

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing




Vorlesung Compilerbau: Syntaktische Analyse - Parsing

Vorlesung des BA-Studiums
Prof. Johann Christoph Freytag, Ph.D.
Institut für Informatik, Humboldt-Universität zu Berlin
SoSe 2018

Handys bitte ausschalten

© Prof. J.C. Freytag, Ph.D. 4.1



Überblick

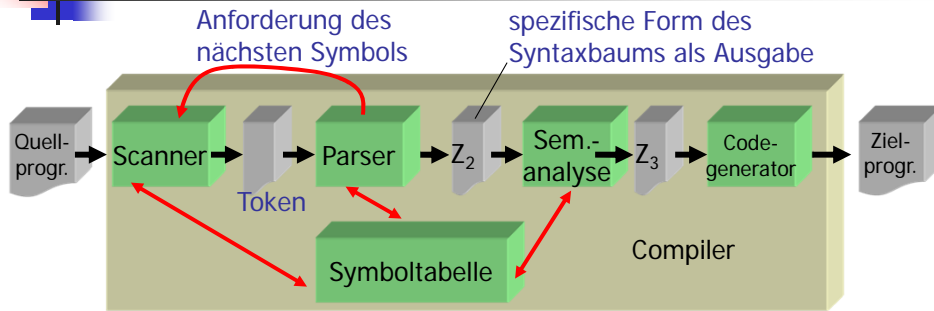
- Einführung
 - Syntaktische Analyse allgemein
- Theoretische Grundlagen
 - Sprachklassen und ihre Erkennung
 - Kurzeinführung Kellerautomat
- Implementationstechniken von Parsern
 - Top-Down/Bottom-Up-Verfahren

© Prof. J.C. Freytag, Ph.D. 4.2

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Die Rolle des Parsers



Aufgaben eines Parsers

- Prüfung, ob Token-Folge des Scanners von der **Grammatik** der Quellsprache erzeugt werden kann (Zusammenfassung von Symbolen zu **syntaktischen Einheiten**)
- Erstellung aussagekräftiger Mitteilungen bei **Syntaxfehlern**
- Fehlerbehandlung, so dass sichere Fortsetzung der Analyse des Restquelltextes möglich ist

© Prof. J.C. Freytag, Ph.D.

4.3

Syntax von Programmiersprachen

- zu jeder Programmiersprache gehören **Regeln**, die festlegen, wie die **syntaktische Struktur** wohlgeformter Programme auszusehen hat.
- Syntax lässt sich durch **kontextfreie Grammatiken** oder mit **BNF** (Backus-Naur-Form) beschreiben.
- Grammatiken (allgemein) sind sowohl für den **Sprachentwurf** als auch für den **Compilerbau** hilfreich.
- Grammatik beschreibt **Syntax exakt**.
- für **gewisse** Grammatikklassen können **automatisch effiziente Parser** abgeleitet werden.
- eine Sprache zu **erweitern** ist einfacher, wenn sie bereits auf Basis einer Grammatik implementiert wurde.

© Prof. J.C. Freytag, Ph.D.

4.4

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Beispiel für eine Grammatik

<Satz>	→ <Subjekt><Prädikat><Objekt>
<Subjekt>	→ <Artikel><Attribut><Substantiv>
<Artikel>	→ ε
<Artikel>	→ der
<Artikel>	→ die
<Artikel>	→ das
<Attribut>	→ ε
<Attribut>	→ <Adjektiv>
<Attribut>	→ <Adjektiv><Attribut>
<Adjektiv>	→ kleine
<Adjektiv>	→ bissige
<Adjektiv>	→ große
<Substantiv>	→ Hund
<Substantiv>	→ Katze
<Prädikat>	→ jagt
<Objekt>	→ <Artikel><Attribut><Substantiv>

Beispiel für Satz

der kleine bissige Hund jagt die große Katze

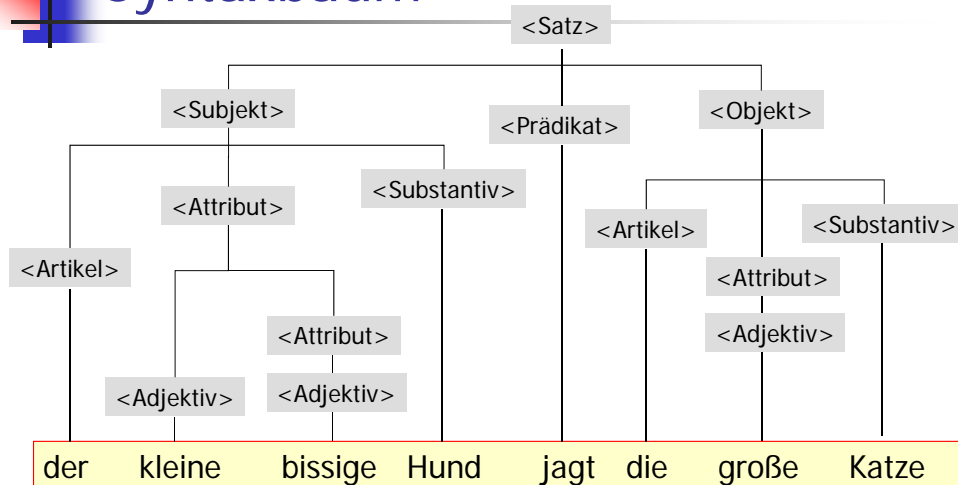
bereits unendliche Sprache

der Hund jagt die kleine kleine kleine... Katze

© Pro

4.5

Syntaxbaum



© Prof. J.C. Freytag, Ph.D.

