Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Kontextfreie Sprachen und Grammatiken

- Rekursive Muster und ihre Beschreibung
- Grammatiken: eindeutige / mehrdeutig, Ableitungen, Ableitungsbäume
- Präzendenzen und Assoziativitäten
- Abstrakte Syntaxbäume

Übersicht

Im letzten Abschnitt wurden nicht-rekursive Muster, ihre Definition und Mechanismen zum Erkennen dieser Muster behandelt.

Im Compilerbau werden nicht-rekursive Muster vor allem zur Definition und dem effizienten Erkennen von lexikalischen Elementen eingesetzt. Die behandelten Formalismen und Technologien haben aber weit darüber hinaus außerordentlich viele praktische Anwendungen.

In diesem Abschnitt

beschäftigen wir uns mit der Definition und dem Erkennen von (eventuell) rekursiven Mustern

Thematik 1: Grammatiken

Grammatiken sind ein Formalismus zur Definition von Sprachen. Eine Sprache ist eine Menge von „Wörtern“, also ein Muster.

Thematik 2: Syntaxbäume und ASTs, abstrakte Syntaxbäume

Ein Ableitungsbaum gibt die syntaktische Struktur eines Textes wieder. Typischerweise enthält ein Ableitungsbaum viele redundante Information, es ist darum besser mit einer „bereinigten Version“, einem AST, weiter zu arbeiten.

Thematik 3: Parser

Parser sind die Muster-Erkenner, der durch Grammatiken definierten Muster (Sprachen)

Parser werden im nächsten und übernächsten Foliensatz behandelt

Muster und ihre rekursive Beschreibung

Grammatik

Beispiel 1: Vollständig geklammerte arithmetische Ausdrücke

In Foliensatz 1 haben wir schon eine Grammatik kennen gelernt:

Expression → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Expression Operator Expression*)
Operator → + | – | * | /
Startsymbol: *Expression*

Grammatik: Nonterminale, die durch Produktionen definiert werden.

Diese Grammatik beschreibt in intuitiver Art vollständig geklammerte arithmetische Ausdrücke über Ziffern:

- Das Muster / die Sprache: Die Menge aller Zeichenketten, die als „vollständig geklammerter arithmetischer Ausdruck“ gelten
- Die syntaktische Struktur der Elemente dieser Menge

Die Definition ist produktiv

Es wird also definiert, welche Texte „dazugehören“,

nicht wie man erkennt, ob ein Text dazu gehört.

Die Definition ist induktiv / rekursiv

Es wird ein Prozess definiert, mit dem alle, „die dazu gehören“, erzeugt werden können.

Der Prozess ist rekursiv: in einer Produktion können beliebige Nonterminale verwendet werden. (Ohne Rekursion wäre eine solche Grammatik eine Sammlung regulärer Ausdrücke)

Grammatik vs Regulärer Ausdruck

Gemeinsames

- Beide – Grammatiken und reguläre Ausdrücke – sind **produktive Definitionen**:
  - Sie legen fest: Wie die Elemente der zu definierenden Menge erzeugt werden
- Der definierte Produktionsprozess ist jeweils **induktiv / rekursiv**, er legt fest:
  - Was ist die Basis
  - Wie wird aus Gegebenem Neues erzeugt

```
Expression → Number | Operation
Number     → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation  → ( Expression Operator Expression )
Operator   → + | – | * | /
Startsymbol: Expression
```

Definiert die Menge aller Texte die so produziert werden können:

- **Basis**: Eine ***Number*** ist eine ***Expression***
- **Induktion**: Sind e_1 und e_2 Expressions, dann ist auch $(e_1\ op\ e_2)$, wobei **op** ein ***Operator*** ist, eine ***Expression***.

Eine Grammatik als Definition eines Produktionsalgorithmus' für Textmengen

```
a (b | c)*
```

Definiert die Menge aller Texte die so produziert werden können:

- **Basis**: Produziere $\mathcal{A}$= {a}, $\mathcal{B}$={b}, $\mathcal{C}$={c}
- **Induktion**: Wende die Operatoren an:

  $\mathcal{BC} = \mathcal{B} \cup \mathcal{B}$

  $\mathcal{BC}Star = \bigcup_{n \in \mathbf{N}} \mathcal{BC}^n$

  $\mathcal{A}\_\mathcal{BC}Star = \{ x\ y \mid x \in \mathcal{A}\ ,\ y \in \mathcal{BC}Star\}$

Ein regulärer Ausdruck als Definition eines Produktionsalgorithmus' für Textmengen

Grammatik vs Regulärer Ausdruck

Die definierten Produktionsprozesse unterscheiden sich:

- **Reguläre Ausdrücke definieren Operationen auf (unendlichen) Textmengen**

  Ein Text gehört zu der definierten Sprache, wenn er Element der definierten (unendlichen) Menge ist.

