2. Syntax

Themen dieses Kapitels:

- 2.1 Grundsymbole
- 2.2 Kontext-freie Grammatiken
- Schema für Ausdrucksgrammatiken
- Erweiterte Notationen für kontext-freie Grammatiken
- Entwurf einfacher Grammatiken
- abstrakte Syntax
- 2.3 XML

2.1 Grundsymbole

Grundsymbole:

Programme bestehen aus einer **Folge von Grundsymbolen**. (Ebene (a) auf GPS-1-16)
Jedes Grundsymbol ist eine **Folge von Zeichen**.
Ihre Schreibweise wird z.B. durch **reguläre Ausdrücke** festgelegt.
Grundsymbole sind die **Terminalsymbole der konkreten Syntax**. (Ebene (b) GPS-1-16)

Folgende 4 **Symbolklassen** sind typisch für Grundsymbole von Programmiersprachen:
Bezeichner, Wortsymbole, Literale, Spezialsymbole

1. **Bezeichner (engl. identifier):**

   zur Angabe von Namen, z. B. **`maximum findValue res_val _MIN2`**

   Definition einer Schreibweise durch reg. Ausdruck: *Buchstabe (Buchstabe | Ziffer)**

2. **Wortsymbole (engl. keywords):**

   kennzeichnen Sprachkonstrukte

   Schreibweise fest vorgegeben; meist wie Bezeichner, z. B.**`class static if for`**
   Dann müssen Bezeichner verschieden von Wortsymbolen sein.

   Nicht in PL/1; dort unterscheidet der Kontext zwischen Bezeichener und Wortsymbol:

   `IF THEN THEN THEN = ELSE ELSE ELSE = THEN;`

   Es gibt auch gekennzeichnete Wortsymbole, z.B. **`$begin`**

Literale und Spezialsymbole

2. Literale (engl. literals):

Notation von Werten, z. B.

ganze Zahlen: `7  077  0xFF`
Gleitpunktzahlen: `3.7e-5  0.3`
Zeichen: `'x'  '\n'`
Zeichenreihen: `"Hallo"`

Unterscheide Literal und sein Wert: `"Sage \"Hallo\""` und `Sage "Hallo"`
verschiedene Literale - gleicher Wert: `63  077  0x3F`

Schreibweisen werden durch reguläre Ausdrücke festgelegt

4. Spezialsymbole (engl. separator, operator):

Operatoren, Trenner von Sprachkonstrukten, z. B. `; , = * <=`

Schreibweise festgelegt, meist Folge von Sonderzeichen

Bezeichner und Literale tragen außer der Klassenzugehörigkeit weitere Information:
Identität des Bezeichners und **Wert des Literals**.

Wortsymbole und Spezialsymbole stehen nur für sich selbst, tragen keine weitere Information.

Trennung von Grundsymbolen

In den meisten Sprachen haben
die Zeichen **Zwischenraum, Zeilenwechsel, Tabulator** und **Kommentare**
keine Bedeutung außer zur Trennung von Grundsymbolen; auch **white space** genannt.
z. B. `int pegel;` statt `intpegel;`

Ausnahme **Fortran**:
Zwischenräume haben auch innerhalb von Grundsymbolen keine Bedeutung
z. B. Zuweisung `DO 5 I = 1.5` gleichbedeutend wie `DO5I=1.5` aber
Schleifenkopf `DO 5 I = 1,5`

In **Fortran, Python, Occam** können Anweisungen
durch Zeilenwechsel getrennt werden.
In **Occam** und **Python** werden Anweisungen durch
gleiche Einrücktiefe zusammengefasst

```
if (x < y)
   a = x
   b = y
print (x)
```

Häufigste Schreibweisen von **Kommentaren**:
geklammert , z. B.

```
int pegel; /* geklammerter Kommentar */
```

oder **Zeilenkommentar** bis zum Zeilenende, z. B.

```
int pegel; // Zeilenkommentar
```

Geschachtelte Kommentare z.B. in **Modula-2**:

```
/* aeusserer /* innerer */ Kommentar */
```

2.2 Kontext-freie Grammatiken; Definition

Kontext-freie Grammatik (KFG, engl. CFG):
formaler Kalkül zur **Definition von Sprachen** und **von Bäumen**

Die **konkrete Syntax** einer Programmiersprache oder anderen formalen Sprache wird durch eine KFG definiert. (Ebene b, GPS 1-16)

Die **Strukturbäume** zur Repräsentation von Programmen in Übersetzern werden als **abstrakte Syntax** durch eine KFG definiert.

Eine **kontext-freie Grammatik G = (T, N, P, S)** besteht aus:

T	Menge der **Terminalsymbole**	Daraus bestehen Sätze der Sprache; Grundsymbole
N	Menge der **Nichtterminalsymbole**	Daraus werden Sprachkonstrukte abgeleitet.
$S \in N$	**Startsymbol** (auch **Zielsymbol**)	Daraus werden Sätze abgeleitet.
$P \subseteq N \times V^*$	Menge der **Produktionen**	Regeln der Grammatik.

außerdem wird $V = T \cup N$ als **Vokabular** definiert; T und N sind disjunkt

Produktionen haben also die Form **A ::= x**, mit $A \in N$ und $x \in V^*$
d.h. x ist eine evtl. leere Folge von Symbolen des Vokabulars.

KFG Beispiel: Grammatik für arithmetische Ausdrücke

G_{aA} = (T, N, P, S) besteht aus:

T	**Terminalsymbole**	{'(', ')', '+', '-', '*', '/', Ident}
N	**Nichtterminalsymbole**	{Expr, Fact, Opd, AddOpr, MulOpr}
$S \in N$	**Startsymbol**	**Expr**
$P \subseteq N \times V^*$	**Produktionen**	

P Menge der Produktionen:

Häufig gibt man Produktionen Namen:

```
p1:  Expr   ::= Expr AddOpr Fact
p2:  Expr   ::= Fact
p3:  Fact   ::= Fact MulOpr Opd
p4:  Fact   ::= Opd
p5:  Opd    ::= '(' Expr ')'
p6:  Opd    ::= Ident
p7:  AddOpr ::= '+'
p8:  AddOpr ::= '-'
p9:  MulOpr ::= '*'
p10: MulOpr ::= '/'
```

Unbenannte Terminalsymbole kennzeichnen wir in Produktionen, z.B. '+'

Es werden meist nur die Produktionen (und das Startsymbol) einer kontext-freien Grammatik angegeben, wenn sich die übrigen Eigenschaften daraus ergeben.

