Compilerbau

Wintersemester 2009 / 2010

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

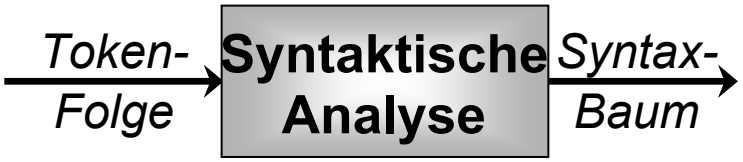
Kapitel 4

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: $V_N \cap V_T = \varnothing$

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

$S \rightarrow$ `id :=` E

$E \rightarrow$ `id`

$E \rightarrow$ `num`

$E \rightarrow E$ `*` E

$E \rightarrow E$ `+` E

$E \rightarrow E$ `-` E

- *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

$S \rightarrow$ `id :=` E

$E \rightarrow$ `id`

$E \rightarrow$ `num`

$E \rightarrow E$ `*` E

$E \rightarrow E$ `+` E

$E \rightarrow E$ `-` E

- 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`

$\underline{S} \rightarrow$

`id :=` $\underline{E} \rightarrow$

`id :=` E `+` $\underline{E} \rightarrow$

`id :=` $\underline{E}$ `+` E `*` $E \rightarrow$

`id := id +` $\underline{E}$ `*` $E \rightarrow$

`id := id + id *` $\underline{E} \rightarrow$

`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:

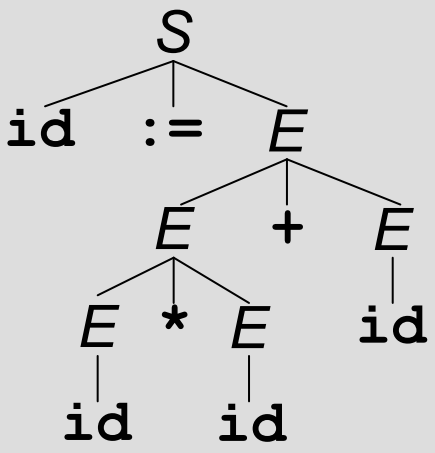
- Für einen inneren Knoten markiert mit $A \in V_N$ sind seine Kinder von links nach rechts markiert mit $N_1, N_2, \ldots, N_k \in V_N \cup V_T$ und P enthält eine Regel $A \rightarrow N_1 N_2 \ldots N_k$, oder A's einziges Kind ist markiert mit ε und $A \rightarrow \varepsilon$ ist eine Regel aus P.
- 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.

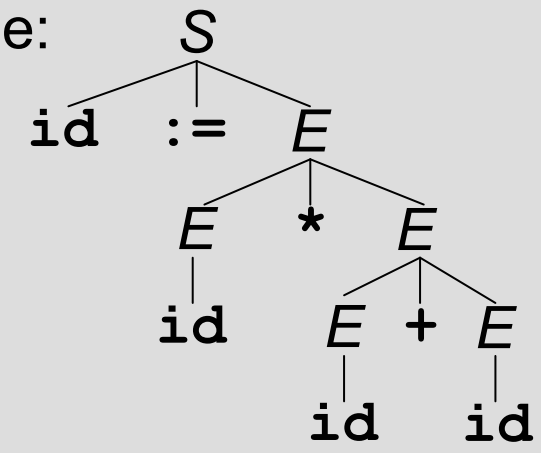
Syntaxbäume (2)

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*.

Eindeutige Grammatik

Beispiel: Grammatik G_2

$S \to$ `id :=` E

$E \to E$ `+` T | $E \to E$ `-` T | $E \to T$

$T \to T$ `*` F | $T \to F$

$F \to$ `id` | $F \to$ `num` | $F \to$ `(` E `)`

- Einfügen neuer Nichtterminale T und F
- Bedeutung: S – Statements, E – Expressions, T – Terme, F – Faktoren
- Dadurch, dass `+` und `*` in G_2 nun nur aus verschiedenen Nichtterminalen abgeleitet werden können, ist G_2 eindeutig:
- ☞ Die Interpretation `id := id * ( id + id )` für die Tokenfolge `id := id * id + id` ist in G_2 nicht mehr möglich.

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 rechen Seite von p: Erzeuge einen Kind-Knoten für v.
  3. Entspricht aktuelles Wort der Eingabe nicht mehr dem Saum von B: Backtracking!

***Hauptproblem:* Wahl der passenden Regel in Schritt 1!**

Beispiel (1)

Grammatik G_2

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T | $E \rightarrow E$ `-` T | $E \rightarrow T$

$T \rightarrow T$ `*` F | $T \rightarrow F$

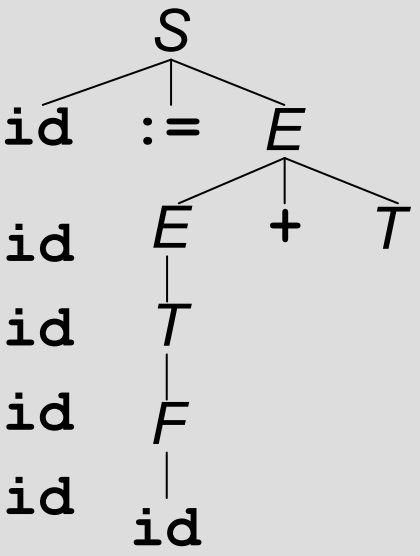
$F \rightarrow$ `id` | $F \rightarrow$ `num` | $F \rightarrow$ `(` E `)`

- 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 ↑ markiert.

Beispiel (2)

Eingabe: `id := id – num * id`

	Saum	Eingabe
1.	$\underline{S}$	↑`id := id – num * id`
2.	`id :=` E	↑`id := id – num * id`
3.	`id :=` E	`id` ↑`:= id – num * id`
4.	`id :=` $\underline{E}$	`id :=` ↑`id – num * id`
5.	`id :=` $\underline{E}$ `+` T	`id :=` ↑`id – num * id`
6.	`id :=` $\underline{T}$ `+` T	`id :=` ↑`id – num * id`
7.	`id :=` $\underline{F}$ `+` T	`id :=` ↑`id – num * id`
8.	`id := id +` T	`id :=` ↑`id – num * id`
9.	`id := id +` T	`id := id` ↑`– num * id`

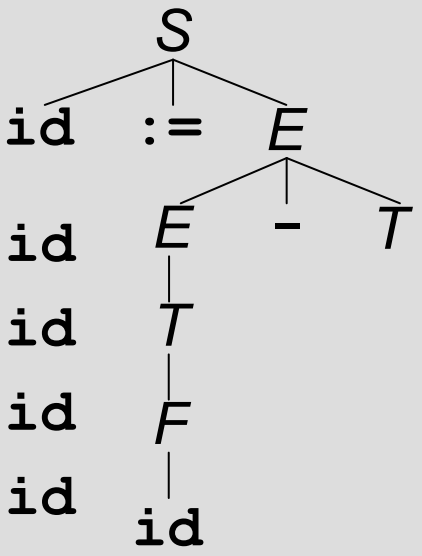


+ (Saum) und **–** (Eingabe) passen nicht: *Zurück zu Schritt 4!*

Beispiel (3)