4.6

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing



Grammatiken (allgemein)

- linke Seite einer Regel (auch: Produktion) enthält Wörter, die sich
 - sowohl aus (evtl. mehreren) Variablen (heißen auch Nichtterminal-Symbole)
 - als auch aus Terminal-Symbolen zusammensetzen
- Anwendung einer Regel bedeutet, dass in dem bislang erzeugten Wort ein Teilwort, das einer linken Regelseite entspricht, durch die rechte Seite der Regel ersetzt wird
- Ableitungsschritte werden solange ausgeführt, bis das entstandene Wort nur noch aus Terminalsymbolen besteht
 - Reihenfolge der Schritte ist nicht eindeutig
- jedes so erzeugte Wort gehört dann zu der von der Grammatik erzeugten oder definierten Sprache

© Prof. J.C. Freytag, Ph.D.

4.7



Syntaktische Einheiten

... in imperativen Sprachen

- Variablen und Ausdrücke,
- Anweisungen und Anweisungsfolgen,
- Deklarationen und Spezifikationen

... in funktionalen Sprachen

- Variablen und Ausdrücke,
- Muster,
- Definitionen und Deklarationen

... in logischen Sprachen

- Variablen und Terme,
- Listen von Termen,
- Ziele und Klauseln

© Prof. J.C. Freytag, Ph.D.

4.8



Vergleich von Scannen und Parsen

Worin besteht der Unterschied?

- Scannen: Zusammenfassen von Zeichen zu Wörtern
- Parsen: Erkennen von Satzstrukturen

Beispiel:

aber Übergänge sind fließend

```
term ::= [a-zA-Z] ([a-zA-Z] | [0-9])*
      | 0 | [1-9] [0-9]*
op   ::= + | - | * | /
expr ::= (term op)* term
```

© Prof. J.C. Freytag, Ph.D.

4.9



Vergleich von Scannen und Parsen (2)

RAs klassifizieren:

- Identifikatoren, Zahlen, Schlüsselwörter
- RAs sind für Token-Erkennung kompakter und einfacher zu verstehen als Grammatiken
- für Token ist Konstruktion effizienterer Scanner möglich, wenn sie aus RAs abgeleitet werden können: DFAs

kontextfreie Grammatiken »zählen« :

- Klammern: (), begin...end, if...then...else
- Struktur von Ausdrücken

syntaktische Analyse ist selbst komplex genug:

- Separierung der lexikalischen Analyse vereinfacht Realisierung eines Compilers

© Prof. J.C. Freytag, Ph.D.

4.10



Backus-Naur-Form (BNF) und Erweiterungen


Beispiel:

1.	<Satz>	::= <Subjekt> <Prädikat> <Objekt>
2.	<Subjekt>	::= <Artikel> <Attribut> <Substantiv>
3.	<Artikel>	::= ε der die das
4.	<Attribut>	::= ε <Adjektiv> <Attribut> <Adjektiv>
5.	<Adjektiv>	::= kleine bissige große
6.	<Substantiv>	::= Hund Katze
7.	<Prädikat>	::= jagt
8.	<Objekt>	::= <Artikel> <Attribut> <Substantiv>

BNF-Notation (es gibt auch andere Formen)

- Nicht-Terminals in spitzen Klammern oder in Großbuchstaben
- Terminals in Typewriter-Font oder unterstrichen oder ...
- Produktionsregeln wie im Beispiel

© Prof. J.C. Freytag, Ph.D. 4.11



Generierung von Parsern

- Grundlage für Syntaxanalysatoren/Parser: **Kellerautomat**
 - englisch "push-down automata (PDA)"
- Kellerautomat kann als Parser
 - **automatisch** generiert werden, wenn Sprache durch kontextfreie Grammatik darstellbar ist
 - Genauer: **deterministische**, kontextfreie Grammatik
 - oder **per Hand realisiert** werden als
 - Kellerautomat oder mittels
 - Umsetzung durch „rekursiven Abstieg“
- **Bemerkung:**
 - Von-Hand-Implementation empfiehlt sich nicht, solange sich die Sprache noch in Entwicklung befindet

© Prof. J.C. Freytag, Ph.D. 4.12




Behandlung von Fehlern

- Herausforderungen für einen Parser
 - Normalfall für einen Compiler
 - Programme sind fehlerhaft
- Fehler, die einfach zu behandeln sind
 - lexikalische Fehler
 - Fehler in der statischen Semantik (z.B. Typfehler)
- Fehler, die „schwieriger“ sind
 - Syntaxfehler (z.B. Klammerstrukturierung)




Behandlung von Fehlern (2)

- Schritte
 - melde und lokalisier den Fehler
 - diagnostiziere den Fehler
 - korrigiere den Fehler (wenn möglich)
 - um zumindest das nächste Token verarbeiten und Endlos-Schleifen in der Analyse abwehren zu können
 - fasse wieder Tritt, um evtl. vorhandene weitere Fehler zu entdecken
 - wenigstens müssen bis zum nächsten Token (z.B. Semikolon) Zeichen »geschluckt« werden
- Bemerkung:
 - effektive Umsetzung ist jedoch schwierig
 - manche Fehler werden erst viel später entdeckt als sie auftreten



... ein wenig über Sprachen und Grammatiken ...