Ableitungen

Produktionen sind **Ersetzungsregeln**:
Ein Nichtterminal A in einer Symbolfolge u A v kann durch die rechte Seite x einer Produktion A ::= x ersetzt werden.

Das ist ein **Ableitungsschritt** u **A** v ⇒ u **x** v
z. B. `Expr AddOpr` **`Fact`** ⇒ `Expr AddOpr` **`Fact MulOpr Opd`** mit Produktion p3

Beliebig viele Ableitungsschritte nacheinander angewandt heißen **Ableitung**: u ⇒* v

Eine kontext-freie Grammatik **definiert eine Sprache**, d. h. die Menge von Terminalsymbolfolgen, die aus dem Startsymbol S ableitbar sind:

$$L(G) = \{ w \mid w \in T^* \text{ und } S \Rightarrow^* w \}$$

Die Grammatik aus GPS-2-4a definiert z. B. Ausdrücke als Sprachmenge:

$$L(G) = \{ w \mid w \in T^* \text{ und } Expr \Rightarrow^* w \}$$

$$\{\texttt{Ident, Ident + Ident, Ident + Ident * Ident}\} \subset L(G)$$

oder mit verschiedenen Bezeichnern für die Vorkommen des Grundsymbols `Ident`:

$$\{\texttt{a, b + c, a + b * c}\} \subset L(G)$$

Beispiel für eine Ableitung

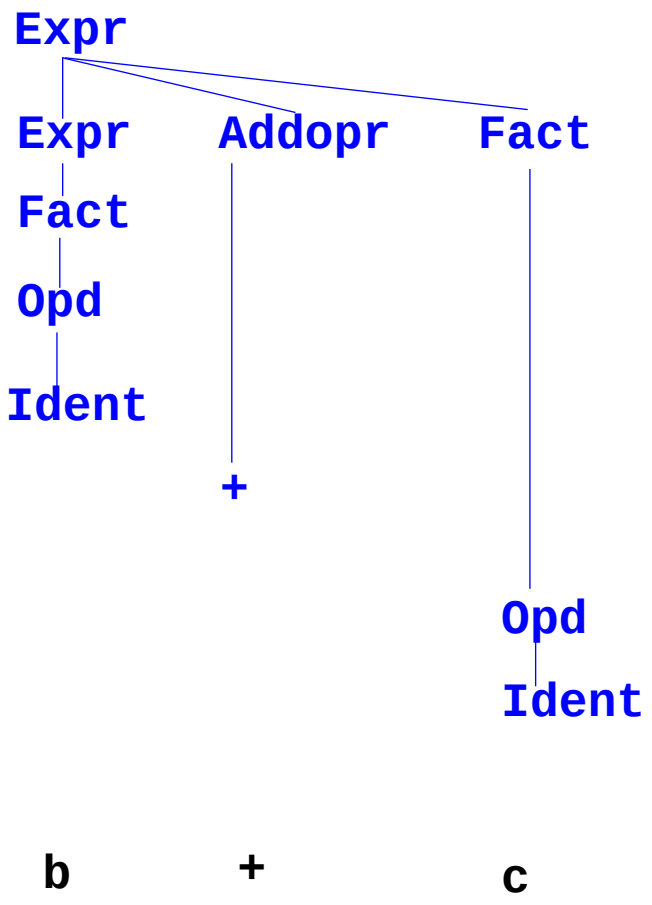
Satz der Ausdrucksgrammatik **b + c**

Ableitung:

```
                Expr
p1      => Expr     Addopr Fact
p2      => Fact     Addopr Fact
p4      => Opd      Addopr Fact
p6      => Ident    Addopr Fact
p7      => Ident    +      Fact
p4      => Ident    +      Opd
p6      => Ident    +      Ident

           b        +      c
```

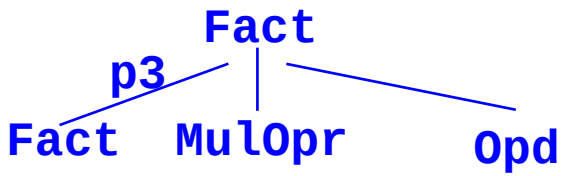
Ableitungsbaum:



Ableitungsbäume

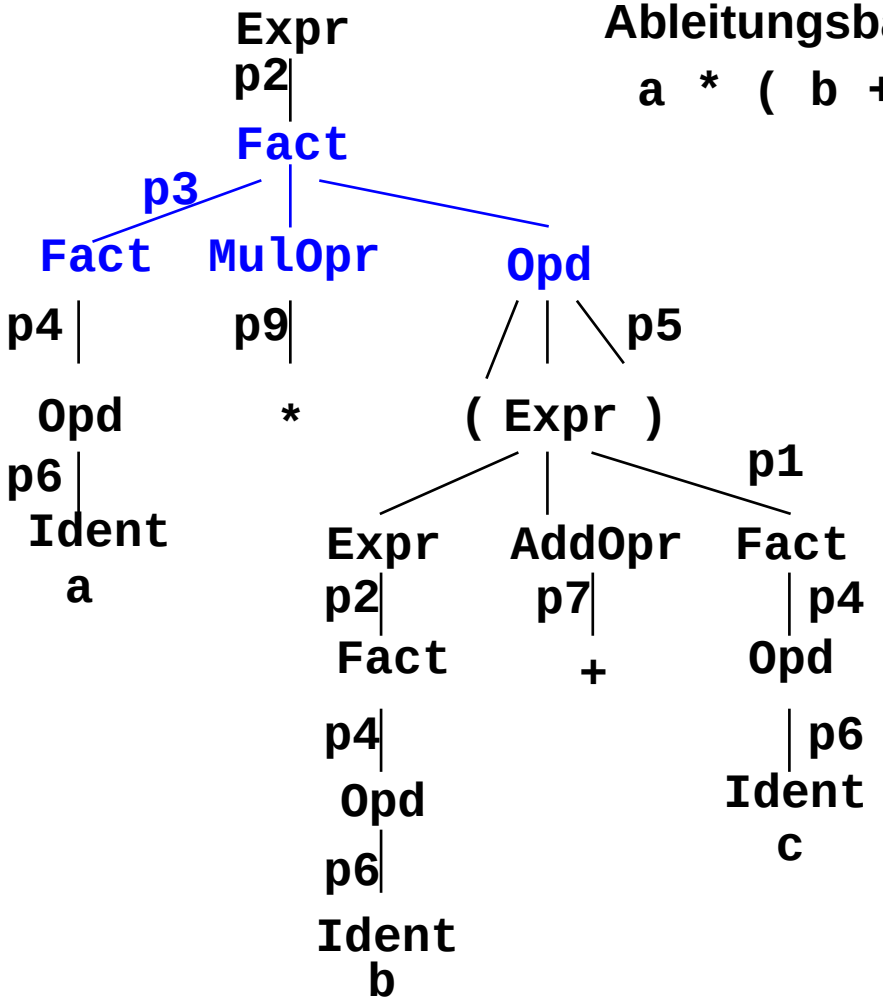
Jede Ableitung kann man als **Baum** darstellen. Er **definiert die Struktur des Satzes**.
Die **Knoten** repräsentieren **Vorkommen von Terminalen und Nichtterminalen**.
Ein **Ableitungsschritt** mit einer Produktion wird dargestellt durch Kanten zwischen dem Knoten für das Symbol der linken und denen für die Symbole der rechten Seite der Produktion:

Anwendung der Produktion p3:



Ableitungsbaum für

a * (b + c)



Mehrdeutige kontext-freie Grammatik

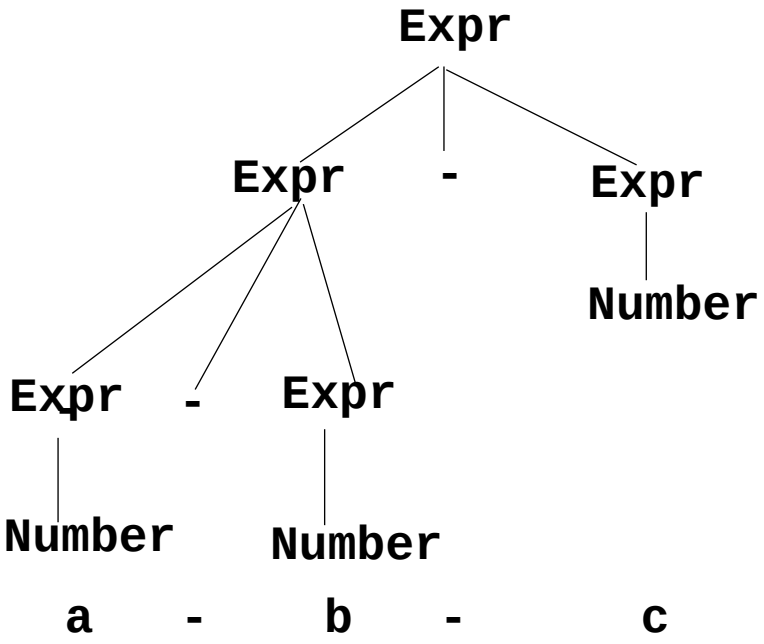
Eine kontext-freie Grammatik ist genau dann **mehrdeutig**, wenn
es einen **Satz aus ihrer Sprache gibt**,
zu dem es **zwei verschiedene Ableitungsbäume gibt.**

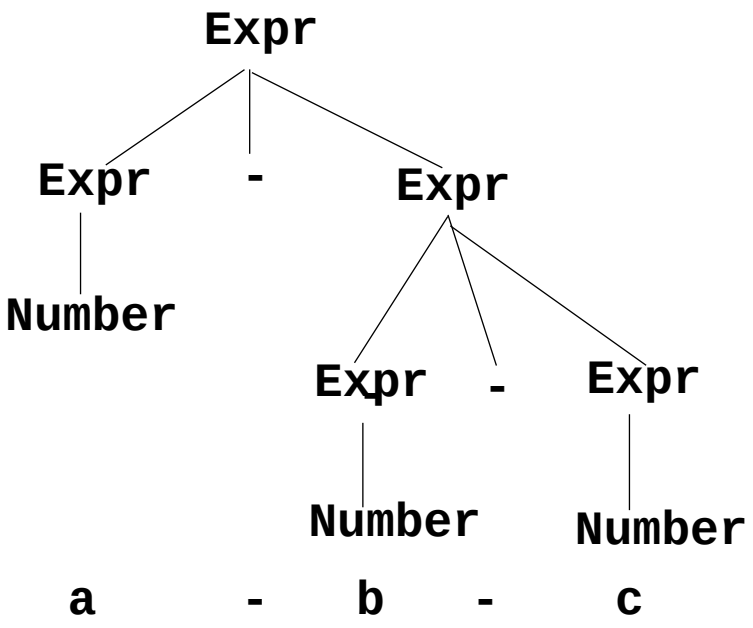
Beispiel für eine mehrdeutige KFG:

```
Expr ::= Expr '-' Expr

Expr ::= Number
```

ein Satz, der 2 verschiedene Ableitungsbäume hat:

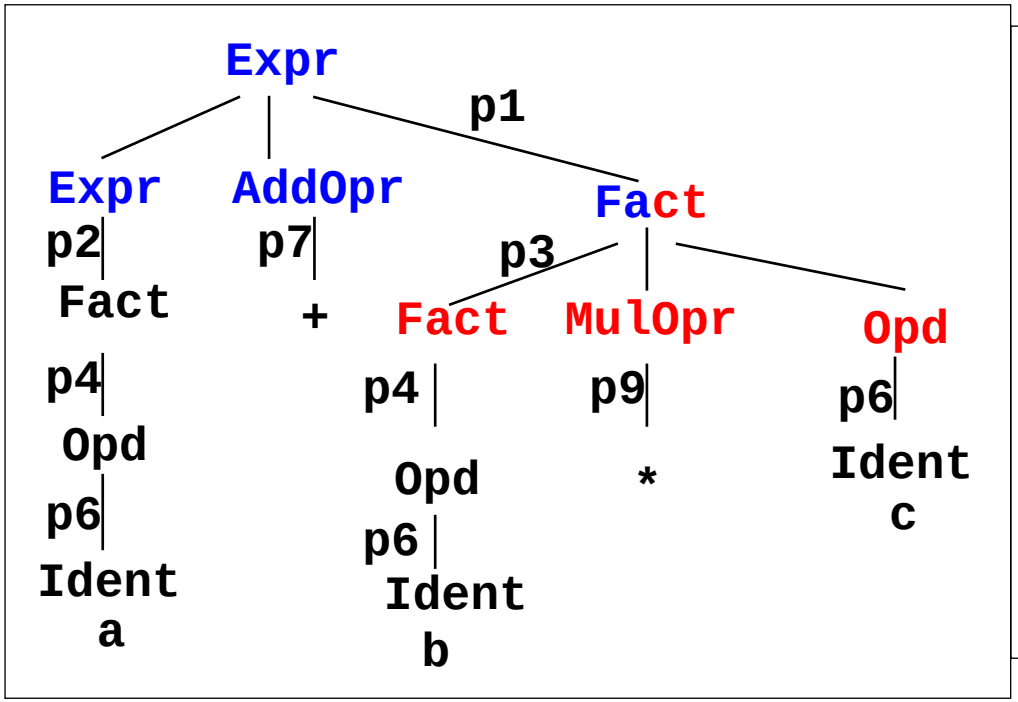




Ausdrucksgrammatik

Die Struktur eines Satzes wird durch seinen Ableitungsbaum bestimmt.
Ausdrucksgrammatiken legen dadurch die **Präzedenz** und **Assoziativität** von Operatoren fest.

Im Beispiel hat **`AddOpr` geringere Präzedenz als `MulOpr`**, weil er **höher in der Hierarchie der Kettenproduktionen** `Expr ::= Fact`, `Fact ::= Opd` steht.



Name	Produktion
p1:	`Expr ::= Expr AddOpr Fact`
p2:	`Expr ::= Fact`
p3:	`Fact ::= Fact MulOpr Opd`
p4:	`Fact ::= Opd`
p5:	`Opd ::= '(' Expr ')'`
p6:	`Opd ::= Ident`
p7:	`AddOpr ::= '+'`
p8:	`AddOpr ::= '-'`
p9:	`MulOpr ::= '*'`
p10:	`MulOpr ::= '/'`

Im Beispiel sind `AddOpr` und `MulOpr` **links-assoziativ**, weil ihre **Produktionen links-rekursiv** sind, d. h. **`a + b - c`** entspricht **`(a + b) - c`**.

Schemata für Ausdrucksgrammatiken

Ausdrucksgrammatiken konstruiert man **schematisch**, sodass **strukturelle Eigenschaften** der Ausdrücke definiert werden:

eine Präzedenzstufe, binärer Operator, linksassoziativ:

```
A ::= A Opr B
A ::= B
```

eine Präzedenzstufe, binärer Operator, **rechtsassoziativ**:

```
A ::= B Opr A
A ::= B
```

eine Präzedenzstufe, **unärer Operator**, präfix:

```
A ::= Opr A
A ::= B
```

eine Präzedenzstufe, unärer Operator, **postfix**:

```
A ::= A Opr
A ::= B
```

Elementare Operanden: nur aus dem Nichtterminal der höchsten Präzedenzstufe (sei hier H) abgeleitet:

```
H ::= Ident
```

Geklammerte Ausdrücke: nur aus dem Nichtterminal der höchsten Präzedenzstufe (sei hier H) abgeleitet; enthalten das Nichtterminal der niedrigsten Präzedenzstufe (sei hier A)

```
H ::= '(' A ')'
```

Notationen für kontext-freie Grammatiken

Eine kontext-freie Grammatik wurde 1959 erstmals zur Definition einer Programmiersprache (Algol 60) verwendet. Name für die Notation - noch heute: **Backus Naur Form (BNF)**.

Entweder werden **Symbolnamen gekennzeichnet**,
z. B. durch Klammerung **<Expr>** oder durch den Schrifttyp *Expr.*

oder unbenannte **Terminale**, die für sich stehen, werden **gekennzeichnet**, z. B. **'('**

Zusammenfassung von Produktionen mit gleicher linker Seite:

```
Opd  ::=  '(' Expr ')'
      |   Ident
```

oder im Java -Manual:

```
Opd:
   ( Expr )
   Ident
```

Erweiterte Notation EBNF

Backus Naur Form (BNF) erweitert um Konstrukte regulärer Ausdrücke zu **Extended BNF**

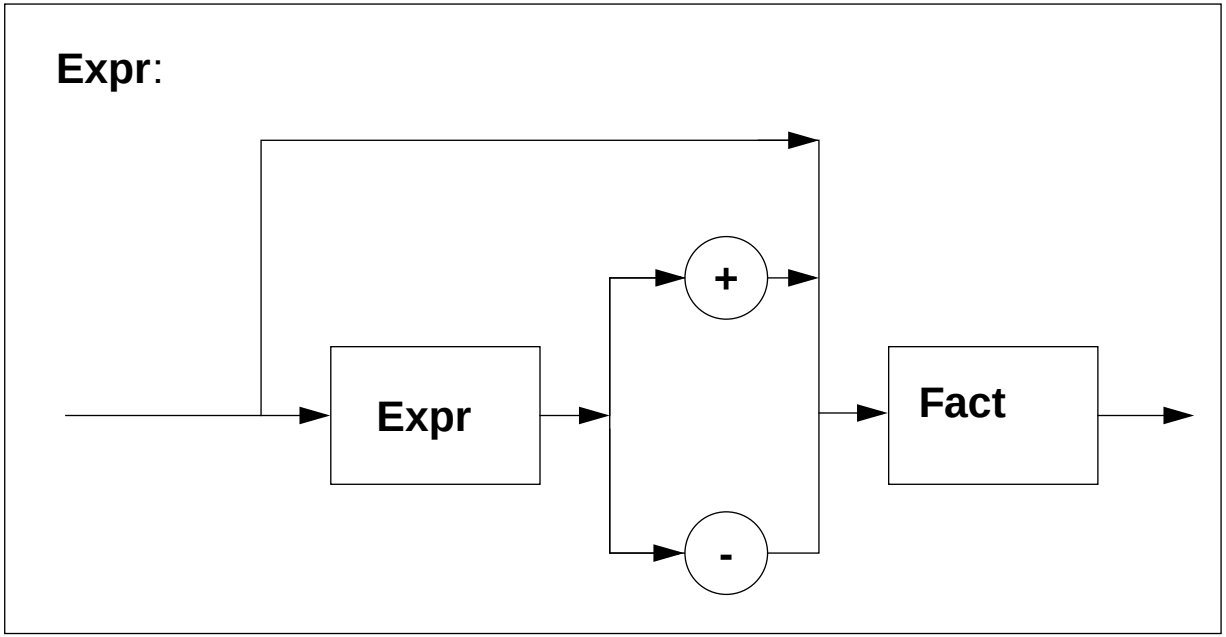
EBNF		gleichbedeutende BNF-Produktionen		
X ::= u **(v)** w	Klammerung	X ::= u Y w	Y ::= v	
X ::= u **[v]** w	optional	X ::= u Y w	Y ::= v	Y ::= ε
X ::= u **s*** w	optionale	X ::= u Y w	Y ::= s Y	Y ::= ε
X ::= u **{ s }** w	Wiederholung			
X ::= u **s ...** w	Wiederholung	X ::= u Y w	Y ::= s Y	Y ::= s
X ::= u **s+** w				
X ::= u (**v || s**) w	Wdh. mit Trenner	X ::= u Y w	Y ::= v s Y	Y ::= v
X ::= u (**v1 | v2**) w	Alternativen	X ::= u Y w	Y ::= v1	Y ::= v2