  Bei der Definition des Produktionsprozesses (eines regulären Ausdrucks) ist allgemeine Rekursion nicht erlaubt, nur die „Schleifen-Rekursion“ (Wiederholungen) kann verwendet werden.

- **Grammatiken definieren nichtdeterministische Operationen auf Texten**

  Ein Text gehört zu der definierten Sprache, wenn er eine Ableitung vom Startsymbol hat, also wenn er erzeugt werden kann durch irgendeine Folge von Ersetzungen eines Nonterminals durch eine seiner Produktionen.

  Bei der Definition des Produktionsprozesses (einer Grammatik) ist Rekursion erlaubt.

Grammatik vs Regulärer Ausdruck

Die definierten Produktionsprozesse haben unterschiedliche Mächtigkeit

- **Jede Sprache (Textmenge), die mit einem regulären Ausdruck definiert werden kann, kann auch mit einer Grammatik definiert werden.**
- **Es gibt Sprachen (Textmengen) die mit einer Grammatik definiert werden können, aber nicht mit einem regulären Ausdruck.**

Grammatik vs Regulärer Ausdruck

Jeder reguläre Ausdruck kann in eine Grammatik umgewandelt werden.

Mit einem einfachen, induktiv über ihre Struktur definierten Algorithmus können reguläre Ausdrücke r in Grammatiken umgewandelt werden:

- r = a ~> $S_r \rightarrow a$
- r = $r_1 \mid r_2$ ~> $S_r \rightarrow S_{r1} \mid S_{r2}$
- r = $r_1\ r_2$ ~> $S_r \rightarrow S_{r1}\ S_{r2}$
- r = r_1^* ~> $S_r \rightarrow S_{r1}\ S_r \mid \varepsilon$ ε ist das „leere Wort“

Beispiel:

a | bc* ~>

A → a
B → b
C → c
D → C D | ε
E → B D
F → A | B

oder kurz:

S → a | b T
T → c T | ε

Grammatik vs Regulärer Ausdruck

Es gibt Sprachen, die mit Grammatiken, aber nicht mit regulären Ausdrücken definiert werden können.

Beispiel: Die Sprache

S = { ab, aabb, aaabbb, … }

der Texte, die aus Sequenzen von a's gefolgt von einer gleich langen Sequenz von b's bestehen, kann mit einer Grammatik definiert werden:

S → aSb | ab

Es gibt jedoch keinen regulären Ausdruck mit dem diese Sprache definiert werden kann.

Man beachte:

- Weder a*b* noch (ab)* definieren S
- Reguläre Ausdrücke der Form

  ab | aabb | aaabbb | aaaabbbb | … $a^n b^n$

  definieren eine beliebig große Teilmenge von S

  Für die gesamte Menge S müsste der Ausdruck jedoch unendlich groß sein,
- … oder Rekursion müsste für reguläre Ausdrücke erlaubt sein

  S = ab | aSb
- Rekursion ist aber nicht erlaubt:

  Mit Rekursion ginge die Äquivalenz mit endlichen Automaten und damit die effiziente Implementierung verloren.

Zusammenfassung: Grammatik vs Regulärer Ausdruck

Grammatiken

- sind ein mächtigerer Definitions-Formalismus als reguläre Ausdrücke und endliche Automaten

  (Letztlich sind Grammatiken „reguläre Ausdrücke mit Rekursion“)

- Grammatiken benötigen dafür aber aufwendigere Erkennungsalgorithmen.

Rolle der Rekursion

- Grammatiken und reguläre Ausdrücke definieren beide (unterschiedliche) induktive / rekursive Produktionsprozesse für (meist unendliche) Mengen.
- Nur bei Grammatiken darf in der Definition des Produktionsprozesses selbst Rekursion unbeschränkt verwendet werden.

Terminologie und Formalismus

Kontextfreie Grammatik

Definition

Eine *kontextfreie Grammatik* besteht aus

- einer endlichen Menge T von *Terminalsymbolen* (oder Tokens)
- einer endlichen Menge N von *Nonterminalsymbolen* (oder Variablen), $N \cap T = \{\}$
- einer endlichen Menge P von Ableitungsregeln (oder Produktionen) der Form