© Prof. J.C. Freytag, Ph.D. 4.15



Definition der Syntax

- **kontextfreie Grammatiken** beschreiben auf natürliche Weise hierarchische Programmstrukturen

if (Ausdruck) Anweisung **else** Anweisung

in C gibt es kein then

<stmt> → if (<expr>) <stmt> else <stmt>

*Regel heißt **Produktion***

gelesen als "kann die Form haben"

- lexikalische Elemente (if, Klammern) heißen **Terminale (Terminalsymbole)**
- Variablen (expr, stmt) heißen **Nichtterminale** (repräsentieren Symbolfolgen)

© Prof. J.C. Freytag, Ph.D. 4.16

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Kontextfreie Grammatik

... ist ein 4-Tupel $G = (V_N, V_T, P, s)$ mit

- V_N : ist die (endliche) Menge der Variablen (Nichtterminalsymbole) der Grammatik
sie werden benutzt, um die Grammatik zu strukturieren
- V_T : ist die (endliche) Menge der Terminalsymbole
in unserem Falle ist sie identisch mit der Token-Menge, die der Scanner zurück gibt
- $P \subseteq V_N \times (V_N \cup V_T)^*$ ist die (endliche) Menge der Produktionsregeln,
sie legen fest, wie Terminal- und Nicht-Terminalsymbole kombiniert werden können, um einen Satz der Sprache zu bilden
jede (Produktions-) Regel darf **nur ein einziges Nicht-Terminalsymbol** auf der linken Seite haben
- $s \in V_N$ ist als ausgezeichnetes Nicht-Terminalsymbol das Startsymbol, aus dem die Sprache $L(G) \subseteq V_T^*$ gebildet wird
- Die Menge $V = V_N \cup V_T$ bildet das Vokabular von G (es gilt: $V_N \cap V_T = \emptyset$)

© Prof. J.C. Freytag, Ph.D.

4.17

Allgemeine Grammatik

... ist ein 4-Tupel $G = (V_N, V_T, P, s)$ mit

- V_N : ist die (endliche) Menge der Variablen (Nichtterminalsymbole) der Grammatik
sie werden benutzt, um die Grammatik zu strukturieren
- V_T : ist die (endliche) Menge der Terminalsymbole
in unserem Falle ist sie identisch mit der Token-Menge, die der Scanner zurück gibt
- $P \subseteq (V_N \cup V_T)^* V_N (V_N \cup V_T)^* \times (V_N \cup V_T)^*$ ist die (endliche) Menge der Produktionsregeln,
sie legen fest, wie Terminal- und Nicht-Terminalsymbole kombiniert werden können, um einen Satz der Sprache zu bilden.
linke Seite einer Produktion enthält Wörter, die sowohl (evtl. mehrere) Variablen als auch Terminal-Symbole enthalten
- $s \in V_N$ ist als ausgezeichnetes Nicht-Terminalsymbol das Startsymbol

© Prof. J.C. Freytag, Ph.D.

4.18

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Allgemeine Grammatik (Unterschied: rot markiert)

... ist ein 4-Tupel $G = (V_N, V_T, P, s)$ mit

- V_N : ist die (endliche) Menge der Variablen (Nichtterminalsymbole) der Grammatik *sie werden benutzt, um die Grammatik zu strukturieren*
- V_T : ist die (endliche) Menge der Terminalsymbole *in unserem Falle ist sie identisch mit der Token-Menge, die der Scanner zurück gibt*
- $P \subseteq (V_N \cup V_T)^* V_N (V_N \cup V_T)^* \times (V_N \cup V_T)^*$ ist die (endliche) Menge der Produktionsregeln, *sie legen fest, wie Terminal- und Nicht-Terminalsymbole kombiniert werden können, um einen Satz der Sprache zu bilden.*
linke Seite einer Produktion enthält Wörter, die sowohl (evtl. mehrere) Variablen als auch Terminal-Symbole enthalten
- $s \in V_N$ ist als ausgezeichnetes Nicht-Terminalsymbol das Startsymbol

© Prof. J.C. Freytag, Ph.D.

4.19

Notation und Terminologie

Symbole:

- $a, b, c, \dots \in V_T$
- $A, B, C, \dots \in V_N$
- $U, V, W, \dots \in V \quad (= V_T \cup V_N)$
- $\alpha, \beta, \gamma, \dots \in V^*$
- $u, v, w, \dots \in V_T^*$

Notation:

- Falls $A \rightarrow \gamma$, dann ist $\alpha A \beta \Rightarrow \alpha \gamma \beta$ eine "Ein-Schritt"-Ableitung, die $A \rightarrow \gamma$ benutzt
- \Rightarrow^* und \Rightarrow^+ bezeichnen Ableitungen mit ≥ 0 und ≥ 1 Schritten
- Falls $S \Rightarrow^* \beta$, so heißt β eine Satzform (engl. *sentential form*) der Grammatik G
- $L(G) = \{w \in V_T^* \mid S \Rightarrow^* w\}$, $w \in L(G)$ wird als Wort oder Satz bezeichnet
- Bemerkung: $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_T^*$

© Prof. J.C. Freytag, Ph.D.

