

Compilerbau

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Kapitel 4

Syntaktische Analyse (Parser)

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
 - Einführung
 - Top-Down Syntax-Analyse
 - Bottom-Up Syntax-Analyse
 - Praktische Umsetzung: BISON
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Aufgabe der Syntaktischen Analyse



Syntaktische Analyse (*Parser*):

- Sei G Grammatik der Quellsprache
- Entscheidung, ob Tokenfolge aus G ableitbar ist.
- Syntaxbaum: Baumförmige Darstellung des Codes anhand während Ableitung benutzter Regeln aus G
- Fehlerbehandlung

Kontextfreie Grammatiken (1)

- *Reguläre Ausdrücke*: Spezifikation der lexikalischen Einheiten
- ☞ *Grammatiken*: Spezifikation der Sprach-Syntax
- I.a.: keine beliebigen Grammatiken, sondern:

Definition (*Kontextfreie Grammatik*):

Eine *kontextfreie Grammatik* ist ein Tupel $G = (V_N, V_T, P, S)$ mit

- V_N : Menge der **Nichtterminale**
- V_T : Menge der **Terminale**
- $P \subseteq V_N \times (V_N \cup V_T)^*$: Menge der **Produktionsregeln**
- $S \in V_N$: **Startsymbol**

Nichtterminal- und Terminal-Mengen sind disjunkt: $V_N \cap V_T = \emptyset$

Kontextfreie Grammatiken (2)

- *Terminale*: stehen für konkrete Token, die in der zu analysierenden Programmiersprache tatsächlich auftreten. (z. B. **if**, **else**, **begin**, **end**, ...)
- *Nichtterminale*: stehen für Mengen von Wörtern der Programmiersprache, die die Grammatik über ihr Regelwerk produziert.

Kontextfreie Grammatiken (3)

Beispiel: Grammatik G_1

$$\begin{array}{ll}
 S \rightarrow \text{id} := E & E \rightarrow \text{id} \\
 & E \rightarrow \text{num} \\
 & E \rightarrow E * E \\
 & E \rightarrow E + E \\
 & E \rightarrow E - E
 \end{array}$$

- *Terminale:* `id := num * + -`
- *Nichtterminale:* `S, E`
- *Startsymbol:* Wenn nicht explizit ausgewiesen, das Symbol auf der linken Seite der ersten Regel.
Hier: `S`
- Linke Seiten von Regeln dürfen nur ein Nichtterminal enthalten.

Kontextfreie Grammatiken (4)

Beispiel: Grammatik G_1

$$\begin{aligned}
 S &\rightarrow \text{id} := E & E &\rightarrow \text{id} \\
 & & E &\rightarrow \text{num} \\
 & & E &\rightarrow E * E \\
 & & E &\rightarrow E + E \\
 & & E &\rightarrow E - E
 \end{aligned}$$

- Ursprünglicher Quellcode vor lexikalischer Analyse:
b := c + d * 5
- Tokenfolge nach lexikalischer Analyse:
id := id + id * num
- ☞ Tokenfolge ist aus obiger Grammatik G_1 ableitbar.

Ableitungen (1)

Beispiel:

`id := id + id * num`

S →

`id := E` →

`id := E + E` →

`id := E + E * E` →

`id := id + E * E` →

`id := id + id * E` →

`id := id + id * num`

Ableitungen (2)

- Es wird mit dem Startsymbol der Grammatik begonnen.
- Nacheinander wird ein Nichtterminal durch die rechte Seite einer Regel ersetzt.
- Enthält ein Satz ausschließlich Terminale, so ist dieser Satz aus G ableitbar.

- *Linksableitung*: Es wird immer das äußerst linke Nichtterminal ersetzt.
- *Rechtsableitung*: analog

- ☞ Beispiel von [Folie 9](#) ist weder Links- noch Rechtsableitung.

Syntaxbäume (1)

Definition (*Syntaxbaum*):

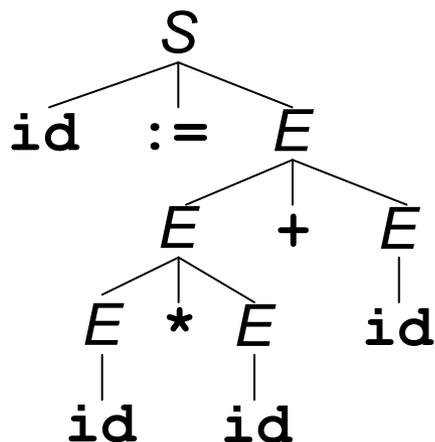
Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik, $x \in V_T^*$ eine Tokenfolge. B sei ein geordneter Baum, d.h. die Ausgangskanten jedes Knotens sind geordnet. Die Blätter von B sind markiert mit Symbolen aus $V_T \cup \{ \varepsilon \}$, innere Knoten mit Symbolen aus V_N . B heißt *Syntaxbaum* für x , wenn gilt:

- Für einen inneren Knoten markiert mit $A \in V_N$ sind seine Kinder von links nach rechts markiert mit $N_1, N_2, \dots, N_k \in V_N \cup V_T$ und P enthält eine Regel $A \rightarrow N_1 N_2 \dots N_k$, oder A 's einziges Kind ist markiert mit ε und $A \rightarrow \varepsilon$ ist eine Regel aus P .
 - Die Wurzel von B ist markiert mit dem Startsymbol S .
 - Die Symbole der Blätter ergeben von links nach rechts gelesen x .
- ☞ Im folgenden: Der *Saum* von B = die Symbole der Blätter.

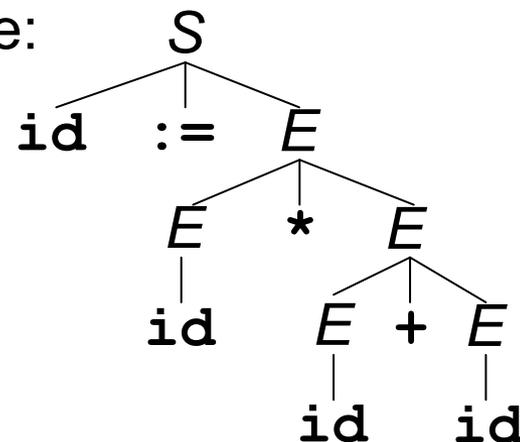
Syntaxbäume (2)

Beispiel:

Ein Syntaxbaum für die Tokenfolge $id := id * id + id$ und G_1 ist:



Aber: Dies ist auch ein gültiger Syntaxbaum für die gleiche Folge:



☞ G_1 ist *mehrdeutig*:

Es existiert eine Tokenfolge, die über mehr als einen Syntaxbaum aus G_1 ableitbar ist.

Mehrdeutige / Eindeutige Grammatiken

Problem:

- Mehrdeutige Grammatiken sind zur Beschreibung von Programmiersprachen ungeeignet: Für einen gültigen Satz dieser Programmiersprache gäbe es mehrere gültige Bedeutungen!
- Beispiel: Wie ist `id := id * id + id` zu interpretieren? Als `id := (id * id) + id` oder als `id := id * (id + id)`? Naheliegend wäre erste Variante, aber G_1 lässt beides zu.
- ☞ Ein Programmierer könnte nicht eindeutig beschreiben, welche Bedeutung in seinem Programm gemeint ist.

Ziel:

Eliminierung von Mehrdeutigkeit durch Grammatik-Transformation, d.h. Herstellen von *Eindeutigkeit*.

Übersicht über Kapitel 4 (1)

Ausgangspunkt des restlichen Kapitels:

- Eindeutige kontextfreie Grammatik G für eine Programmiersprache

Erster Ansatz zur syntaktischen Analyse:

- Beispiel-Ableitung von [Folie 9](#) begann bei Startsymbol von G und hat dieses sukzessive „verfeinert“.
- 👉 Erzeugung des Syntaxbaums von der Wurzel zu den Blättern
- 👉 *Top-Down Syntax-Analyse*

Übersicht über Kapitel 4 (2)

Top-Down Syntax-Analyse:

- Für „einfache“ Grammatiken G ist top-down Ansatz zur Syntax-Analyse sehr nützlich
- Per-Hand Implementierung eines Parsers für „einfache“ Grammatiken leicht möglich
- Verfahren des *rekursiven Abstiegs* (*recursive descent*)
- „Einfache“ Grammatiken: LL(1)
(Left-to-right parse, Leftmost-derivation, 1-symbol look-ahead)

Problem der Top-Down Analyse:

- Parser muss „vorhersagen“, welche Produktionsregel in jeweiligem Stand des Ableitungsprozesses zu verwenden ist.

Übersicht über Kapitel 4 (3)

Bottom-Up Syntax-Analyse:

- Erzeugung des Syntaxbaums von den Blättern zu der Wurzel
- Parsen „komplexer“ Grammatiken möglich
- „Komplexe“ Grammatiken: LR(1)
(Left-to-right parse, Rightmost-derivation, 1-symbol look-ahead)
- Maschinenmodell, das solche Grammatiken akzeptiert:
(deterministischer) Kellerautomat

Automatische Generierung von Parsern:

- Standard-Werkzeug BISON für LR(1)-Grammatiken

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
 - Einführung
 - Top-Down Syntax-Analyse
 - Bottom-Up Syntax-Analyse
 - Praktische Umsetzung: BISON
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Top-Down Analyse

Grundlegender Algorithmus:

- Erzeuge mit Startsymbol S markierten Wurzel-Knoten des Syntaxbaums B
- Wiederhole, bis der Saum von B gleich der Eingabe ist:
 1. Für ein mit Nichtterminal $N \in V_N$ markiertes Blatt v wähle eine Produktionsregel $p \in P$, die N auf linker Seite enthält.
 2. Für jedes Symbol auf der rechten Seite von p : Erzeuge einen Kind-Knoten für v .
 3. Entspricht aktuelles Wort der Eingabe nicht mehr dem Saum von B : Backtracking bis zu der Stelle, wo man in Schritt 1 einen unpassende Regel gewählt hat!

Hauptproblem: Wahl der passenden Regel in Schritt 1!

Beispiel (1)

Grammatik G_2

$$\begin{array}{llll}
 S \rightarrow \text{id} := E & E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\
 & E \rightarrow E - T & T \rightarrow F & F \rightarrow \text{num} \\
 & E \rightarrow T & & F \rightarrow (E)
 \end{array}$$

- Eingabe: `id := id - num * id`
- Im folgenden:
 - Versuch der Linksableitung.
 - Dumme Wahl der Regeln, die zu Problemen führt.
 - Aktuelle Position im Eingabe-Satz ist mit \uparrow markiert.