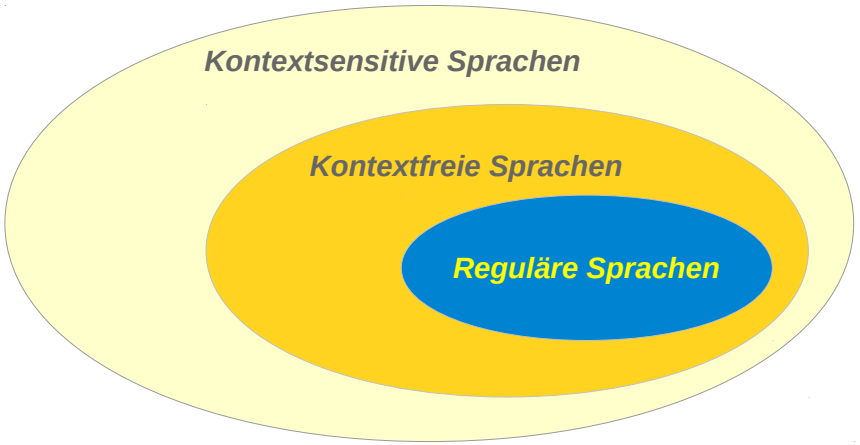
  $\alpha \rightarrow w$, wobei $a \in N, w \in (N \cup T)^*$

  a heißt linke Seite, w rechte Seite der Regel.
- einem Startsymbol S $\in$ N.

Quelle: M. Jäger, *Compilerbau*

Bemerkung: Kontextfrei

- Eine Kontextfreie Grammatik ist eine von mehreren Varianten von Grammatiken
- Die kontextfreien Grammatiken sind die wichtigste Variante, mit „Grammatik“ ist hier – und ansonsten meist auch – stets „kontextfreie Grammatik“ gemeint.
- Folgende Varianten von Grammatiken werden in der Theorie der formalen Sprachen unterschieden:
  - Reguläre Grammatiken definieren reguläre Sprachen
  - Kontextfreie Grammatiken definieren kontextfreie Sprachen
  - Kontextsensitive Grammatiken definieren kontextsensitive Sprachen



Reguläre Grammatiken entsprechen regulären Ausdrücken

Notation

Produktion / Regel

– **Ein Grammatik enthält als wichtigsten Bestandteil eine oder mehrere Produktionen (auch Regeln) der Form**

Head → *Body*

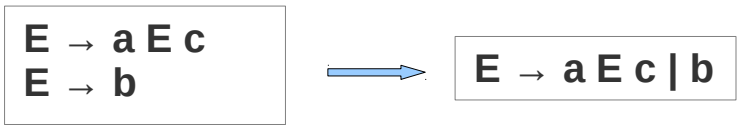
Dabei ist

- *Head* **ein Nichtterminal und**
- *Body* **eine Folge von Terminalen und Nichtterminalen**

– **Abgekürzte Notation**

Alle Produktionen mit dem gleichen *Head* werden in einer Produktion zusammengefasst.

Beispiel:



Notation: EBNF

BNF / EBNF

- Notationen für kontextfreie Grammatiken, benannt nach J. W. Backus und P. Naur
- BNF: Backus / Naur Form,
- EBNF: Extended Backus / Naur Form

EBNF

::= statt →

Nichtterminale in spitzen Klammern, Terminale in Hochkomma

EBNF

Die rechte Seite einer Regel ist ein regulärer Ausdruck

Beispiel:

```
Exp      ::= Term { ('+' | '-') Term }
Term     ::= Factor { ('+' | '-') Term }
Factor   ::= Number | '(' Exp ')'
```

Metasymbole:

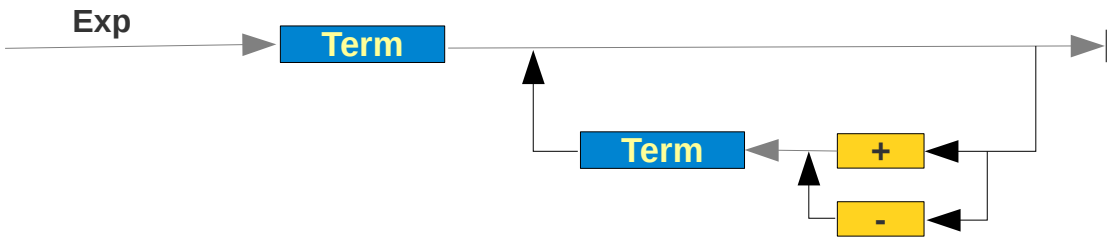
{ .. } Wiederholung
| Alternative
(..) Gruppierung

Notation: EBNF

Syntaxdiagramm

Graphische Darstellung einer EBNF-Produktion

```
Exp  ::= Term { ('+' | '-') Term }
```



Notation

Terminal / Token

- Ein Terminal ist ein elementares Zeichen:

  Ein Element des „Alphabets“ aus dem „die Worte der Sprache“ zusammengesetzt sind

  oder

  Ein „Wort“ aus der „Menge der Worte“ aus denen die „Sätze der Sprache“ zusammengesetzt sind.

- **Notation: Terminale**

  Terminale werden in theoretischem Kontext stets mit **Kleinbuchstaben**, Operatoren oder Ziffern bezeichnet.