4.20

Grammatiktypen nach Noam Chomsky

- **Typ-0**
Phrasenstrukturgrammatik: **keinerlei Einschränkung**,
jede Grammatik ist vom Typ 0
- **Typ-1**
kontextsensitive Grammatik: falls zusätzlich für alle Regeln $w_1 \rightarrow w_2$ in P gilt:
 $|w_1| \leq |w_2|$
- **Typ-2**
kontextfreie Grammatik: falls zusätzlich für alle Regeln $w_1 \rightarrow w_2$ in P gilt,
dass **w₁ eine einzelne Variable ist** ($w_1 \in V_N$)
- **Typ-3**
reguläre Grammatik: falls zusätzlich gilt: dass $w_2 \in (V_T \cup V_T V_N)$
d.h. rechte Seiten sind entweder ein einzelnes Terminalsymbol oder ein Terminalsymbol gefolgt von einem Nicht-Terminalsymbol

© Prof. J.C. Freytag, Ph.D.

4.21

Kontextfrei gegenüber Kontextsensitiv

- **bei kontextfreier Regel: $A \rightarrow x$**
kann die Variable A – unabhängig vom Kontext, in dem A steht –
bedingungslos durch x ersetzt werden
- **bei kontextsensitiver Regel: $Av \rightarrow uxv$**
kann die Variable A durch x nur ersetzt werden, wenn die
Variable A im Kontext zwischen u und v steht
ebenfalls: $uAv \rightarrow u'xv'$

© Prof. J.C. Freytag, Ph.D.

4.22



Die ε - Sonderregelung

unerwünschte Eigenschaft von Typ 1,2,3 Sprachen: $\varepsilon \notin L(G)$

- wegen $|w_1| \leq |w_2|$ kann das leere Wort ε nicht abgeleitet werden

Sonderregelung, falls $\varepsilon \in L(G)$ erwünscht:

- Regel erlaubt: $s \rightarrow \varepsilon$
- s darf dann aber nicht auf der rechten Seite vorkommen

weitere Sonderregelung (nur) für kontextfreie Sprachen

Regel erlaubt: $A \rightarrow \varepsilon$ ($A \neq s$)

- Bedeutung für Optionalität von Alternativen



Wortproblem

Frage

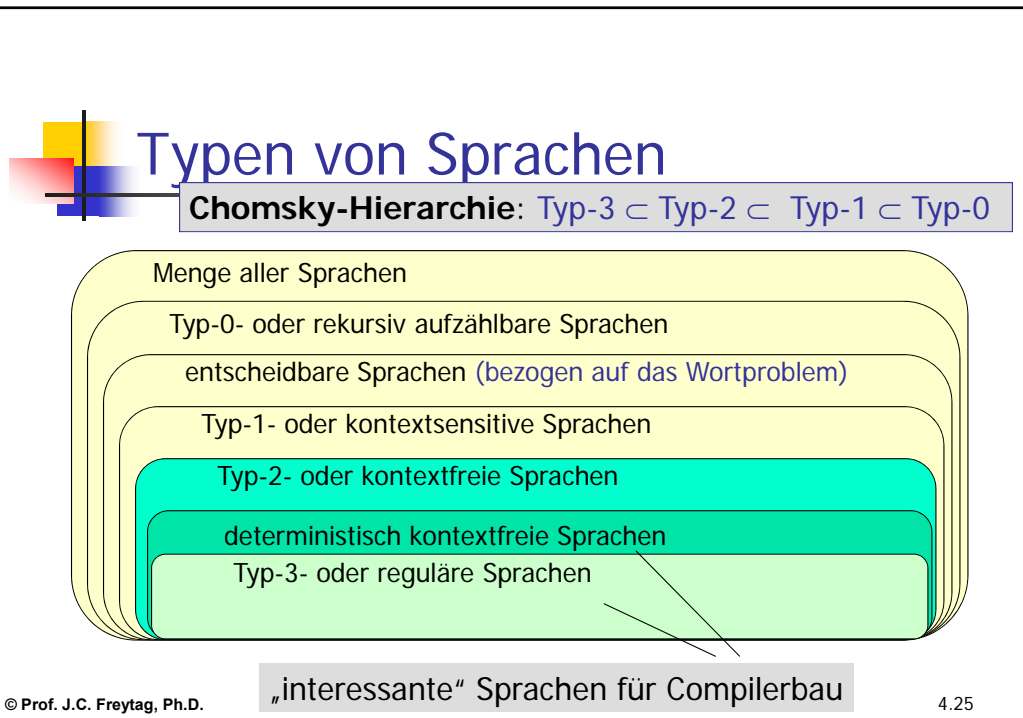
- Gibt es einen Algorithmus, der bei Eingabe einer Grammatik $G = (V_N, V_T, P, s)$ und eines Wortes $x \in V_T^*$ in endlicher Zeit entscheidet, ob $x \in L(G)$ oder $x \notin L(G)$?

