Konkrete Ausprägung zur Entity-Menge Angestellte aus Abb. 6.26

Abb. 6.27 zeigt zwei Entities der Menge Angestellte mit ihren Attributwerten. Der Wertebereich eines Attributes kann im Modell zum Attributnamen explizit angegeben werden, wie z. B. int beim Attribut Gehalt in Abb. 6.2. Natürlich kann derselbe Attributwert vielfach im System vorkommen, z. B. können zwei Angestellte dieselbe Telefonnummer haben. Hierin unterscheiden sich Attributwerte von Objekten, die eindeutig identifizierbar sind.

Die Identität von Entities kann auch durch Attributwerte explizit gemacht werden:

Definition 6.8: Schlüsselattribut

Ein Attribut, dessen Wert jede Entity einer konkreten Ausprägung einer Entity-Menge eindeutig identifiziert, heißt ***Schlüsselattribut****. Sein Name wird im Modell durch Unterstreichen hervorgehoben. Auch die Werte mehrerer Attribute können zusammen die eindeutige Unterscheidung leisten. Sie werden dann alle als Schlüsselattribute gekennzeichnet.* ■

Im Beispiel von Abb. 6.2 ist das Attribut Name als Schlüsselattribut gekennzeichnet. Eine konkrete Ausprägung darf dann nicht mehrere Entities enthalten, deren Werte der Schlüsselattribute übereinstimmen. In unserem Beispiel müssen ggf. bei Namensgleichheit die Namen, z. B. durch Anhängen einer Nummer, eindeutig gemacht werden.

6.3.3 Relationen

Relationen modellieren Beziehungen zwischen Entities aus Entity-Mengen.

Definition 6.9: Relation

Eine ***n-stellige Relation R*** *verknüpft Entities aus n Entity-Mengen* ***E1, ..., En,*** *wobei* $n \geq 2$*. Sie wird grafisch repräsentiert durch eine Raute, die mit den n Entity-Mengen verbunden ist (siehe Abb. 6.28). Eine* ***konkrete Ausprägung*** *von R ist eine endliche Menge von n-Tupeln* $(e_1, ..., e_n)$ *mit* $e_i \in E_i$ *für* $i = 1, .., n$*. In der Grafik verbinden wir die Komponenten eines Tupels durch Linien und fassen alle Tupel durch eine Ellipse zusammen.* ■

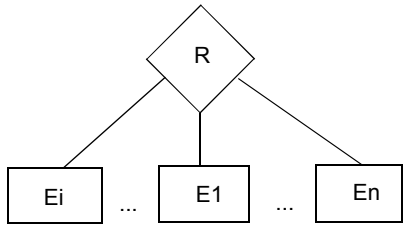


Abbildung 6.28: Grafische Notation einer n-stelligen Relation

Abb. 6.29 zeigt die 2-stellige Relation arbeitet in zwischen den Entity-Mengen Angestellte und Abteilungen sowie eine konkrete Ausprägung mit zwei Tupeln. Der Relationsbegriff entspricht dem über Wertemengen definierten Begriff. Allerdings sind hier die Wertebereiche auf Entity-Mengen eingeschränkt.

Die grafische Notation einer Relation legt die Reihenfolge der Komponenten im Tupel nicht fest. Beim Übergang auf eine textuelle Darstellung ist dafür zusätzliche Information nötig. Häufig kann man aus den Namen der Relation und der beteiligten Entity-Mengen erkennen, welche Rollen die Entities jeweils in der Relation spielen: Zum Beispiel ist in der Relation arbeitet in klar, dass eine Angestellte in einer Abteilung arbeitet und nicht um-

gekehrt. Wir können die Rolle, die eine Entity-Menge in einer Relation spielt, auch explizit machen, indem wir die Kante mit einem Namen beschriften.

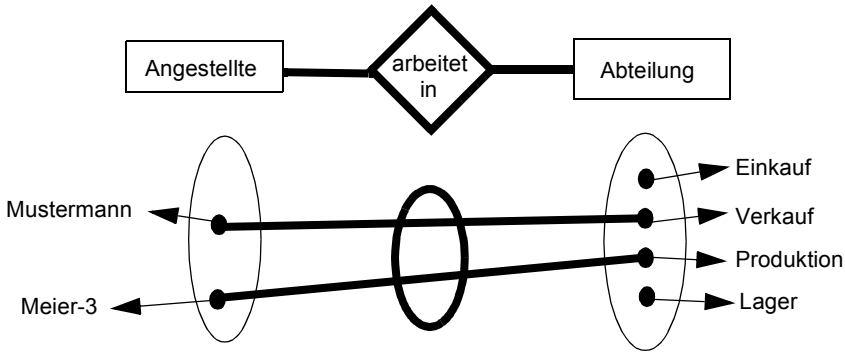


Abbildung 6.29: Relation „arbeitet in“ und eine konkrete Ausprägung dazu

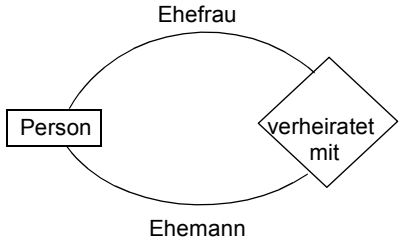


Abbildung 6.30: Relation mit Angabe von Rollen

In Abb. 6.30 haben wir so die beiden Rollen der Relation verheiratet mit unterschieden, die beide auf die Entity-Menge Person führen.

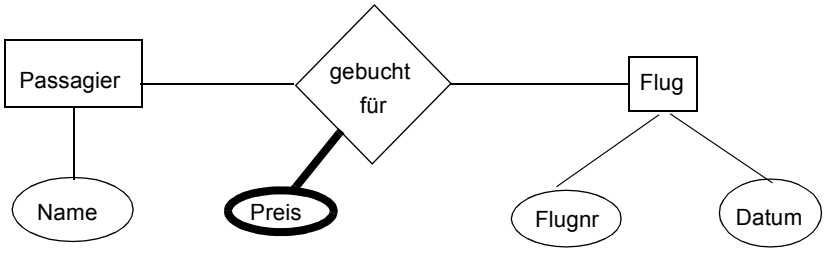


Abbildung 6.31: Relation mit Attribut

Auch Relationen können Attribute zugeordnet werden. Sie beschreiben Eigenschaften zu jedem Tupel der Relation. Das Beispiel der Relation gebucht für in Abb. 6.31 macht die

Notwendigkeit dieses Konzeptes deutlich. Das Attribut Preis modelliert eine Eigenschaft jeder Buchung eines Fluges und nicht etwa eine Eigenschaft der Passagiere oder der Flüge.

An diesem Beispiel wollen wir unterschiedliche Ausdrucksmöglichkeiten der Modellierung mit Relationen verdeutlichen: Man könnte natürlich Buchungen auch als Entity-Menge statt als Relation modellieren, wie in Abb. 6.32. Dann zerfällt die ursprüngliche Relation gebucht für in zwei Relationen hat gebucht zwischen Passagier und Buchung und ist gebucht zwischen Buchung und Flug. Das Attribut Preis wird dann der Entity-Menge Buchung zugeordnet. Die beiden Varianten drücken jedoch Unterschiedliches aus: In der Entity-Menge Buchung kann es mehrere Entities geben, die jeweils denselben Passagier mit demselben Flug verknüpfen. Damit ist zulässig, dass eine Person mehrere Plätze für denselben Flug gekauft hat. Das ist in der Relation gebucht für nicht möglich, weil die Tupelmenge gleiche Tupel nicht mehrfach enthalten kann. Dies auszudrücken, ist bei der Modellierung aber möglicherweise nötig, weil zum Beispiel beschrieben werden soll, welche Personen sich an Bord eines Fluges befinden.