  In praktischem Kontext (Compilerbau) ist es üblich

  - Anderweitig – z.B. durch reguläre Ausdrücke – definierte Tokens zu verwenden oder
  - Zeichenfolgen mit oder ohne Hochkommas zu verwenden

Notation

Nichtterminal / Syntaktische Kategorie

- Terminale sind vorgegeben,
  Nichtterminale sind das, was definiert wird.
- Jedes Nichtterminal steht für eine syntaktische Kategorie, d.h. für eine Menge von Zeichenfolgen die syntaktisch gleichwertig sind.
- Beispiel: Alle Elemente der syntaktischen Kategorie Operator sind syntaktisch gleichwertig: Jeder Operator kann in gleicher Art verwendet werden um Ausdrücke aufzubauen.
- Notation: Nichtterminale
  Nichtterminale werden in theoretischem Kontext stets mit Großbuchstaben bezeichnet.
  In praktischem Kontext (Compilerbau) ist es üblich sie in spitzen Klammern einzuschließen.
- Beispiele

```
E → E + E | N
N → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

```
<Expression> → <Expression> + <Expression> | <Number>
<Number> = [1-9][0-9]*
```

```
<Statement> → 'while' <Expression> '{' <StatementList> '}'
```

Terminologie und Formalismus

Ableitbar / direkt ableitbar

Definition

Sei G = (T,N,P,S) eine Grammatik, u, x, y, w $\in$ (T $\cup$ N)*.

Aus *uxw* ist *uyw* **direkt ableitbar** (Notation: $uxw \rightarrow_G uyw$), wenn $(x \rightarrow y) \in$ P.

Aus *uxw* ist *uyw* **ableitbar** (Notation: $uxw \overset{*}{\rightarrow}_G uyw$), wenn für ein $n > 0$ Wörter $v_0, \ldots, v_n$ existieren, so dass

$$uxw = uv_0w \rightarrow_G uv_1w \cdots \rightarrow_G uv_nw = uyw$$

(Die Ableitbarkeitsrelation ist demzufolge der transitive und reflexive Abschluss der direkten Ableitbarkeit.)

Quelle: M. Jäger, *Compilerbau*

Terminologie und Formalismus

Erzeugte Sprache / Satzform

Definition

Die von G erzeugte Sprache L(G) ist die Menge aller aus dem Startsymbol ableitbaren Wörter, die nur aus Terminalsymbolen bestehen:

$L(G) = \{w \in T^* \mid S \overset{*}{\rightarrow}_G w\}$

Definition

Sei G = (N, T, P, S) eine Grammatik. Ein Wort $w \in (N \cup T)^*$, das sich aus S ableiten lässt, $S \overset{*}{\rightarrow}_G w$, heißt **Satzform** (zu G).

Quelle: M. Jäger, *Compilerbau*

Erzeugte Sprache

Die erzeugte Sprache ist die Semantik einer Grammatik:

Eine Grammatik bedeutet / definiert die von ihr erzeugte Sprache

Die erzeugte Sprache ist die Menge der korrekten Sätze / Wörter der Sprache