Antwort

- erfolgt in Abhängigkeit der Eigenschaften von G
(und motiviert damit die Klasseneinteilung der Grammatiken)

Vorlesung Compilerbau (PI 3) (SoSe 2018)


Teil 4: Syntaktische Analyse - Parsing



Grammatiktypen und das Wortproblem

- **Typ-0**
Wortproblem ist nicht entscheidbar (auf der linken Seite können auch bereits analysierte Terminale stehen, aus denen aber weiter abgeleitet werden kann)
- **Typ-1**
kontextsensitive Grammatik: falls zusätzlich für alle Regeln $w_1 \rightarrow w_2$ in P gilt:
 $|w_1| \leq |w_2|$
Bei der sequentiellen Betrachtung alternativer Ableitungen interessiert man sich nur für die mit geringsten Expansion
Wortproblem ist entscheidbar, läßt sich also prinzipiell analysieren (exponentielle Komplexität)
- **Typ-2**
kontextfreie Grammatik: falls zusätzlich für alle Regeln $w_1 \rightarrow w_2$ in P gilt,
dass w_1 eine einzelne Variable ist ($w_1 \in V_N$)
Wortproblem ist entscheidbar (polynomiale Komplexität)
(Deterministisch kontextfreie Sprachen: lineare Komplexität)
- **Typ-3**
reguläre Grammatik: falls zusätzlich gilt: dass $w_2 \in (V_T \cup V_T V_N)$
Wortproblem effizient entscheidbar (lineare Komplexität);
Grammatik aber sehr beschränkt

© Prof. J.C. Freytag, Ph.D. 4.26




Beispiele für Sprachen (aus jeweiliger Differenzmenge)

- $L = \{a^n b^n \mid n \geq 1\}$ ist vom Typ 2, aber nicht vom Typ 3
- $L' = \{a^n b^n c^n \mid n \geq 1\}$ ist vom Typ 1, aber nicht vom Typ 2
- $L'' = H$ (Halteproblem) ist vom Typ 0, aber nicht vom Typ 1
- es gibt entscheidbare Sprachen, die nicht vom Typ 1 sind

Alle Sprachen vom Typ 1,2,3 sind **entscheidbar**, d.h. es gibt einen Algorithmus, der bei Eingabe von G und w in endlicher Zeit feststellt, ob $w \in L(G)$ ist

© Prof. J.C. Freytag, Ph.D. 4.27



Sprachen in der Praxis

... werfen jedoch Fragestellungen

- **kontextsensitiver** oder sogar **Typ-0-Art** auf

dennoch arbeitet man wegen schwieriger algorithmischer Handhabung von Typ-0,1-Sprachen lieber mit kontextfreien sprachen (Grammatiken)

Problemlösung

- notwendige Behandlung von unterschiedlichen Kontextbedingungen und Sonderfällen erfolgt durch **nicht-grammatikalische Zusatzalgorithmen**

z.B. in C: Typverträglichkeit, korrekte Parameteranzahl beim Funktionsruf, ausschließliche Verwendung vorab deklarerter Objekte lassen sich nicht durch kontextfreie Grammatiken (Syntaxdiagramme, BNF) beschreiben

© Prof. J.C. Freytag, Ph.D. 4.28



Backus-Naur-Form (BNF)

$A \rightarrow \beta_1$
 $A \rightarrow \beta_2$
.....
 $A \rightarrow \beta_n$



$A ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Formalismus zur kompakten
Notation von kontextfreien
Grammatiken

eingeführt mit Algol-60

Erweiterung: **EBNF**

$B ::= \alpha[\beta]\gamma$, β kann, muss aber nicht eingefügt werden

$C ::= \alpha\{\beta\}\gamma$, β kann beliebig oft (auch null mal) eingefügt werden



Zusammenhänge: Sprachen und Automaten

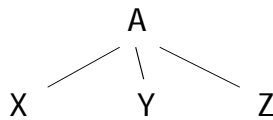
- jede durch **endliche Automaten** erkennbare Sprache ist regulär, also Typ-3 (s. Scanner)
- jede von einem **NFA** akzeptierte Sprache ist auch durch einen **DFA** akzeptierbar
- für jede **reguläre Grammatik** G gibt es einen NFA M mit $L(G) = L(M)$
- eine Sprache L ist **kontextfrei** genau dann, wenn L von einem **nichtdeterministischen Kellerautomaten** erkannt wird (*kommt später*)
- eine Sprache L ist **deterministisch kontextfrei** genau dann, wenn L von einem **deterministischen Kellerautomaten** erkannt wird (*kommt später*)
- die von **linear beschränkten, nichtdeterministischen Turing-Maschinen** akzeptierten Sprachen sind genau die **kontextsensitiven (Typ-1-) Sprachen** (*kommt in der TI*)

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Syntaxbaum (Parse-Baum)

- einer Ableitung eines Wortes in einer Typ-2 oder 3-Grammatik G kann man einen **Syntaxbaum** (oder Ableitungsbaum) zuordnen
- ein **Syntaxbaum** stellt grafisch dar, wie aus einem **Start-Symbol** einer Grammatik ein **Wort der Sprache** hergeleitet wird
- wenn es für ein Nichtterminalsymbol A eine Produktion $A \rightarrow XYZ$ gibt, könnte ein Syntaxbaum
 - einen mit A markierten inneren Knoten besitzen,
 - der wiederum 3 Nachfolger mit den Marken X, Y, Z besitzt

