E2. Symbole und Syntax
Überblick

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

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:

Σ =	{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 $a_1\ a_2\ ...\ a_n$, mit $a_i \in A$, für i = 1,..,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

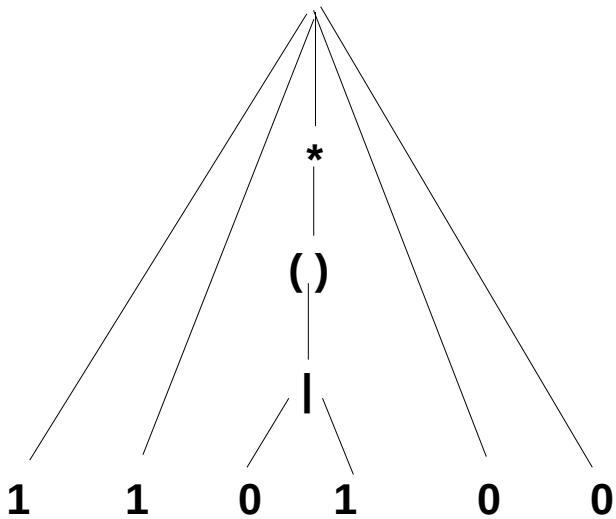
Ein **regulärer Ausdruck *R***
definiert eine **Mengen von Worten** über einem Alphabet A, die **Sprache L (*R*)**.

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
F G	{ f g | 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

1 1 (0 | 1)* 0 0



Zusammenfügen, 5-fach

Folge

Klammern

Alternative

Zeichen

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 | ε)	ac bc ad bd a b
Anrede =	Sehr geehrte(r | ε) (Frau | Herr)	Sehr geehrte Frau
Dig =	0 | 1 | ... | 9	7
sLet =	a | b | ... | z	x
cLet =	A | B ... | 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* ((_ | ε) (*Let* | *Dig*))*
PHP_Var_Identifier =	$ (*Let* | _) (*Let* | _ | *Dig*)*
Pascal_Real =	((*Dig*$^{+}$. *Dig*$^{+}$) ((e | E) (+ |- | ε) *Dig*$^{+}$) | ε) | (*Dig*$^{+}$ (e | E) (+ | - | ε) *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 ews(0|1|2|3|4|5|6|7|8|9)^{3}.html

in der Schreibweise von Unix-sh
```
ls ews[0-9][0-9][0-9].html
```

Aufruf einer PHP-Funktion
```
preg_match ("/[dD]aß/", $absatz)
```
sucht ein Textmuster in einer Zeichenreihe — Muster — Zeichenreihe

```
$d = "[0-9]";
preg_match ("/ews$d$d$d\.html/", $files)
```

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
F G	Zusammenfügen von 2 Worten
F | *G*	Alternativen
(F)	Klammerung
F?	Option; wie *F* | ε
*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 Zeichenreihenliterale (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-Z_0-9]*";

$ADA_Identifier =     "[a-zA-Z](_?[a-zA-Z0-9])*";

$PHP_Var_Identifier = "\$[a-zA-Z_][a-zA-Z_0-9]*";

$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

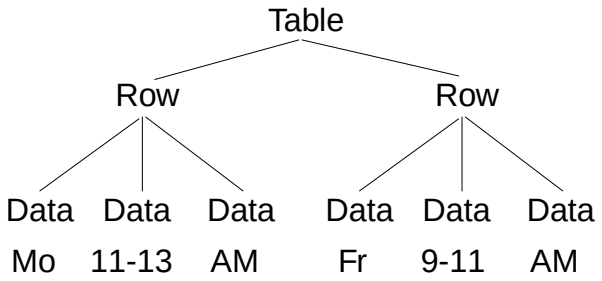
- **Sprache** als Menge von Sätzen; jeder **Satz** ist eine **Folge von Symbolen**
- **Menge von Bäumen**; jeder Baum repräsentiert die **Struktur eines Satzes** der Sprache