Sie wird oben – wie in der Theorie der formalen Sprachen üblich – über den Prozess der Ableitung definiert.

Diese Definition ist äquivalent zu einer induktiven Definition der Sprache:

- **Für jedes Nichtterminal N wird $\mathcal{L}(N)$, die *von N erzeugte Sprache* induktiv definiert**
- **Die erzeugte Sprache der Grammatik G ist die erzeugte Sprache des Startsymbols S der Grammatik:**

  $L(G) = \mathcal{L}(S)$

Die von einem Nichtterminal N erzeugte Sprache $\mathcal{L}(N)$ ist induktiv wie folgt definiert:

- **Basis: $\mathcal{L}(N) = \{\}$**
- **Induktion:**

  Angenommen $N \rightarrow X_1\ X_2\ \ldots\ X_n$ ist eine Produktion, dann ist auch

  $s_1\ s_2\ \ldots\ s_n$ in $\mathcal{L}(N)$

  wobei $s_i = X_i$ falls X_i ein Terminal ist

  $s_i = x$ falls X_i ein Nichtterminal ist und $x \in \mathcal{L}(X_i)$

Terminologie und Formalismus

Ableitungsbaum

Definition

Ein **Ableitungsbaum** ist ein geordneter Baum, der aus einer Ableitung A eines Worts w bezüglich einer Grammatik G = (N, T, P, S) wie folgt konstruiert wird:

- Der Wurzelknoten wird mit dem Startsymbol S markiert
- Zu jeder in der Ableitung angewandten Regel $X \rightarrow x_1 ... x_n$ werden dem mit X markierten Knoten im Baum, der das abgeleitete Nonterminal repräsentiert, n neue, mit $x_1, ..., x_n$ markierte Knoten als Nachfolgeknoten zugeordnet.

Quelle: M. Jäger, *Compilerbau*

Chomsky-Hierarchie: Varianten von Grammatiken

- **und der mit ihnen definierbaren Sprachen**

- Bei Chomsky-0-Grammatiken gibt es keine Restriktionen für die Ableitungsregeln. Ob ein Wort w mit den Regeln der Grammatik aus dem Startsymbol ableitbar ist, ist **nicht entscheidbar**.

- Chomsky-1-Grammatiken nennt man auch **kontextsensitive** Grammatiken. Für diese Grammatiken gibt es bestimmte Restriktionen für die Ableitungsregeln und die theoretische Aussage, dass die Zugehörigkeit eines Wort zu einer Chomsky-1-Sprache entschieden werden kann.

  Für den Compilerbau sind Chomsky-1-Grammatiken zu komplex.

Kontextsenisitive Grammatiken: In Produktionen sind als Head (links) auch Kombinationen aus Terminalen und Nichtterminalen erlaubt.

- Chomsky-2-Grammatiken sind die **kontexfreien** Grammatiken.

  Die kontexfreien Sprachen eignen sich zur Spezifikation der hierarchischen Programmstruktur für alle gängigen Programmiersprachen. Zur Syntaxanalyse können Automaten mit einem Stack (Kellermaschinen) verwendet werden.