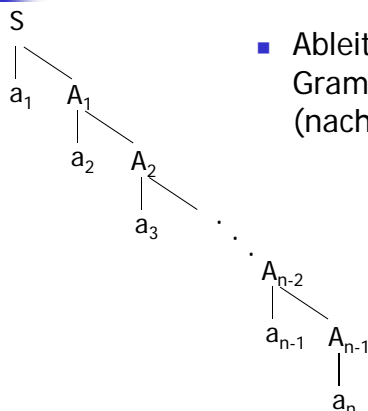


- ein Syntaxbaum stellt grafisch dar, wie aus einem **Start-Symbol** einer Grammatik ein **Wort der Sprache** hergeleitet wird

© Prof. J.C. Freytag, Ph.D.

4.31

Syntaxbaum (2)



- Ableitungsbäume bei regulären Grammatiken sind immer entartet (nach rechts geneigte lineare Ketten)

© Prof. J.C. Freytag, Ph.D.

4.32

Syntaxbaum einer kontextfreien Grammatik

... ist ein **Baum mit folgenden Eigenschaften:**

1. Wurzel ist mit Startsymbol markiert
2. jedes Blatt ist entweder mit einem **Terminal** oder mit ϵ markiert
3. jeder **innere Knoten** ist mit einem **Nichtterminal** markiert
4. **wenn** ein innerer Knoten mit dem Nichtterminal A markiert ist und die Kinder dieses Knoten von links nach rechts die Marken $N_1, N_2, \dots, N_k \in (V_N \cup V_T)$ tragen, dann ist $A \rightarrow N_1 N_2 \dots N_k$ eine Produktion in P
 - Hierbei stehen N_1, N_2, \dots, N_k jeweils für ein Symbol, das entweder Terminal oder Nichtterminal ist.
 - Im besonderen Fall einer Produktion $A \rightarrow \epsilon$ hat ein mit A markierter Knoten evtl. nur einen einzigen, mit ϵ markierten Nachfolger.

© Prof. J.C. Freytag, Ph.D.

4.33

Beispiel: einfache kontextfreie Grammatik

- Grammatik für einfache Ausdrücke

```
1. <goal> ::= <expr>
2. <expr> ::= <expr> <op> <expr>
3.           | num
4.           | id
5. <op>    ::= *
6.           | +
7.           | -
8.           | /
```

- Beispielwort: $x + 2 * y$

© Prof. J.C. Freytag, Ph.D.

4.34

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Beispiel: Ableitungen

- Beispiel für Ableitungen einer kontextfreien Grammatik :

```
<goal> ⇒ <expr>
        ⇒ <expr> <op> <expr>
        ⇒ <expr> <op> <expr> <op> <expr>
        ⇒ <id, x> <op> <expr> <op> <expr>
        ⇒ <id, x> + <expr> <op> <expr>
        ⇒ <id, x> + <num, 2> <op> <expr>
        ⇒ <id, x> + <num, 2> * <expr>
        ⇒ <id, x> + <num, 2> * <id, y>
```

Linksableitung:

immer die weitesten links stehende Variable wurde durch Regelanwendung ersetzt,

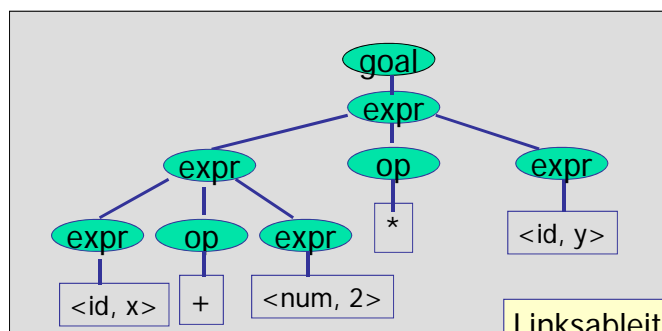
ABER: nicht eindeutig, welche Regel dabei zum Einsatz kommen muss Grammatik hat dies nicht festgelegt

- es wurde das Wort $x+2*y$ abgeleitet
- Ableitung oder Parse: $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$
- Prozess der Konstruktion der Ableitung wird als »Parsing« bezeichnet

© Prof. J.C. Freytag, Ph.D.

4.35

Beispiel: Ableitungsbaum (oder auch Syntaxbaum)



Baumdurchlauf

berechnet: $(x+2)*y$

Linksableitung:

immer das weitesten links stehende Nicht-Terminalsymbol wurde durch Regelanwendung ersetzt

© Prof. J.C. Freytag, Ph.D.

4.36

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Ableitungen

- **Frage:** In jedem Schritt wird ein Nicht-Terminal zur Ersetzung gewählt, aber: welches?
- Wahl bestimmt unterschiedliche Ableitungen
 - **(Linkeste) Linksableitung** (*left-most derivation*)
das am weitesten links stehende Nicht-Terminal wird zur Ersetzung ausgewählt
 - **(Rechteste) Rechtsableitung** (*right-most derivation*)
das am weitesten rechts stehende Nicht-Terminal wird zur Ersetzung gewählt
- **Frage:** Sind die entstehenden Syntaxbäume trotzdem gleich?
- **Frage:** Habe ich evtl. mehrere Ableitungsmöglichkeiten bei Ersetzung eines Nicht-Terminals?
- **Frage:** Sind die entstehenden Syntaxbäume immer noch gleich?