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

Terminale **T** = { **(** , **)** }

Nichtterminale **N** = { **`Klammern`**, **`Liste`** }

Startsymbol **S** = **`Klammern`**

Produktionen **P** =

{

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

p2: **`Liste ::= Klammern Liste`**

p3: **`Liste ::=`**

}

Vokabular V = T ∪ N =
{ **(** , **)**, **`Klammern`**, **`Liste`** }

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

∩ ∩

Namen **N** **V***

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

`(()())` **`((()(()()()))`**

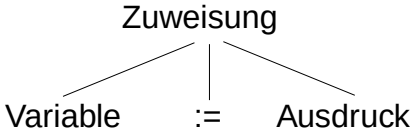
Bedeutung der Produktionen

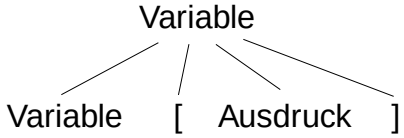
Eine Produktion A ::= x ist eine **Strukturregel**: A **besteht aus** x

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):



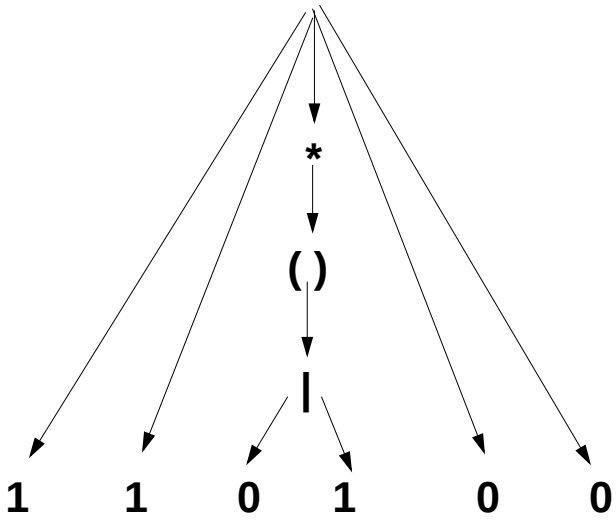


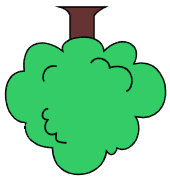
Bäume als Abstraktion

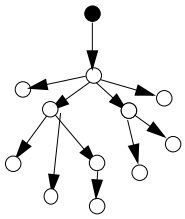
Bäume bezeichnen in der Informatik **abstrakte Datenstrukturen** mit speziellen Eigenschaften.

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:







Begriffe zu Bäumen

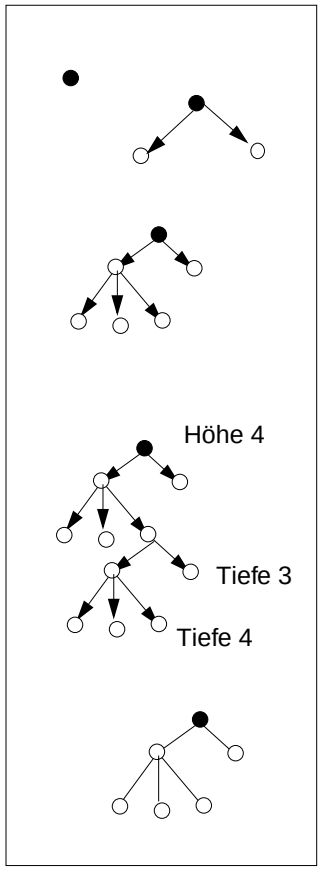
Definition: Ein (gewurzelter, gerichteter) **Baum** besteht aus

- 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.