Beispiel (2)

Eingabe: `id := id - num * id`

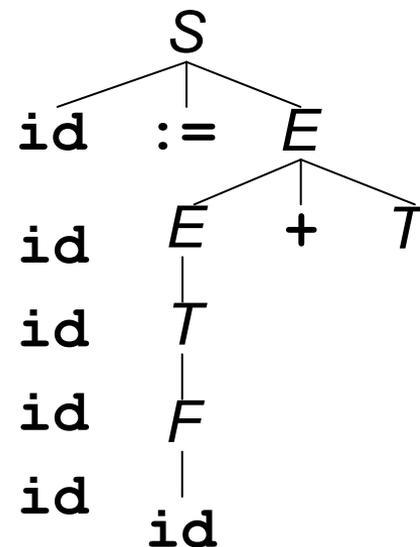
Saum

1. S
2. `id := E`
3. `id := E`
4. `id := E`
5. `id := E + T`
6. `id := T + T`
7. `id := F + T`
8. `id := id + T`
9. `id := id + T`



Eingabe

- ↑`id := id - num * id`
- ↑`id := id - num * id`
- `id` ↑ `:= id - num * id`
- `id :=` ↑`id - num * id`
- `id := id` ↑`- num * id`



+ (Saum) und
- (Eingabe)
passen nicht:
*Zurück zu
Schritt 4!*

Beispiel (3)

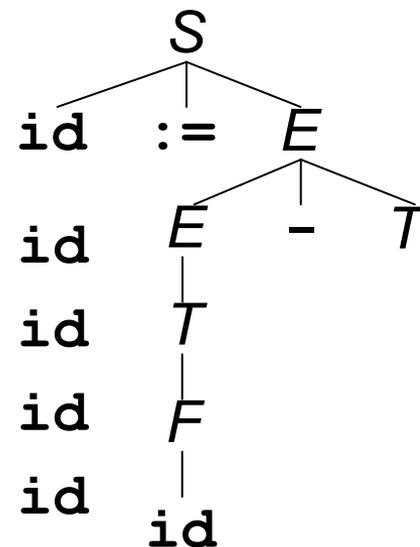
Eingabe: `id := id - num * id`

Saum

1. S
2. `id := E`
3. `id := E`
4. `id := E`
5. `id := E - T`
6. `id := T - T`
7. `id := F - T`
8. `id := id - T`
9. `id := id - T`

Eingabe

1. `↑id := id - num * id`
2. `↑id := id - num * id`
3. `id ↑ := id - num * id`
4. `id := ↑id - num * id`
5. `id := ↑id - num * id`
6. `id := ↑id - num * id`
7. `id := ↑id - num * id`
8. `id := ↑id - num * id`
9. `id := id ↑ - num * id`



- (Saum) und
 - (Eingabe)
 passen:
*Weiter mit
 Eingabe!*

Beispiel (4)

Eingabe: `id := id - num * id`

Saum

10. `id := id - T`

11. `id := id - F`

12. `id := id - num`

13. `id := id - num` 

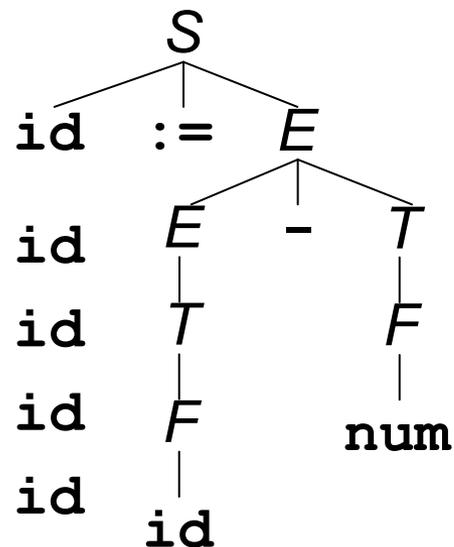
Eingabe

`id := id - ↑num * id`

`id := id - ↑num * id`

`id := id - ↑num * id`

`id := id - num ↑* id`

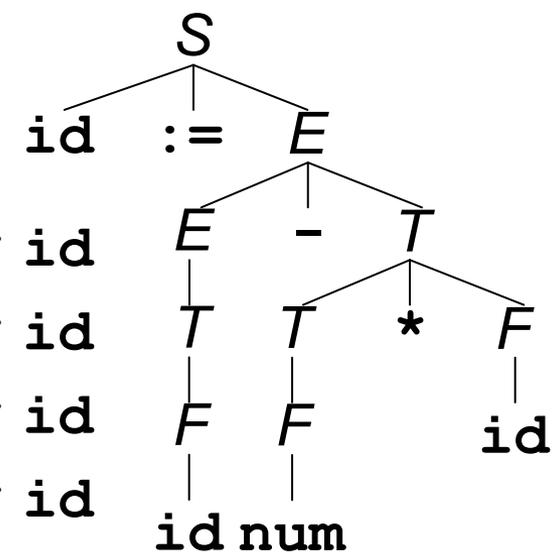


num aus Saum und **num** aus Eingabe passen, aber es gibt keine weiteren Nichtterminale für restliche Eingabe: *Zurück zu Schritt 10!*

Beispiel (5)

Eingabe: `id := id - num * id`

	Saum	Eingabe
10.	<code>id := id - <u>T</u></code>	<code>id := id - ↑num * id</code>
11.	<code>id := id - <u>T</u> * F</code>	<code>id := id - ↑num * id</code>
12.	<code>id := id - <u>F</u> * F</code>	<code>id := id - ↑num * id</code>
13.	<code>id := id - num * F</code>	<code>id := id - ↑num * id</code>
14.	<code>id := id - num * F</code>	<code>id := id - num ↑* id</code>
15.	<code>id := id - num * <u>F</u></code>	<code>id := id - num * ↑id</code>
16.	<code>id := id - num * id</code>	<code>id := id - num * ↑id</code>
17.	<code>id := id - num * id</code>	<code>id := id - num * id↑</code>



Eingabe passt:
Fertig!

Viel schlimmeres Beispiel: Endlos-Parser!

Gleiche Grammatik G_2 , gleiche Eingabe: $id := id - num * id$

	Saum	Eingabe
1.	<u>S</u>	$\uparrow id := id - num * id$
2.	$id := E$	$\uparrow id := id - num * id$
3.	$id := E$	$id \uparrow := id - num * id$
4.	$id := \underline{E}$	$id := \uparrow id - num * id$
5.	$id := \underline{E} + T$	$id := \uparrow id - num * id$
6.	$id := \underline{E} + T + T$	$id := \uparrow id - num * id$
7.	$id := \underline{E} + T + T + T$	$id := \uparrow id - num * id$
...		
∞	$id := \underline{E} + T + \dots + T$	$id := \uparrow id - num * id$

Beliebig häufige Anwendung von Regeln, ohne jemals weitere Eingabe zu konsumieren!

Linksrekursion

Definition (*Linksrekursive Grammatik*):

Eine Grammatik G ist *linksrekursiv*, wenn es ein Nichtterminal $N \in V_N$ gibt, so dass über die Produktionsregeln P eine Ableitung $N \rightarrow^+ N\alpha$ möglich ist.

α ist hierbei eine beliebige Folge von Wörtern, d.h. $\alpha \in (V_N \cup V_T)^*$

- Linksrekursivität kann dazu führen, dass ein Top-Down Parser nicht terminiert!
- Zur Top-Down Syntax-Analyse muss die Grammatik rechtsrekursiv sein: $N \rightarrow^+ \alpha N$
 (wobei im Gegensatz zur Linksrekursivität oben für α zusätzlich gelten muss: $\alpha \neq \emptyset$ und α beginnt nicht mit N)

 Umwandlung der Linksrekursivität in Rechtsrekursivität.

Eliminierung von Linksrekursion (1)

Grammatik G_2

$$\begin{array}{llll}
 S \rightarrow \text{id} := E & E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\
 & E \rightarrow E - T & T \rightarrow F & F \rightarrow \text{num} \\
 & E \rightarrow T & & F \rightarrow (E)
 \end{array}$$

- Offensichtlich: Folgende Produktionsregeln sind linksrekursiv:
 $E \rightarrow E + T$ $E \rightarrow E - T$ $T \rightarrow T * F$
- Für z.B. das linksrekursive Nichtterminal E wird ein neues Nichtterminal E' eingeführt.
- Linksrekursive Produktionsregeln bzgl. E werden umgeformt, z.B. wird aus $E \rightarrow E + T$  $E \rightarrow T E'$ $E' \rightarrow + T E'$
 $E' \rightarrow$

Eliminierung von Linksrekursion (2)

Allgemeines Schema der Umformung:

- Für Produktionen $N \rightarrow N \alpha$ und $N \rightarrow \gamma$ weiß man unter der Voraussetzung, dass γ nicht mit N beginnt:
 Aus diesen Regeln sind Sätze $\gamma \alpha^*$ ableitbar, d.h. ein γ gefolgt von Null oder mehreren α .
- Umformung:

$$\begin{array}{l}
 N \rightarrow N \alpha_1 \\
 N \rightarrow N \alpha_2 \\
 N \rightarrow \gamma_1 \\
 N \rightarrow \gamma_2
 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{l}
 N \rightarrow \gamma_1 N' \\
 N \rightarrow \gamma_2 N' \\
 N' \rightarrow \alpha_1 N' \\
 N' \rightarrow \alpha_2 N' \\
 N' \rightarrow
 \end{array}$$

Eliminierung von Linksrekursion (3)

Grammatik G_2

$S \rightarrow \text{id} := E$	$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow \text{id}$
	$E \rightarrow E - T$	$T \rightarrow F$	$F \rightarrow \text{num}$
	$E \rightarrow T$		$F \rightarrow (E)$



Grammatik G_3

$S \rightarrow \text{id} := E$	$E \rightarrow T E'$	$T \rightarrow F T'$	$F \rightarrow \text{id}$
			$F \rightarrow \text{num}$
	$E' \rightarrow + T E'$	$T' \rightarrow * F T'$	$F \rightarrow (E)$
	$E' \rightarrow - T E'$	$T' \rightarrow$	
	$E' \rightarrow$		

Vorausschauendes Parsen

Bisheriges Problem des Top-Down Parsers:

- Wahl der passenden Regel kann zu Backtracking des Parsers führen.

Alternative:

- Gezielte Bestimmung der passenden Regel, indem man in der zu parsenden Eingabe vorausschaut und Wörter nach aktueller Position mit betrachtet (*look-ahead*).

Im folgenden:

- Vorausschau genau eines Wortes, d.h. look-ahead von 1
- ☞ *Predictive Parsers*

Parsen mit Rekursivem Abstieg (1)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$L \rightarrow \text{end}$

$S \rightarrow \text{begin } S L$

$L \rightarrow ; S L$

$S \rightarrow \text{print } E$

$E \rightarrow \text{num} = \text{num}$

Grundlegender Aufbau eines solchen Parsers:

- Eine Funktion \mathfrak{N} für jedes Nichtterminal N der Grammatik
- In einer solchen Funktion \mathfrak{N} jeweils einen Zweig einer Fallunterscheidung pro Produktionsregel, die N auf linker Seite enthält.

Parsen mit Rekursivem Abstieg (2)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$L \rightarrow \text{end}$

$S \rightarrow \text{begin } S L$

$L \rightarrow ; S L$

$S \rightarrow \text{print } E$

$E \rightarrow \text{num} = \text{num}$

Hilfsgerüst:

```
enum token IF, THEN, ELSE, BEGIN, END, ...;
```

```
extern enum token getToken(); // This is the scanner
```

```
enum token tok; // This is the current input token
```

```
void advance() { tok = getToken(); } // Read 1 token
```

```
void eat( enum token t ) // Match t with current input
```

```
{ if ( tok == t ) advance(); else error(); }
```

Parsen mit Rekursivem Abstieg (3)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$L \rightarrow \text{end}$

$S \rightarrow \text{begin } S L$

$L \rightarrow ; S L$

$S \rightarrow \text{print } E$

$E \rightarrow \text{num} = \text{num}$

Nichtterminal S:

```

void S() { switch( tok ) // Check token, i.e., look-ahead
  case IF: eat(IF); E(); eat(THEN); S(); eat(ELSE); S();
           break;
  case BEGIN: eat(BEGIN); S(); L(); break;
  case PRINT: eat(PRINT); E(); break;
  default: error(); }
  
```

Parsen mit Rekursivem Abstieg (4)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$L \rightarrow \text{end}$

$S \rightarrow \text{begin } S L$

$L \rightarrow ; S L$

$S \rightarrow \text{print } E$

$E \rightarrow \text{num} = \text{num}$

Nichtterminal L :

```

void L() { switch( tok ) // Check token, i.e., look-ahead
  case END: eat(END); break;
  case SEMICOLON: eat(SEMICOLON); S(); L(); break;
  default: error(); }
  
```

Parsen mit Rekursivem Abstieg (5)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; S L$

$E \rightarrow \text{num} = \text{num}$

Nichtterminal E :

```
void E() {  
    eat(NUM) ; eat(EQ) ; eat(NUM) ; }  
}
```

Parsen mit Rekursivem Abstieg (6)

Beispiel: Grammatik G_4

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$L \rightarrow \text{end}$

$S \rightarrow \text{begin } S L$

$L \rightarrow ; S L$

$S \rightarrow \text{print } E$

$E \rightarrow \text{num} = \text{num}$

Sehr schön zu erkennen:

- Parsen erfolgt *rekursiv*, da Funktionen, die Nichtterminale repräsentieren, sich potenziell rekursiv selbst aufrufen.
- Parsen erfolgt im *Abstieg*, d.h. top-down von Wurzel zu Blättern
- Parsen erfolgt *vorausschauend*, d.h. nutzt für Nichtterminale S und L look-ahead von 1, um korrekte Regel zweifelsfrei zu ermitteln.

