

E2. Symbole und Syntax Überblick

E EWS-3.1

Grundbegriffe zur **formalen Definition von Sprachen**:

- **Alphabet**: Menge von Zeichen
- **Wort**: Folge von Zeichen aus Alphabet - gebildet nach bestimmten Regeln (Ebene 1)
Satz: Folge von Symbolen aus Vokabular - gebildet nach bestimmten Regeln (Ebene 2)
(Wort, Satz), (Zeichen, Symbol) und (Alphabet, Vokabular) paarweise synonym
- **Sprache**: Menge von Worten bzw. Sätzen

Kalküle zur Bildung von Worten und Sätzen:

- **reguläre Ausdrücke**
zur Definition der **Notation von Grundsymbolen**
(Ebene 1 der Spracheigenschaften)

zur Definition von **Textmustern**
angewandt zum Suchen und Ersetzen von Zeichenfolgen
programmiert in PHP, Perl, Unix-sh
- **kontextfreie Grammatik**
zur Definition der **Menge der syntaktisch korrekten Sätze** einer Sprache
(Ebene 2 der Spracheigenschaften)

Struktur der syntaktisch korrekten Sätze einer Sprache

Alphabete und Zeichenfolgen

E EWS-3.2

Ein **Alphabet** ist eine nicht-leere **Menge von Zeichen** zur Bildung von Zeichenfolgen.

Wir betrachten hier nur Alphabete mit endlich vielen Zeichen.

Alphabete werden in Formeln häufig mit Σ bezeichnet;
sonst gibt man ihnen individuelle Namen oder benutzt sie unbenannt.

Beispiele:

$\Sigma =$ {T, F}
Dualziffern = {0, 1}
Dezimalziffern = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Kleinbuchstaben = {a, b, ..., z}
Nukleotide = {A, C, G, U}
ASCII = standardisierter Zeichensatz mit 128 Zeichen

Ein **Wort über einem Alphabet A** ist eine Folge von Zeichen aus A.

formal: eine Folge $a_1 a_2 \dots a_n$, mit $a_i \in A$, für $i = 1, \dots, n$.

n ist die **Länge der Folge** bzw. die **Länge des Wortes**.

Beispiele: Wort der Länge 7 über dem Alphabet Dualziffern: 1001101

Die **leere Folge** bzw. das **leere Wort** wird mit ϵ (epsilon) bezeichnet.

Reguläre Ausdrücke

E EWS-3.3

Ein **regulärer Ausdruck R**

definiert eine **Mengen von Worten** über einem Alphabet A, die **Sprache L (R)**.

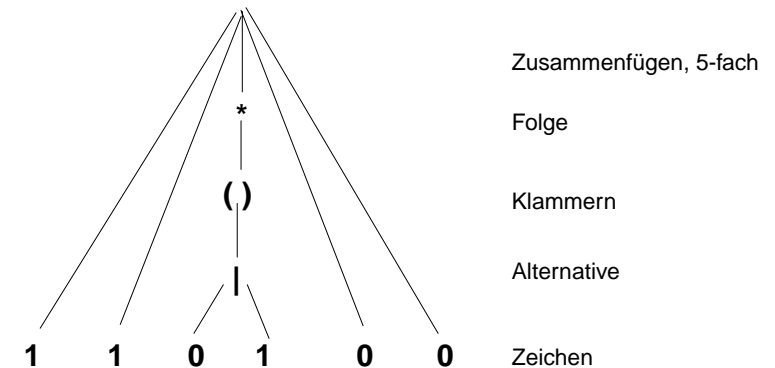
Ein regulärer Ausdruck kann aus folgenden 8 Formen rekursiv zusammengesetzt sein.
F und G seien reguläre Ausdrücke.