© Prof. J.C. Freytag, Ph.D.

4.37

Rechtsableitung

- Zeichenkette: $x+2*y$

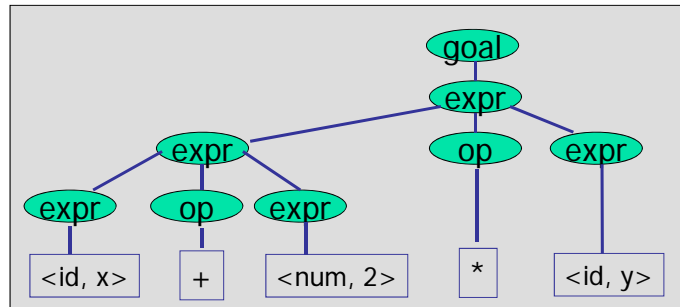
```
<goal>  ⇒ <expr>
        ⇒ <expr> <op> <expr>
        ⇒ <expr> <op> <id,y>
        ⇒ <expr> * <id,y>
        ⇒ <expr> <op> <expr> * <id,y>
        ⇒ <id,x> <op> <num,2> * <id,y>
        ⇒ <expr> + <num,2> * <id,y>
        ⇒ <id,x> + <num,2> * <id,y>
```

- wiederum Ableitung: $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$

© Prof. J.C. Freytag, Ph.D.

4.38

Syntaxbaum & Rechtsableitung



Baumdurchlauf **berechnet**: $(x+2)*y$

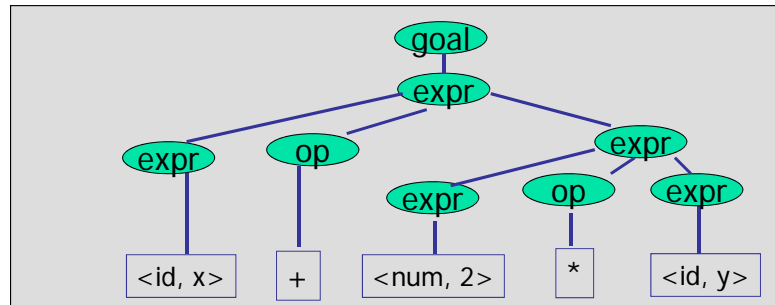
Alternative Links-Ableitung

- Zeichenkette: $x+2*y$

```
<goal>  =>  <expr>
          =>  <expr> <op> <expr>
          =>  <id,x> <op> <expr>
          =>  <id,x> + <expr>
          =>  <id,x> + <expr> <op> <expr>
          =>  <id,x> + <num,2> <op> <expr>
          =>  <id,x> + <num,2> * <expr>
          =>  <id,x> + <num,2> * <id,y>
```

- wiederum Ableitung: $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$

Syntaxbaum nach alternativer Linksableitung



Baumdurchlauf **berechnet**: $x + (2 * y)$

© Prof. J.C. Freytag, Ph.D.

4.41

Präzedenzen

Problem bisheriger Ableitungen

- keine Berücksichtigung der Präzedenzen oder implizite Reihenfolge der Berechnung (nicht entscheidend, ob Rechts-oder Linksableitung)
- zusätzliche Symbole notwendig, um dies zwingend zu berücksichtigen

neue Grammatik

1. $\langle \text{goal} \rangle ::= \langle \text{expr} \rangle$
2. $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$
3. | $\langle \text{expr} \rangle - \langle \text{term} \rangle$
4. | $\langle \text{term} \rangle$
5. $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle$
6. | $\langle \text{term} \rangle / \langle \text{factor} \rangle$
7. | $\langle \text{factor} \rangle$
8. $\langle \text{factor} \rangle ::= \text{num}$
9. | id

■ *term* muss von *expression* abgeleitet werden, um korrekten Baum zu erzeugen

© Prof. J.C. Freytag, Ph.D.

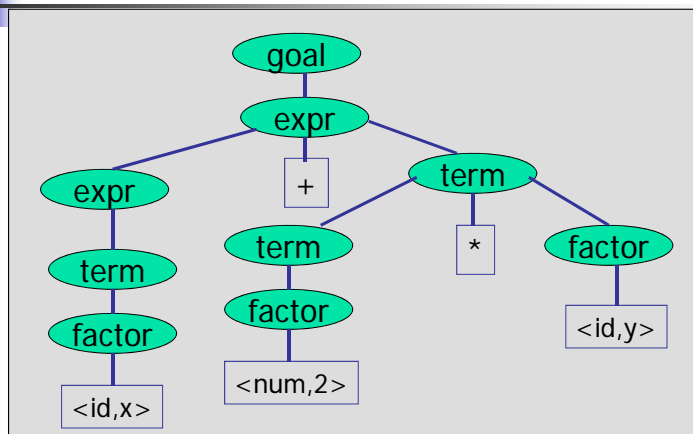
4.42

Präzedenzen (2)