Eingabe: `id := id – num * id`

	Saum	Eingabe
1.	$\underline{S}$	↑`id := id – num * id`
2.	`id :=` E	↑`id := id – num * id`
3.	`id :=` E	`id` ↑`:= id – num * id`
4.	`id :=` $\underline{E}$	`id :=` ↑`id – num * id`
5.	`id :=` $\underline{E}$ `–` T	`id :=` ↑`id – num * id`
6.	`id :=` $\underline{T}$ `–` T	`id :=` ↑`id – num * id`
7.	`id :=` $\underline{F}$ `–` T	`id :=` ↑`id – num * id`
8.	`id := id –` T	`id :=` ↑`id – num * id`
9.	`id := id –` T	`id := id` ↑`– num * id`

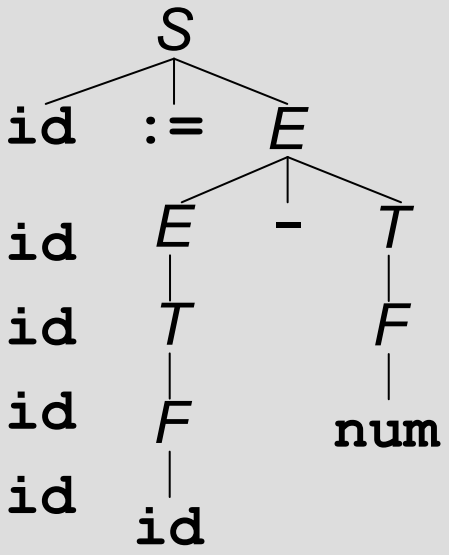


– (Saum) und
– (Eingabe)
passen:
Weiter mit Eingabe!

Beispiel (4)

Eingabe: `id := id – num * id`

	Saum	Eingabe
10.	`id := id –` $\underline{T}$	`id := id –` ↑`num * id`
11.	`id := id –` $\underline{F}$	`id := id –` ↑`num * id`
12.	`id := id – num`	`id := id –` ↑`num * id`
13.	`id := id – num`	`id := id – num` ↑`* id`

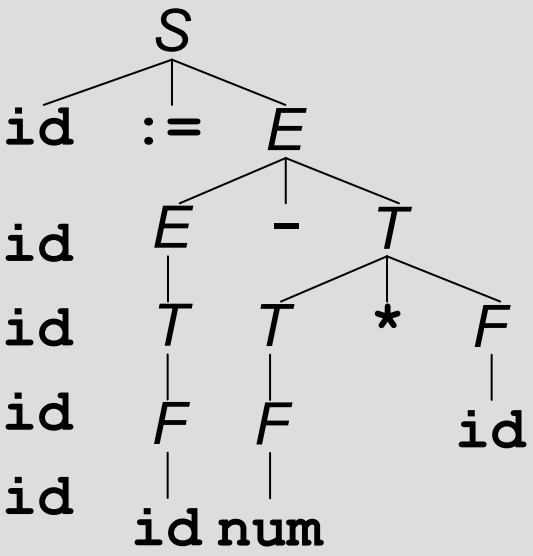


num aus Saum und **num** aus Eingabe passen, aber es gibt keine weiteren Nichtterminale für restliche Eingabe: *Zurück zu Schritt 10!*

Beispiel (5)

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

	Saum	Eingabe
10.	`id := id -` *T*	`id := id -` ↑`num * id`
11.	`id := id -` *T* `* F`	`id := id -` ↑`num * id`
12.	`id := id -` *F* `* F`	`id := id -` ↑`num * id`
13.	`id := id - num * F`	`id := id -` ↑`num * id`
14.	`id := id - num * F`	`id := id - num` ↑`* id`
15.	`id := id - num *` *F*	`id := id - num *` ↑`id`
16.	`id := id - num * id`	`id := id - num *` ↑`id`
17.	`id := id - num * id`	`id := id - num * id`↑



Eingabe passt:
Fertig!

Viel schlimmeres Beispiel: Endlos-Parser!

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

	Saum	Eingabe
1.	$\underline{S}$	↑`id := id - num * id`
2.	`id :=` $\underline{E}$	↑`id := id - num * id`
3.	`id :=` E	`id` ↑`:= id - num * id`
4.	`id :=` $\underline{E}$	`id :=` ↑`id - num * id`
5.	`id :=` $\underline{E}$ `+` T	`id :=` ↑`id - num * id`
6.	`id :=` $\underline{E}$ `+` T `+` T	`id :=` ↑`id - num * id`
7.	`id :=` $\underline{E}$ `+` T `+` T `+` T	`id :=` ↑`id - num * id`
...		
∞	`id :=` $\underline{E}$ `+` T `+` ... `+` T	`id :=` ↑`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: $\alpha \neq \varnothing$ und α beginnt nicht mit N)

☞ Umwandlung der Linksrekursivität in Rechtsrekursivität.

Eliminierung von Linksrekursion (1)

Grammatik G_2

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T | $E \rightarrow E$ `-` T | $E \rightarrow T$

$T \rightarrow T$ `*` F | $T \rightarrow F$

$F \rightarrow$ `id` | $F \rightarrow$ `num` | $F \rightarrow$ `(` E `)`

- Offensichtlich: Folgende Produktionsregeln sind linksrekursiv:
  $E \rightarrow E$ `+` T $\quad$ $E \rightarrow E$ `-` T $\quad$ $T \rightarrow T$ `*` 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 ⇒ $E \rightarrow T\ E'$ $\quad$ $E' \rightarrow$ `+` $T\ E'$ $\quad$ $E' \rightarrow$

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

$S \rightarrow$ `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 `)`

Grammatik G_3

$S \rightarrow$ `id :=` E	$E \rightarrow T\ E'$	$T \rightarrow F\ T'$	$F \rightarrow$ `id`
			$F \rightarrow$ `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

$S \rightarrow$ **`if`** E **`then`** S **`else`** S

$S \rightarrow$ **`begin`** $S\ L$

$S \rightarrow$ **`print`** E

$L \rightarrow$ **`end`**

$L \rightarrow$ **`;`** $S\ L$

$E \rightarrow$ **`num`** = **`num`**

Grundlegender Aufbau eines solchen Parsers:

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

Parsen mit Rekursivem Abstieg (2)

Beispiel: Grammatik G_4

$S \rightarrow$ **if** E **then** S **else** S

$S \rightarrow$ **begin** $S\ L$

$S \rightarrow$ **print** E

$L \rightarrow$ **end**

$L \rightarrow$ **;** $S\ L$

$E \rightarrow$ **num** = **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

$S \rightarrow$ `if` E `then` S `else` S

$S \rightarrow$ `begin` $S\ L$

$S \rightarrow$ `print` E

$L \rightarrow$ `end`

$L \rightarrow$ `;` $S\ L$

$E \rightarrow$ `num = 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

$S \rightarrow$ **if** E **then** S **else** S

$S \rightarrow$ **begin** $S\ L$

$S \rightarrow$ **print** E

$L \rightarrow$ **end**

$L \rightarrow$ **;** $S\ L$

$E \rightarrow$ **num** = **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