regulärer Ausdruck R	Sprache L (R)	Erklärung
a	{ a }	Zeichen a als Wort
FG	{ fg f ∈ L(F), g ∈ L(G) }	Zusammenfügen von 2 Worten
F G	{ f f ∈ L(F) } ∪ { g g ∈ L(G) }	Alternativen
ϵ	{ ϵ }	das leere Wort
(F)	L(F)	Klammerung
F ⁺	{ f ₁ f ₂ ... f _n f _i ∈ L(F), für n ≥ 1, i = 1, ..., n }	nicht-leere Folgen von Worten aus L(F)
F [*]	{ ϵ } ∪ L(F ⁺)	Folgen von Worten aus L(F)
F ⁿ	{ f ₁ f ₂ ... f _n f _i ∈ L(F), für i = 1, ..., n }	Folgen von genau n Worten aus L(F)

Struktur eines regulären Ausdruckes

E EWS-3.4

1 1 (0 | 1) * 0 0



verbal:

Jedes Wort aus der Sprache dieses regulären Ausdruckes besteht aus zwei 1, gefolgt von beliebig vielen 0 oder 1, gefolgt von zwei 0.

Beispiele für reguläre Ausdrücke

Name	regulärer Ausdruck A	Worte aus seiner Sprache L(A)
$Abc =$	$(a b) (c d \epsilon)$	ac bc ad bd a b
$Anrede =$	Sehr geehrte(r ϵ) (Frau Herr)	Sehr geehrte Frau
$Dig =$	$0 1 \dots 9$	7
$sLet =$	$a b \dots z$	x
$cLet =$	$A B \dots Z$	B
$Let =$	$sLet cLet$	m N
$Bezeichner =$	$Let (Let Dig)^*$	Maximum min3 a
$GeldBetrag =$	Dig^+, Dig^2	23,95 0,50
$KFZ =$	$(cLet cLet^2 cLet^3) - (cLet cLet^2) - (Dig Dig^2 Dig^3 Dig^4)$	PB-AX-123
$Dual =$	$1^3 (1 0)^* 0^3$	1111000 111000 1111101010000

Wenn **Namen** von regulären Ausdrücken in **regulären Ausdrücken** verwendet werden, müssen sie von den Zeichen unterschieden werden können; hier *Namen* kursiv.

Beispiele für Definitionen von Grundsymbolen

$Pascal_Identifier =$	$Let (Let Dig)^*$
$C_Identifier =$	$(Let _) (Let _ Dig)^*$
$ADA_Identifier =$	$Let (_ \epsilon) (Let Dig)^*$
$PHP_Var_Identifier =$	$\$ (Let _) (Let _ Dig)^*$
$Pascal_Real =$	$((Dig^+ \cdot Dig^+) ((e E) (+ - \epsilon) Dig^+) \epsilon) (Dig^+ (e E) (+ - \epsilon) Dig^+)$
$HexDig =$	$Dig a b c d e f A B C D E F$
$HTML_CharRef =$	$\& (Let^* \# Dig^* \#x HexDig^*) ;$

Reguläre Ausdrücke als Textmuster

In Sprachen, die zur **Textverarbeitung** eingesetzt werden, benutzt man **reguläre Ausdrücke, um Textmuster zu definieren**.

Beispiele:

suche alle Dateinamen der Form	<code>ews(0 1 2 3 4 5 6 7 8 9)³.html</code>
in der Schreibweise von Unix-sh	<code>ls ews[0-9][0-9][0-9].html</code>
Aufruf einer PHP-Funktion	<code>preg_match ("/[dD]aß/", \$absatz)</code>
sucht ein Textmuster in einer Zeichenreihe	Muster Zeichenreihe
	<code>\$d = "[0-9]";</code>
	<code>preg_match ("/ews\$d\$d\$d\..html/", \$files)</code>

Reguläre Ausdrücke als Textmuster sind umfassend in der Skriptsprache Perl definiert und so in PHP übernommen.

Auf den vorigen Folien haben wir die **grundlegenden Begriffe** für reguläre Ausdrücke mit der **dafür üblichen Notation** eingeführt. In **PHP, Perl, Unix-sh** werden die gleichen Begriffe aber in anderer **Notation** verwendet.