Dabei sind u, v, v1, v2, w ∈ V* s ∈ V X, Y ∈ N
Y ist ein Nichtterminal, das sonst nicht in der Grammatik vorkommt.

Beispiele:

Block ::= '{' **Statement*** '}'	Block ::= '{' **Y** '}'	Y ::= Statement Y	Y ::= ε
Decl ::= Type **(Ident || ',')** ';'	Decl ::= Type **Y** ';'	Y ::= Ident ',' Y	Y ::= Ident

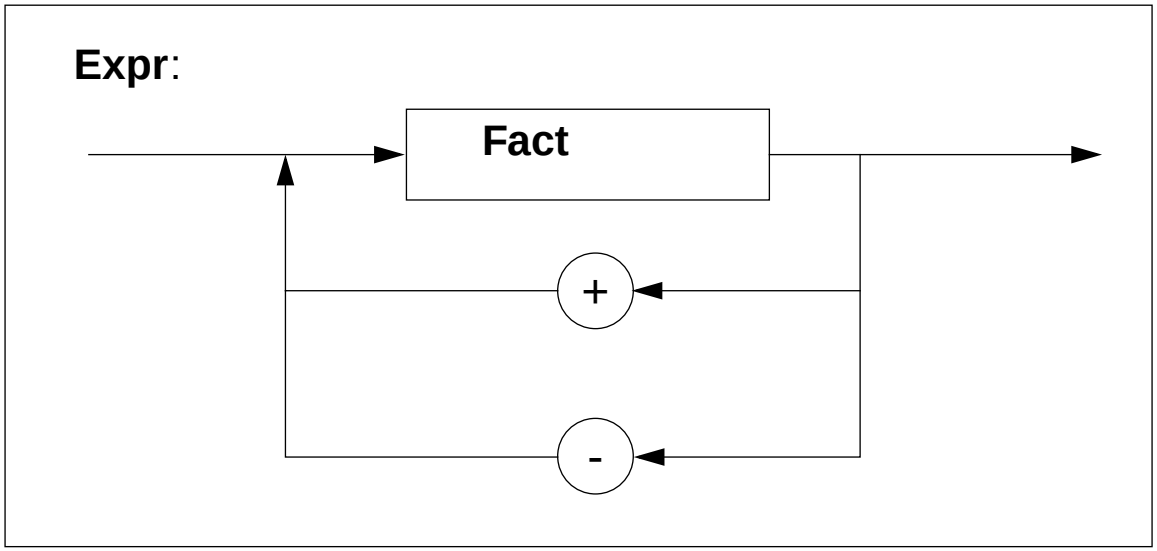
Syntaxdiagramme



Ein **Syntaxdiagramm repräsentiert eine EBNF-Produktion**:

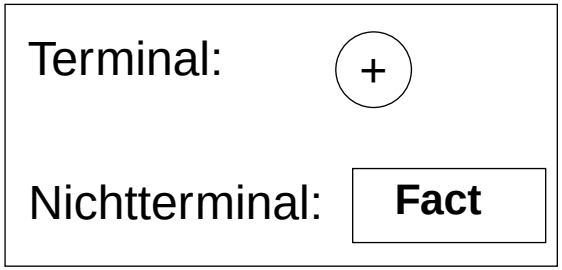
Expr ::= [Expr ('+' | '-')] Fact

Option und Alternative



Expr ::= (Fact || ('+' | '-'))

Wiederholung mit alternativem Trenner



Produktionen-Schemata für Folgen

Beschreibung	Produktionen	Sprachmenge
nicht-leere Folge von b	A ::= A b | b	{b, bb, bbb, ...}
nicht-leere Folge von b	A ::= b A | b	{b, bb, bbb, ...}
evtl. leere Folge von b	A ::= A b |	{ε, b, bb, bbb, ...}
evtl. leere Folge von b	A ::= b A |	{ε, b, bb, bbb, ...}
nicht-leere Folge von b getrennt durch t	A ::= A t b | b	{b, btb, btbtb, ...}
nicht-leere Folge von b getrennt durch t	A ::= b t A | b	{b, btb, btbtb, ...}

Grammatik-Entwurf: Folgen

Produktionen für **Folgen von Sprachkonstrukten** systematisch konstruieren.
Schemata hier am Beispiel von Anweisungsfolgen (`Stmts`)

Folgen mit Trenner:

```
a. Stmts    ::=  Stmts ';' Stmt  | Stmt                    linksrekursiv
b. Stmts    ::=  Stmt  ';' Stmts | Stmt                    rechtsrekursiv
c. Stmts    ::=  ( Stmt || ';')                            EBNF
d. StmtsOpt ::=  Stmts |                                   mit leerer Folge
```

Folgen mit Terminator:

```
a. Stmts    ::=  Stmt ';' Stmts | Stmt ';'                 rechtsrekursiv
b. Stmts    ::=  Stmt     Stmts | Stmt                     Terminator an den Elementen
   Stmt     ::=  Assign ';' | ...
c. Stmts    ::=  Stmts Stmt| Stmt                          linksrekursiv
   Stmt     ::=  Assign ';' | ...
d. Stmts    ::=  ( Stmt ';')+                              EBNF
```