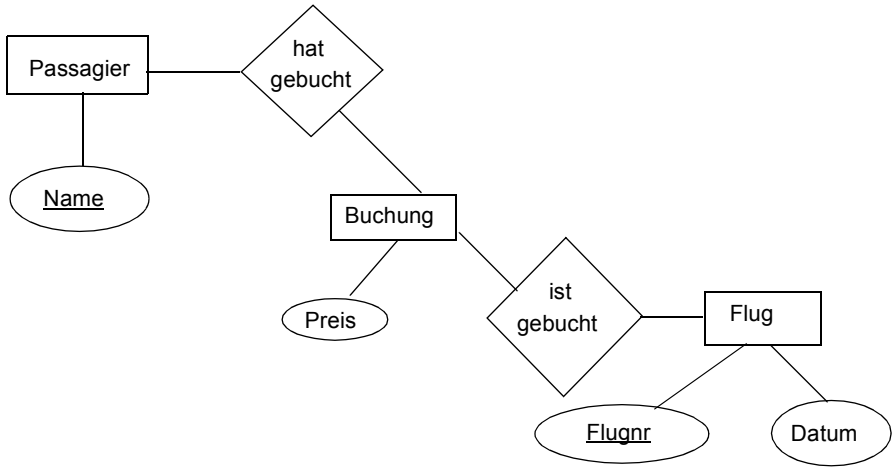


Abbildung 6.32: Relation aus Abb. 6.31 zerlegt in eine Entity-Menge und zwei Relationen

Abb. 6.33 zeigt konkrete Ausprägungen für die beiden Varianten, die diesen Unterschied verdeutlichen.

Relationen in der Form, wie wir sie bisher eingeführt haben, sagen über konkrete Ausprägungen nur aus, dass einige Entities aus den beteiligten Mengen in der angegebenen Beziehung stehen können. Häufig möchte man aber schärfere Bedingungen für konsistente Ausprägungen formulieren. Man möchte etwa spezifizieren, dass jeder Angestellte durch die Relation arbeitet in mit genau einer Abteilung verbunden ist. Solche Restriktionen werden durch Kardinalitäten formuliert.

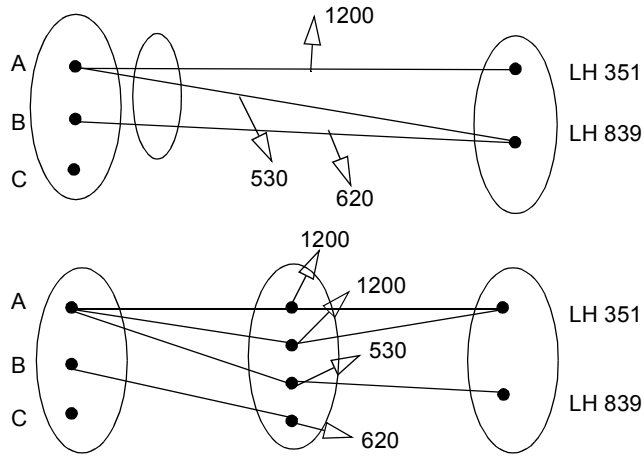


Abbildung 6.33: Konkrete Ausprägungen zu den Relationen in Abb. 6.31 und 6.32

Definition 6.10: Kardinalität

Die Kante, die eine Entity-Menge mit der Relation verbindet, kann mit einer Angabe zur ***Kardinalität*** *in der Form* ***[m, n]*** *markiert werden. Sie bestimmt, dass in einer konkreten Ausprägung jede Entity aus der entsprechenden Menge in mindestens m Tupeln der Relation vorkommen muss und in höchstens n Tupeln vorkommen darf. m und n sind in der Form [m, n] nicht negative ganze Zahlen mit* $m \leq n$. *Ein * statt n drückt aus, dass die Anzahl nach oben unbegrenzt ist.* ■

Häufig verwendet werden folgende Kardinalitätsangaben:

[1] Jede Entity kommt in genau einem Tupel vor; die Relation ist eine totale Funktion dieser Entity-Menge auf die übrigen Rollen der Relation.

[0, 1] Jede Entity kommt in höchstens einem Tupel vor, d. h. die Relation ist eine partielle Funktion auf die übrigen Rollen.

[0 ,*] Jede Entity kann in beliebig vielen Tupeln vorkommen; es wird keine Einschränkung gemacht, die Angabe kann auch weggelassen werden.

In unserem Beispiel würde man die Angestellten in der Relation arbeitet in mit [1,1] markieren. Wenn man ausdrücken wollte, dass jede Abteilung mindestens einen Angestellten hat, würde man diese Rolle mit der Kardinalität [1, *] markieren. In der Praxis sollte man solche Restriktionen jedoch nicht zu scharf wählen. Das obige Beispiel lässt eine leere Abteilung nicht zu; deshalb würden beim Einführen neuer Abteilungen und beim Entfernen bestehender vorübergehend inkonsistente Ausprägungen entstehen.

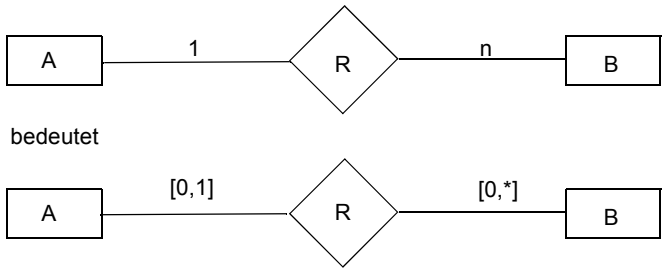


Abbildung 6.34: Kurznotation für spezielle Kardinalitäten

Für Kardinalitäten in 2-stelligen Relationen ist auch die Kurznotation aus Abb. 6.34 gebräuchlich.

Es gibt auch ER-Dialekte, in denen die hier eingeführten Notationen eine komplementäre Bedeutung haben: [m, n] an einer Rolle *E* bedeutet, dass sich mindestens m und höchstens n Tupel nur in Entities aus *E* unterscheiden und in den übrigen Rollen übereinstimmen. Bei 2-stelligen Relationen sind dann, gegenüber der oben eingeführten Bedeutung, die Kardinalitätsangaben gerade zwischen den Rollen vertauscht.

Abb. 6.35 zeigt einige Relationen mit Kardinalitätsangaben, die für häufig vorkommende Muster stehen. Beim Verstehen und beim Entwerfen solcher Relationen sollte man die Bedeutung der Kardinalitäten auch systematisch verbal formulieren:

1. Jedes Auto-Exemplar hat genau eine Automarke. Zu einer Automarke können beliebig viele Autos modelliert sein.
2. Jede Publikation hat mindestens einen Autor.
3. Eine Sinfonie stammt von genau einem Komponisten.
4. Es gibt auch unverheiratete Personen. Polygamie ist in diesem Modell nicht vorgesehen.
5. Jede Person hat höchstens einen Vater. Von manchen Personen ist der Vater nicht modelliert. (Andernfalls gäbe es keine konsistente Ausprägung!)
6. Veranstaltungen werden höchstens dreimal (pro Woche) angeboten. Im Stundenplan sind Termine nicht mehrfach belegt.

Als letztes ER-Konstrukt führen wir eine Relation ein, mit der Spezialisierungen modelliert werden. Sie hat den Namen **IST** (engl. *is-a*).