Notation von regulären Ausdrücken in PHP

a	das Zeichen a
FG	Zusammenfügen von 2 Worten
$F G$	Alternativen
(F)	Klammerung
$F?$	Option; wie $F \epsilon$
$F+$	nicht-leere Folge von Worten aus $L(F)$
F^*	beliebig lange Folge von Worten aus $L(F)$
$F\{m,n\}$	Folge mit mindestens m und höchstens n von Worten aus $L(F)$
$F\{m\}$	Folge mit genau m Worten aus $L(F)$
[abc]	alternativ ein Zeichen aus der Klammer
[^abc]	alternativ ein anderes Zeichen als die in der Klammer
[a-zA-Z]	alternativ ein Zeichen aus Zeichenbereichen
.	beliebiges Zeichen
^	Anfang der Zeichenfolge (nichts darf vorangehen)
\$	Ende der Zeichenfolge (nichts darf darauf folgen)

Beispiele für reguläre Ausdrücke in PHP-Notation

Reguläre Ausdrücke kommen in PHP-Programmen immer als Zeichenreihenlitterale (Strings) vor. Diese werden in " eingeschlossen, z. B. "1|0".

Statt Namen für reguläre Ausdrücke zu definieren, weisen wir die Zeichenreihe einer Variablen zu, z. B. \$binDig = "(1|0)";

Variable = regulärer Ausdruck als PHP-String

```
$Abc = "(a|b)(c|d)?";
$Anrede = "Sehr geehrte(r)? (Frau|Herr)";
$Dig = "[0-9]";
$sLet = "[a-z]";
$cLet = "[A-Z]";
$Let = "[a-zA-Z]";
$Bezeichner = "[a-zA-Z][a-zA-Z0-9]*";
$GeldBetrag = "$Dig+,$Dig{2}";
$KFZ = "$cLet{1,3}-$cLet{1,2}-$Dig{1,4}";
$Dual = "1{3}[10]*0{3}";
```

Beispiele für Definitionen von Grundsymbolen in PHP-Notation

```
$Pascal_Identifier = "[a-zA-Z][a-zA-Z0-9]*";
$C_Identifier = "[a-zA-Z_][a-zA-Z0-9]*";
$ADA_Identifier = "[a-zA-Z](_[a-zA-Z0-9])?";
$PHP_Var_Identifier = "\$[a-zA-Z_][a-zA-Z0-9]*";
$Pascal_Exponent = "((e|E)(\+|-)?[0-9]+)";
$Pascal_Real = "(((0-9)+\.[0-9]+)$Exponent?)|((0-9)+$Exponent)";
$HexDig = "[0-9a-fA-F]";
$HTML_CharRef = "&(([a-zA-Z]+)|(#[0-9]+)|(#x$HexDig+));";
```

E2.2 Kontextfreie Grammatiken

Kontextfreie Grammatik (KFG): formaler Kalkül zur Definition einer

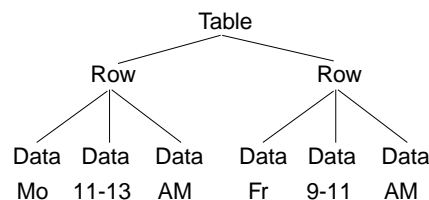
- **Sprache** als Menge von Sätzen; jeder **Satz** ist eine **Folge von Symbolen**
- **Menge von Bäumen**; jeder Baum repräsentiert die **Struktur eines Satzes** der Sprache

Anwendungen:

- Programme einer **Programmiersprache** und deren **Struktur**, z. B. Java, Pascal, C
- **Sprachen als** Schnittstellen zwischen Software-Werkzeugen, **Datenaustauschformate**, z. B. HTML, XML
- Bäume zur Repräsentation **strukturierter Daten**, z. B. in HTML