Grammatik mit Konflikten

Zurück zu Grammatik G_2 :

$$\begin{array}{llll}
 S \rightarrow \text{id} := E & E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\
 & E \rightarrow E - T & T \rightarrow F & F \rightarrow \text{num} \\
 & E \rightarrow T & & F \rightarrow (E)
 \end{array}$$

Nichtterminal E :

```

void E() { switch( tok ) // Check token, i.e., look-ahead
  case ???: E(); eat(PLUS); T(); break;
  case ???: E(); eat(MINUS); E(); break;
  case ???: T(); break;
  default: error(); }

```

Wie soll die
Fallunter-
scheidung
funktionieren?

Vorausschauendes Parsen & 1. Terminal

Problem bei Grammatik G_2 :

- Vorausschauendes Parsen funktioniert nur, wenn das *erste aus der rechten Seite einer Regel ableitbare Terminal genügend Information liefert*, um die passende Regel auszuwählen.
 - Für Nichtterminal E müssten rechte Regel-Seiten $E + T$ und $E - T$ unterschieden werden.
 - Aus $E + T$ können ausschließlich Sätze der Form $\mathbf{id}\alpha^*$, $\mathbf{num}\beta^*$ oder $(\gamma^*$ abgeleitet werden.
 - Gleiches gilt dummerweise genauso für $E - T$.
 - Regeln mit rechter Seite $E + T$ und $E - T$ sind für Parser mit look-ahead 1 nicht unterscheidbar.
- 👉 G_2 ungeeignet für vorausschauendes Parsen

FIRST-Mengen

Definition (*FIRST-Menge*):

Für eine beliebige Folge von Wörtern α , d.h. $\alpha \in (V_N \cup V_T)^*$, ist $\text{FIRST}(\alpha)$ die Menge aller Terminale, die beliebige, aus α ableitbaren Sätze beginnen können.

Beispiel zu G_2 :

- $\text{FIRST}(E + T) = \{ \text{id}, \text{num}, (\}$
- Zusätzlich: $\text{FIRST}(E - T) = \{ \text{id}, \text{num}, (\}$
- 👉 D.h. $\text{FIRST}(E + T)$ und $\text{FIRST}(E - T)$ enthalten für Nichtterminal E gemeinsame Terminale.
- 👉 Problem von G_2 : $\text{FIRST}(E + T) \cap \text{FIRST}(E - T) \neq \emptyset$

Neutralisierbare Nichtterminale

Berechnung der FIRST-Mengen:

- Scheinbar einfach: $\text{FIRST}(X Y Z)$ scheint nur von X abzuhängen, Y und Z können ignoriert werden.
- Problem: Was, wenn z.B. X und Y nach ε abgeleitet werden können, d.h. X und Y neutralisierbar sind?
- ☞ $\text{FIRST}(X Y Z)$ hängt nun von Z ab.
- Falls zusätzlich Z neutralisierbar: $\text{FIRST}(X Y Z)$ hängt von dem ab, was auf Z folgen kann.

Definition (**Neutralisierbare Nichtterminale**):

Ein Nichtterminal N heißt *neutralisierbar*, falls aus N das leere Wort ε abgeleitet werden kann: $N \xrightarrow{*} \varepsilon$.

FOLLOW-Mengen

Definition (*FOLLOW-Menge*):

Für ein Nichtterminal N ist $\text{FOLLOW}(N)$ die Menge aller Terminale, die in Ableitungen unmittelbar auf N folgen können:

$$\text{FOLLOW}(N) = \{ T \in V_T \mid S \rightarrow^* \beta N \gamma \text{ und } T \in \text{FIRST}(\gamma) \}$$

Beispiel:

- $\text{FOLLOW}(X)$ kann Terminal t enthalten, wenn
 - Folge $X Y Z t$ ableitbar ist und
 - Y und Z beide neutralisierbar sind.

Berechnung von FIRST und FOLLOW (1)

Gegeben:

- Kontextfreie Grammatik $G = (V_N, V_T, P, S)$

Initialisierung:

for (<alle Nichtterminale $N \in V_N$ >)

 FIRST(N) = FOLLOW(N) = \emptyset ;

 NEUTRAL(N) = false;

for (<alle Terminale $T \in V_T$ >)

 FIRST(T) = { T };

Berechnung von FIRST und FOLLOW (2)

Algorithmus:

```

while ( <FIRST, FOLLOW und NEUTRAL wurden verändert> )
  for ( <alle Produktionsregeln  $p \in P: N \rightarrow Y_1 Y_2 \dots Y_k$ > )
    if ( NEUTRAL(  $Y_1$  ) == ... == NEUTRAL(  $Y_k$  ) == true )
      NEUTRAL(  $N$  ) = true;
    for ( <alle  $i$  von 1 bis  $k$ > )
      if (  $i == 1$  ) // Fall 1a
        FIRST(  $N$  ) = FIRST(  $N$  ) U FIRST(  $Y_1$  );
      else if ( NEUTRAL(  $Y_1$  ) == ... == NEUTRAL(  $Y_{i-1}$  ) == true )
        // Fall 1b:  $N \rightarrow \varepsilon \varepsilon \dots \varepsilon Y_i \dots Y_k$ 
        FIRST(  $N$  ) = FIRST(  $N$  ) U FIRST(  $Y_i$  );
      ...
  
```

Berechnung von FIRST und FOLLOW (3)

Algorithmus:

```

while ( <FIRST, FOLLOW und NEUTRAL wurden verändert> )
  for ( <alle Produktionsregeln  $p \in P: N \rightarrow Y_1 Y_2 \dots Y_k$ > )
    ...
    for ( <alle  $i$  von 1 bis  $k$ > )
      ...
      if (  $i == k$  ) // Fall 2a
        FOLLOW(  $Y_k$  ) = FOLLOW(  $Y_k$  ) U FOLLOW(  $N$  );
      else if ( NEUTRAL(  $Y_{i+1}$  ) == ... == NEUTRAL(  $Y_k$  ) == true )
        // Fall 2b:  $N \rightarrow \dots Y_i \varepsilon \varepsilon \dots \varepsilon$ 
        FOLLOW(  $Y_i$  ) = FOLLOW(  $Y_i$  ) U FOLLOW(  $N$  );
      ...
  
```

Berechnung von FIRST und FOLLOW (4)

Algorithmus:

```

while ( <FIRST, FOLLOW und NEUTRAL wurden verändert> )
  for ( <alle Produktionsregeln  $p \in P: N \rightarrow Y_1 Y_2 \dots Y_k$ > )
    ...
    for ( <alle  $i$  von 1 bis  $k$ > )
      ...
      for ( <alle  $j$  von  $i+1$  bis  $k$ > )
        if (  $j == i+1$  ) // Fall 3a
          FOLLOW(  $Y_i$  ) = FOLLOW(  $Y_i$  ) U FIRST(  $Y_j$  );
        else
          if ( NEUTRAL(  $Y_{i+1}$  ) == ... == NEUTRAL(  $Y_{j-1}$  ) == true )
            // Fall 3b:  $N \rightarrow \dots Y_i \varepsilon \varepsilon \dots \varepsilon Y_j \dots$ 
            FOLLOW(  $Y_i$  ) = FOLLOW(  $Y_i$  ) U FIRST(  $Y_j$  );
  
```

Parser-Tabellen

Parsen mit Rekursivem Abstieg:

- Parser-Funktion für Nichtterminal N enthält einen Zweig der Fallunterscheidung pro Regel mit N auf linker Seite.
- Parser-Funktion muss für Nichtterminal N und für aktuelles Eingabe-Token T entscheiden, welche Produktionsregel, d.h. welcher Zweig der Fallunterscheidung zu verwenden ist.
- Entscheidungen des Parsers können in zweidimensionaler Tabelle dargestellt werden

Parser-Tabelle:

- X-Achse: Alle Terminale Y-Achse: Alle Nichtterminale
- Tabelleneintrag: zu verwendende Produktionsregel

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	\Rightarrow	$S \rightarrow \text{id} := E$
	$E = \{ \}$		$E \rightarrow E + T$
	$T = \{ \}$		$E \rightarrow E - T$
	$F = \{ \}$		$E \rightarrow T$
FOLLOW:	$S = \{ \}$		$T \rightarrow T * F$
	$E = \{ \}$		$T \rightarrow F$
	$T = \{ \}$		$F \rightarrow \text{id}$
	$F = \{ \}$		$F \rightarrow \text{num}$
NEUTRAL:			$F \rightarrow (E)$

Wird für dieses Beispiel unterschlagen, da ohnehin für alle Nichtterminale stets gleich false (es gibt keine Regel in G_2 , die nach ε ableitet).

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{id_{1/1a}$	}		$S \rightarrow id := E$
	$E = \{$	}		$E \rightarrow E + T$
	$T = \{$	}		$E \rightarrow E - T$
	$F = \{$	}		$E \rightarrow T$
FOLLOW:	$S = \{$	}		$T \rightarrow T * F$
	$E = \{+_{2/3a}$	}		$T \rightarrow F$
	$T = \{$	}		$F \rightarrow id$
	$F = \{$	}		$F \rightarrow num$
				$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id} \}$		$S \rightarrow \text{id} := E$
	$E = \{ \}$		$E \rightarrow E + T$
	$T = \{ \}$		$E \rightarrow E - T$
	$F = \{ \}$		$E \rightarrow T$
FOLLOW:	$S = \{ \}$		$T \rightarrow T * F$
	$E = \{ +, - \}$		$T \rightarrow F$
	$T = \{ + \}$		$F \rightarrow \text{id}$
	$F = \{ \}$		$F \rightarrow \text{num}$
			$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \}$	$E \rightarrow T$
		$T \rightarrow T * F$
		$T \rightarrow F$
		$F \rightarrow \text{id}$
		$F \rightarrow \text{num}$
		$F \rightarrow (E)$

FOLLOW:

$S = \{ \}$
$E = \{ +, -, (\}$
$T = \{ +, - \}$
$F = \{ \}$



Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} \}$	$F \rightarrow \text{num}$
		$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a} \}$	$F \rightarrow \text{num}$
		$F \rightarrow (E)$



Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} \ -_{4/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} \ -_{5/2a} \ *_{7/3a} \}$	$\Rightarrow F \rightarrow \text{id}$
	$F = \{ +_{6/2a} \ -_{6/2a} \ *_{8/2a} \}$	$F \rightarrow \text{num}$
		$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \text{ num}_{10/1a} \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a} \}$	 $F \rightarrow \text{num}$
		$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \text{ num}_{10/1a} (_{11/1a} \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a})_{12/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a} \}$	$F \rightarrow \text{num}$
		 $F \rightarrow (E)$