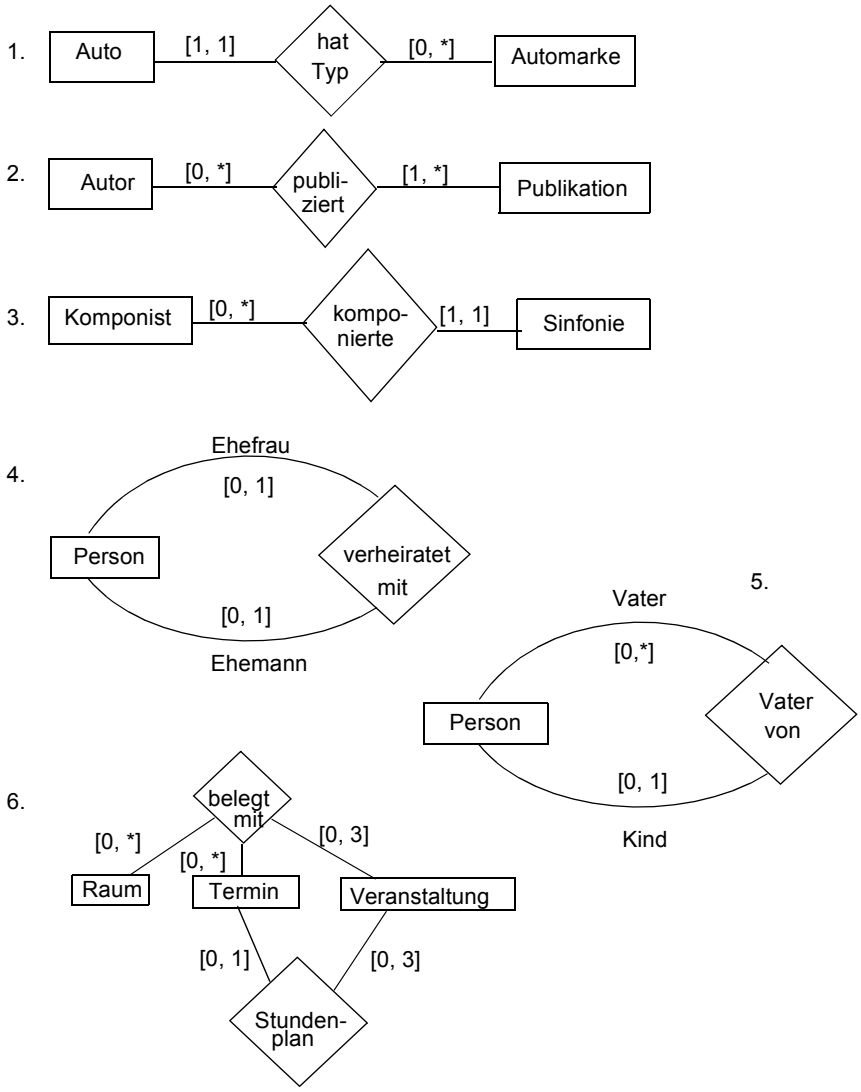


Abbildung 6.35: Einige Relationen mit Kardinalitätsangaben

Definition 6.11: IST-Relation

*Zwei Entity-Mengen A und B stehen in der **Relation A IST B**. Es bedeutet, dass in konkreten Ausprägungen jedes Element aus A auch Element aus B ist. B modelliert die allgemeinere Entity-Menge, A eine speziellere dazu. Die Relation wird durch einen Pfeil von A nach B notiert mit einer Route, die mit IST beschriftet ist (siehe Abb. 6.36). Die Entities in A erben alle Attribute von B*

und können noch weitere Attribute haben, die spezielle Eigenschaften von A beschreiben. ■

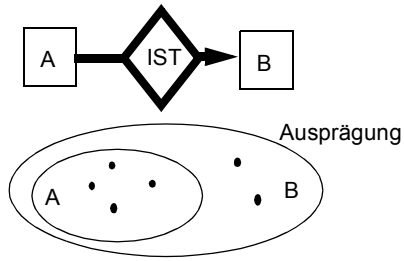


Abbildung 6.36: Notation der IST-Relation und einer Ausprägung

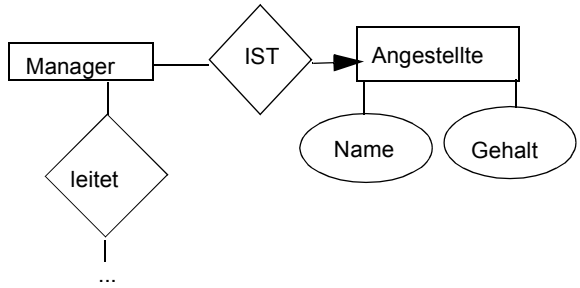


Abbildung 6.37: Eine IST-Relation spezialisiert Angestellte zu Managern

Abb. 6.37 zeigt ein Beispiel, in dem zu den Angestellten einer Firma eine speziellere Entity-Menge der Manager modelliert wird. In diesem Beispiel wird ausgedrückt, dass nur Angestellte aus der Teilmenge der Manager Leitungsaufgaben übernehmen können. Die Entities der Menge Manager erben alle Attribute der allgemeineren Menge Angestellte. Das gilt natürlich auch für Schlüsselattribute mit ihrer Schlüsseleigenschaft: Wenn der Name für alle Angestellten eindeutig ist, dann ist er es auch für die Teilmenge der Manager.

Mit der IST-Relation werden Teilmengenbeziehungen zwischen Entity-Mengen eingeführt. Auch wenn manche Entities in mehreren Entity-Mengen enthalten sein können, so gilt doch weiterhin, dass jede Entity eine eindeutige Identität hat.

Die IST-Relationen können zu mehrstufigen Hierarchien zusammengesetzt werden. Sie dürfen natürlich keine Zyklen bilden.

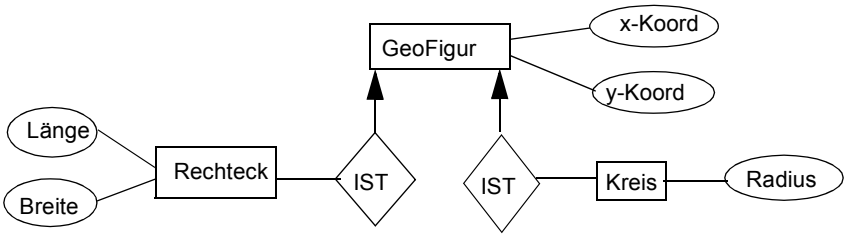


Abbildung 6.38: IST-Relation mit disjunkten Spezialisierungen

Häufig modelliert man zu einer einzigen allgemeinen Entity-Menge mehrere verschiedene spezialisierte Mengen: Abb. 6.38 zeigt als Beispiel GeoFigur mit ihren Spezialisierungen Rechteck und Kreis. Hier wird auch deutlich, dass die Attribute auf der jeweils passenden Hierarchieebene zugeordnet werden sollen: Jede geometrische Figur hat Koordinaten, die sie auf der Zeichenfläche lokalisieren. Es wäre eine schlechte Modellierung, diese Attribute nicht der allgemeinen Entity-Menge, sondern jeder ihrer Spezialisierungen zuzuordnen.

In diesem Beispiel sind die Ausprägungen der spezialisierten Entity-Mengen immer *disjunkt*: Eine Kreis-Entity kann nicht zugleich auch eine Rechteck-Entity sein. Das ist in den meisten sinnvollen Modellen so, wird aber vom ER-Modell nicht vorgeschrieben.