Beispiel: Tabellen in HTML:

```
<table><tr><td>Mo</td>
  <td>11-13</td>
  <td>AM</td>
</tr>
<tr><td>Fr</td>
  <td>9-11</td>
  <td>AM</td>
</tr>
</table>
```



Definition: Kontextfreie Grammatik

Eine kontextfreie Grammatik $G = (T, N, P, S)$ besteht aus:

- T Menge der Terminalsymbole** (kurz: Terminale)
- N Menge der Nichtterminalsymbole** (kurz: Nichtterminale)
(T und N sind disjunkt)
- S Startsymbol**; S ist ein Nichtterminal: $S \in N$
- P Menge der Produktionen**
jede Produktion hat die Form $A ::= x$
A ist ein Nichtterminal, d. h. $A \in N$ und
x ist eine (evtl. leere) Folge von Terminalen und Nichtterminalen,
d. h. $x \in (T \cup N)^* = V^*$

$V = T \cup N$ heißt auch **Vokabular**, seine Elemente heißen **Symbole**

Man sagt

„In der Produktion $A ::= x$ steht A auf der **linken Seite** und x auf der **rechten Seite**.“

Man gibt Produktionen häufig **Namen**: **p1**: $A ::= x$

In Symbolfolgen aus V^* werden die Elemente nur durch Zwischenraum getrennt: $A ::= B C D$

Beispiel zur Definition einer KFG

E EWS-3.13

Terminale $T = \{ (,) \}$

Nichtterminale $N = \{ \text{Klammern}, \text{Liste} \}$

Startsymbol $S = \text{Klammern}$

Produktionen $P =$

{

p1: Klammern ::= '(' Liste ')'

p2: Liste ::= Klammern Liste

p3: Liste ::=

}

Vokabular $V = T \cup N = \{ (,), \text{Klammern}, \text{Liste} \}$

Unbenannte Terminale werden in ' eingeschlossen, um Verwechslungen mit KFG-Zeichen zu vermeiden: '('

Namen $\begin{matrix} \text{m} & \text{m} \\ \text{N} & \text{V}^* \end{matrix}$

Diese Grammatik definiert eine Sprache, deren Sätze Folgen von geschachtelten Klammerpaaren sind, z. B.

$((())) \quad (())(())$

© 2003 bei Prof. Dr. Uwe Kastens

Bedeutung der Produktionen

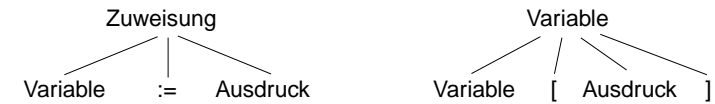
E EWS-3.14

Eine Produktion $A ::= x$ ist eine **Strukturregel**: A besteht aus x

Beispiele:

DeutscherSatz ::=	Subjekt Prädikat Objekt
Ein DeutscherSatz besteht aus (der Folge)	Subjekt Prädikat Objekt
Klammern ::=	'(' Liste ')'
Zuweisung ::=	Variable '=' Ausdruck
Variable ::=	Variable '[' Ausdruck ']'

Produktion graphisch dargestellt als gewurzelter **Baum** mit geordneten Kanten und mit Symbolen als Knotenmarken (zum Begriff „Baum“ siehe nächste Folie):



© 2003 bei Prof. Dr. Uwe Kastens

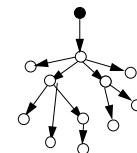
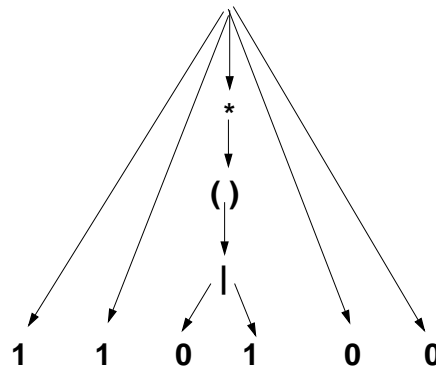
Bäume als Abstraktion

E EWS-3.15

Bäume bezeichnen in der Informatik **abstrakte Datenstrukturen** mit speziellen Eigenschaften.