$S \rightarrow$ `if` E `then` S `else` S

$S \rightarrow$ `begin` $S\ L$

$S \rightarrow$ `print` E

$L \rightarrow$ `end`

$L \rightarrow$ `;` $S\ L$

$E \rightarrow$ `num` = `num`

Nichtterminal *E*:

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

Parsen mit Rekursivem Abstieg (6)

Beispiel: Grammatik G_4

$S \rightarrow$ **`if`** E **`then`** S **`else`** S

$S \rightarrow$ **`begin`** $S\ L$

$S \rightarrow$ **`print`** E

$L \rightarrow$ **`end`**

$L \rightarrow$ **`;`** $S\ L$

$E \rightarrow$ **`num`** = **`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:

$S \rightarrow$ `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 `)`

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 Fallunterscheidung 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 **`id`**α^*, **`num`**β^* oder **`(`**γ^* 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 FIRST(α) die Menge aller Terminale, die beliebige, aus α ableitbaren Sätze beginnen können.

Beispiel zu G_2:

- FIRST(*E* **+** *T*) = { **`id`**, **`num`**, **`(`** }
- Zusätzlich: FIRST(*E* **–** *T*) = { **`id`**, **`num`**, **`(`** }

☞ D.h. FIRST(*E* **+** *T*) und FIRST(*E* **–** *T*) enthalten für Nichtterminal *E* gemeinsame Terminale.

☞ Problem von G_2: FIRST(*E* **+** *T*) $\cap$ FIRST(*E* **–** *T*) $\neq \emptyset$

Neutralisierbare Nichtterminale

Berechnung der FIRST-Mengen:

- Scheinbar einfach: 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?

☞ FIRST($X\ Y\ Z$) hängt nun von Z ab.

- Falls zusätzlich Z neutralisierbar: 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: $N \rightarrow^* \varepsilon$.

FOLLOW-Mengen

Definition *(FOLLOW-Menge)*:

Für ein Nichtterminal N ist 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:

- 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) = $\varnothing$;
  - neutralisierbar(N) = false;
- for (<alle Terminale $T \in V_T$>)
  - FIRST(T) = { T };

Berechnung von FIRST und FOLLOW (2)

Algorithmus:

- while (<FIRST, FOLLOW und neutralisierbar wurden verändert>)
  - for (<alle Produktionsregeln $p \in P$: $N \rightarrow Y_1\ Y_2\ \ldots\ Y_k$>)
    - if ($\forall\ l = 1\ \ldots\ k$: neutralisierbar(Y_l) == true)
      - neutralisierbar(N) = true;
    - for (<alle i von 1 bis k und alle j von i+1 bis k>)
      - if ($\forall\ l = 1\ \ldots\ i$-1: neutralisierbar(Y_l) == true)
        - FIRST(N) = FIRST(N) U FIRST(Y_i);
      - if ($\forall\ l = i$+1 $\ldots\ k$: neutralisierbar(Y_l) == true)
        - FOLLOW(Y_i) = FOLLOW(Y_i) U FOLLOW(N);
      - if ($\forall\ l = i$+1 $\ldots\ j$-1: neutralisierbar(Y_l) == true)
        - 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

FIRST / FOLLOW und Parser-Tabellen

Konstruktion der Parser-Tabelle:

Eine Produktionsregel $N \rightarrow \alpha$ mit $\alpha \in (V_N \cup V_T)^*$

- ist in Zeile N und Spalte T für alle $T \in$ FIRST[α] einzutragen UND
- ist in Zeile N und Spalte T für alle $T \in$ FOLLOW[N] einzutragen, falls jedes Wort in α neutralisierbar ist.

Randnotiz:

Obige Konstruktionsvorschrift benutzt FIRST[α] für ganze Folgen von Worten, während Algorithmus FIRST-Mengen nur pro Terminal bzw. Nichtterminal berechnet. *Verallgemeinerung:*

FIRST[$X\beta$] = FIRST[X] falls X nicht neutralisierbar

FIRST[$X\beta$] = FIRST[X] ∪ FIRST[β] sonst

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{$ `id`$_1$ $\}$

$E = \{$ $\}$

$T = \{$ $\}$

$F = \{$ $\}$

FOLLOW:

$S = \{$ $\}$

$E = \{$ $\}$

$T = \{$ $\}$

$F = \{$ $\}$

neutralisierbar:

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

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T

$E \rightarrow E$ `-` T

$E \rightarrow T$

$T \rightarrow T$ `*` F

$T \rightarrow F$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

S = { `id`$_1$ }

E = { }

T = { }

F = { }

FOLLOW:

S = { }

E = { `+`$_2$ }

T = { }

F = { }

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T

$E \rightarrow E$ `-` T

$E \rightarrow T$

$T \rightarrow T$ `*` F

$T \rightarrow F$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{$ **id**$_1$	$\}$	$S \rightarrow$ **id** $:=$ E
	$E = \{$	$\}$	$E \rightarrow E$ **+** T
	$T = \{$	$\}$	⇒ $E \rightarrow E$ **−** T
	$F = \{$	$\}$	$E \rightarrow T$
FOLLOW:	$S = \{$	$\}$	$T \rightarrow T$ ***** F
	$E = \{$**+**$_2$ **−**$_4$	$\}$	$T \rightarrow F$
	$T = \{$**+**$_3$	$\}$	$F \rightarrow$ **id**
	$F = \{$	$\}$	$F \rightarrow$ **num**
			$F \rightarrow$ **(** E **)**

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{$ `id`$_1$ $\}$

$E = \{$ $\}$

$T = \{$ $\}$

$F = \{$ $\}$

FOLLOW:

$S = \{$ $\}$

$E = \{$ `+`$_2$ `-`$_4$ $\}$

$T = \{$ `+`$_3$ `-`$_5$ $\}$

$F = \{$ $\}$

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T

$E \rightarrow E$ `-` T

$E \rightarrow T$

$T \rightarrow T$ `*` F

$T \rightarrow F$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{\texttt{id}_1$	$\}$	$S \rightarrow \texttt{id} := E$
	$E = \{$	$\}$	$E \rightarrow E + T$
	$T = \{$	$\}$	$E \rightarrow E - T$
	$F = \{$	$\}$	$E \rightarrow T$
FOLLOW:	$S = \{$	$\}$	⇒ $T \rightarrow T * F$
	$E = \{+_2 \ -_4$	$\}$	$T \rightarrow F$
	$T = \{+_3 \ -_5 \ *_7$	$\}$	$F \rightarrow \texttt{id}$
	$F = \{+_6 \ -_6$	$\}$	$F \rightarrow \texttt{num}$
			$F \rightarrow \texttt{(}\ E\ \texttt{)}$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{$ `id`$_1$ $\}$

$E = \{$ $\}$

$T = \{$ $\}$

$F = \{$ $\}$

FOLLOW:

$S = \{$ $\}$

$E = \{$ `+`$_2$ `-`$_4$ $\}$

$T = \{$ `+`$_3$ `-`$_5$ `*`$_7$ $\}$

$F = \{$ `+`$_6$ `-`$_6$ `*`$_8$ $\}$

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T

$E \rightarrow E$ `-` T

$E \rightarrow T$

$T \rightarrow T$ `*` F

$T \rightarrow F$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{\texttt{id}_1$	$\}$	$S \to \texttt{id} := E$
	$E = \{$	$\}$	$E \to E + T$
	$T = \{$	$\}$	$E \to E - T$
	$F = \{\texttt{id}_9$	$\}$	$E \to T$
FOLLOW:	$S = \{$	$\}$	$T \to T * F$
	$E = \{+_2 \ -_4$	$\}$	$T \to F$
	$T = \{+_3 \ -_5 \ *_7$	$\}$	⇒ $F \to \texttt{id}$
	$F = \{+_6 \ -_6 \ *_8$	$\}$	$F \to \texttt{num}$
			$F \to \texttt{(}\ E\ \texttt{)}$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:	$S = \{$ **id**$_1$	$\}$	$S \rightarrow$ **id** := E
	$E = \{$	$\}$	$E \rightarrow E$ **+** T
	$T = \{$	$\}$	$E \rightarrow E$ **-** T
	$F = \{$ **id**$_9$ **num**$_{10}$	$\}$	$E \rightarrow T$
FOLLOW:	$S = \{$	$\}$	$T \rightarrow T$ ***** F
	$E = \{$ **+**$_2$ **-**$_4$	$\}$	$T \rightarrow F$
	$T = \{$ **+**$_3$ **-**$_5$ *****$_7$	$\}$	$F \rightarrow$ **id**
	$F = \{$ **+**$_6$ **-**$_6$ *****$_8$	$\}$	$F \rightarrow$ **num**
			$F \rightarrow$ **(** E **)**

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

S = { **id**$_1$ }

E = { }

T = { }

F = { **id**$_9$ **num**$_{10}$ **(**$_{11}$ }

FOLLOW:

S = { }

E = { **+**$_2$ **-**$_4$ **)**$_{12}$ }

T = { **+**$_3$ **-**$_5$ *****$_7$ }

F = { **+**$_6$ **-**$_6$ *****$_8$ }

$S \rightarrow$ **id :=** E

$E \rightarrow E$ **+** T

$E \rightarrow E$ **-** T

$E \rightarrow T$

$T \rightarrow T$ ***** F

$T \rightarrow F$

$F \rightarrow$ **id**

$F \rightarrow$ **num**

⇒ $F \rightarrow$ **(** E **)**

Iteration 1 des Algorithmus: Schritte 1 bis 12

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{$ **id**$_1$ $\}$

$E = \{$ $\}$

$T = \{$ $\}$

$F = \{$ **id**$_9$ **num**$_{10}$ (**$_{11}$ $\}$

FOLLOW:

$S = \{$ $\}$

$E = \{$ +$_2$ -$_4$)$_{12}$ $\}$

$T = \{$ +$_3$ -$_5$ *$_7$)$_{13}$ $\}$

$F = \{$ +$_6$ -$_6$ *$_8$ $\}$

$S \rightarrow$ **id** := E

$E \rightarrow E$ + T

$E \rightarrow E$ - T

$E \rightarrow T$

$T \rightarrow T$ * F

$T \rightarrow F$

$F \rightarrow$ **id**

$F \rightarrow$ **num**

$F \rightarrow$ (E)

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{\texttt{id}_1 \}$

$E = \{ \}$

$T = \{ \}$

$F = \{\texttt{id}_9 \;\; \texttt{num}_{10} \;\; (_{11} \}$

FOLLOW:

$S = \{ \}$

$E = \{+_2 \;\; -_4 \;\;)_{12} \}$

$T = \{+_3 \;\; -_5 \;\; *_7 \;\;)_{13} \}$

$F = \{+_6 \;\; -_6 \;\; *_8 \;\;)_{14} \}$

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

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow \texttt{id}$

$F \rightarrow \texttt{num}$

$F \rightarrow (E)$

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{\texttt{id}_1$ }

$E = \{$ }

$T = \{\texttt{id}_{15}\ \texttt{num}_{15}\ (_{15}$ }

$F = \{\texttt{id}_9\ \texttt{num}_{10}\ (_{11}$ }

FOLLOW:

$S = \{$ }

$E = \{+_2\ -_4\ )_{12}$ }

$T = \{+_3\ -_5\ *_7\ )_{13}$ }

$F = \{+_6\ -_6\ *_8\ )_{14}$ }

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

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

⇒ $T \rightarrow F$

$F \rightarrow \texttt{id}$

$F \rightarrow \texttt{num}$

$F \rightarrow \texttt{(}\ E\ \texttt{)}$

Iteration 2 des Algorithmus: Schritte 13 bis 15

Beispiel zu FIRST / FOLLOW-Mengen: G_2

FIRST:

$S = \{\texttt{id}_1 \}$

$E = \{\texttt{id}_{16}\ \texttt{num}_{16}\ (_{16} \}$

$T = \{\texttt{id}_{15}\ \texttt{num}_{15}\ (_{15} \}$

$F = \{\texttt{id}_{9}\ \texttt{num}_{10}\ (_{11} \}$

FOLLOW:

$S = \{ \}$

$E = \{+_2\ -_4\ )_{12} \}$

$T = \{+_3\ -_5\ *_7\ )_{13} \}$

$F = \{+_6\ -_6\ *_8\ )_{14} \}$

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

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow \texttt{id}$

$F \rightarrow \texttt{num}$

$F \rightarrow (\ E\ )$

Iteration 3 des Algorithmus: Schritt 16

Iteration 4 des Algorithmus: Verändert Mengen nicht – Stopp

Beispiel: Parser-Tabelle zu G_2

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

$S = \{\texttt{id}_1$ }
$E = \{\texttt{id}_{20}\ \texttt{num}_{20}\ (_{20}$ }
$E' = \{+_2\ -_4$ }
$T = \{\texttt{id}_{17}\ \texttt{num}_{17}\ (_{17}$ }
$T' = \{*_8$ }
$F = \{\texttt{id}_{11}\ \texttt{num}_{12}\ (_{13}$ }

FOLLOW:

$S = \{$ }
$E = \{)_{14}$ }
$E' = \{)_{16}$ }
$T = \{+_3\ -_5\ )_{15}$ }
$T' = \{+_7\ -_7\ )_{19}$ }
$F = \{*_9\ +_{18}\ -_{18}\ )_{18}$ }

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

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

Beispiel: Parser-Tabelle zu G_3

	`id`	`:=`	`+`	`-`	`*`	`num`	`(`	`)`
S	$S \to$ `id :=` E							
E	$E \to T\ E'$					$E \to T\ E'$	$E \to T\ E'$	
E'			$E' \to$ `+` $T\ E'$	$E' \to$ `-` $T\ E'$				$E' \to$
T	$T \to F\ T'$					$T \to F\ T'$	$T \to F\ T'$	
T'			$T' \to$	$T' \to$	$T' \to$ `*` $F\ T'$			$T' \to$
F	$F \to$ `id`					$F \to$ `num`	$F \to$ `(` 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)*:

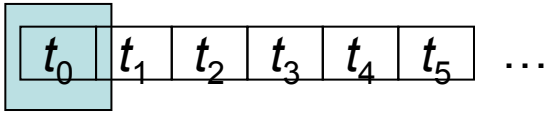
Sei $K = (\Sigma, \Gamma, Q, q_0, Z_0, F, \Delta)$ ein KA. Ein Paar (q, w, γ) mit $q \in Q$, $w \in \Sigma^*$ und $\gamma \in \Gamma^*$ heißt *Konfiguration*. Die *Schrittrelation* $\overset{}{\underset{K}{\vdash}}$ setzt Konfigurationen in Beziehung:

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

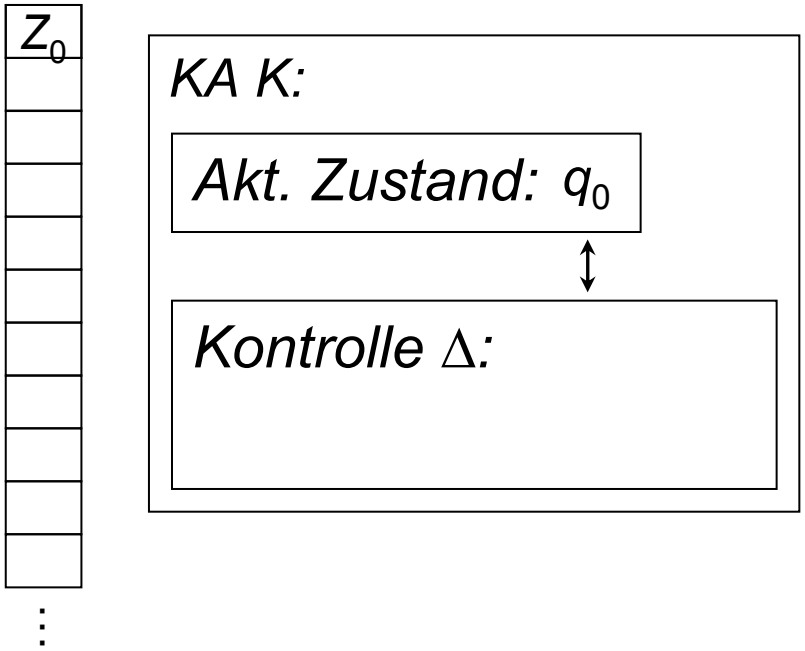
In Worten: $\overset{}{\underset{K}{\vdash}}$ modelliert einen einzelnen Zustandsübergang von K von aktuellem Zustand q in Folgezustand p für ein Eingabezeichen e bzw. ε und ein Rest-Wort w. Dabei „verbraucht“ K die untersten Elemente γ_2 des Kellers und speichert neue Elemente γ_3 unten im Stack. $\overset{*}{\underset{K}{\vdash}}$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

Arbeitsweise von Kellerautomaten

Eingabeband:



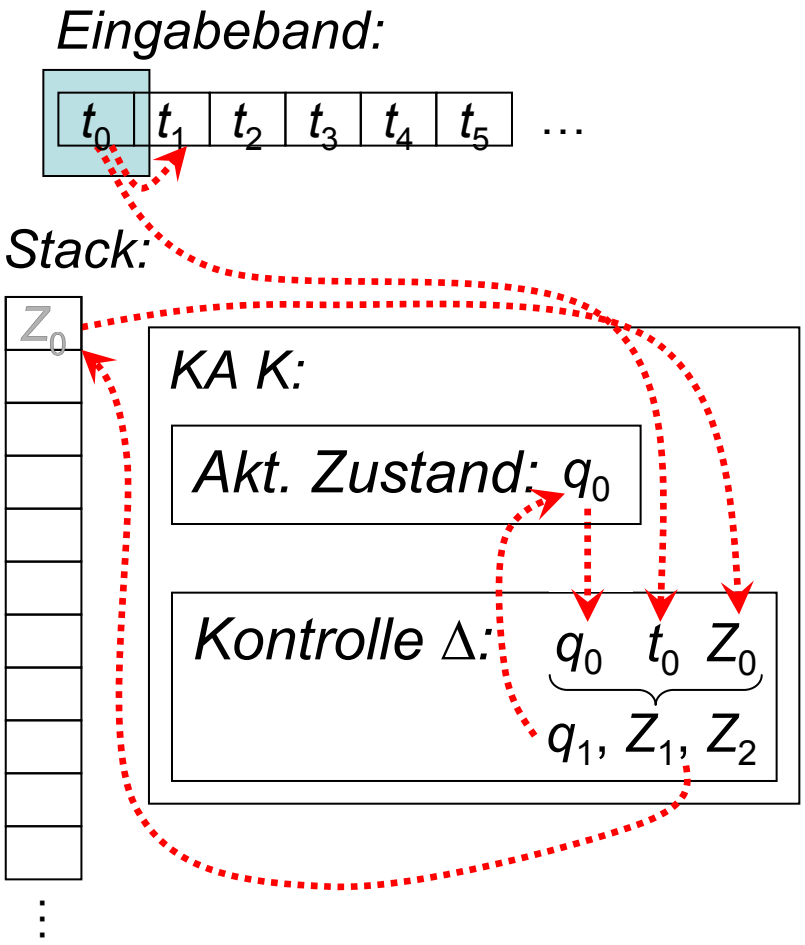
Stack:



Start des Automaten:

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

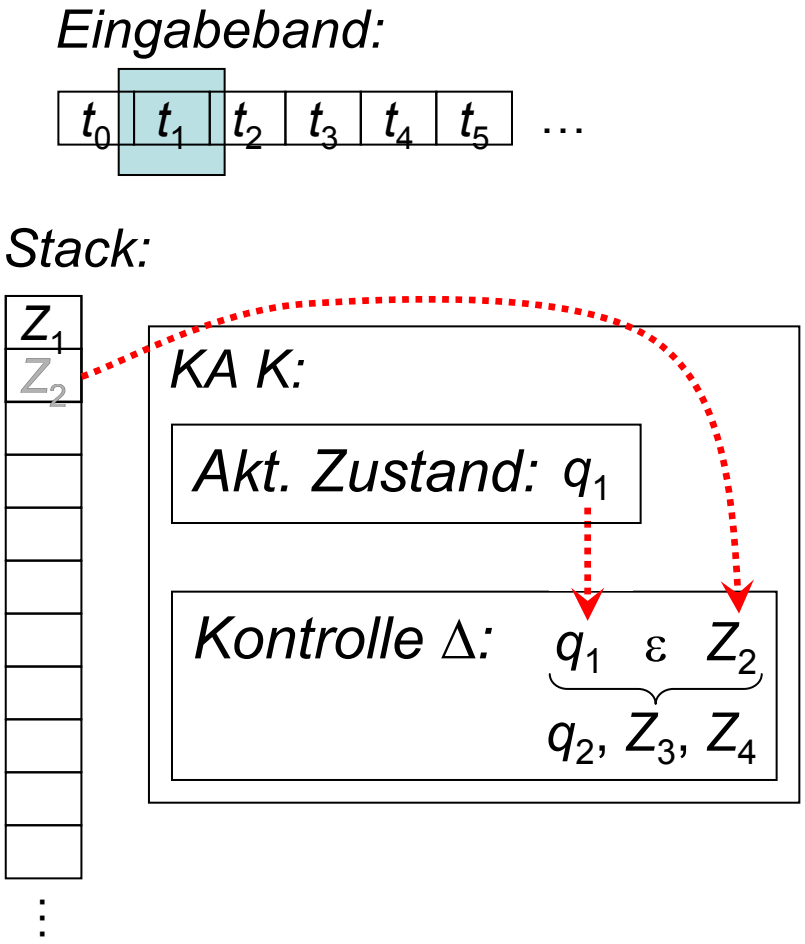
Arbeitsweise von Kellerautomaten



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}, ….
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, …, Z_n: $(q_{i+1}, Z_1, \ldots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \ldots)$
- Abspeichern von q_{i+1}, Z_1, …, Z_n
- Nur falls Token der Eingabe gelesen: Lesekopf nach rechts.

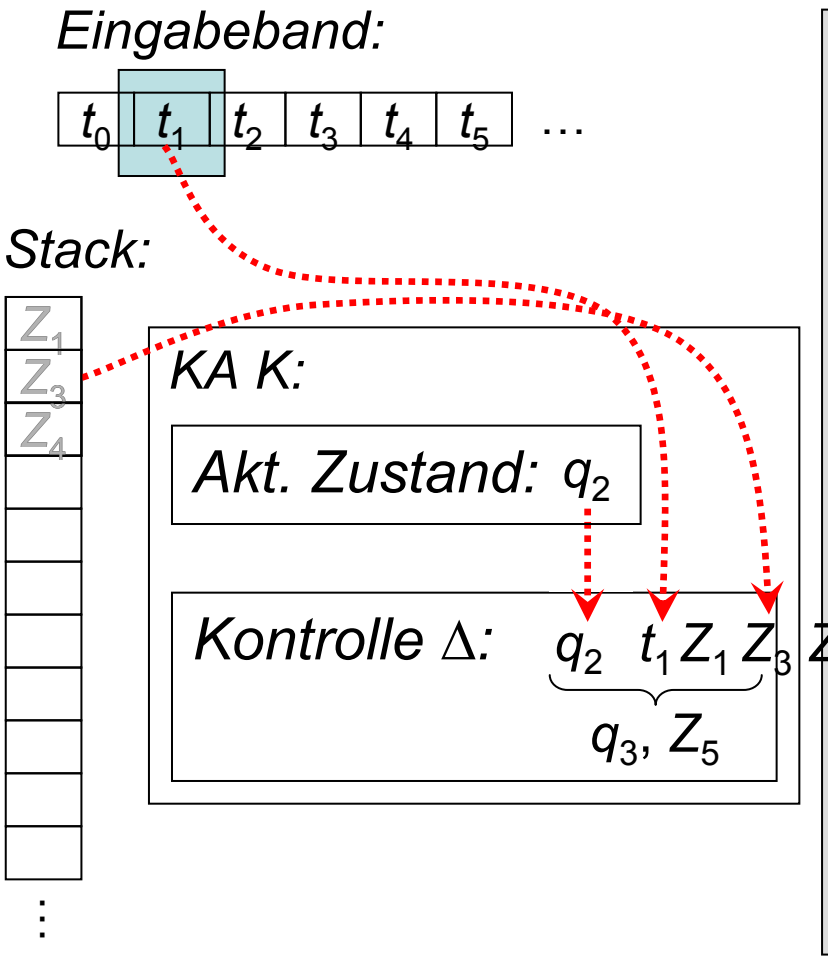
Arbeitsweise von Kellerautomaten



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}, ….
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, …, Z_n: $(q_{i+1}, Z_1, \ldots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \ldots)$
- Abspeichern von q_{i+1}, Z_1, …, Z_n
- Nur falls Token der Eingabe gelesen: Lesekopf nach rechts.

Arbeitsweise von Kellerautomaten

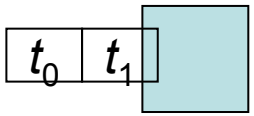


Pro Verarbeitungsschritt:

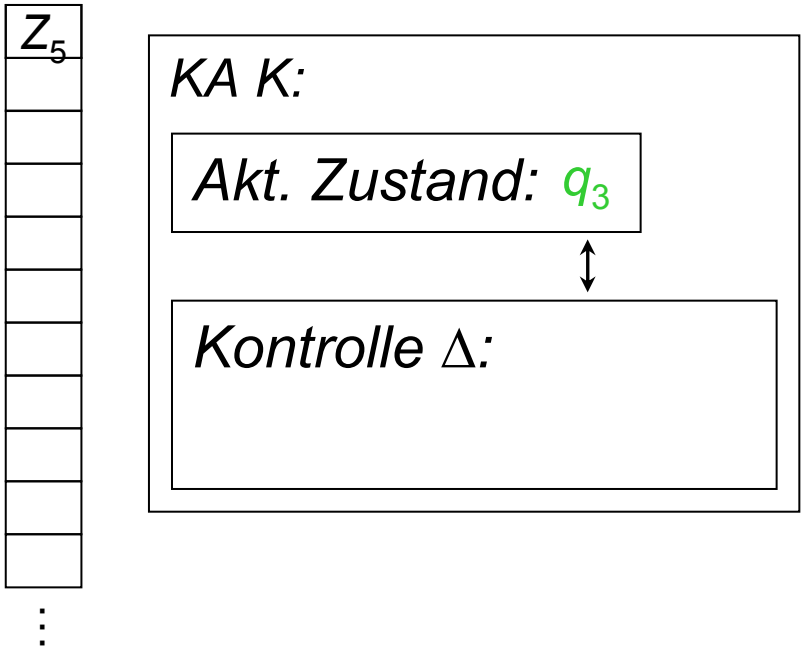
- „Raten" der aktuellen Eingabe *e*: entweder aktuelles Token unter Lesekopf oder 'ε'.
- Lesen und Entfernen der unteren Stack-Elemente Z_l, Z_{l-1}, ….
- Berechnen des Folgezustands q_{i+1} und neuer Stack-Inhalte Z_1, …, Z_n: $(q_{i+1}, Z_1, \ldots, Z_n) = \Delta(q_i, e, Z_l, Z_{l-1}, \ldots)$
- Abspeichern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

Arbeitsweise von Kellerautomaten

Eingabeband:



Stack:



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 $(q_0, w, Z_0) \overset{*}{\vdash_K} (q_i, \varepsilon, \gamma)$ für $q_i \in F$ durchgeführt.