Die IST-Relation mit der Vererbung von Attributen und ihrer Hierarchieeigenschaft entspricht der Vererbungsrelation zwischen Ober- und Unterklassen in objektorientierten Programmiersprachen.

Zum Abschluss stellen wir ein etwas größeres Modell im Zusammenhang vor. Es werden darin einige Aspekte einer Fluggesellschaft modelliert. Das Modell ist in Abb. 6.39 angegeben. Die meisten Angaben darin sind durch die Wahl der Namen selbsterklärend. Wir wollen hier nur auf Modellierungstechniken hinweisen, die für den Entwurf solcher Modelle wichtig sind:

1. Es kommt eine IST-Relation vor. Sie drückt aus, dass Piloten spezielle Angestellte sind. Dies wird insbesondere benötigt, um die Relation kann fliegen hinreichend präzise formulieren zu können und das Attribut Flugstunden nicht allen Angestellten zuordnen zu müssen.
2. Entities in allen Mengen werden durch Schlüsselattribute eindeutig identifiziert. Bei den Passagieren wird das Paar aus Name und Adresse als eindeutig gefordert. Die Namen der übrigen Schlüsselattribute deuten an, dass man jeweils eine Nummerierung eingeführt hat, um die Eindeutigkeit zu erreichen. Solch ein Vorgehen ist sehr empfehlenswert, besonders wenn das Modell praktisch in einer Datenbank realisiert werden soll.
3. Wir möchten besonders hinweisen auf die Relation Typ, welche die Entity-Mengen Flugzeug und Flugzeugtyp verbindet: Die Kardinalität [1, 1] sagt aus, dass eine totale Funktion jedem Flugzeug (das die Fluggesellschaft benutzt) dessen Typ zuordnet. Sol-

che Unterscheidungen von *Typ* und *Exemplar* kommen vielfach in Modellierungen vor und sind wichtig, damit Relationen und Attribute sachgerecht zugeordnet werden können. Die Fähigkeit eines Piloten soll durch bestimmte Flugzeugtypen ausgedrückt werden und nicht durch Exemplare, die diesem Typ angehören.

Das Muster „totale Funktion von Exemplaren auf ihren Typ“ kommt ein zweites Mal vor: Es verbindet die Abflüge als Exemplare mit den Flügen als Typen. Der Unterschied wird deutlich, wenn man sich vorstellt, dass die Flüge Einträge im Flugplan repräsentieren, der z. B. wöchentlich abgearbeitet wird, während jede Abflug-Entity an einem bestimmten Datum stattfindet. Damit ist klar, dass sich die Relation für das Buchen auf Abflüge beziehen muss.

Häufig wird auch versucht, Typ-Exemplar-Beziehungen bei der Modellierung durch die Spezialisierung IST auszudrücken. Das ist jedoch ein schwerer Entwurfsfehler: Typen und Exemplare sind verschiedenartige Entities; zwischen ihren Entity-Mengen kann nicht die Teilmengenbeziehung bestehen.

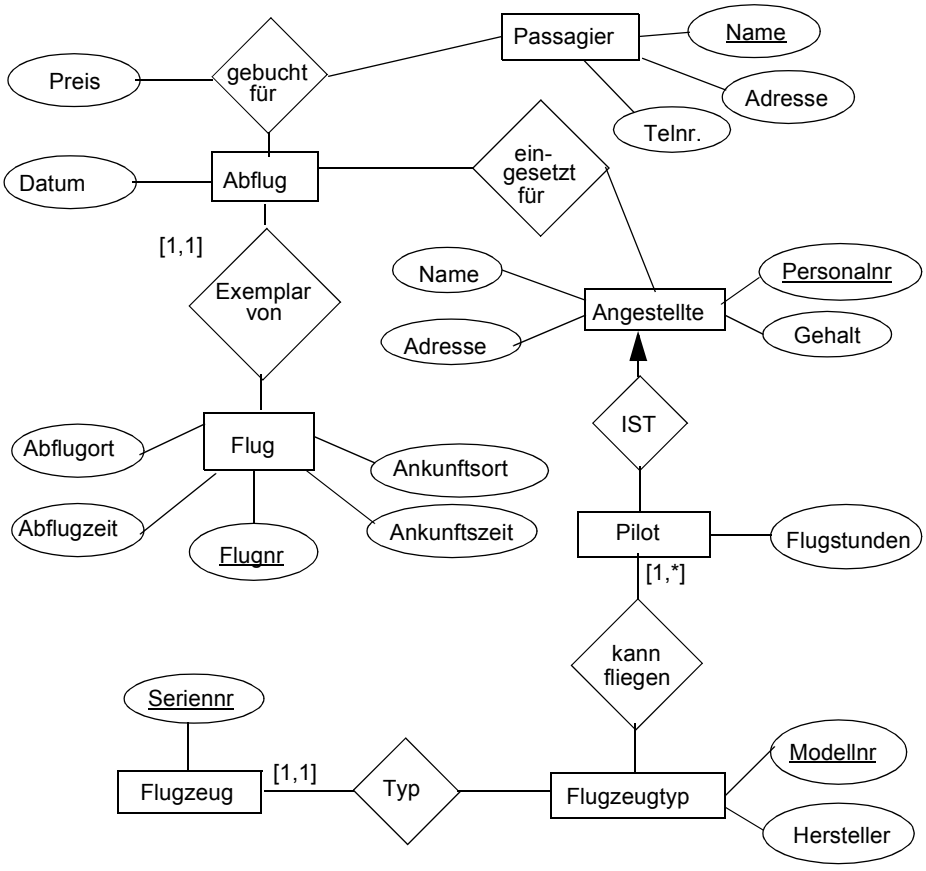


Abbildung 6.39: Ausschnitt aus der Modellierung einer Fluggesellschaft

6.4 Klassendiagramme in UML

Die *Unified Modeling Language (UML)* ist derzeit wohl die am weitesten verbreitete und am häufigsten eingesetzte Sprache zur Modellierung von Systemen. Wir stellen hier Klassendiagramme als Teilsprache von UML vor, da sie eng mit dem Entity-Relationship-Modell aus Abschnitt 6.3 verwandt sind und sich für die gleichen Modellierungsaufgaben eignen. Im Abschnitt 7.1.5 stellen wir auch die Teilsprache der *Statecharts* in UML vor, die auf dem Kalkül der endlichen Automaten basiert.

UML wird für den Gebrauch durch Menschen als visuelle Sprache verwendet: Die Sprachelemente werden durch geometrische Formen mit textuellen Annotationen dargestellt. Eine spezielle XML-Sprache, *XML Metadata Interchange (XMI)*, ist definiert, in der UML-Werkzeuge Entwürfe speichern und lesen.

UML versucht die Modellierung unterschiedlicher Aspekte von Systemen möglichst umfassend abzudecken. Es umfasst 13 Teilsprachen, die hier *Diagrammtypen* genannt werden und jeweils auf die Modellierung bestimmter Aspekte spezialisiert sind. So modelliert man mit *Klassendiagrammen* die Systemstruktur, statische Eigenschaften und Beziehungen zwischen Strukturelementen; mit der Teilsprache für *Statecharts* werden Abläufe von Operationen modelliert. Ein Entwurf in UML besteht im Allgemeinen aus mehreren Diagrammen unterschiedlicher Typen, die durch Namen von Elementen verbunden sind und verschiedene Sichten auf das System beschreiben.