Sie werden zur abstrakten Beschreibung bestimmter Zusammenhänge eingesetzt. Ein besonders wichtiger ist die „besteht-aus“-Beziehung zwischen einem Objekt und seinen Teilen.

Z. B. ein regulärer Ausdruck besteht aus Teilausdrücken:



© 2003 bei Prof. Dr. Uwe Kastens

Begriffe zu Bäumen

E EWS-3.16

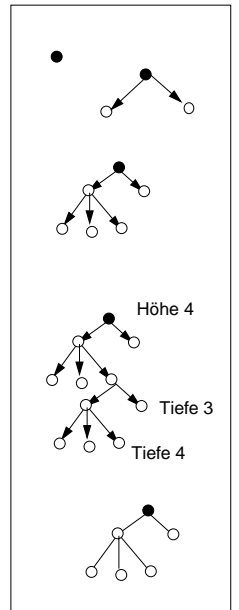
Definition: Ein (gewurzelter, gerichteter) **Baum** besteht aus

- einem **Knoten** k und
- einer (evtl. leeren) **Folge von Bäumen** und **Kanten** von k zu jedem Element der Folge.
- In **einen** Knoten mündet keine Kante, in alle anderen genau eine.

Begriffe und Eigenschaften:

- **Wurzel:** Knoten in den keine Kante mündet.
- **Blätter:** Knoten, von denen keine Kante ausgeht
- **innere Knoten:** es mündet eine Kante und es gehen welche aus
- Es gibt genau eine Wurzel.
- Wenn ein Baum n Knoten hat, dann hat er n-1 Kanten.
- **Tiefe eines Blattes:** Anzahl der Kanten auf dem Weg von der Wurzel zu dem Blatt.
- **Höhe des Baumes:** größte Tiefe aller seiner Blätter.

Knoten und/oder Kanten können **beschriftet** werden. Man kann die Pfeilspitzen weglassen, wenn die Wurzel bekannt ist.



© 2006 bei Prof. Dr. Uwe Kastens

Ableitungen

E EWS-3.17

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**; er wird notiert als $u A v \Rightarrow u x v$

z. B. **Klammern Klammern** Liste \Rightarrow **Klammern (Liste)** Liste
mit Produktion p1: **Klammern** ::= **(Liste)**

Beliebig viele Ableitungsschritte nacheinander angewandt heißen **Ableitung**; notiert als $u \Rightarrow^* v$

Eine kontextfreie Grammatik **definiert eine Sprache**; das ist eine **Menge von Sätzen**.
Jeder Satz ist eine Folge von Terminalsymbolen, die aus dem Startsymbol ableitbar ist:
 $L(G) = \{ w \mid w \in T^* \text{ und } S \Rightarrow^* w \}$

Grammatik auf EWS-3.13 definiert geschachtelte Folgen paariger Klammern als Sprachmenge:
 $\{ (), (()), (())(), ((())()), \dots \} \subseteq L(G)$

Ableitung des Satzes $((())())$:

S	= Klammern	p1
	\Rightarrow (Liste)	p2
	\Rightarrow (Klammern Liste)	p2
	\Rightarrow (Klammern Klammern Liste)	p1
	\Rightarrow (Klammern (Liste) Liste)	p1
	\Rightarrow ((Liste) (Liste) Liste)	p3
	\Rightarrow (() (Liste) Liste)	p3
	\Rightarrow (() (Liste)	p3
	\Rightarrow (() ()	p3
	\Rightarrow (() ()	

Ableitungsbäume

E EWS-3.18

Jede Ableitung kann man als gewurzelten **Baum** darstellen:

Die **Knoten** mit ihren Marken repräsentieren **Vorkommen von Symbolen**.

Ein Knoten mit seinen direkten Nachbarn repräsentiert die **Anwendung einer Produktion**.