Ableitung für $x+2*y$ mit „korrektem“ Baum

```
<goal>  => <expr>
          => <expr> + <term>
          => <expr> + <term> * <factor>
          => <expr> + <term> * <id,y>
          => <expr> + <num,2> * <id,y>
          => <term> + <num,2> * <id,y>
          => <factor> + <num,2> * <id,y>
          => <id,x> + <num,2> * <id,y>
```

Präzedenzen (3)



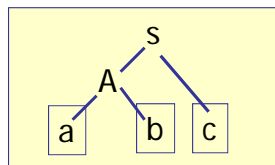
Baumdurchlauf berechnet: $x + (2 * y)$

Mehrdeutigkeit von Grammatiken

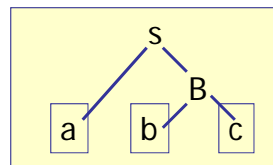
- **Def.:** Erlaubt eine Grammatik mehr als eine Ableitung für eine syntaktische Einheit, dann ist die Grammatik **mehrdeutig**.

- **Beispiel**

```
<s> ::= a<B>
      | <A>c
<A> ::= ab
<B> ::= bc
```



Wort abc



mit: $\langle s \rangle \rightarrow abc$ wird die Grammatik eindeutig

© Prof. J.C. Freytag, Ph.D.

4.45

Mehrdeutigkeit von Grammatiken (2)

- nicht immer lassen sich Mehrdeutigkeiten durch Grammatik-Restrukturierungen beseitigen
- eine Sprache L heißt **inhärent mehrdeutig**, wenn jede Grammatik G mit $L(G)=L$ mehrdeutig ist
- im Allg. kann man algorithmisch nicht feststellen, ob eine Grammatik (bzw. Sprache) inhärent mehrdeutig ist
- **Beispiel** (einer inhärent mehrdeutigen Sprache):
$$L = \{a^i b^j c^k \mid i=j \text{ oder } j=k\}$$

© Prof. J.C. Freytag, Ph.D.

4.46

Mehrdeutigkeit von Grammatiken (3)

Beispiel

```
<stmt> ::= if <expr> then <stmt>  
        | if <expr> then <stmt> else <stmt>  
        | other stmts
```

- für die syntaktische Einheit:
if E_1 then if E_2 then S_1 else S_2
gibt es mehr als eine Ableitung
- diese Mehrdeutigkeit ist grammatikalisch begründet
sie ist eine **kontextfreie** Mehrdeutigkeit

Mehrdeutigkeit von Grammatiken (4)

Möglichkeit der Vermeidung (aber schlecht lesbar):

- Restrukturierung der Grammatik

Beispiel:

```
<stmt>      ::= <matched>  
              | <unmatched>  
  
<matched>   ::= if <expr> then <matched> else <matched>  
              | other stmts  
  
<unmatched> ::= if <expr> then <stmt>  
              | if <expr> then <matched> else <unmatched>
```

- generiert die gleiche Sprache mit der offensichtlichen Regel

Ordne jedes **else** dem am nächsten stehenden (noch) nicht zugeordneten **then** zu

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

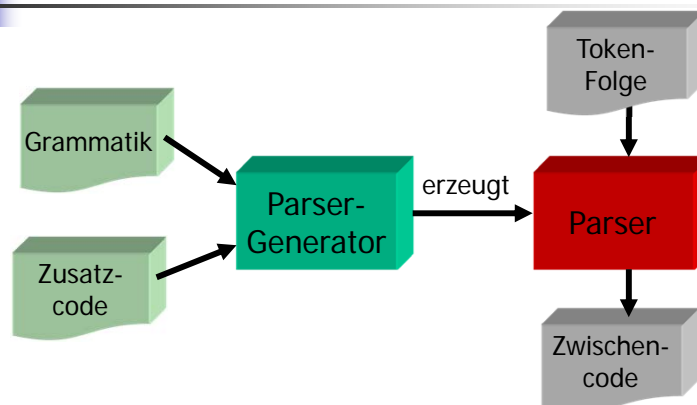
Mehrdeutigkeit von Grammatiken (5)

- ist häufig auf unklare/verwirrende Spezifikation der Regeln einer kontextfreien Grammatik zurückzuführen
- kann durch Überladen verursacht werden
 $A = f(17)$
f kann eine Funktion oder ein Array sein (in Algol-ähnlichen Sprachen)
- Auflösung der Mehrdeutigkeit durch *Kontextwissen*
 - Zugriff auf Deklarationen über Symboltabelle (sinnvoll für Compilerbau...)
 - nicht kontextfrei
 - Typisierungsproblem
- statt durch Parsing werden andere Möglichkeiten der Auflösung behandelt (später)

© Prof. J.C. Freytag, Ph.D.

4.49

Konstruktion eines Parsers



Ziel: Automatische Konstruktion eines flexiblen Parsers

© Prof. J.C. Freytag, Ph.D.

4.50

Vorlesung Compilerbau (PI 3) (SoSe 2018)

Teil 4: Syntaktische Analyse - Parsing

Fragen

A cartoon illustration of a man with a mustache, wearing a pink polo shirt, sitting on the ground with his hands clasped in a questioning or confused pose. Three blue question marks '???' float above his head. He is surrounded by light blue motion lines, suggesting he has just arrived or is in a state of surprise.

© Prof. J.C. Freytag, Ph.D.

4.51