Kontextfreie Grammatiken: Rekursion in den Regeln erlaubt. In Produktionen sind nur Nonterminale als Head (links) erlaubt.

- Chomsky-3-Grammatiken nennt man auch **reguläre** Grammatiken. Hier sind nur links- oder rechtsrekursive Regeln ($X \rightarrow Xw$ bzw. $X \rightarrow wX$) erlaubt, aber keine allgemeineren Rekursionsformen.

  Die regulären Sprachen eignen sich zur Spezifikation der lexikalischen Ebene der Sprachsyntax. Zur Erkennung können Endliche Automaten verwendet werden, ein Stack ist im Gegensatz zu den Chomsky-2-Grammatiken nicht notwendig

Reguläre Grammatiken sind äquivalent zu regulären Ausdrücken. Nur „schleifen-artige“, keine „eingebettete Rekursion“ in Produktionen. Eine reguläre Sprache kann mit einer einzigen EBNF-Produktion definiert werden.

Relevant im Compilerbau

Quelle: M. Jäger, *Compilerbau*

Ableitungsbaum (*Parse Tree*)

Zu jedem korrekten Satz einer Grammatik gibt es (mindestens) einen Ableitungsbaum.

Der Ableitungsbaum gibt den Prozess der Erzeugung des Satzes wider.

Der Prozess der Erzeugung aller Sätze der Sprache, die durch eine Grammatik definiert wurde, kann leicht in den Prozess der Konstruktion aller Ableitungsbäume umgeändert werden.

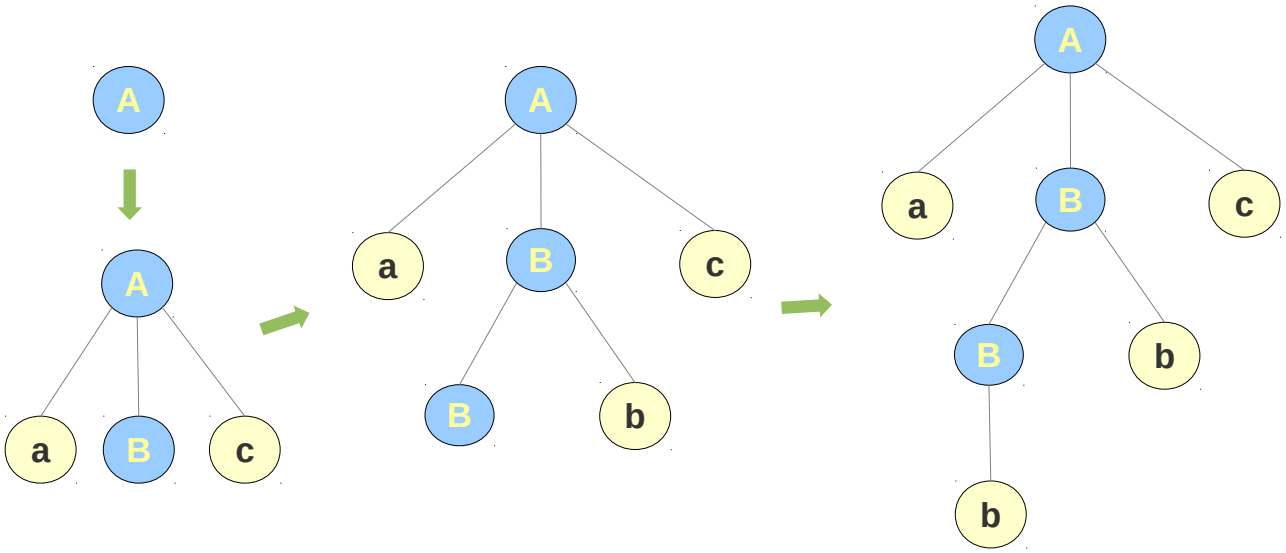
Beispiel:

A → aBc
B → Bb | b

Grammatik

A → aBc
→ aBbc
→ abbc

abbc ist ein Satz der definierten Sprache: Der Satz kann nach den Regel der Grammatik abgeleitet / konstruiert werden



Regeln zur Produktion von Ableitungsbäumen können genauso leicht auf Basis einer Grammatik definiert werden. Statt abbc wäre dann der Ableitungsbaum ein Element dieser „Baum-Sprache“.

Eindeutige und mehrdeutige Grammatiken

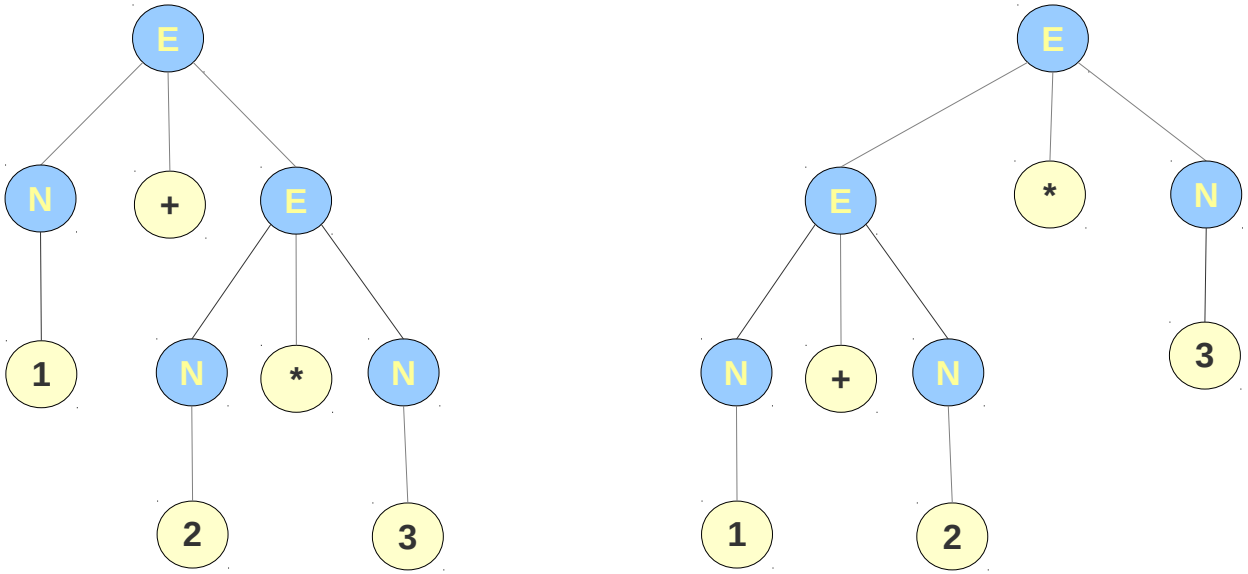
Eine Grammatik ist eindeutig, wenn jeder ihrer Sätze auf genau eine Art konstruiert / abgeleitet werden kann.

Eine Grammatik ist mehrdeutig, wenn es mindestens einen Satz gibt, der auf mehr als eine Art erzeugt / abgeleitet werden kann.

Beispiel:

$$E \rightarrow E + E \mid E * E \mid N$$
$$N \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$$

Eine mehrdeutige Grammatik



*Zwei Ableitungen für 1+2*3*

Eindeutige und mehrdeutige Grammatiken

Die Bedeutung eines Satzes ergibt sich aus seiner Struktur.

Ist eine Grammatik mehrdeutig, dann gibt es Sätze mit mehreren Strukturen, diese Sätze sind mehrdeutig.

Im Compilerbau (und in den natürlichen Sprachen meist auch) sind mehrdeutige Sätze nicht erwünscht: Ein Programm muss eindeutig zu interpretieren sein.