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**; er wird notiert als u A v => u x v

z. B. **`Klammern Klammern Liste => Klammern ( Liste ) Liste`**
mit Produktion **`p1: Klammern ::= ( Liste )`**

Beliebig viele Ableitungsschritte nacheinander angewandt heißen **Ableitung**; notiert als u =>* 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 | w ∈ T* und S =>* w }

Grammatik auf EWS-3.13 definiert geschachtelte Folgen paariger Klammern als Sprachmenge:
{ (), (()), (()()()),((()())(())), ... } ⊆ L(G)

Ableitung des Satzes `(()())`:

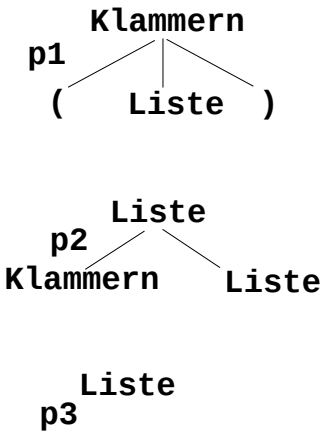
S	**= Klammern**	**p1**
	=> (Liste)	**p2**
	=> (Klammern Liste)	**p2**
	=> (Klammern Klammern Liste)	**p1**
	=> (KLammern (Liste) Liste)	**p1**
	=> ((Liste) (Liste) Liste)	**p3**
	=> (( ) (Liste) Liste)	**p3**
	=> (( ) ( ) Liste)	**p3**
	=> (( ) ( ))	

Ableitungsbäume

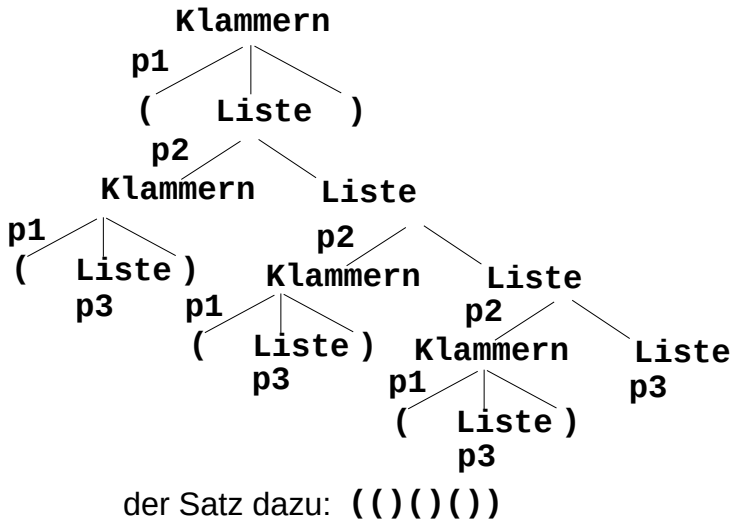
Jede Ableitung kann man als gewurzelten **Baum** darstellen:
Die **Knoten** mit ihren Marken repräsentieren **Vorkommen von Symbolen**.
Ein Knoten mit seinen direkten Nachbarn repräsentiert die **Anwendung einer Produktion**.
Die **Wurzel** ist mit dem **Startsymbol** markiert.
Terminale kommen nur an **Blättern** vor.

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

Produktionen:



ein **Ableitungsbaum**:



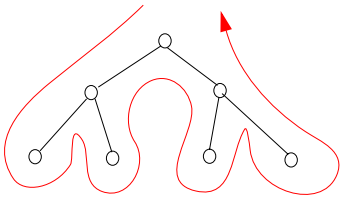
der Satz dazu: **`(()())`**

Satz zum Ableitungsbaum

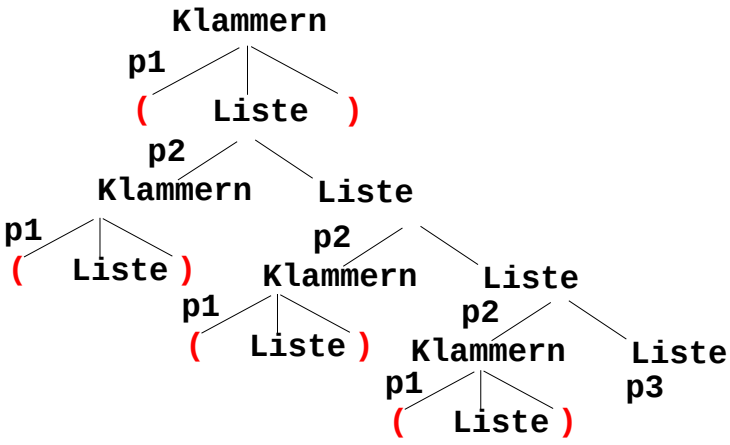
Man erhält den **Satz** aus einem **Ableitungsbaum**, wenn man seine **Terminale** in der Reihenfolge eines **links-abwärts Durchlaufes** aufschreibt.