LR-Parsen mit Kellerautomaten

LR(*k*)-Parser:

- Left-to-right parse, Rightmost-derivation, *k*-symbol look-ahead
- Merkwürdig: *Rechts*ableitung 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

$S \rightarrow$ `id :=` E

$E \rightarrow E$ `+` T | $E \rightarrow E$ `-` T | $E \rightarrow T$

$T \rightarrow T$ `*` F | $T \rightarrow F$

$F \rightarrow$ `id` | $F \rightarrow$ `num` | $F \rightarrow$ `(` E `)`

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

Beispiel (2)

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

	Stack	Eingabe	Aktion
1.	Z_0	↑`id := id - num * id`	*shift*
2.	`id`	`id` ↑`:= id - num * id`	*shift*
3.	`id :=`	`id :=` ↑`id - num * id`	*shift*
4.	`id := id`	`id := id` ↑`- num * id`	*reduce(* $F \rightarrow$ `id` *)*
5.	`id :=` F	`id := id` ↑`- num * id`	*reduce(* $T \rightarrow F$ *)*
6.	`id :=` T	`id := id` ↑`- num * id`	*reduce(* $E \rightarrow T$ *)*
7.	`id :=` E	`id := id` ↑`- num * id`	*shift*
8.	`id :=` E `-`	`id := id -` ↑`num * id`	*shift*
9.	`id :=` E `- num`	`id := id - num` ↑`* id`	*reduce(* $F \rightarrow$ `num` *)*

Beispiel (3)

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

	Stack	Eingabe	Aktion
10.	`id :=` *E* `-` *F*	`id := id - num` ↑`* id`	*reduce(* $T \rightarrow F$ *)*
11.	`id :=` *E* `-` *T*	`id := id - num` ↑`* id`	*shift*
12.	`id :=` *E* `-` *T* `*`	`id := id - num * ` ↑`id`	*shift*
13.	`id :=` *E* `-` *T* `* id`	`id := id - num * id`↑	*reduce(* $F \rightarrow$ `id` *)*
14.	`id :=` *E* `-` *T* `*` *F*	`id := id - num * id`↑	*reduce(* $T \rightarrow T$ `*` F *)*
15.	`id :=` *E* `-` *T*	`id := id - num * id`↑	*reduce(* $E \rightarrow E$ `-` T *)*
16.	`id :=` *E*	`id := id - num * id`↑	*reduce(* $S \rightarrow$ `id :=` E *)*
17.	*S*	`id := id - num * id`↑	*accept*

Bottom-Up Parser

Einige notwendige Begriffe:

- Ziel des Parsers ist, ausgehend vom Startsymbol S einer Grammatik einen Eingabesatz abzuleiten:
  $S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_{n-1} \rightarrow \gamma_n \rightarrow$ 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 $<N \rightarrow \alpha, k>$; $N \rightarrow \alpha \in P$, k bezeichnet die Position des äußerst rechten Symbols von α in γ.
Für einen Griff $<N \rightarrow \alpha, k>$ 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 | $E \rightarrow E$ `-` T | $E \rightarrow T$

$T \rightarrow T$ `*` F | $T \rightarrow F$

$F \rightarrow$ `id` | $F \rightarrow$ `num` | $F \rightarrow$ `(` E `)`

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

Griffe: (auch in grün angezeigt)

$\underline{S}$ $\rightarrow$	%
`id :=` $\underline{E}$ $\rightarrow$	<$S \rightarrow$ `id :=` E, 3>
`id :=` $\underline{T}$ $\rightarrow$	<$E \rightarrow T$, 3>
`id :=` T `*` $\underline{F}$ $\rightarrow$	<$T \rightarrow T$ `*` F, 5>
`id :=` $\underline{T}$ `*` `num` $\rightarrow$	<$F \rightarrow$ `num`, 5>
`id :=` $\underline{F}$ `*` `num` $\rightarrow$	<$T \rightarrow F$, 3>
`id :=` `id` `*` `num`	<$F \rightarrow$ `id`, 3>

Pseudocode des Shift-Reduce Parsers

Algorithmus:

- shift(Z_0);
  tok = getToken();
- while (! (<bottom of stack == S> && <tok == EOF>))
  - if (<bottom of stack == handle <$N \rightarrow \alpha$, k>)
    - 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 Griffs $<N \rightarrow \alpha, k>$ 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

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

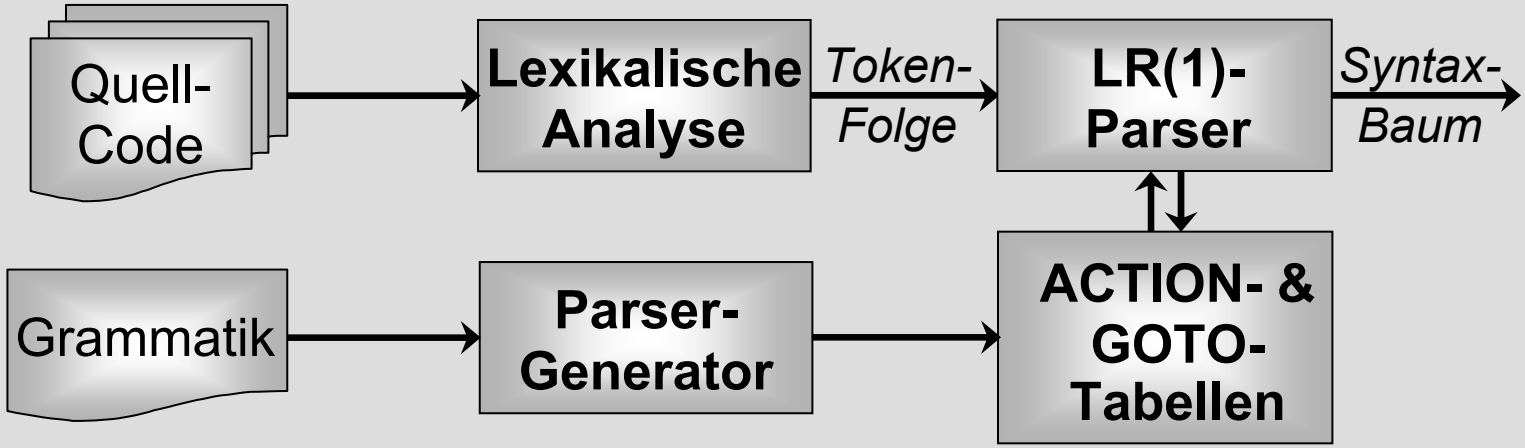
zulässt. *G* heißt *LR(1)-Grammatik*, wenn

- 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($N \rightarrow \alpha$)“)
    - stack.popnum(2 * $|\alpha|$); *// pop 2 * $|\alpha|$ 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)*

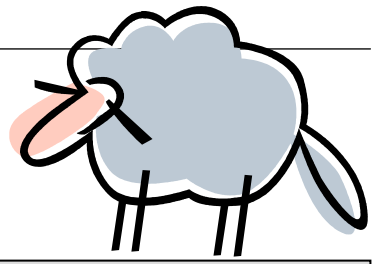
- while (!found)
  - s = stack.top(); *// get current state of handle-finding DFA*
  - if (ACTION[s, tok] == „reduce($N \rightarrow \alpha$)“)
    - *<siehe vorige Folie>*
  - else if (ACTION[s, tok] == „shift(s_i)“)
    - stack.push(tok); *// push current token PLUS*
    - stack.push(s_i); *// follow-up 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: $S \rightarrow$ *Blöken* R2: *Blöken* $\rightarrow$ *Blöken* **`määäh`**