```
Die Beschimpfungen dieses
Schriftstellers sind
unerträglich.*
```

*** Quelle: https://files.ifi.uzh.ch/cl/volk/SyntaxVorl/Vorl_10.Ambig.html**

Eindeutige und mehrdeutige Grammatiken

Eine Grammatik kann Eindeutigkeit erreichen

- durch den vermehrten Einsatz von Terminalen
- durch komplexere Produktionsregeln

Beispiel arithmetische Ausdrücke

Expression → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Expression Operator Expression*)
Operator → + | – | * | /

Mehr Terminale (hier Klammern) machen die Ableitung / Struktur eindeutig.

Expression → *Expression AddOp* Term | *Term*
Term → *Term MultOp* Faktor | *Faktor*
Faktor → *Number* | (*Expression*)
AddOp → + | -
MultOp → * | /
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Die komplexere Grammatik macht die Ableitung / Struktur ebenfalls eindeutig.

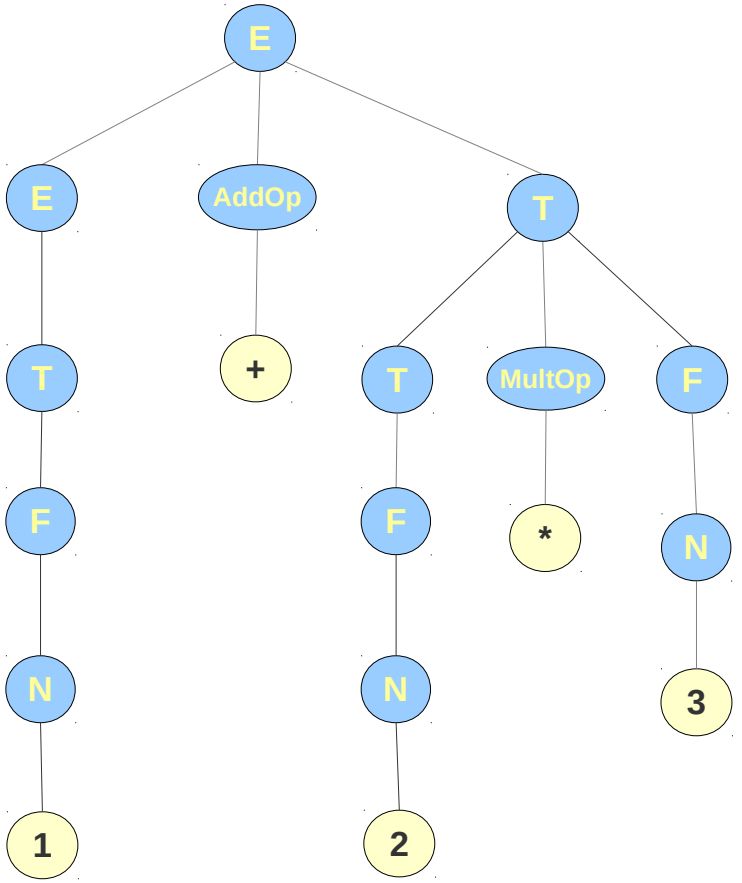
Prioritäten durch Hierarchie der Nontermninale: Expression, Term, Faktor
Links-Assoziativität der Operatoren durch Links-Rekursion in der Grammatik.

Eindeutige und mehrdeutige Grammatiken

Beispiel arithmetische Ausdrücke

Expression	→	*Expression AddOp Term* | *Term*
Term	→	*Term MultOp* Faktor | *Faktor*
Faktor	→	*Number* | (*Expression*)
AddOp	→	+ | -
MultOp	→	* | /
Number	→	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

*Eindeutige Ableitung für 1+2*3*



Präzedenzen und Assoziativitäten

Operatoren können auf unterschiedliche Arten syntaktisch mit ihren Argumenten verbunden werden:

- **Präzedenzen**

  Die Operatoren können unterschiedliche Bindungsstärken haben,

  z.B.: * bindet stärker als +

  2+3*4 = (2 + (3*4))

- **Assoziativitäten**

  Die Operatoren können von links oder rechts gruppiert werden:

  - **Linksassoziativ**

    $a \circ b \circ c = ((a \circ b) \circ c)$

  - **Rechtsassoziativ**

    $a \circ b \circ c = (a \circ (b \circ c))$

Mit Präzedenzen und Assoziativiäten können Ausdrücke ohne Klammern kurz und trotzdem eindeutig dargestellt werden. Sie komplizieren allerdings die Grammatik und erschweren das Parsen.

Präzedenzen und Grammatiken