Klassendiagramme in UML basieren auf den gleichen Grundkonzepten wie das Entity-Relationship-Modell: Entity-Mengen (hier Klassen) mit Attributen und Relationen und Hierarchie-Beziehungen zwischen den Entity-Mengen. Ein Modell im ER-Kalkül kann recht direkt in ein Klassendiagramm umgeformt werden, das die gleichen Strukturen und Zusammenhänge beschreibt. Abb. 6.40 zeigt ein Klassendiagramm, das dem Modell aus Abschnitt 6.2 entspricht. Die Sprache der Klassendiagramme ist jedoch reichhaltiger, sodass Modelle bei Bedarf detaillierter beschrieben werden können. Wenn UML-Modelle hinreichend konkret ausgearbeitet sind, können daraus Implementierungen, ggf. mit noch auszufüllenden Lücken, erzeugt werden. Hier beschränken wir uns, soweit möglich, auf die Darstellung der UML-Konstrukte, die denen entsprechen, die wir in Abschnitt 6.3 für den ER-Kalkül vorgestellt haben.

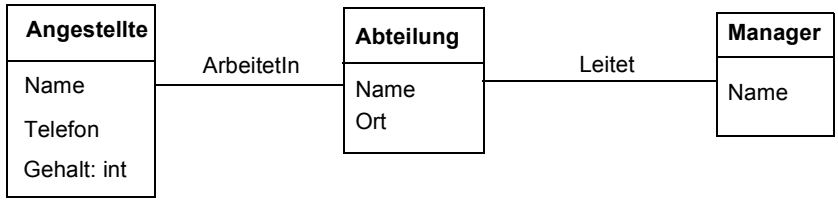


Abbildung 6.40: Ausschnitt aus dem Modell einer Firmenorganisation, gemäß Abb. 6.2

6.4.1 Klassen mit Attributen

Eine Klasse repräsentiert ebenso wie eine Entity-Menge eine Zusammenfassung von Objekten, die im Modell als gleichartig angesehen werden. Sie wird wie im ER-Modell als Rechteck mit Namen dargestellt. Attribute der Klasse werden angegeben, indem man das Rechteck horizontal in zwei Felder einteilt; im oberen steht der Klassenname, im unteren die Folge von Attributen mit oder ohne Typangaben. In einem dritten Feld könnten darunter Operationen angegeben werden. Abb. 6.41 stellt eine ER-Menge und eine Klasse einander gegenüber, die das Gleiche modellieren. Eine Kennzeichnung von Attributen als Schlüsselattribute ist in UML nicht vorgesehen.

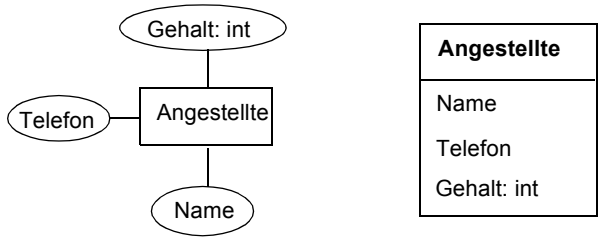


Abbildung 6.41: Entity-Menge und Klasse mit Attributen

Konkrete Ausprägungen zu einer Klasse, also Objekte, können in UML auch dargestellt werden. Es wird dafür die gleiche Grafik wie für Klassen verwendet. Zum Namen des Objektes wird der Name der Klasse als Typ angegeben und beides unterstrichen, siehe Abb. 6.42. Einzelne Objekte, die für die statische Modellierung relevant sind, können so in einem Klassendiagramm angegeben werden. Eine Darstellung der Menge aller Objekte einer Klasse ist in UML nicht vorgesehen.

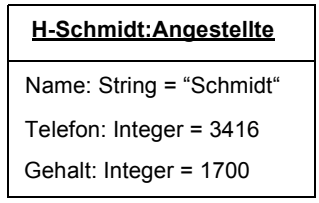


Abbildung 6.42: Ein Objekt

6.4.2 Assoziationen

Relationen des ER-Modells heißen in UML *Assoziationen*. Zweistellige Assoziationen werden als Linie zwischen zwei Klassen dargestellt, die mit dem Namen der Assoziation beschriftet ist. Die Enden der Linie können mit Namen beschriftet sein, die die Rollen der

Klassen in der Assoziation beschreiben. Assoziationen zwischen mehr als zwei Klassen werden wie im ER-Kalkül durch eine Raute dargestellt. Häufig ist es nützlich, durch einen Pfeil neben dem Namen einer zweistelligen Assoziation die „Leserichtung“ anzugeben. Abb. 6.43 zeigt einige Beispiele für Relationen aus Abschnitt 6.3 in UML-Notation.

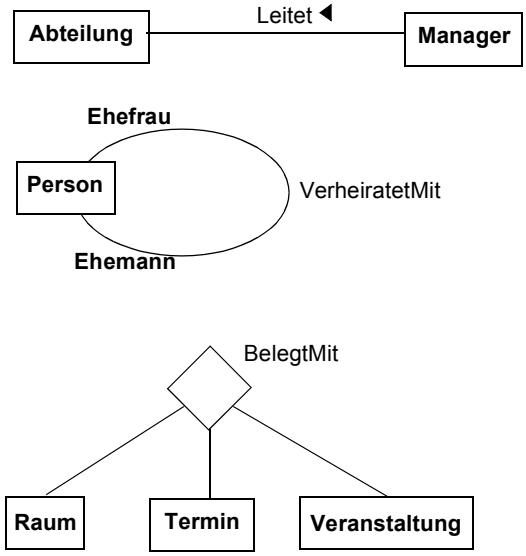


Abbildung 6.43: Einige Assoziationen

In UML können Assoziationen auch als Klassen aufgefasst werden und insbesondere auch Attribute haben. In diesem Fall wird die Assoziation mit einer Klassen-Grafik mit gleichem Namen verbunden, wie in Abb. 6.44.

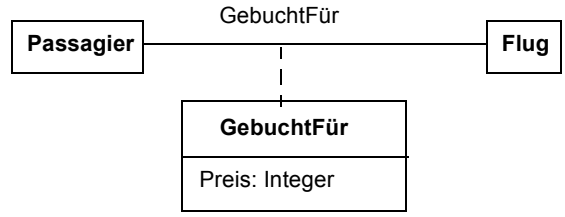


Abbildung 6.44: Eine Assoziation mit Attributen

Wie im ER-Kalkül können die Enden einer Assoziation mit Angaben zu Kardinalitäten *(UML: multiplicities)* annotiert werden. In UML werden zwar fast die gleichen Notationen für Kardinalitäten verwendet wie im ER-Kalkül: *[m, n]* in der ER-Notation entspricht *m..n* in UML. Ihre Bedeutung ist jedoch unterschiedlich definiert. Dies hat zur Konse-

quenz, dass in binären Assoziationen die entsprechende Kardinalität gerade am gegenüberliegenden Ende der Assoziationslinie annotiert wird, siehe Abb. 6.45.

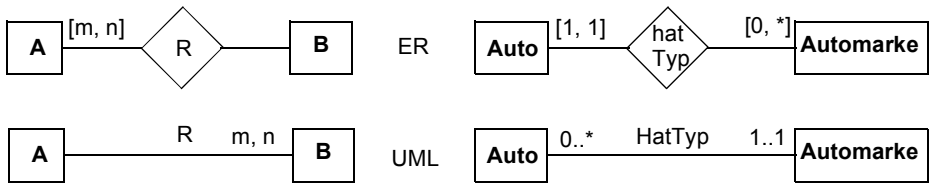


Abbildung 6.45: Kardinalität in binären Assoziationen

Die beiden folgenden Aussagen sind gleichwertig:

- Jedes Objekt aus *A* kommt in den Tupeln der Relation *R* mindestens *m* und höchstens *n* mal vor.
- Jedem Objekt aus *A* ordnet die Relation *R* mindestens *m* und höchstens *n* verschiedene Objekte aus *B* zu.