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.



ein **Ableitungsbaum**:



der Satz dazu: **(()()())**

Grammatik für arithmetische Ausdrücke

Grammatik für arithmetische Ausdrücke
Produktionen:

```
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 ::= '/'
```

Beispiele:

```
b + c

a * (b + c)

a * b + c

a + b * c
```

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

N = { **Expr, Fact, Opd, AddOpr, MulOpr** }

S = **Expr**

Beispiel für eine Ableitung zur Ausdrucksgrammatik

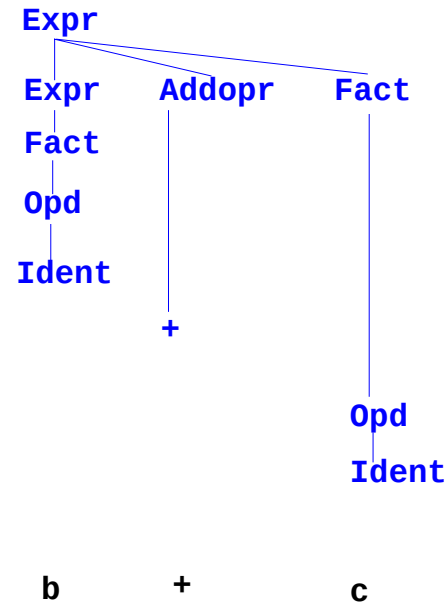
Satz der Ausdrucksgrammatik **`b + c`**

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:

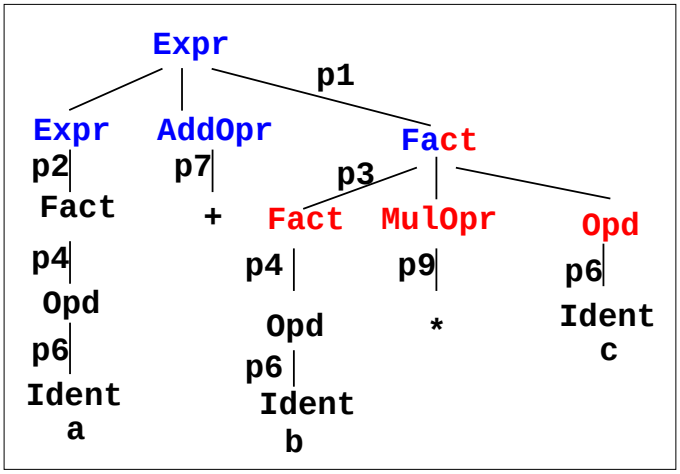


Präzedenz und Assoziativität in Ausdrucksgrammatiken

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:

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 ')'
```


Nützliche Grammatik-Konstrukte

beliebig lange Folgen von **`Stmt`** z. B. in:

```
Block ::= '{' Stmts '}'

Stmts ::= Stmts Stmt
Stmts ::=
```

Abkürzung für **beliebig lange Folgen**:

```
Block ::= '{' Stmt* '}'
```

Die 2 Produktionen für **`Stmts`** entfallen.

Entsprechend für **nicht-leere Folgen**:

```
Block ::= '{' Stmt+ '}'
```

nicht leere Folgen von **`Stmt`**:

```
Stmts ::= Stmts Stmt
Stmts ::= Stmt
```

nicht-leere Folgen von **`Stmt`** **getrennt durch `;`**:

```
Stmts ::= Stmts ';' Stmt
Stmts ::= Stmt
```

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

```
Stmts ::=  Stmts ';' Stmt |
           Stmt
```

Optionale `Parameter` z. B. in:

```
Call ::= Name '(' ParamOpt ')'

ParamOpt ::= Parameter
ParamOpt ::=
```

Optionales mit [ ] klammern, z. B.

```
Call ::= Name '(' [ Parameter ] ')'
```

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 KFG**n 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 | ...
```