Iteration 1 des Algorithmus: Schritte 1 bis 12

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$		$S \rightarrow \text{id} := E$
	$E = \{ \}$		$E \rightarrow E + T$
	$T = \{ \}$		$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \text{ num}_{10/1a} (_{11/1a} \}$		$E \rightarrow T$
FOLLOW:	$S = \{ \}$		$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a})_{12/3a} \}$		$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a})_{13/2a} \}$		$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a} \}$		$F \rightarrow \text{num}$
			$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \}$	$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \text{ num}_{10/1a} (_{11/1a} \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a})_{12/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a})_{13/2a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a})_{14/2a} \}$	$F \rightarrow \text{num}$
		$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{ \text{id}_{1/1a} \}$	$S \rightarrow \text{id} := E$
	$E = \{ \}$	$E \rightarrow E + T$
	$T = \{ \text{id}_{15/1a} \text{ num}_{15/1a} (\text{)}_{15/1a} \}$	$E \rightarrow E - T$
	$F = \{ \text{id}_{9/1a} \text{ num}_{10/1a} (\text{)}_{11/1a} \}$	$E \rightarrow T$
FOLLOW:	$S = \{ \}$	$T \rightarrow T * F$
	$E = \{ +_{2/3a} -_{4/3a})_{12/3a} \}$	$T \rightarrow F$
	$T = \{ +_{3/2a} -_{5/2a} *_{7/3a})_{13/2a} \}$	$F \rightarrow \text{id}$
	$F = \{ +_{6/2a} -_{6/2a} *_{8/2a})_{14/2a} \}$	$F \rightarrow \text{num}$
		$F \rightarrow (E)$

Iteration 2 des Algorithmus: Schritte 13 bis 15

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{id_{1/1a} \}$	$S \rightarrow id := E$
	$E = \{id_{16/1a} \text{ num}_{16/1a} (\text{ }_{16/1a} \}$	$E \rightarrow E + T$
	$T = \{id_{15/1a} \text{ num}_{15/1a} (\text{ }_{15/1a} \}$	$E \rightarrow E - T$
	$F = \{id_{9/1a} \text{ num}_{10/1a} (\text{ }_{11/1a} \}$	$E \rightarrow T$
		$T \rightarrow T * F$
FOLLOW:	$S = \{ \}$	$T \rightarrow F$
	$E = \{+_{2/3a} -_{4/3a})_{12/3a} \}$	$F \rightarrow id$
	$T = \{+_{3/2a} -_{5/2a} *_{7/3a})_{13/2a} \}$	$F \rightarrow num$
	$F = \{+_{6/2a} -_{6/2a} *_{8/2a})_{14/2a} \}$	$F \rightarrow (E)$

Iteration 3 des Algorithmus: Schritt 16

Iteration 4 des Algorithmus: Verändert Mengen nicht – Stopp

Beispiel: Parser-Tabelle zu G_2

	id	:=	+	-	*	num	()
S	$S \rightarrow id := E$							
E	$E \rightarrow E + T$ $E \rightarrow E - T$ $E \rightarrow T$					$E \rightarrow E + T$ $E \rightarrow E - T$ $E \rightarrow T$	$E \rightarrow E + T$ $E \rightarrow E - T$ $E \rightarrow T$	
T	$T \rightarrow T * F$ $T \rightarrow F$					$T \rightarrow T * F$ $T \rightarrow F$	$T \rightarrow T * F$ $T \rightarrow F$	
F	$F \rightarrow id$					$F \rightarrow num$	$F \rightarrow (E)$	

Problem von G_2 :

- Parser-Tabelle hat etliche Zellen, die mehrere Regeln enthalten.

LL(1)-Grammatiken

Informale Definition (*LL(1)-Grammatik*):

Eine kontextfreie Grammatik heißt *LL(1)-Grammatik*, wenn alle Zellen ihrer Parser-Tabelle höchstens eine Produktionsregel enthalten.

Beispiel:

- G_2 ist keine LL(1)-Grammatik
- G_2 kann daher grundsätzlich nicht durch vorausschauenden Parser erkannt werden.

Beispiel zu FIRST / FOLLOW-Mengen: G_3

FIRST:

$$\begin{aligned}
 S &= \{ \text{id}_{1/1a} \} \\
 E &= \{ \text{id}_{20/1a} \text{ num}_{20/1a} (\text{ }_{20/1a} \} \\
 E' &= \{ +_{2/1a} \text{ }_{-4/1a} \} \\
 T &= \{ \text{id}_{17/1a} \text{ num}_{17/1a} (\text{ }_{17/1a} \} \\
 T' &= \{ *_{8/1a} \} \\
 F &= \{ \text{id}_{11/1a} \text{ num}_{12/1a} (\text{ }_{13/1a} \}
 \end{aligned}$$

$$S \rightarrow \text{id} := E$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E'$$

$$E' \rightarrow - T E'$$

$$E' \rightarrow$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T'$$

$$T' \rightarrow$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

FOLLOW:

$$\begin{aligned}
 S &= \{ \} \\
 E &= \{)_{14/3a} \} \\
 E' &= \{)_{16/2a} \} \\
 T &= \{ +_{3/3a} \text{ }_{-5/3a})_{15/2b} \} \\
 T' &= \{ +_{7/2a} \text{ }_{-7/2a})_{19/2a} \} \\
 F &= \{ *_{9/3a} +_{18/2b} \text{ }_{-18/2b})_{18/2b} \}
 \end{aligned}$$

NEUTRAL: $E' = \text{true}_6$ $T' = \text{true}_{10}$

Beispiel: Parser-Tabelle zu G_3

	id	:=	+	-	*	num	()
S	$S \rightarrow \text{id} := E$							
E	$E \rightarrow TE'$					$E \rightarrow TE'$	$E \rightarrow TE'$	
E'			$E' \rightarrow +TE'$	$E' \rightarrow -TE'$				$E' \rightarrow$
T	$T \rightarrow FT'$					$T \rightarrow FT'$	$T \rightarrow FT'$	
T'			$T' \rightarrow$	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$
F	$F \rightarrow \text{id}$					$F \rightarrow \text{num}$	$F \rightarrow (E)$	

👉 G_3 ist eine LL(1)-Grammatik

Zusammenfassung: Top-Down Analyse

Vorgehensweise zum Schreiben eines Top-Down Parsers:

Gegeben: Kontextfreie Grammatik G

1. Berechne FIRST- und FOLLOW-Mengen
2. Erzeuge Parser-Tabelle
3. Falls G keine LL(1)-Grammatik ist:
Transformiere G , so dass LL(1)-Eigenschaft gegeben ist
(z.B. Eliminierung von Linksrekursion)
4. Generiere eine Prozedur pro Nichtterminal N :
 - Je 1 Case-Zweig pro zu N passender Regel
 - Aufruf anderer Prozeduren, je nach vorhandenen Nichtterminalen in rechten Seiten von Regeln

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
 - Einführung
 - Top-Down Syntax-Analyse
 - Bottom-Up Syntax-Analyse
 - Praktische Umsetzung: BISON
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Motivation Bottom-Up Syntax-Analyse

Problem der Top-Down Analyse:

- Parser muss „vorhersagen“, welche Produktionsregel in jeweiligem Stand des Ableitungsprozesses zu verwenden ist.
- Einzige Information, die ein Top-Down Parser dazu hat: k Eingabe-Token look-ahead.

Schema der Bottom-Up Analyse:

- Führt im Gegensatz zur Top-Down Analyse keine Links- sondern eine Rechtsableitung durch.
- Bestimmung der anzuwendenden Regel erst, wenn alle Eingabe-Token evtl. passender rechter Regel-Seiten gelesen wurden.
- Ebenfalls: Nutzung von k weiteren Eingabe-Token look-ahead
- 👉 Viel mehr Informationen zum Parsen verfügbar!

Kellerautomaten

Definition (*Kellerautomat, KA*):

Ein *Kellerautomat* ist ein Tupel $K = (\Sigma, \Gamma, Q, q_0, Z_0, F, \Delta)$ mit

- Σ : endliches **Eingabealphabet**
- Γ : endliches **Stackalphabet**
- Q : endliche Menge von **Zuständen**
- $q_0 \in Q$: **Anfangszustand**
- $Z_0 \in \Gamma$: die **Initialisierung des Stacks**
- $F \subseteq Q$: Menge der **akzeptierenden Endzustände**
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times (Q \times \Gamma^*)$: Relation der **Zustandsübergänge**

Schritte eines Kellerautomaten

Definition (*Schrittrelation*):

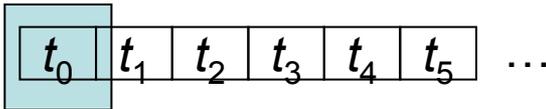
Sei $K = (\Sigma, \Gamma, Q, q_0, Z_0, F, \Delta)$ ein KA. Ein Paar (q, w, γ) mit $q \in Q$, $w \in \Sigma^*$ und $\gamma \in \Gamma^*$ heißt *Konfiguration*. Die *Schrittrelation* \xrightarrow{K} setzt Konfigurationen in Beziehung:

$(q, ew, \gamma_1\gamma_2) \xrightarrow{K} (p, w, \gamma_1\gamma_3) \Leftrightarrow (q, e, \gamma_2, (p, \gamma_3)) \in \Delta$ und $e \in \Sigma \cup \{ \varepsilon \}$,
 $\gamma_1, \gamma_2, \gamma_3 \in \Gamma^*$. \xrightarrow{K}^* ist die transitive Hülle der Schrittrelation.

In Worten: \xrightarrow{K} modelliert einen einzelnen Zustandsübergang von K von aktuellem Zustand q in Folgezustand p für ein Eingabezeichen e bzw. ε und ein Rest-Wort w . Dabei „verbraucht“ K die untersten Elemente γ_2 des Kellers und speichert neue Elemente γ_3 unten im Stack. \xrightarrow{K}^* modelliert eine ganze Folge einzelner Schritte für ein Wort w .

Arbeitsweise von Kellerautomaten

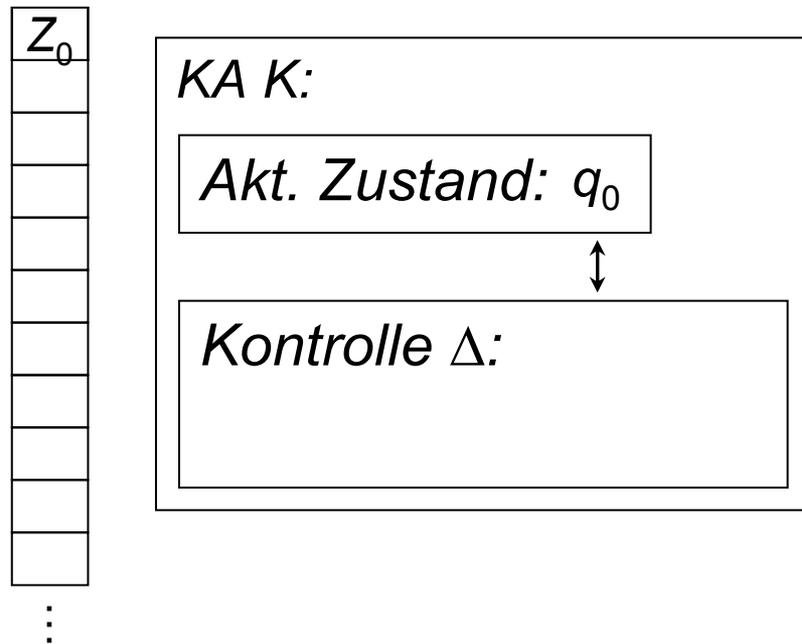
Eingabeband:



Start des Automaten:

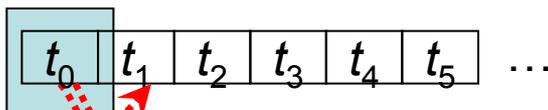
- Lesekopf steht auf erstem Token der Eingabe
- Speicher des Automaten enthält Anfangszustand q_0
- Kellerspeicher ist mit Z_0 initialisiert

Stack:

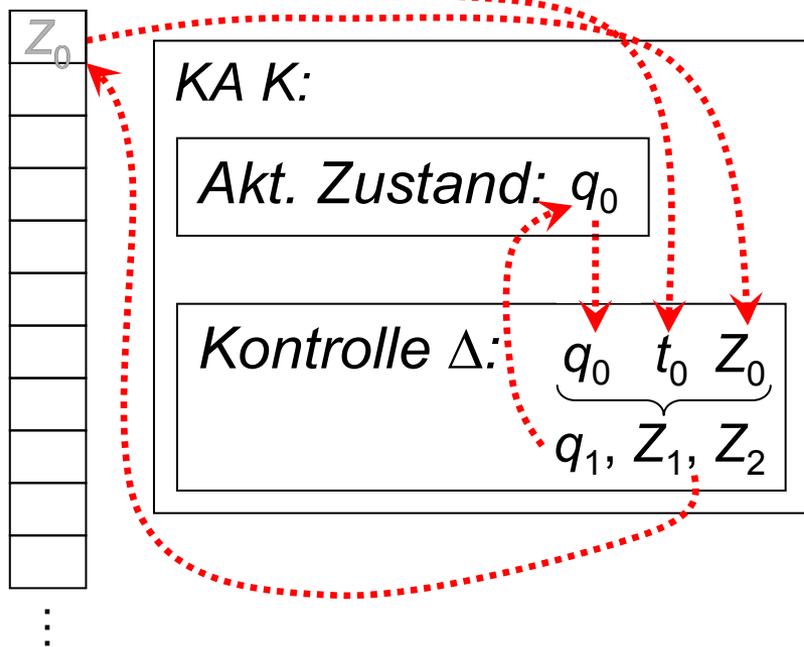


Arbeitsweise von Kellerautomaten

Eingabeband:



Stack:

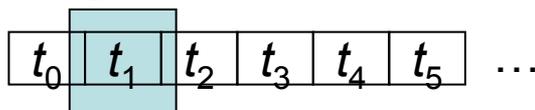


Pro Verarbeitungsschritt:

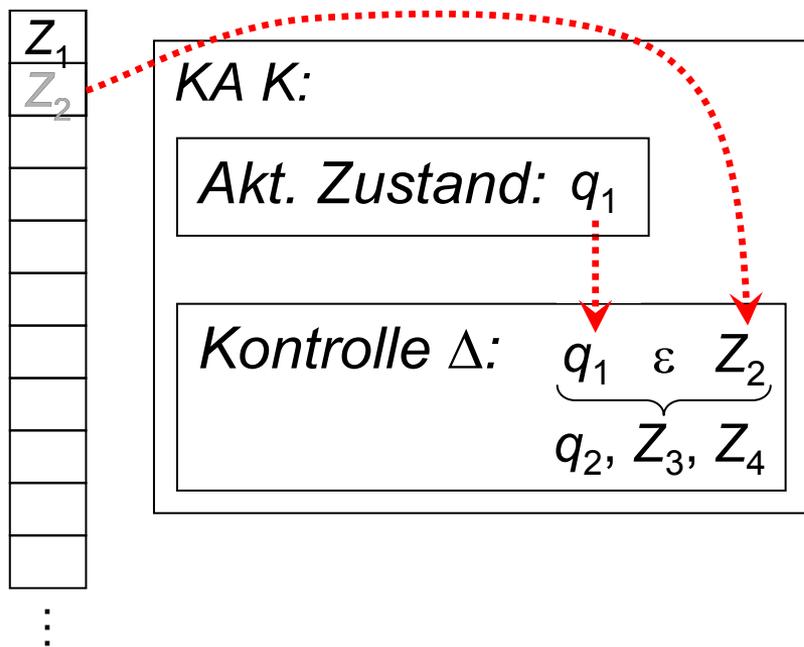
- „Raten“ der aktuellen Eingabe e : entweder aktuelles Token unter Lesekopf oder ‘ ε ’.
- Lesen und Entfernen der unteren Stack-Elemente Z_l, Z_{l-1}, \dots
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, \dots, Z_n :
 $(q_{i+1}, Z_1, \dots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \dots)$
- Abspeichern von q_{i+1}, Z_1, \dots, Z_n
- Nur falls Token der Eingabe gelesen: Lesekopf nach rechts.

Arbeitsweise von Kellerautomaten

Eingabeband:



Stack:

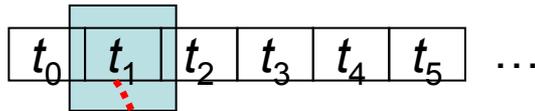


Pro Verarbeitungsschritt:

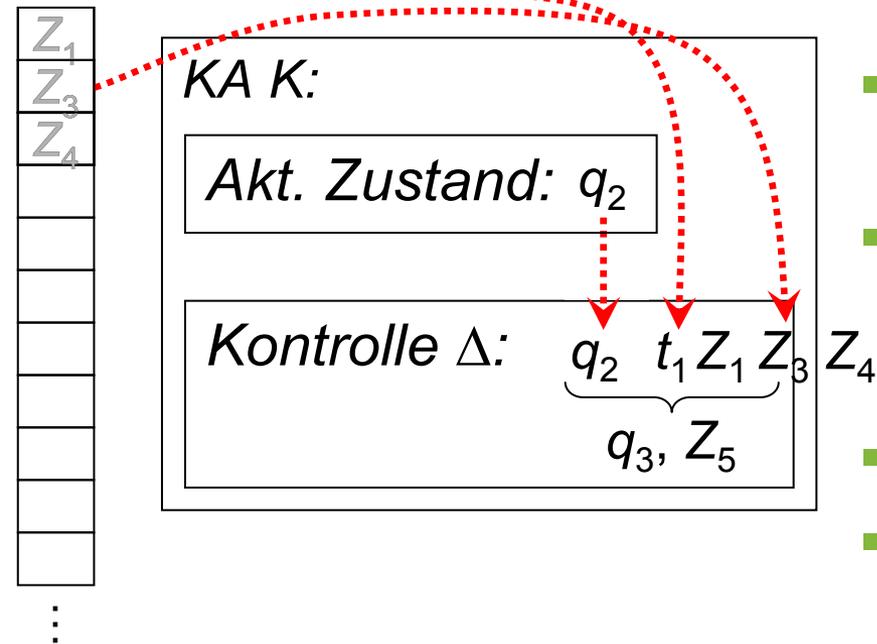
- „Raten“ der aktuellen Eingabe e : entweder aktuelles Token unter Lesekopf oder ‘ ε ’.
- Lesen und Entfernen der unteren Stack-Elemente Z_l, Z_{l-1}, \dots
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, \dots, Z_n :
 $(q_{i+1}, Z_1, \dots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \dots)$
- Abspeichern von q_{i+1}, Z_1, \dots, Z_n
- Nur falls Token der Eingabe gelesen: Lesekopf nach rechts.

Arbeitsweise von Kellerautomaten

Eingabeband:



Stack:

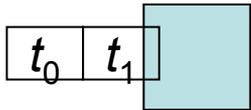


Pro Verarbeitungsschritt:

- „Raten“ der aktuellen Eingabe e : entweder aktuelles Token unter Lesekopf oder ‘ ε ’.
- Lesen und Entfernen der unteren Stack-Elemente Z_l, Z_{l-1}, \dots
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, \dots, Z_n :
 $(q_{i+1}, Z_1, \dots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \dots)$
- Abspeichern von q_{i+1}, Z_1, \dots, Z_n
- Nur falls Token der Eingabe gelesen: Lesekopf nach rechts.

Arbeitsweise von Kellerautomaten

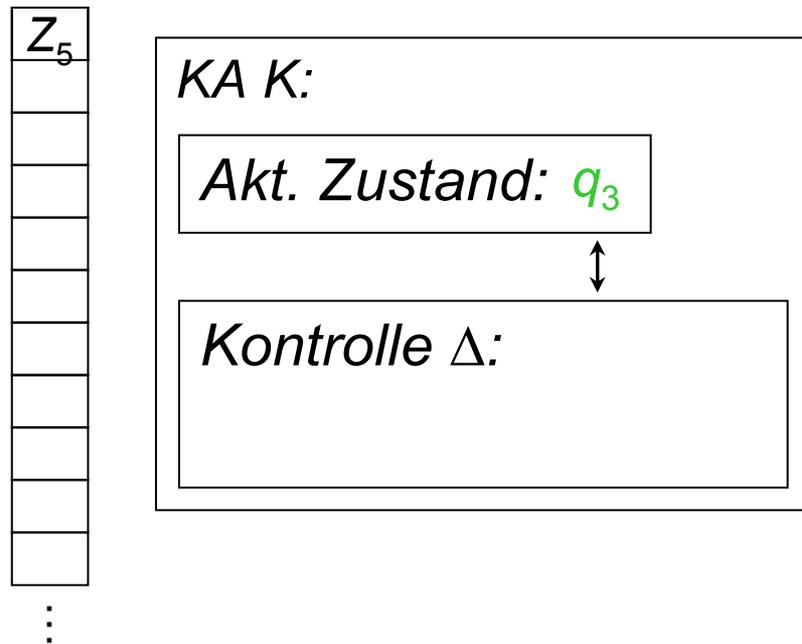
Eingabeband:



Gültige Eingabe gefunden, wenn:

- aktueller Zustand q_i ein Endzustand ist, d.h. $q_i \in F$, und
 - Eingabe komplett abgearbeitet wurde.
- ☞ Dann: K hat akzeptierende Endkonfiguration $(q_i, \varepsilon, \gamma)$ erreicht.
- ☞ Kellerautomat hat Schrittfolge $(q_0, w, Z_0) \xrightarrow{*} (q_i, \varepsilon, \gamma)$ für $q_i \in F$ durchgeführt.

Stack:



LR-Parsen mit Kellerautomaten

LR(k)-Parser:

- Left-to-right parse, Rightmost-derivation, k -symbol look-ahead
- Merkwürdig: Rechtsableitung bei Verarbeitung der Eingabe von links?
- ☞ Realisierung mit Hilfe des Stacks des Kellerautomaten.

Kellerautomat als LR(k)-Parser:

- Erste k Token der Eingabe sind look-ahead des Parsers.
- Mit Hilfe des look-aheads und aktuellem Stack-Inhalt wählt der LR(k)-Parser zwischen zwei Möglichkeiten:
 - Shift: Sichere 1. Token der aktuellen Eingabe auf Stack, oder
 - Reduce: Wähle Produktionsregel $X \rightarrow A B C$; entferne A , B und C vom Ende des Stacks; speichere X auf Stack.

Beispiel (1)

Grammatik G_2

$$\begin{array}{llll}
 S \rightarrow \text{id} := E & E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\
 & E \rightarrow E - T & T \rightarrow F & F \rightarrow \text{num} \\
 & E \rightarrow T & & F \rightarrow (E)
 \end{array}$$

- Eingabe: `id := id - num * id`
- Im folgenden:
 - Versuch der Rechtsableitung.
 - Aktuelle Position im Eingabe-Satz ist mit \uparrow markiert.