Die erste Aussage wird besser durch die ER-Notation illustriert, die zweite besser durch die UML-Notation.

Der Unterschied in der Bedeutung der Kardinalitäten für allgemeine *n*-stellige Relationen ist subtiler. Für die in Abb. 6.46 angegebene Relation *s* wird die Bedeutung der Kardinalitätsangabe wie folgt erklärt:

> ER: Jedes Objekt aus E_1 kommt in den Tupeln der Relation *S* mindestens *m* und höchstens *n* mal vor (vergl. Def. 6.10).
>
> UML: Jeder Kombination von Objekten aus E_2, ..., E_n ordnet die Relation *S* mindestens *m* und höchstens *n* Objekte aus E_1 zu.

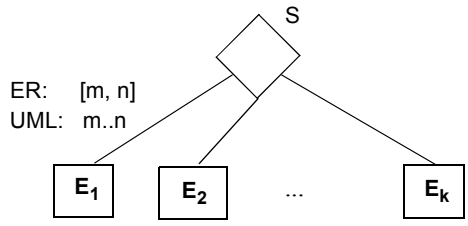


Abbildung 6.46: Kardinalität in k-stelliger Relation

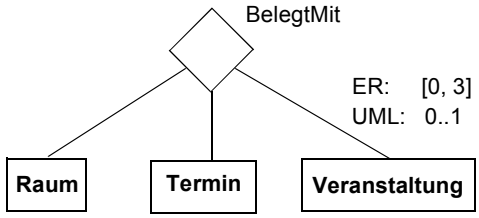


Abbildung 6.47: Unterschiedliche Einschränkungen in dreistelliger Assoziation

Das Beispiel in Abb. 6.47 zeigt, dass mit den beiden unterschiedlichen Bedeutungen der Kardinalitäten unterschiedliche Restriktionen formuliert bzw. nicht formuliert werden können: Die ER-Angabe *[0, 3]* an der *Veranstaltung* der *belegtMit* Relation drückt aus, dass für jede Veranstaltung zwischen *0* und *3 Raum-Termin*-Kombinationen vorgesehen sind. Die UML-Angabe *0..1* an derselben Stelle drückt aus, dass für jede *Raum-Termin*-Kombination höchstens eine *Veranstaltung* vorgesehen ist. Diese Restriktionen sind in der jeweils anderen Notation nicht formulierbar, weil Kombinationen von Klassen nicht mit Kardinalitäten annotiert werden.

UML bietet zwei weitere Kennzeichen von Assoziationsenden, um zu modellieren, dass Objekte einander enthalten (Abb. 6.48): Die *Komposition* (gefüllte) Raute drückt aus, dass jedes Teil unverzichtbar zu genau einem Ganzen gehört. Die *Aggregation* (offene Raute) drückt aus, dass Objekte zu größeren Objekten zusammengefasst werden, aber prinzipiell auch allein existieren können.

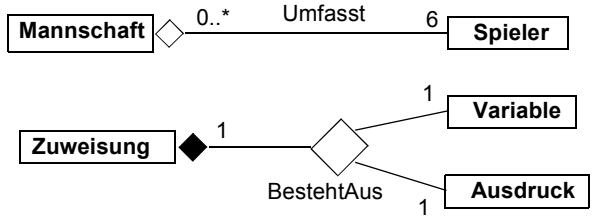


Abbildung 6.48: Aggregation und Komposition

Die Generalisierung ist eine weitere spezielle Relation zwischen Klassen. Sie dient, wie die IST-Relation, im ER-Modell zur Modellierung von Abstraktions-Hierarchien. In Abb. 6.49 ist *GK* eine generalisierte Klasse zu den Klassen *SK1* und *SK2*. Man nennt *GK* auch die *Oberklasse* und *SK1*, *SK2 Unterklassen*. Jede Generalisierung kann benannt (hier *Arten*) und durch Eigenschaften wie *complete/incomplete, disjoint/overlapping* charakterisiert werden.

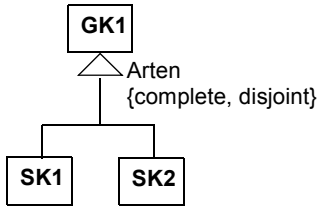


Abbildung 6.49: Notation der Generalisierung

Hinsichtlich der Mengen der Objekte definiert die Generalisierung eine Teilmengenbeziehung: Jedes Objekt einer der Unterklassen ist auch ein Objekt der Oberklasse. Ist eine Generalisierung als *complete* angegeben, so gibt es in dem Modell keine weiteren außer den angegebenen Unterklassen. (Es kann trotzdem Objekte geben, die der Oberklasse, aber keiner Unterklasse angehören.) Eine Generalisierung ist *disjoint*, wenn kein Objekt in mehr als einer Unterklasse liegt. Letzteres ist möglich, wenn eine Unterklasse zwei verschiedene Oberklassen hat. Die Generalisierung lässt sich auch als Vererbungsbeziehung auffassen: Objekte einer Unterklasse haben auch alle Attribute und Operationen der Oberklasse, es sei denn, sie werden überschrieben durch Attribute oder Operationen, die mit demselben Namen in der Unterklasse definiert sind. Abb. 6.50 zeigt das Beispiel aus Abb. 6.38 in UML-Notation. Hier kann insbesondere zum Ausdruck gebracht werden, dass Rechteck und Kreis disjunkte Unterklassen von *GeoFigur* sind und dass noch weitere (z. B. Dreieck) modelliert werden sollen.

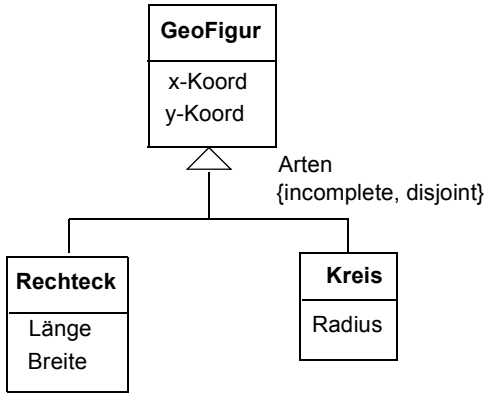


Abbildung 6.50: Beispiel für Generalisierung

Abschließend zeigen wir in Abb. 6.51 das größere Modell aus Abb. 6.39 in UML-Notation. Die Erläuterungen aus Abschnitt 6.3.3 treffen hierauf ebenso zu.