Präzedenzen können in einer Grammatik über eine Hierarchie der Nonterminale zum Ausdruck gebracht werden.

Exp → *Exp* '+' *Term* | *Term*
Term → *Term* '*' *Factor* | *Factor*
Factor → *Number* | '(' *Exp* ')'

zunehmende Präzedenz

Assoziativitäten und Grammatiken

Assoziativitäten können in einer Grammatik über Links- und Rechts-Rekursion zum Ausdruck gebracht werden.

Exp → *Exp* '+' *Term* | *Term* — **Links-Rekursion: Links-Assoziativ**

Exp → *Term* '+' *Exp* | *Term* — **Rechts-Rekursion: Rechts-Assoziativ**

Assoziativitäten, Präzedenzen und Grammatiken

Schema zum Einbringen von Präzedenz und Assoziativität

Eine Ableitungsregel der Form $X \to Xw$ heißt *linksrekursiv*. Eine Ableitungsregel der Form $X \to wX$ heißt *rechtsrekursiv*.

Präzedenzgruppen und Assoziativitätsregeln für zweistellige Operatoren lassen sich in einer Grammatik wie folgt umsetzen:

- Jeder Präzedenzstufe i wird in der Grammatik ein eigenes Nonterminalsymbol $Expr_i$ zugeordnet.
- Ein linksassoziativer Operator Op_i der Stufe i wird durch die linksrekursive Regel

  $$Expr_i \to Expr_i Op_i Expr_{i+1}$$

  beschrieben. Dadurch ist sichergestellt, dass der rechte Operand weder den Operator selbst, noch Operatoren gleicher oder geringerer Präzedenz (z.B. Stufe i-1) in ungeklammerter Form enthält.
- Entsprechend wird ein rechtsassoziativer Operator durch die rechtsrekursive Regel

  $$Expr_i \to Expr_{i+1} Op_i Expr_i$$

  beschrieben.
- Für einen nichtassoziativen Operator *Op* der Stufe i lautet die Regel

  $$Expr_i \to Expr_{i+1} Op_i Expr_{i+1}$$

Quelle: M. Jäger, *Compilerbau*

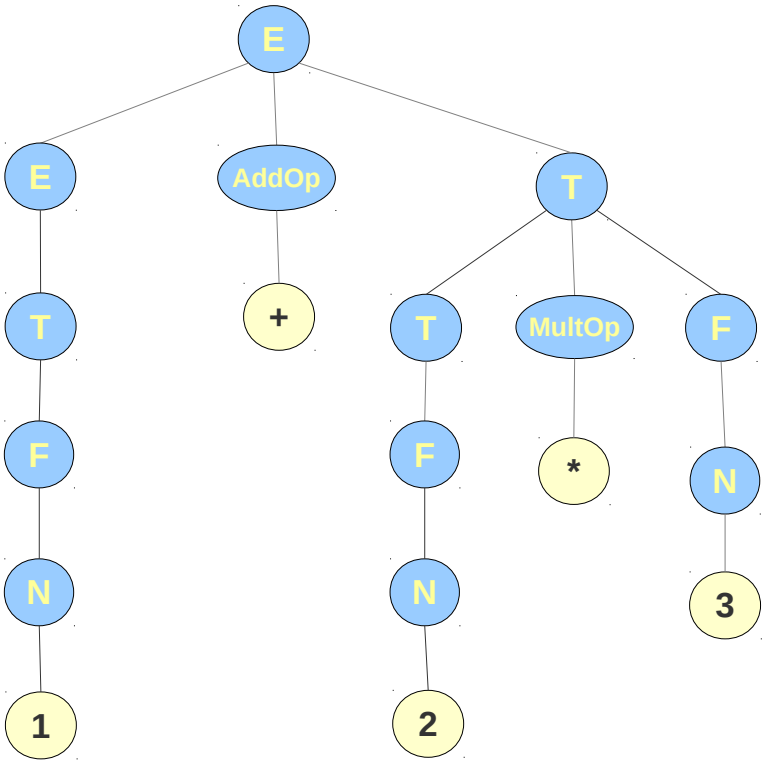
AST: Abstrakter Syntaxbaum / *Abstract Syntax Tree*

Ableitungsbäume sind oft sehr komplex

Sie spiegeln die Komplexität der Grammatik wider

Diese muss komplex sein, damit ein Text eindeutig interpretiert werden kann.

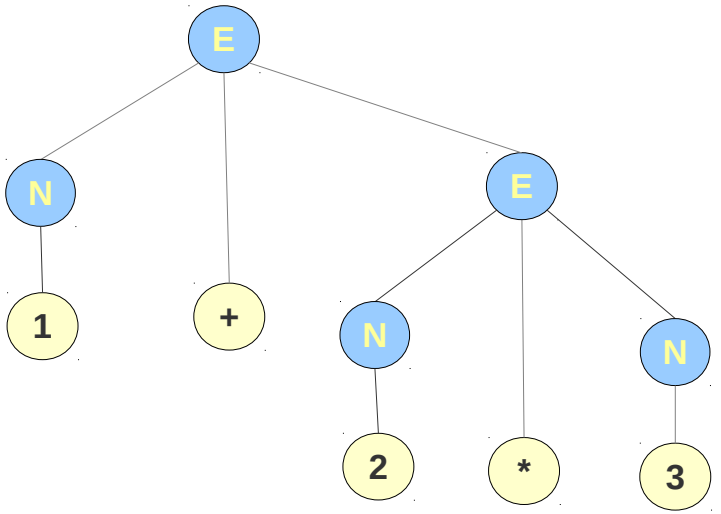
Bäume haben eine klarere Struktur, können darum einfacher sein.



*Ableitungsbaum für 1+2*3*

Abstraktion:

Konzentration auf das Wesentliche



*AST für 1+2*3, enthält die gleiche relevante Information*