Beispiel (2)

Eingabe: `id := id - num * id`

	Stack	Eingabe	Aktion
1.	Z_0	$\uparrow id := id - num * id$	<i>shift</i>
2.	<code>id</code>	<code>id</code> $\uparrow := id - num * id$	<i>shift</i>
3.	<code>id :=</code>	<code>id :=</code> $\uparrow id - num * id$	<i>shift</i>
4.	<code>id := id</code>	<code>id := id</code> $\uparrow - num * id$	<i>reduce</i> ($F \rightarrow id$)
5.	<code>id := F</code>	<code>id := id</code> $\uparrow - num * id$	<i>reduce</i> ($T \rightarrow F$)
6.	<code>id := T</code>	<code>id := id</code> $\uparrow - num * id$	<i>reduce</i> ($E \rightarrow T$)
7.	<code>id := E</code>	<code>id := id</code> $\uparrow - num * id$	<i>shift</i>
8.	<code>id := E -</code>	<code>id := id -</code> $\uparrow num * id$	<i>shift</i>
9.	<code>id := E - num</code>	<code>id := id - num</code> $\uparrow * id$	<i>reduce</i> ($F \rightarrow num$)

Beispiel (3)

Eingabe: `id := id - num * id`

	Stack	Eingabe	Aktion
10.	<code>id := E - F</code>	<code>id := id - num ↑ * id</code>	<i>reduce(T → F)</i>
11.	<code>id := E - T</code>	<code>id := id - num ↑ * id</code>	<i>shift</i>
12.	<code>id := E - T *</code>	<code>id := id - num * ↑ id</code>	<i>shift</i>
13.	<code>id := E - T * id</code>	<code>id := id - num * id ↑</code>	<i>reduce(F → id)</i>
14.	<code>id := E - T * F</code>	<code>id := id - num * id ↑</code>	<i>reduce(T → T * F)</i>
15.	<code>id := E - T</code>	<code>id := id - num * id ↑</code>	<i>reduce(E → E - T)</i>
16.	<code>id := E</code>	<code>id := id - num * id ↑</code>	<i>reduce(S → id := E)</i>
17.	<code>S</code>	<code>id := id - num * id ↑</code>	<i>accept</i>

Bottom-Up Parser

Einige notwendige Begriffe:

- Ziel des Parsers ist, ausgehend vom Startsymbol S einer Grammatik einen Eingabesatz abzuleiten:

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_{n-1} \rightarrow \gamma_n \rightarrow \text{Eingabesatz}$$
- Alle γ_i sind *Satzformen*, d.h. Ableitungen, die Terminale und Nichtterminale enthalten. Eine *Rechtssatzform* entsteht bei Anwendung von Rechtsableitung.
- Ein Bottom-Up Parser startet den Ableitungsprozess beim Satz und arbeitet sich schrittweise von γ_i nach γ_{i-1} , bis das Startsymbol S erreicht wird.
- Dazu: Bestimme Regel $N \rightarrow \alpha$, so dass $\alpha \in \gamma_i$; erzeuge γ_{i-1} , indem α in γ_i durch N ersetzt wird, so dass N in γ_{i-1} das äußerst rechte Nichtterminal ist (\Rightarrow Rechtsableitung).

Problem beim LR-Parsen

„Zudecken“ gebotener Reduktionen:

- Shift-Reduce Parser dürfen niemals eine *gebotene Reduktion* verpassen, d.h. durch ein eingelesenes Symbol im Keller „zudecken“.
- ☞ Voriges Beispiel: In Zeilen 5, 6 und 7 muss id nach E reduziert werden, damit später in Zeile 15 Regel $E \rightarrow E - T$ anwendbar ist.
- „Gebotene Reduktion“ heißt hier: Reduktion muss gemacht werden, um Rechtsableitung bis zum Startsymbol zu erhalten.
- Ist eine rechte Seite einmal zugedeckt, wird sie nicht mehr am unteren Ende des Stacks auftauchen.
- ☞ Die zugedeckte, gebotene Reduktion kann nicht mehr durchgeführt werden.

Griffe / Handles

Motivation:

Der Parser muss für eine Rechtssatzform γ_i die gebotene Reduktion $N \rightarrow \alpha$ finden, so dass die daraus erzeugte Rechtssatzform γ_{i-1} auch tatsächlich zum Startsymbol S führt und keine „Sackgasse“ ist.

Definition (*Griff / Handle*):

Ein *Griff* einer Rechtssatzform γ ist ein Paar $\langle N \rightarrow \alpha, k \rangle$; $N \rightarrow \alpha \in P$, k bezeichnet die Position des äußerst rechten Symbols von α in γ . Für einen Griff $\langle N \rightarrow \alpha, k \rangle$ liefert das Ersetzen von α durch N an Position k diejenige Rechtssatzform, aus der γ durch Rechtsableitung hervorgeht.

Wichtig: Für eine eindeutige Grammatik G existiert ein eindeutiger Griff für jede Rechtssatzform!

Beispiel

Grammatik G_2

$S \rightarrow id := E$

$E \rightarrow E + T$

$T \rightarrow T * F$

$F \rightarrow id$

$E \rightarrow E - T$

$T \rightarrow F$

$F \rightarrow num$

$E \rightarrow T$

$F \rightarrow (E)$

▪ *Eingabe:* $id := id * num$

$id := id * num \rightarrow$

$id := F * num \rightarrow$

$id := T * num \rightarrow$

$id := T * F \rightarrow$

$id := T \rightarrow$

$id := E \rightarrow$

S

Griffe: (auch in grün angezeigt)

$\langle F \rightarrow id, 3 \rangle$

$\langle T \rightarrow F, 3 \rangle$

$\langle F \rightarrow num, 5 \rangle$

$\langle T \rightarrow T * F, 5 \rangle$

$\langle E \rightarrow T, 3 \rangle$

$\langle S \rightarrow id := E, 3 \rangle$

%

Pseudocode des Shift-Reduce Parsers

Algorithmus:

```

shift(  $Z_0$  );
tok = getToken();
while ( ! ( <bottom of stack ==  $S$ > && <tok == EOF> ) )
  if ( <bottom of stack == handle  $\langle N \rightarrow \alpha, k \rangle$  )
    reduce(  $N \rightarrow \alpha$  );
  else if ( tok != EOF )
    shift( tok );
    tok = getToken();
  else
    error();
accept();
  
```

Aktionen des Shift-Reduce Parsers

Parsen beruht auf lediglich 4 Aktionen:

- *accept* und *error*: Parsen beenden und ggfs. sinnvolle Fehlermeldung ausgeben. (*simpel*)
- *shift*: Aktuelles Token auf den Stack legen, Scanner für nächstes Token aufrufen. (*trivial*)
- *reduce*: Teil α des benötigten Griffes $\langle N \rightarrow \alpha, k \rangle$ liegt am unteren Ende des Stacks.
Ersetze α durch N auf dem Stack.

Bestimmen von Griffen ist der Schlüssel des Parsers:

- Griff liegt unten auf Stack, und es gibt nur endlich viele Griffe für eine Grammatik.

👉 Konstruktion eines DEAs zum Bestimmen von Griffen!

Schwierigkeit des Bestimmens von Griffen

Eigenschaft eines Griffs: Damit ein Teilstring α Griff einer Rechtssatzform γ ist, müssen zwei Bedingungen gelten:

- α muss rechte Seite einer Regel $N \rightarrow \alpha$ sein, und
- es muss irgendeine Rechtsableitung vom Startsymbol S aus geben, die die Satzform γ erzeugt, wobei Regel $N \rightarrow \alpha$ als letzter Schritt angewendet wird.

Schwierigkeit:

- Bloße Betrachtung rechter Regelseiten zur Bestimmung von Griffen reicht nicht aus.
- Wie können Griffe bestimmt werden, ohne alle möglichen potenziellen Rechtsableitungen von S aus zu generieren?

Ansatz zum Bestimmen von Griffen

Erzeugung eines Endlichen Automaten:

- Ausnutzung von look-ahead und von Tabellen, die durch Analyse der Grammatik aufgebaut werden können.
- LR(1)-Parser erzeugen einen DEA, der anhand des look-aheads und der Tabellen den Stack untersucht und Griffe findet.

LR(1)-Grammatiken

Informale Definition (*LR(1)-Grammatik*):

Sei G eine kontextfreie Grammatik, die eine Rechtsableitung

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_{n-1} \rightarrow \gamma_n \rightarrow \text{Eingabesatz}$$

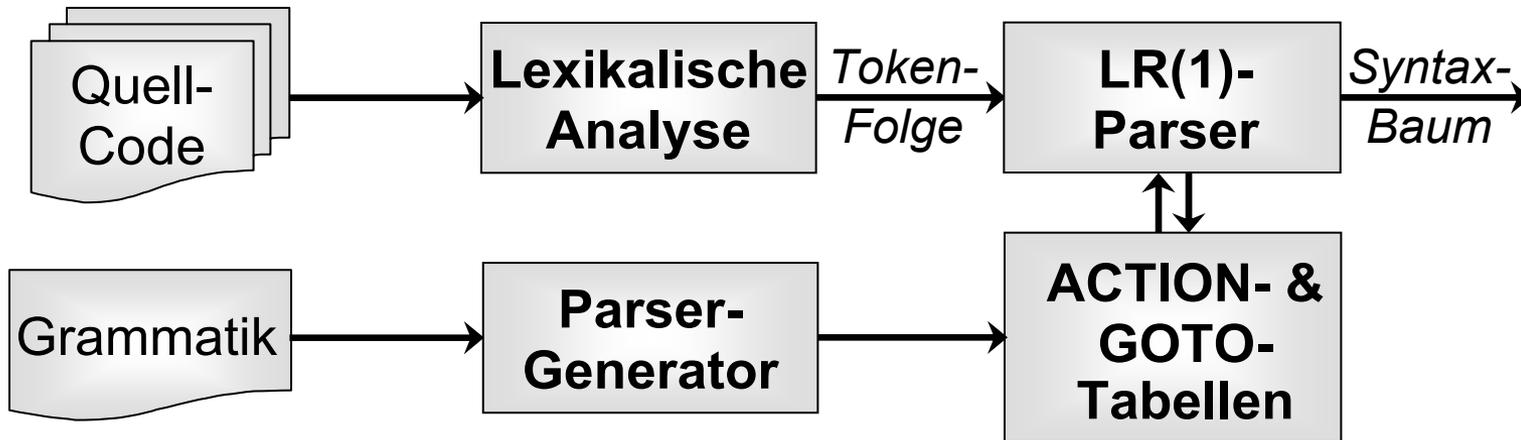
zulässt. G heißt *LR(1)-Grammatik*, wenn

- für jede Rechtssatzform γ_i der Griff sowie
- die zur Reduktion benötigte Regel bestimmt werden können, indem

γ_i von links nach rechts analysiert wird und höchstens 1 Symbol hinter dem rechten Ende des Griffs zusätzlich betrachtet wird.

Tabellenbasierter LR(1)-Parser

Generelle Struktur:



- Tabellen können per Hand erzeugt werden...
- Erzeugung lässt sich allerdings perfekt automatisieren!

Skelett des LR(1)-Parsers mit Tabellen (1)

Initialisierung:

```
stack.push(  $Z_0$  );
```

```
DFAstate s =  $s_0$ ;
```

```
stack.push(  $s_0$  );           // push initial state of handle-finding DFA
```

```
bool found = false;
```

```
tok = scanner.getToken();    // call scanner for very first token
```

Skelett des LR(1)-Parsers mit Tabellen (2)

Verarbeitung:

```

while ( !found )
  s = stack.top();           // get current state of handle-finding DFA
  if ( ACTION[ s, tok ] == „reduce( N → α )“ )
    stack.popnum( 2 * |α| );           // pop 2 * |α| symbols
    s = stack.top();           // get resulting state of handle-finding DFA
    stack.push( N );           // push LHS of handle PLUS novel
    stack.push( GOTO[ s, N ] ); // state of handle-finding DFA
  else ...
  