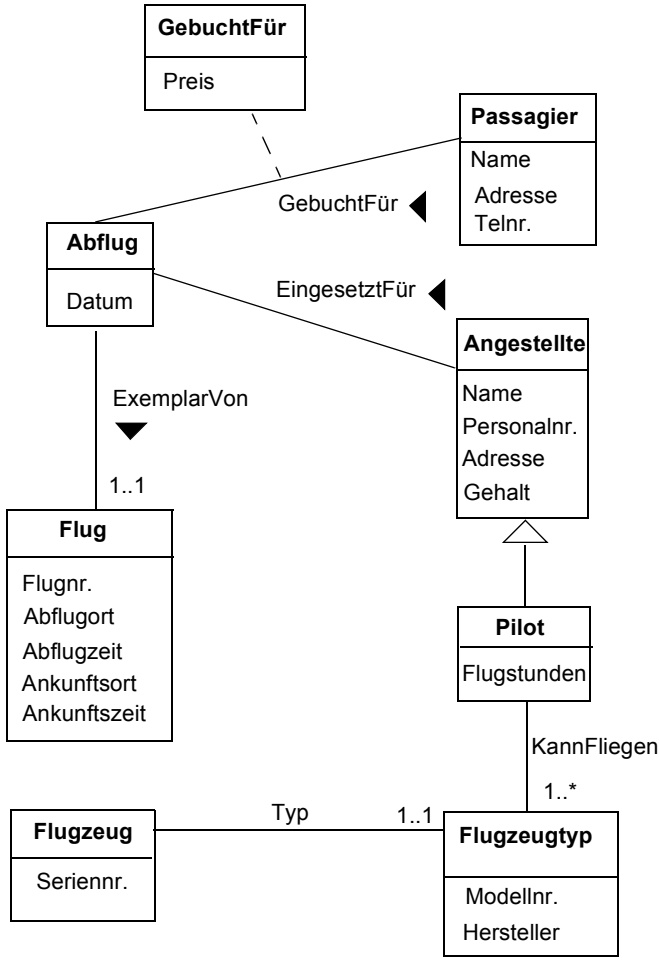


Abbildung 6.51: Klassendiagramm modelliert Ausschnitt einer Fluggesellschaft

Übungen

6.1 Baumstrukturen beschreiben

Menüstrukturen in Benutzungsoberflächen lassen sich durch Bäume darstellen. Die Struktur solcher Bäume wird hier mit Hilfe der folgenden kontextfreien Grammatik *G = (T, N, P, S)* beschrieben:

Startsymbol: S = Menü
Terminale: T = { Befehlsname, Untermenüname}
Produktionen: P = { p1: Menü ::= Menüliste

p2: Menüliste ::= Menüeintrag Menüliste,
p3: Menüliste ::= Menüeintrag,
p4: Menüeintrag ::= Befehlsname,
p5: Menüeintrag ::= Untermenüname Menü }

a) Stellen Sie folgende Menüstruktur als Ableitungsbaum der angegebenen Grammatik dar:

Datei
+ - - -Laden
+ - - -Speichern
+ - - -Exportieren als
+ - - -Text
+ - - - Postscript
Bearbeiten
+ - - -Kopieren
+ - - -Einfügen

Hinweis: Beachten Sie, dass Befehlsname und Untermenüname Terminale der Grammatik sind, deren Schreibweise nicht näher spezifiziert ist. Etwas Vergleichbares findet man in Beispiel Abschnitt 6.4, wo das Terminal Variable für beliebige Variablennamen, z. B. für a oder b, stehen kann.

b) Erweitern Sie die Grammatik so, dass man Menüeinträge durch Trennstriche in Gruppen gliedern kann.

6.2 Menüstrukturen als XML-Sprache

Entwerfen Sie eine XML-Sprache für Menüstrukturen, wie sie Aufgabe 6.1 definiert. Schreiben Sie für die dort angegebene kontextfreie Grammatik eine DTD, die Bäume gleicher Struktur definiert. Geben Sie den in Aufgabe 6.1a gezeigten Baum in XML-Notation zu der DTD an. Erweitern Sie die XML-Sprache wie in Aufgabe 6.1b.

6.3 Notation von Tupeln

Es sei folgende Grammatik $G = (T, N, P, S)$ gegeben:

Startsymbol: S = Start
Produktionen: P = { p1 : Start ::= Tupel,
p2 : Tupel ::= '(' Stelle ',' Stelle ')',
p3 : Stelle ::= Tupel,
p4 : Stelle ::= Ziffer }

a) Geben Sie die Mengen der Terminale und *Nichtterminale* an.

b) Geben Sie drei Sätze an, die von dieser Grammatik erzeugt werden. Die Sätze sollen unterschiedlich viele Ziffern enthalten, aber höchstens 4.

c) Zeichnen Sie den Ableitungsbaum zu einem Satz mit mehr als 2 Ziffern aus Teil (b). Notieren Sie an den Kanten des Baums die Nummer der jeweils benutzten Produktion.

6.4 Tupel-Notation als XML-Sprache

Entwerfen Sie eine XML-Sprache für die in Aufgabe 6.3 definierten Tupelstrukturen. Schreiben Sie für die dort angegebene kontextfreie Grammatik eine DTD, die Bäume gleicher Struktur definiert. Übernehmen Sie die Klammern und Kommata der kontextfreien Grammatik nicht in die DTD. Geben Sie Sätze in XML-Notation zu der DTD an.

6.5 Sprache der Palindrome

Die Sprache der Palindrome lässt sich durch die drei folgenden Bildungsregeln beschreiben:

1. Das leere Wort sowie 0 und 1 sind Palindrome.
2. Falls *w* ein Palindrom ist, sind 0 *w* 0 und 1 *w* 1 Palindrome.
3. Alle Palindrome lassen sich durch endlich viele Anwendungen der Regeln 1 und 2 erzeugen.

a) Geben Sie die Terminale und Nichtterminale sowie das Startsymbol und die Menge der Produktionen der zugehörigen kontextfreien Grammatik an.

b) Untersuchen Sie, ob die folgenden Sätze Element der durch obige kontextfreie Grammatik definierten Sprache sind, und geben Sie gegebenenfalls den zugehörigen Ableitungsbaum an.

Satz1: 010111010
Satz2: 1101101011

6.6 Mehrdeutige Grammatik

Die folgende kontextfreie Grammatik *G = (T, N, P, S)* ist *mehrdeutig*. Die in der Sprache enthaltenen Sätze sind durch Schrägstrich getrennte Listen von Buchstaben.

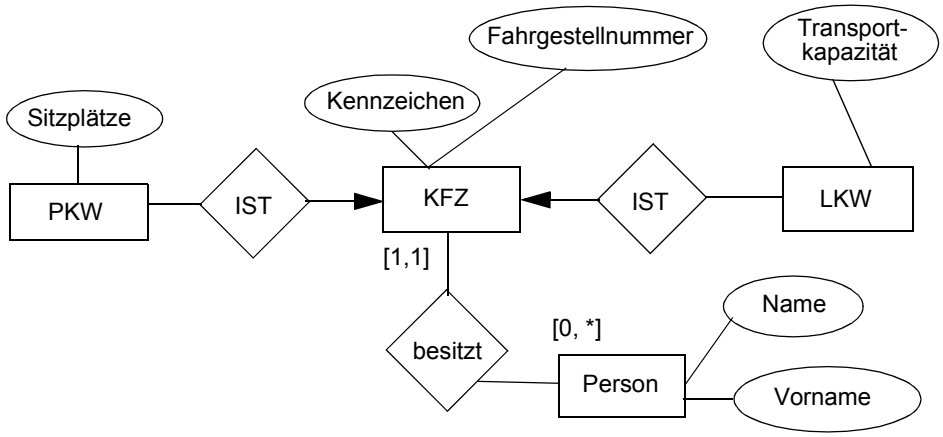
```
Startsymbol:     S = Start
Terminale:       T = {Buchstabe, /}
Nichtterminale:  N = {Start, Liste }
Produktionen:    P = {   p1: Start ::= List,
                         p2: Liste::= Liste ' / ' Liste,
                         p3: Liste::= Buchstabe}
```

a) Zeigen Sie, dass die Grammatik *mehrdeutig* ist.

b) Verändern Sie die Menge der Produktionen so, dass die Grammatik *eindeutig* wird, aber die definierte Sprache gleich bleibt.