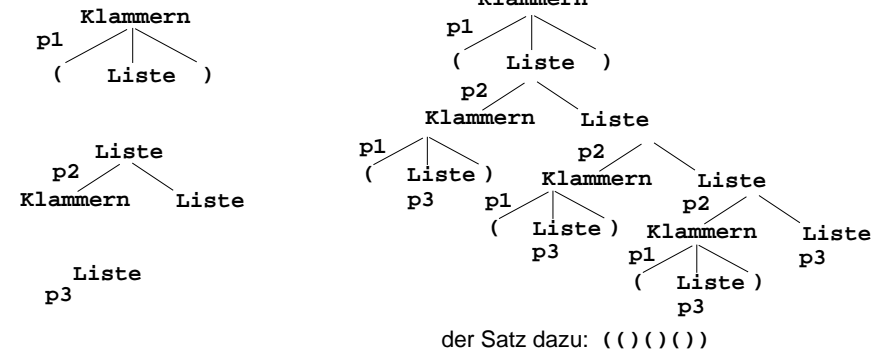
Die **Wurzel** ist mit dem **Startsymbol** markiert.

Terminale kommen nur an **Blättern** vor.

Ein Ableitungsbaum entsteht durch „Zusammensetzen von Kopien der Produktionsbäume“.

Produktionen:

ein **Ableitungsbaum:**



© 2004 bei Prof. Dr. Uwe Kastens

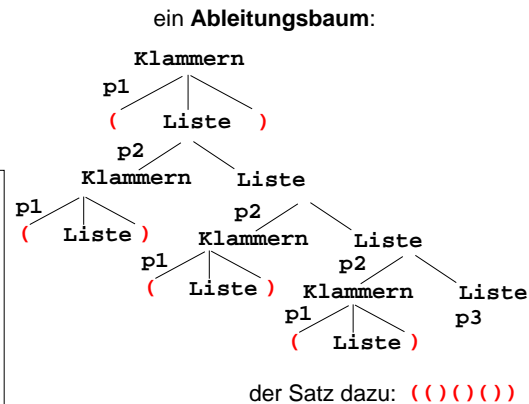
Satz zum Ableitungsbaum

E EWS-3.19

Man erhält den **Satz** aus einem **Ableitungsbaum**, wenn man seine **Terminale** in der Reihenfolge eines **links-abwärts Durchlaufes** aufschreibt.

Ein **links-abwärts Durchlauf**

- **beginnt** mit einem Besuch der **Wurzel**,
- bei dem Besuch eines Knotens werden nacheinander für jeden seiner **Unterbäume von links nach rechts** jeweils ein **links-abwärts Durchlauf** durchgeführt.



© 2006 bei Prof. Dr. Uwe Kastens

Grammatik für arithmetische Ausdrücke

E EWS-3.20

Grammatik für arithmetische Ausdrücke

Produktionen:

Beispiele:

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 ::= '/'

$b + c$
 $a * (b + c)$
 $a * b + c$
 $a + b * c$

$T = \{ (,), +, -, *, /, \text{Ident} \}$

$N = \{ \text{Expr}, \text{Fact}, \text{Opd}, \text{AddOpr}, \text{MulOpr} \}$

$S = \text{Expr}$

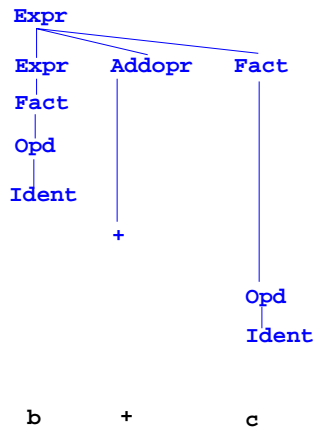
© 2003 bei Prof. Dr. Uwe Kastens

Beispiel für eine Ableitung zur Ausdrucksgrammatik

Satz der Ausdrucksgrammatik $b + c$
Ableitung:

Ableitungsbaum:

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



Präzedenz und Assoziativität in Ausdrucksgrammatiken

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:

<p>eine Präzedenzstufe, binärer Operator, linksassoziativ:</p> <p>$A ::= A \text{ Opr } B$ $A ::= B$</p>	<p>eine Präzedenzstufe, binärer Operator, rechtsassoziativ:</p> <p>$A ::= B \text{ Opr } A$ $A ::= B$</p>
<p>eine Präzedenzstufe, unärer Operator, präfix:</p> <p>$A ::= \text{Opr } A$ $A ::= B$</p>	<p>eine Präzedenzstufe, unärer Operator, postfix:</p> <p>$A ::= A \text{ Opr}$ $A ::= B$</p>

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

$H ::= \text{Ident}$

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

$H ::= \text{'(' } A \text{')'}$

Nützliche Grammatik-Konstrukte

beliebig lange Folgen von **stmt** z. B. in:

$\text{Block} ::= \text{'\{ ' } \text{Stmts} \text{'\}'}$
 $\text{Stmts} ::= \text{Stmts } \text{stmt}$
 $\text{Stmts} ::=$

Abkürzung für **beliebig lange Folgen**:

$\text{Block} ::= \text{'\{ ' } \text{stmt}^* \text{'\}'}$
 Die 2 Produktionen für **stmts** entfallen.

nicht leere Folgen von **stmt**:

$\text{Stmts} ::= \text{Stmts } \text{stmt}$
 $\text{Stmts} ::= \text{stmt}$

Entsprechend für **nicht-leere Folgen**:

$\text{Block} ::= \text{'\{ ' } \text{stmt}^+ \text{'\}'}$

nicht-leere Folgen von **stmt** **getrennt durch ;**:

$\text{Stmts} ::= \text{Stmts } \text{' ; ' } \text{stmt}$
 $\text{Stmts} ::= \text{stmt}$

Alternative Produktionen für dasselbe Nichtterminal mit | zusammenfassen, z. B.

$\text{Stmts} ::= \text{Stmts } \text{' ; ' } \text{stmt} \mid \text{stmt}$

Optionale Parameter z. B. in:

$\text{Call} ::= \text{Name } \text{'(' } \text{ParamOpt} \text{')'}$
 $\text{ParamOpt} ::= \text{Parameter}$
 $\text{ParamOpt} ::=$

Optionales mit [] klammern, z. B.

$\text{Call} ::= \text{Name } \text{'(' } \text{[Parameter] } \text{')'}$
 Die 2 Produktionen für **ParamOpt** entfallen.

Ausschnitte aus einer HTML-Grammatik

Die **Syntax von HTML** ist im Kalkül der „Document Type Definition (DTD)“ formal definiert. Man kann **Ausschnitte zu einigen Aspekten** von HTML in **KFGn** übertragen. Es folgen Beispiele dafür.

Grundstruktur (ohne Attribute innerhalb von Tags):

```
HTMLDoc ::= '<html>' '<head>' HeadContent '</head>'
          '<body>' Block '</body>'
          '</html>'

Block ::= Paragraph | Table | List | Heading | ...

Paragraph ::= '<p>' Inline ['</p>']

Inline ::= ... Fließtext mit Auszeichnungen ohne Blockstrukturen ...

Flow ::= Block | Inline
```

```
Tabellen: Table ::= '<table>' Row* '</table>'
           Row  ::= '<tr>' Cell* '</tr>'
           Cell ::= '<td>' Flow '</td>'
```

HTML-Grammatik: Listen und Attribute

Listen:

```
List ::= '<ol>' ListElement+ '</ol>' |
        '<ul>' ListElement+ '</ul>'

ListElement ::= '<li>' Flow '</li>'
```

Attribute in Anfangs-Tags.

In dieser Grammatik werden Tags weiter zerlegt:

```
AnfangsTag ::= '<' TagName Attribute* '>'

Attribute ::= AttributeName '=' AttributeValue

AttributeName ::= Identifier

AttributeValue ::= StringLiteral | Identifier | Number | ...
```