Grammatik-Entwurf: Klammern

Klammern: Paar von Terminalen, das eine Unterstruktur einschließt:

```
Operand ::=  '(' Expression ')'
Stmt    ::=  'while' Expr 'do' Stmts 'end'
Stmt    ::=  'while' Expr 'do' Stmts 'end'

MethodenDef ::=
   ErgebnisTyp MethodenName '(' FormaleParameter ')' Rumpf
```

Stilregel: Öffnende und schließende Klammer immer in derselben Produktion

gut:
```
Stmt ::=  'while' Expr 'do' Stmts 'end'
```

schlecht:
```
Stmt       ::= WhileKopf Stmts 'end'
WhileKopf  ::= 'while' Expr 'do'
```

Nicht-geklammerte (offene) Konstrukte können **Mehrdeutigkeiten** verursachen:

```
Stmt    ::= 'if' Expr 'then' Stmt
          |   'if' Expr 'then' Stmts 'else' Stmt
```

Offener, optionaler else-Teil verursacht **Mehrdeutigkeit** in C, C++, Pascal, sog. "dangling else"-Problem:

```
if c then if d then S1 else S2
```

In diesen Sprachen gehört `else S2` zur **inneren** if-Anweisung.

Java enthält das gleiche if-Konstrukt. Die Grammatik vermeidet die Mehrdeutigkeit durch Produktionen, die die Bindung des `else` explizit machen.

Abstrakte Syntax

konkrete Syntax

KFG definiert
Symbolfolgen (Programmtexte) und deren **Ableitungsbäume**

konkrete Syntax bestimmt die Struktur von Programmkonstrukten, z. B. Präzedenz und Assozitivität von Operatoren in Ausdrücken

Präzedenzschemata benötigen **Kettenproduktionen**, d.h. Produktionen mit genau einem Nichtterminal auf der rechtenSeite:

```
Expr ::= Fact
Fact ::= Opd
Opd  ::= '(' Expr ')'
```

Mehrdeutigkeit ist problematisch

Alle Terminale sind nötig.

abstrakte Syntax

KFG definiert
abstrakte Programmstruktur durch **Strukturbäume**

statische und dynamische Semantik werden auf der abstrakten Syntax definiert

solche Kettenproduktionen sind hier **überflüssig**

Mehrdeutigkeit ist akzeptabel

Terminale, die nur für sich selbst stehen und **keine Information** tragen, sind hier **überflüssig (Wortsymbole, Spezial-symbole)**, z.B. `class ( ) + - * /`

Abstrakte Ausdrucksgrammatik

konkrete Ausdrucksgrammatik

```
p1:    Expr   ::= Expr AddOpr Fact
p2:    Expr   ::= Fact
p3:    Fact   ::= Fact MulOpr Opd
p4:    Fact   ::= Opd
p5:    Opd    ::= '(' Expr ')'
p6:    Opd    ::= Ident
p7:    AddOpr ::= '+'
p8:    AddOpr ::= '-'
p9:    MulOpr ::= '*'
p10:   MulOpr ::= '/'
```

abstrakte Ausdrucksgrammatik

Name	Produktion
BinEx:	Exp ::= Exp BinOpr Exp
IdEx:	Exp ::= Ident
PlusOpr:	BinOpr ::=
MinusOpr:	BinOpr ::=
TimesOpr:	BinOpr ::=
DivOpr:	BinOpr ::=

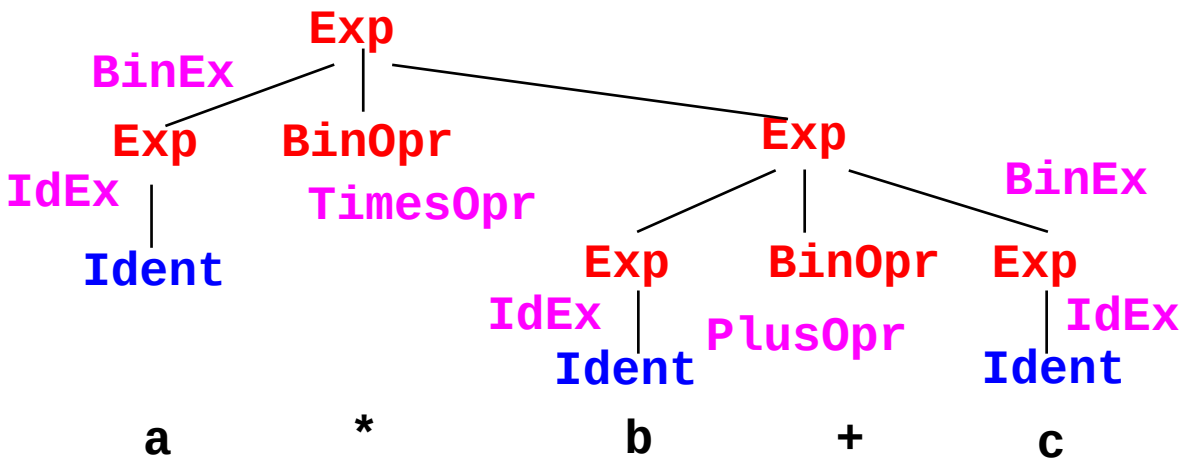
Abbildung konkret -> abstrakt

```
Expr,Fact,Opd -> Exp
AddOpr,MulOpr -> BinOpr

p1,p3 -> BinEx
p2,p4,p5 ->
p6 -> IdEx

p7 -> PlusOpr
...
```

Strukturbaum für a * (b + c)



2.3 XML
Übersicht

XML (Extensible Markup Language, dt.: Erweiterbare Auszeichnungssprache)

- seit 1996 vom W3C definiert, in Anlehnung an SGML
- Zweck: Beschreibungen **allgemeiner Strukturen** (nicht nur Web-Dokumente)
- **Meta-Sprache** (“erweiterbar”):
  Die Notation ist festgelegt (Tags und Attribute, wie in HTML),
  Für beliebige Zwecke kann **jeweils eine spezielle syntaktische Struktur** definiert werden (DTD)
  Außerdem gibt es Regeln (XML-Namensräume), um XML-Sprachen in andere **XML-Sprachen zu importieren**
- X**HTML** ist so als XML-Sprache definiert
- Weitere aus XML **abgeleitete Sprachen**: SVG, MathML, SMIL, RDF, WML
- **individuelle XML-Sprachen** werden benutzt, um strukturierte Daten zu speichern, die von **Software-Werkzeugen geschrieben und gelesen** werden
- XML-Darstellung von strukturierten Daten kann mit verschiedenen Techniken **in HTML transformiert** werden, um sie **formatiert anzuzeigen:**
  XML+CSS, XML+XSL, SAX-Parser, DOM-Parser