6.7 ER-Diagramme verstehen

Es sei folgendes ER-Diagramm gegeben:



Als Beispiel betrachten wir folgende Beschreibung konkreter Entitäten:

PKW1: Kennzeichen PB-XY-123, Fahrgestellnummer 123421, 4 Sitzplätze
PKW2 Kennzeichen PB-KL-188, Fahrgestellnummer 123123, 6 Sitzplätze
PKW3 Kennzeichen HF-AB-345, Fahrgestellnummer 123131, 7 Tonnen Transportkapazität
Person1 Max Meier
Person2 Martha Müller

a) Würde das Beispiel dem Modell widersprechen, wenn
   – Person1 und Person2 den gleichen Nachnamen hätten?
   – PKW1 und PKW2 die gleiche Fahrgestellnummer hätten?
   – PKW1 und LKW1 das gleiche Kennzeichen hätten?

   Begründung!

b) Entsprechen die besitzt-Relationen den im Modell geforderten Kardinalitäten? Begründung!

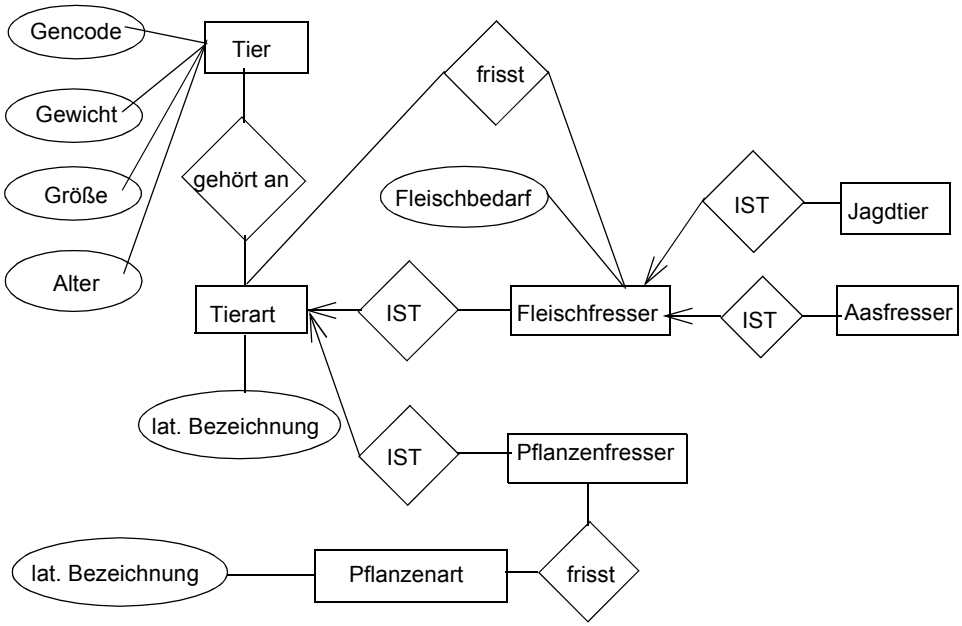
   1. Person1 besitzt PKW1
      Person2 besitzt PKW2
      Person1 besitzt LKW1
   2. Person1 besitzt PKW1
      Person1 besitzt PKW2
      Person1 besitzt LKW1
      Person2 besitzt PKW2
   3. Person 1 besitzt PKW1
      Person 1 besitzt PKW 2
      Person 1 besitzt LKW 1
   4. Person 1 besitzt PKW 1
      Person 2 besitzt LKW 1

c) In dieser Teilaufgabe sollen Sie das Beispiel und die Relation aus (b 1) grafisch darstellen wie in Abb. 6.29: Stellen Sie die konkreten Ausprägungen der Entity-Mengen

KFZ, PKW und LKW grafisch dar. Ergänzen Sie die Attribute der Entitäten. Zeichnen Sie schließlich die besitzt-Relation aus (b 1) in das Diagramm ein.

6.8 ER-Diagramm mit IST-Relationen

Ein Tier hat die Eigenschaften Gencode, Alter, Gewicht und Größe, und es gehört einer Tierart mit lateinischer Bezeichnung an. Eine Tierart ist entweder ein Fleisch- oder ein Pflanzenfresser. Fleischfresser haben einen Fleischbedarf, und sie fressen bestimmte Tierarten. Sie sind entweder Jäger oder Aasfresser. Pflanzenfresser fressen bestimmte Pflanzenarten, die eine lateinische Bezeichnung haben. Das dazugehörige ER-Diagramm sieht so aus:

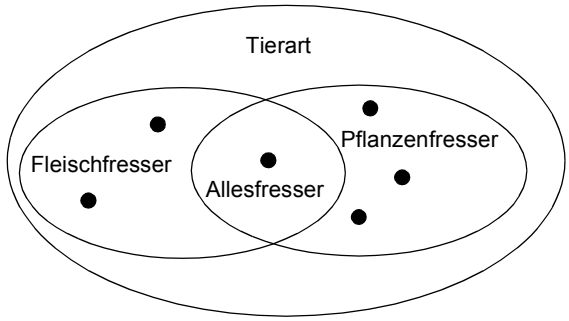


a) Geben Sie alle Attribute der Entity-Menge Fleischfresser an.

b) Unterstreichen Sie Schlüsselattribute aller Entity-Mengen, und begründen Sie Ihre Wahl.

c) Ergänzen Sie die Kardinalitäten aller Relationen.

d) Erweitern Sie das ER-Diagramm um Allesfresser, die sowohl Fleischfresser als auch Pflanzenfresser sind, so dass eine konkrete Ausprägung wie folgt aussieht:



6.9 ER-Diagramme und Getränkeautomaten

a) Formalisieren Sie folgende umgangssprachliche Beschreibung mit Hilfe eines ER-Diagramms:
Eine Getränkesorte hat einen eindeutigen Namen, eine Menge und einen Preis. Die Getränke einer Getränkesorte benötigen zwischen 1 und 3 verschiedene Zutaten. Eine Getränkesorte ist entweder ein Heißgetränk oder ein Kaltgetränk. Heißgetränke haben zudem noch eine Zubereitungszeit.

b) Unterstreichen Sie Schlüsselattribute aller Entity-Mengen, und begründen Sie Ihre Wahl.

c) Geben Sie eine konkrete Ausprägung der im Diagramm vorkommenden Relationen an, die mindestens drei verschiedene Entities enthält.

6.10 Modellieren mit ER-Diagrammen

Betrachten Sie folgenden Sachverhalt: Eine Person sei eindeutig bestimmt durch ihren Namen und ihren Vornamen. Einige Personen sind Studenten, die eine eindeutige Matrikelnummer besitzen. Studenten besuchen normalerweise eine Reihe von Vorlesungen, allerdings finden Vorlesungen nur statt, wenn sich mindestens 5 Studenten dafür interessieren. Vorlesungen haben einen Namen und eine eindeutige Nummer. Jede Vorlesung wird von genau einem Professor gehalten, dem sich neben Namen und Vornamen auch ein Fachgebiet zuordnen lässt.

Modellieren Sie diesen Teil der realen Welt durch ein ER-Diagramm. Vergessen Sie nicht, auch Kardinalitäten und Schlüsselattribute zu definieren.

6.11 ER-Diagramme in UML-Klassendiagramme umschreiben

Geben Sie Klassendiagramme in UML an, die den ER-Diagrammen aus den Aufgaben 6.7, 6.8, 6.9 und 6.10 entsprechen. Übertragen Sie auch die Kardinalitäten aus den ER-Diagrammen.