```

Skelett des LR(1)-Parsers mit Tabellen (3)

Verarbeitung: (Fortsetzung)

```

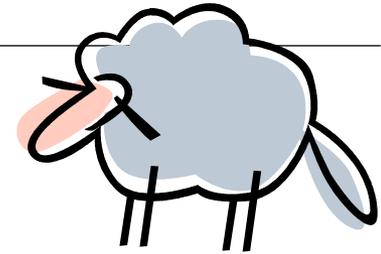
while ( !found )
    s = stack.top();           // get current state of handle-finding DFA
    if ( ACTION[ s, tok ] == „reduce(  $N \rightarrow \alpha$  )“ )
        <siehe vorige Folie>
    else if ( ACTION[ s, tok ] == „shift(  $s_i$  )“ )
        stack.push( tok );           // push current token PLUS
        stack.push(  $s_i$  );           // follow-up DFA state
        tok = scanner.getToken();    // call scanner for next token
    else ...
  
```

Skelett des LR(1)-Parsers mit Tabellen (4)

Verarbeitung: (Fortsetzung)

```

while ( !found )
    s = stack.top();           // get current state of handle-finding DFA
    if ( ACTION[ s, tok ] == „reduce(  $N \rightarrow \alpha$  )“ )
        <siehe vor-vorige Folie>
    else if ( ACTION[ s, tok ] == „shift(  $s_i$  )“ )
        <siehe vorige Folie>
    else if ( ( ACTION[ s, tok ] == „accept“ ) && ( tok == EOF ) )
        found = true;
    else error();
accept();
  
```



Beispiel: Schafsgeblöke

Grammatik G_5 : R1: $S \rightarrow \text{Blöken}$ R2: $\text{Blöken} \rightarrow \text{Blöken määäh}$
 R3: $\text{Blöken} \rightarrow \text{määäh}$

Stack	Eingabe	Aktion
$Z_0 s_0$	määäh määäh EOF	shift(s_2)
$Z_0 s_0 \text{määäh } s_2$	määäh EOF	reduce(R3)
$Z_0 s_0 \text{Blöken } s_1$	määäh EOF	shift(s_3)
$Z_0 s_0 \text{Blöken } s_1 \text{määäh } s_3$	EOF	reduce(R2)
$Z_0 s_0 \text{Blöken } s_1$	EOF	accept()

ACTION:

Zustand	EOF	määäh
s_0	-	shift(s_2)
s_1	accept()	shift(s_3)
s_2	reduce(R3)	reduce(R3)
s_3	reduce(R2)	reduce(R2)

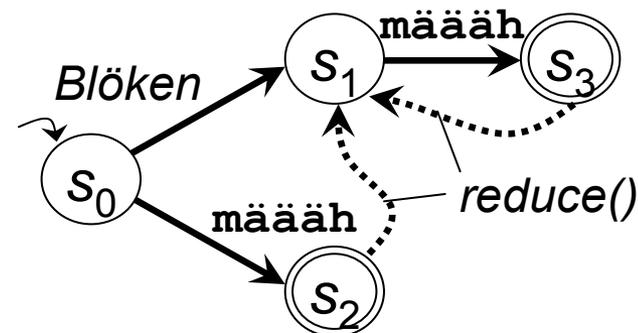
GOTO:

Zustand	Blöken
s_0	s_1
s_1	
s_2	
s_3	

Funktionsweise des LR(1)-Parsers

Eigenschaften:

- Griffe liegen stets unten auf Stack.
- ☞ Parser legt Token auf Stack, bis dessen unteres Ende Teil α des benötigten Griffs $\langle N \rightarrow \alpha, k \rangle$ entspricht.
- Finden von Griffen: durch DEA realisiert, dessen Verhalten in Tabellen ACTION und GOTO codiert ist.
- ☞ Akzeptierende Zustände des DEAs modellieren eine reduce-Aktion, d.h. es wurde ein benötigter Griff gefunden.
- DFA-Folgezustand \cong GOTO[\langle Zustand am unteren Stack-Ende nach pop von α \rangle, N]
- ☞ DEA für G_5 :



LR(1)-Items

Definition (*LR(1)-Item*):

Sei G eine kontextfreie Grammatik, $A \rightarrow \alpha\beta$ eine Produktionsregel.
 $[A \rightarrow \alpha\bullet\beta, T]$ heißt *LR(1)-Item* für $T \in V_T \cup \{ \text{EOF} \}$.

In Worten:

- • repräsentiert die Position des unteren Stack-Endes.
- T repräsentiert den look-ahead von einem Token bzw. dem EOF.
- $[A \rightarrow \alpha\bullet\beta, T]$: die bisher verarbeitete Eingabe stimmt mit der Anwendung von $A \rightarrow \alpha\beta$ überein, und es wurde bereits α erkannt.
- $[A \rightarrow \bullet\alpha\beta, T]$: die bisher verarbeitete Eingabe stimmt mit der Anwendung von $A \rightarrow \alpha\beta$ direkt am unteren Stack-Ende überein.
- $[A \rightarrow \alpha\beta\bullet, T]$: es wurde bisher $\alpha\beta$ erkannt, und das look-ahead Token T stimmt mit der Reduktion mit Hilfe von $A \rightarrow \alpha\beta$ überein.

LR(1)-Items & Auswahl von Regeln

Auswahl der korrekten Regel zur Reduktion:

- Zustände des DEAs zur Griff-Bestimmung sind Mengen von Items.
 - Look-ahead Token T wird in meisten möglichen DEA-Zuständen nicht benötigt und nur zur Buchführung mit verwaltet:
Kein direkter Nutzen von T in Item $[A \rightarrow \alpha \bullet \beta, T]$
 - Erst, wenn aktuelle Stack-Position am rechten Ende einer Regel steht, wird T benötigt:
Ist DEA im Zustand $[A \rightarrow \alpha \beta \bullet, T]$ UND ist aktuelles Eingabe-Token gleich dem look-ahead Token T , so ist $\alpha \beta$ ein benötigter Griff.
 - **Beispiel:** DEA sei in Zustand $\{ [A \rightarrow \alpha \bullet, T], [B \rightarrow \beta \bullet \gamma, U] \}$. Ist aktuelle Eingabe gleich T , so reduziere mit Regel $A \rightarrow \alpha$. Ist Eingabe in $\text{FIRST}(\gamma U)$, so führe shift-Aktion durch.
- ☞ T erlaubt Wahl der korrekten Regel aus mehreren Alternativen.

Berechnung der DEA-Zustände (1)

Vorgehensweise zur Bestimmung von Items & Zuständen:

1. Erweitere Grammatik $G = (V_N, V_T, P, S)$ um neues Startsymbol S' und neue Produktionsregel $S' \rightarrow S$.
(Motivation: Konstruktion des DEAs startet mit einer einzelnen Produktion $S' \rightarrow S$ und nicht mit evtl. vielen Regeln $S \rightarrow \dots$)
2. Setze initialen Zustand s_0 des DEAs auf das Item $[S' \rightarrow \bullet S, \text{EOF}]$ und alle dazu äquivalenten Items.
(Motivation: Zu Beginn hat der Kellerautomat einen leeren Stack, und die Eingabe ist ein kompletter, noch zu parsender, S-Satz, gefolgt von EOF)
3. Äquivalente Items zu s_0 werden als Hülle von s_0 bezeichnet und im folgenden durch Funktion $\text{closure}(s_0)$ berechnet.

Berechnung der DEA-Zustände (2)

Vorgehensweise zur Bestimmung von Items & Zuständen:

4. Wiederhole, so lange sich die Zustandsmenge ändert, für jeden Zustand s_j und jedes $X \in (V_N \cup V_T)$:
5. Berechne den Folgezustand (d.h. die Menge von Items)
 $s_j = goto(s_j, X)$
6. Füge s_j zur Zustandsmenge des DEA hinzu, falls s_j noch nicht enthalten ist.
7. Speichere alle Zustandsübergänge, die durch $goto()$ bestimmt wurden.
8. Überführe Zustandsmenge und -übergänge in Tabellenform und erzeuge ACTION- und GOTO-Tabellen.

Berechnung der DEA-Zustände (3)

Gegeben:

- Kontextfreie Grammatik $G = (V_N, V_T, P, S')$

Initialisierung:

$s_0 = \text{closure}(S' \rightarrow \bullet S, \text{EOF});$

$S = \{ s_0 \};$

$j = 1;$

Berechnung der DEA-Zustände (4)

Algorithmus:

```

while ( <S wurde verändert> )
  for ( <alle Zustände  $s_i \in S$ > )
    for ( <alle Terminale und Nichtterminale  $X \in (V_N \cup V_T)$ > )
       $s_j = goto( s_i, X );$ 
      if (  $s_j \notin S$  )
         $S = S \cup s_j;$ 
        bool increment = true;

Füge Zustandsübergang  $s_i \rightarrow s_j$  unter Symbol  $X$  zum
DEA hinzu.
if ( increment )
   $j = j + 1;$ 
  
```

Berechnung von Hüllen

Hintergrund:

- Für eine Menge s von Items berechnet $closure(s)$ alle Items, die durch s impliziert werden.
- Sei $B \in V_N$. Ein Item $[A \rightarrow \alpha \bullet B \beta, T]$ impliziert $[B \rightarrow \bullet \gamma, X]$ für jede Produktion mit B auf linker Seite, und für jedes $X \in FIRST[\beta T]$.

Algorithmus: $closure(s)$

do

 for (<alle Items $[A \rightarrow \alpha \bullet B \beta, T] \in s$ >)

 for (<alle Regeln $B \rightarrow \gamma \in P$ >)

 for (<alle $X \in FIRST[\beta T]$ >) // β könnte gleich ε sein

$s = s \cup [B \rightarrow \bullet \gamma, X]$

while (< s wurde verändert>)

Berechnung von Folgezuständen

Hintergrund:

- $goto(s, X)$ berechnet den Folgezustand des Parsers, wenn er in Zustand s ist und X erkennt.
- $goto([A \rightarrow \alpha \bullet X \beta, T], X)$ produziert $[A \rightarrow \alpha X \bullet \beta, T]$
- Zusätzlich: alle äquivalenten Items sind ebenfalls enthalten, d.h. $closure([A \rightarrow \alpha X \bullet \beta, T])$.