Dieser Abschnitt orientiert sich eng an **SELFHTML** (Stefan Münz), http://de.selfhtml.org

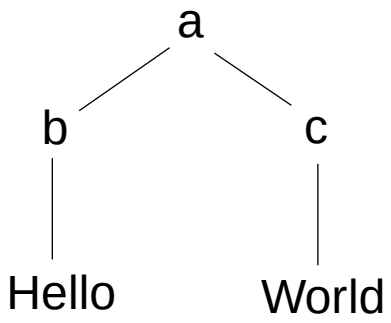
3 elementare Prinzipien

Die XML-Notation basiert auf 3 elementaren Prinzipien:

A: Vollständige Klammerung durch Tags

```
<a>
    <b>Hello</b>
    <c>World</c>
</a>
```

B: Klammerstruktur ist äquivalent zu gewurzeltem Baum



C: Kontextfreie Grammatik definiert Bäume; eine DTD ist eine KFG

```
a ::= b c
b ::= PCDATA
c ::= PCDATA
```

Notation und erste Beispiele

Ein Satz in einer XML-Sprache ist ein Text, der durch Tags strukturiert wird.

Tags werden **immer** in **Paaren von Anfangs- und End-Tag** verwendet:

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

Anfangs-Tags können Attribut-Wert-Paare enthalten:

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

Die **Namen von Tags und Attributen** können für die XML-Sprache **frei gewählt** werden.

Mit Tags gekennzeichnete Texte können geschachtelt werden.

```
<adressBuch>
<adresse>
   <name>
      <nachname>Mustermann</nachname>
      <vorname>Max</vorname>
   </name>
   <anschrift>
      <strasse>Hauptstr 42</strasse>
      <ort>Paderborn</ort>
      <plz>33098</plz>
   </anschrift>
</adresse>
</adressBuch>
```

$(a+b)^2$ in MathML:

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

Ein vollständiges Beispiel

Kennzeichnung des Dokumentes als XML-Datei

Datei mit der Definition der Syntaktischen Struktur dieser XML-Sprache (DTD)

Datei mit Angaben zur Transformation in HTML

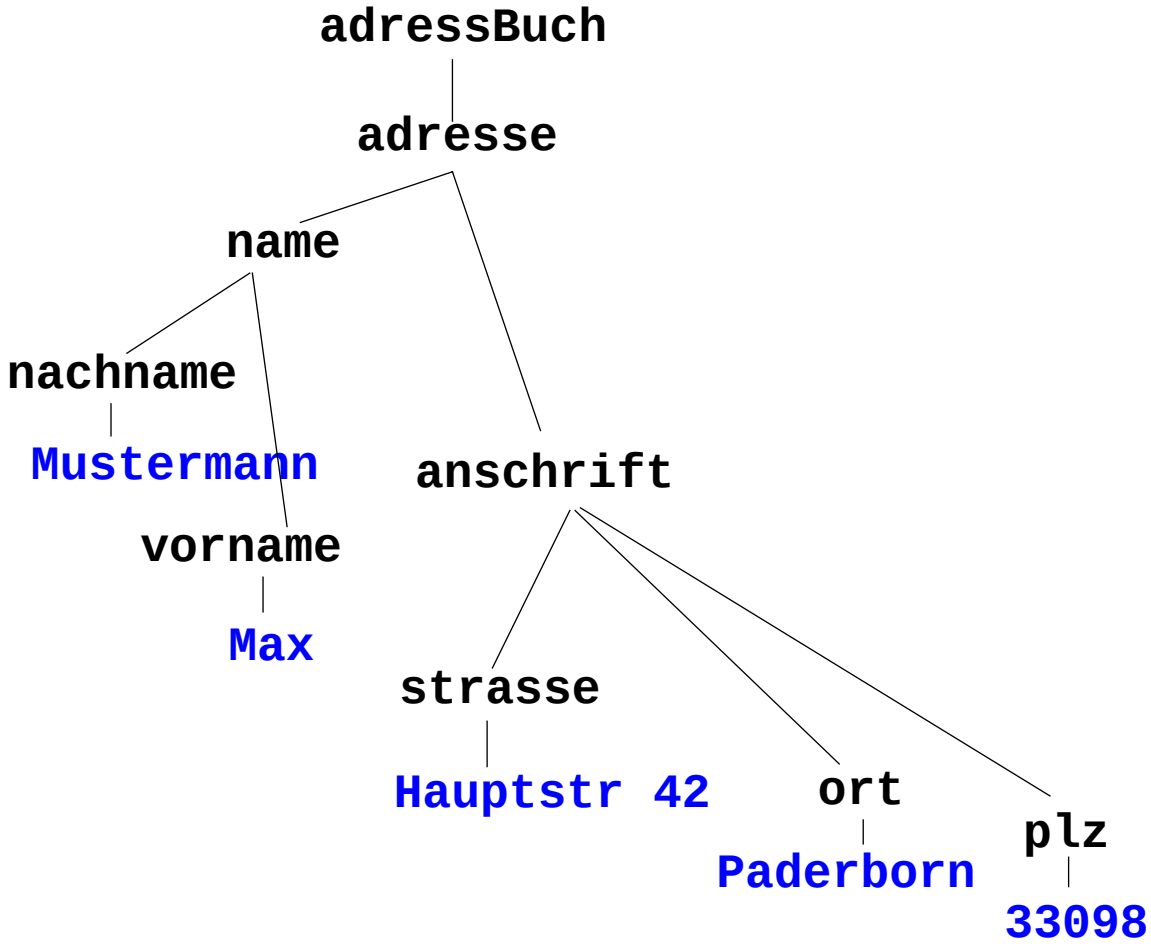
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE produktnews SYSTEM "produktnews.dtd">
<?xml-stylesheet type="text/xsl" href="produktnews.xsl" ?>
<produktnews>
   Die neuesten Produktnachrichten:
   <beschreibung>
      Die Firma <hersteller>Fridolin Soft</hersteller> hat eine neue
      Version des beliebten Ballerspiels
      <produkt>HitYourStick</produkt> herausgebracht. Nach Angaben des
      Herstellers soll die neue Version, die nun auch auf dem
      Betriebssystem <produkt>Ganzfix</produkt> läuft, um die
      <preis>80 Dollar</preis> kosten.
   </beschreibung>
   <beschreibung>
      Von <hersteller>Ripfiles Inc.</hersteller> gibt es ein Patch zu der Sammel-CD
      <produkt>Best of other people's ideas</produkt>. Einige der tollen
      Webseiten-Templates der CD enthielten bekanntlich noch versehentlich nicht
      gelöschte Angaben der Original-Autoren. Das Patch ist für schlappe
      <preis>200 Euro</preis> zu haben.
   </beschreibung>
</produktnews>
```

Baumdarstellung von XML-Texten

Jeder XML-Text ist durch Tag-Paare **vollständig geklammert** (wenn er *well-formed* ist).

Deshalb kann er eindeutig **als Baum dargestellt** werden. (Attribute betrachten wir noch nicht)
Wir markieren die inneren Knoten mit den Tag-Namen; die Blätter sind die elementaren Texte:

```
<adressBuch>
<adresse>
   <name>
      <nachname>Mustermann
      </nachname>
      <vorname>Max
      </vorname>
   </name>
   <anschrift>
      <strasse>Hauptstr 42
      </strasse>
      <ort>Paderborn</ort>
      <plz>33098</plz>
   </anschrift>
</adresse>
</adressBuch>
```



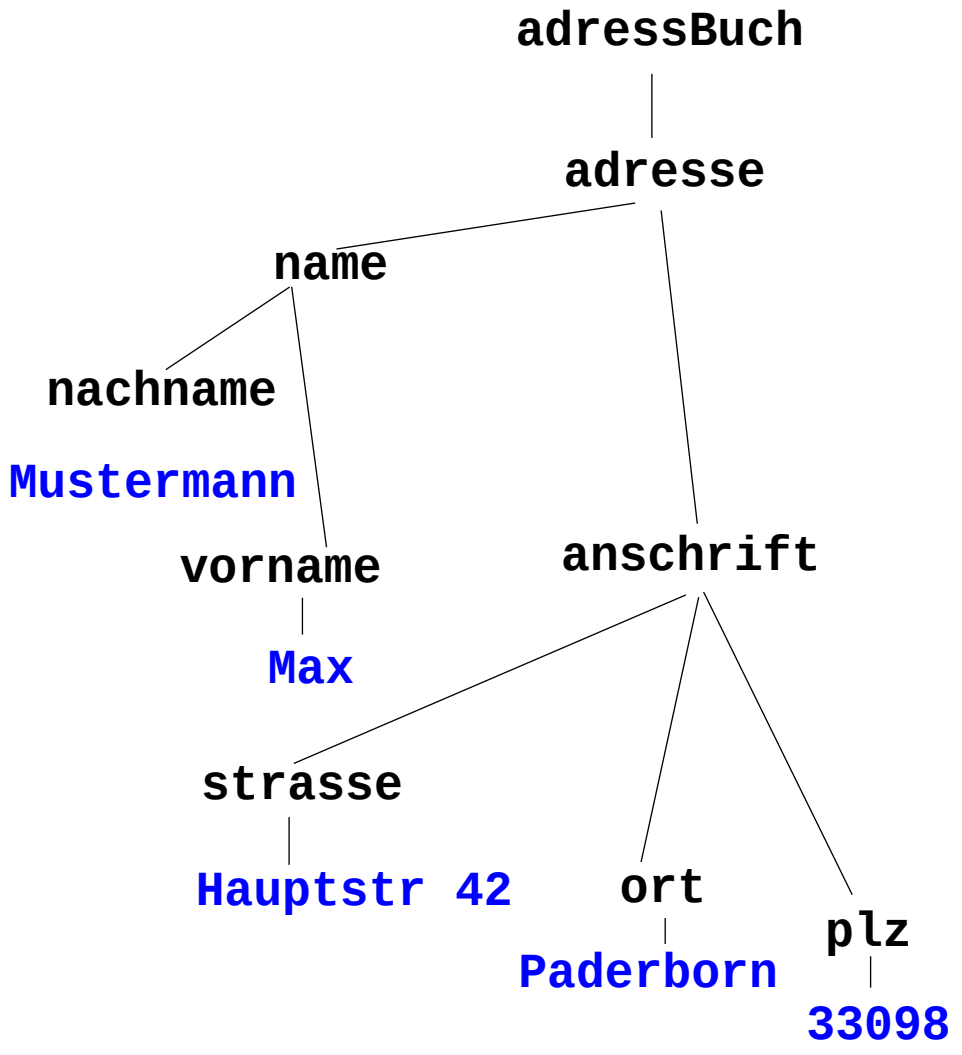
XML-Werkzeuge können die Baumstruktur eines XML-Textes ohne weiteres ermitteln und ggf. anzeigen.

Grammatik definiert die Struktur der XML-Bäume

Mit **kontextfreien Grammatiken (KFG)** kann man **Bäume definieren**.

Folgende KFG definiert korrekt strukturierte Bäume für das Beispiel Adressbuch:

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



Document Type Definition (DTD) statt KFG

Die Struktur von XML-Bäumen und -Texten wird in der **DTD-Notation** definiert. Ihre Konzepte entsprechen denen von KFGn:

KFG

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

DTD

```
<!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)          >
```

weitere Formen von DTD-Produktionen:

`X (Y)+`	nicht-leere Folge
`X (A | B)`	Alternative
`X (A)?`	Option
`X EMPTY`	leeres Element

Zusammenfassung zu Kapitel 2

Mit den Vorlesungen und Übungen zu Kapitel 2 sollen Sie nun Folgendes können:

- Notation und Rolle der Grundsymbole kennen.
- Kontext-freie Grammatiken für praktische Sprachen lesen und verstehen.
- Kontext-freie Grammatiken für einfache Strukturen selbst entwerfen.
- Schemata für Ausdrucksgrammatiken, Folgen und Anweisungsformen anwenden können.
- EBNF sinnvoll einsetzen können.
- Abstrakte Syntax als Definition von Strukturbäumen verstehen.
- XML als Meta-Sprache zur Beschreibung von Bäumen verstehen
- DTD von XML als kontext-freie Grammatik verstehen
- XML lesen können