Algorithmus: $goto(s, X)$

$s' = \emptyset$;

for (<alle Items $[A \rightarrow \alpha \bullet X \beta, T] \in s$ >)

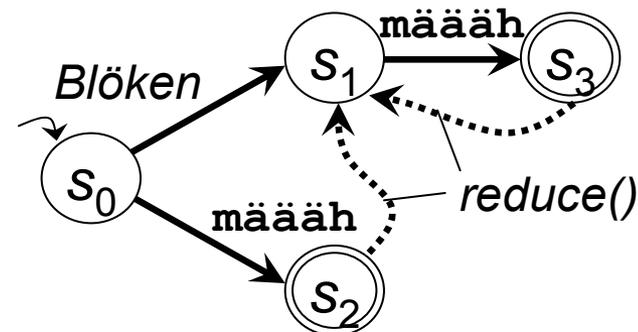
$s' = s' \cup [A \rightarrow \alpha X \bullet \beta, T]$;

return $closure(s')$;

Beispiel: Schafsgeblöke

Algorithmus zur Bestimmung der DEA-Zustände:

- Initialisierung:** $s_0 = \{ [S \rightarrow \bullet \text{Blöken}, \text{EOF}], [\text{Blöken} \rightarrow \bullet \text{Blöken määäh}, \text{EOF}], [\text{Blöken} \rightarrow \bullet \text{määäh}, \text{EOF}], [\text{Blöken} \rightarrow \bullet \text{Blöken määäh}, \text{määäh}], [\text{Blöken} \rightarrow \bullet \text{määäh}, \text{määäh}] \}$
- Iteration 1:** $s_1 = \text{goto}(s_0, \text{Blöken}) = \{ [S \rightarrow \text{Blöken} \bullet, \text{EOF}], [\text{Blöken} \rightarrow \text{Blöken} \bullet \text{määäh}, \text{EOF}], [\text{Blöken} \rightarrow \text{Blöken} \bullet \text{määäh}, \text{määäh}] \}$
 $s_2 = \text{goto}(s_0, \text{määäh}) = \{ [\text{Blöken} \rightarrow \text{määäh} \bullet, \text{EOF}], [\text{Blöken} \rightarrow \text{määäh} \bullet, \text{määäh}] \}$
- Iteration 2:** $s_3 = \text{goto}(s_1, \text{määäh}) = \{ [\text{Blöken} \rightarrow \text{Blöken määäh} \bullet, \text{EOF}], [\text{Blöken} \rightarrow \text{Blöken määäh} \bullet, \text{määäh}] \}$



Berechnung der ACTION- & GOTO-Tabellen

Algorithmus:

```

for ( <alle Zustände  $s_j \in S$ > )
  for ( <alle Items  $i \in s_j$ > )
    if ( (  $i == [A \rightarrow \alpha \bullet T \beta, X]$  für  $T \in V_T$  ) && (  $goto(s_j, T) == s_k$  ) )
      ACTION[  $s_j, T$  ] = shift(  $s_k$  );
    else if (  $i == [S' \rightarrow S \bullet, EOF]$  )
      ACTION[  $s_j, EOF$  ] = accept;
    else if (  $i == [A \rightarrow \alpha \bullet, X]$  )
      ACTION[  $s_j, X$  ] = reduce(  $A \rightarrow \alpha$  );
  for ( <alle Nichtterminale  $N \in V_N$ > )
    if (  $goto(s_j, N) == s_k$  )
      GOTO[  $s_j, N$  ] =  $s_k$ ;
  
```

Zusammenfassung: Bottom-Up Analyse

Vorgehensweise zum Schreiben eines Bottom-Up Parsers:

Gegeben: Kontextfreie Grammatik G

1. Berechne FIRST-Mengen (☞ [Folie 43](#))
2. Konstruiere DEA zum Finden von Griffen (☞ [Folie 99](#))
3. Verwende den DEA von Schritt 2 zum Erzeugen von ACTION- und GOTO-Tabellen (☞ [Folie 104](#))
4. Konstruiere Tabellen-basierten Parser, der wie ein Kellerautomat funktioniert und auf seinem Stack Griffen und Zustände des DEAs von Schritt 2 ablegt (☞ [Folie 89](#))

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
 - Einführung
 - Top-Down Syntax-Analyse
 - Bottom-Up Syntax-Analyse
 - **Praktische Umsetzung: BISON**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Parser-Generierung

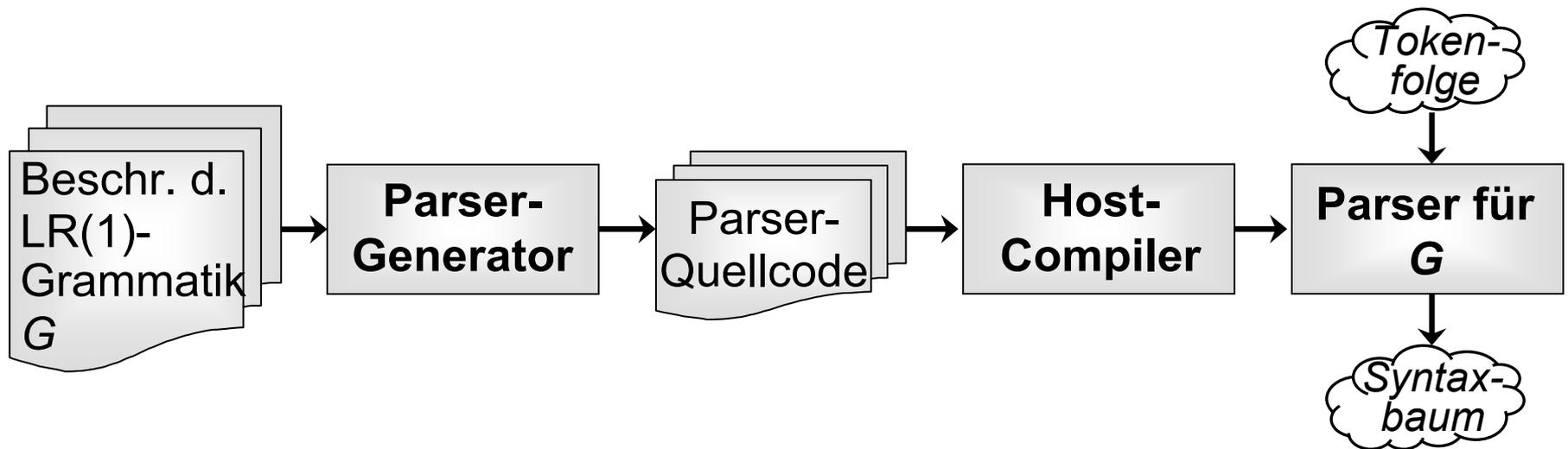
Implementierung eines Parsers:

- *Situation wie bei Scannern:*
Theoretische Grundlagen der syntaktischen Analyse bestens erforscht, effiziente Algorithmen zur Erzeugung von Parsern auf Basis kontextfreier LR(1)-Grammatiken existieren.
 - Per-Hand-Implementierung eines Parsers bei vorhandenem theoretischen Unterbau nicht mehr vertretbar.
(Und auch nicht gewünscht, da diese Aufgabe schlichtweg zu technisch und „langweilig“ ist...)
- 👉 Statt dessen: Verwendung sog. *Parser-Generatoren*



Parser-Generatoren

- Ein *Parser-Generator* erhält eine Beschreibung der zu erkennenden LR(1)-Grammatik als Eingabe und erzeugt daraus einen Parser für eben diese Grammatik.



Parser-Generierung mit BISON (1)

Grundlegender Aufbau einer BISON Parser-Beschreibung

- BISON: Weit verbreiteter Parser-Generator, GNU open source
- Generiert aus Beschreibung C-Code eines Parsers
- Eingabedatei für BISON:

```

%{          /* Präambel */
%}

Deklarationen

%%

C-Funktionen
  
```



The diagram illustrates the structure of a Bison parser description. A red dashed line encloses the preambles, declarations, and rules sections, indicating they are part of the parser's internal state. Another red dashed line connects the closing brace '}' to the C-functions section, showing the transition from the parser's internal state to the generated C code.

Parser-Generierung mit BISON (2)

Präambel (*File parser.y für dieses Beispiel*)

- Wie bei FLEX ( Kapitel 3)
- Eine Präambel enthält beliebigen C-Code, der unverändert an den Beginn des von BISON erzeugten Codes kopiert wird:

```

%{
    #include <stdio.h>

    #include "parser.h" /* Include token declarations */

    extern unsigned int lineNumber; /* Line counter */

    void yyerror( char * ); /* Error handler */
}%
  
```

Parser-Generierung mit BISON (3)

Deklarationen

- Deklarationen umfassen zumindest die Angabe der Terminale und des Startsymbols der Grammatik.
- Nichtterminale brauchen nicht explizit deklariert zu werden, da BISON diese aus den Produktionsregeln und den Informationen zu deklarierten Terminalen generiert:

```
%token tID tWHILE tBEGIN tEND tDO tIF tTHEN
```

```
%token tELSE tSEMI tASSIGN
```

```
%start prog
```

Parser-Generierung mit BISON (4)

Regeln

- Regeln bestehen aus der Spezifikation der Produktionsregeln der zu parsenden Grammatik sowie dem C-Code einer optionalen *semantischen Aktion* in geschweiften Klammern { }.
- Da semantische Aktionen wichtig zur semantischen Analyse sind (☞ *Kapitel 5*), werden diese hier weggelassen.

Parser-Generierung mit BISON (5)

Regeln (*Fortsetzung*)

```
prog : stmlist;
```

```
stm : tID tASSIGN tID  
    | tWHILE tID tDO stm  
    | tBEGIN stmlist tEND  
    | tIF tID tTHEN stm  
    | tIF tID tTHEN stm tELSE stm;
```

```
stmlist : stm  
        | stmlist tSEMI stm;
```

Parser-Generierung mit BISON (6)

C-Funktionen

- Beliebige C-Funktionen können für Hilfszwecke angegeben werden.
- Hilfsfunktionen können innerhalb von semantischen Aktionen aufgerufen werden (hier nicht gezeigt).

```

/* Emit some error message. */
void yyerror( char *s ) {
    fprintf( stderr,
            "%s near line %ld.", s, lineNumber ); }

/* A complete C program needs a main function. */
int main() {
    return yyparse(); }           /* Invoke parser. */
  
```

Parser-Generierung mit BISON (7)

FLEX-basierter Scanner (*File scanner.1* für dieses Beispiel)

```
%{
    #include "parser.h"
    unsigned int lineNumber = 1;
    void incrementLineNumber();
}%

digit [0-9]
letter [a-zA-Z]
identifier {letter}({letter}|{digit})*

%%

while          return tWHILE;
begin          return tBEGIN;
end            return tEND;
```

Parser-Generierung mit BISON (8)

FLEX-basierter Scanner (*Fortsetzung*)

```

do          return tDO;
if          return tIF;
then       return tTHEN;
else       return tELSE;
";"       return tSEMI;
"="       return tASSIGN;
{identif} return tID;
[ \t]     /* Consumes white spaces & tabs. */
\n       incrementLineCounter();

%%

void incrementLineCounter() {
    lineNumber++; }
  
```

Parser-Generierung mit BISON (9)

Übersetzung von Scanner und Parser:

```
bison -v -d -o parser.c parser.y
```

```
flex -oscanner.c scanner.l
```

```
gcc -c parser.c
```

```
gcc -c scanner.o
```

```
gcc parser.o scanner.o -o parser -lfl
```

Aufruf von kombiniertem Scanner und Parser:

- Zu parsendes Programm in Eingabedatei schreiben.
- `./parser < eingabedatei`

Literatur

Top-Down Syntax-Analyse:

- Andrew W. Appel, *Modern compiler implementation in C*,
Cambridge University Press, 1998. (☞ Kapitel 3.2)
ISBN 0-521-58390-X
- J. Eliot B. Moss, *Compiler Techniques – Fall 2008*, 2008.
(☞ Foliensätze 7-9)
<http://www-ali.cs.umass.edu/~moss/610.html>

Literatur

Bottom-Up Syntax-Analyse:

- Andrew W. Appel, *Modern compiler implementation in C*,
Cambridge University Press, 1998. (☞ Kapitel 3.3)
ISBN 0-521-58390-X
- J. Eliot B. Moss, *Compiler Techniques – Fall 2008*, 2008.
(☞ Foliensätze 10-13)
<http://www-ali.cs.umass.edu/~moss/610.html>
- Reinhard Wilhelm, Dieter Maurer, *Übersetzerbau*, 2. Auflage,
Springer, 1997. (☞ Kapitel 8.4)
ISBN 3-540-61692-6

Zusammenfassung

- **Kontextfreie Grammatiken: Eleganter Mechanismus, um Programmiersprachen zu spezifizieren.**
- **Top-Down-, recursive descent- bzw. predictive parsers leicht für LL(1)-Grammatiken per Hand zu implementieren.**
- **LL(1)-Grammatiken für Programmiersprachen u.U. nicht mächtig genug.**
- **Bottom-Up Parser für LR(1)-Grammatiken sind mächtiger, da Stack-Inhalt und look-ahead zum Parsen ausgenutzt werden können.**
- **BISON: Ein Standard-Werkzeug zur Generierung von LR(1)-Parsern.**