R3: *Blöken* $\rightarrow$ **`määäh`**

Stack	Eingabe	Aktion
$Z_0\ s_0$	**`määäh määäh EOF`**	shift(s_2)
$Z_0\ s_0$ **`määäh`** s_2	**`määäh EOF`**	reduce(R3)
$Z_0\ s_0$ *Blöken* s_1	**`määäh EOF`**	shift(s_3)
$Z_0\ s_0$ *Blöken* s_1 **`määäh`** s_3	**`EOF`**	reduce(R2)
$Z_0\ s_0$ *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)

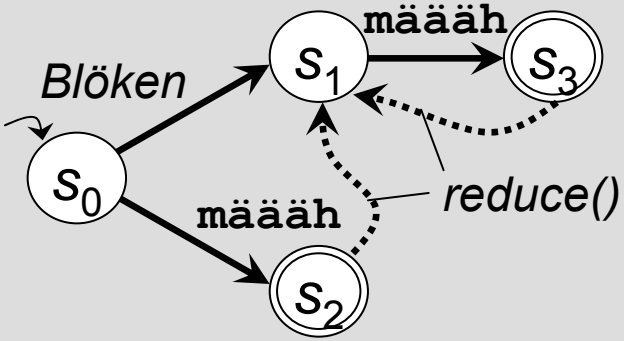
GOTO:

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 $<N \rightarrow \alpha, k>$ 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[<Zustand am unteren Stack-Ende nach pop von α>, N]
- ☞ DEA für G_5:



LR(1)-Items

Definition *(LR(1)-Item)*:
Sei G eine kontextfreie Grammatik, $A \to \alpha\beta$ eine Produktionsregel.
$[A \to \alpha \bullet \beta, T]$ 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.
- $[A \to \alpha \bullet \beta, T]$: die bisher verarbeitete Eingabe stimmt mit der Anwendung von $A \to \alpha\beta$ überein, und es wurde bereits α erkannt.
- $[A \to \bullet \alpha\beta, T]$: die bisher verarbeitete Eingabe stimmt mit der Anwendung von $A \to \alpha\beta$ direkt am unteren Stack-Ende überein.
- $[A \to \alpha\beta \bullet, T]$: es wurde bisher $\alpha\beta$ erkannt, und das look-ahead Token T stimmt mit der Reduktion mit Hilfe von $A \to \alpha\beta$ 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 FIRST(γU), 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 $S \rightarrow \ldots$)
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 *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_i und jedes $X \in (V_N \cup V_T)$:
   5. Berechne den Folgezustand (d.h. die Menge von Items) $s_j = goto(\ s_i, X\ )$
   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:

- $s_0 = closure(\ S' \rightarrow \bullet S,\ \text{EOF}\ )$;
- $S = \{\ s_0\ \}$;
- $j = 1$;

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 $X \in \text{FIRST}(\beta T)$.

Algorithmus: *closure*(*s*)

- while (<*s* wurde verändert>)
  - for (<alle Items $[A \rightarrow \alpha \bullet B \beta, T] \in s$>)
    - for (<alle Regeln $B \rightarrow \gamma \in P$>)
      - for (<alle $X \in \text{FIRST}(\beta T)$>) // β könnte gleich ε sein
        - if ($[B \rightarrow \bullet \gamma, X] \notin s$)
          - $s = s \cup [B \rightarrow \bullet \gamma, X]$

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*]).

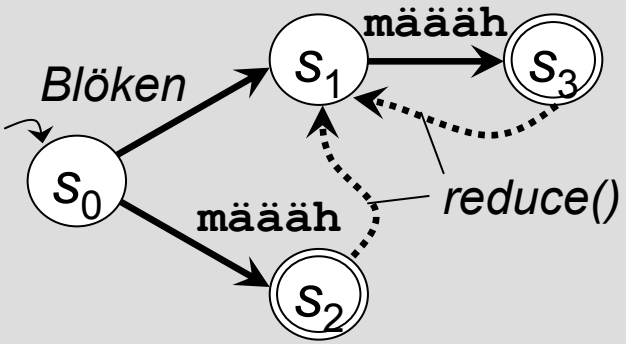
Algorithmus: *goto*(*s*, *X*)

- *s*' = $\emptyset$;
- for (<alle Items [$A \rightarrow \alpha \bullet X \beta$, *T*] $\in$ *s*>)
  - *s*' = *s*' U [$A \rightarrow \alpha X \bullet \beta$, *T*];
- return *closure*(*s*');

Beispiel: Schafsgeblöke

Algorithmus zur Bestimmung der DEA-Zustände:

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

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

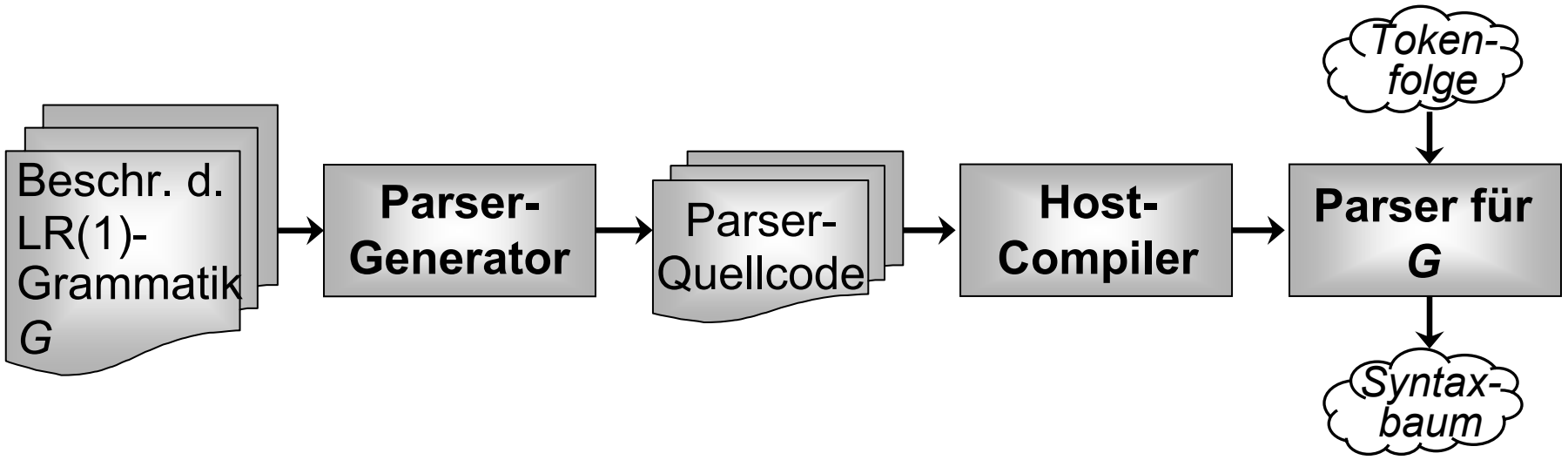
Implementierung eines Parsers:

- *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

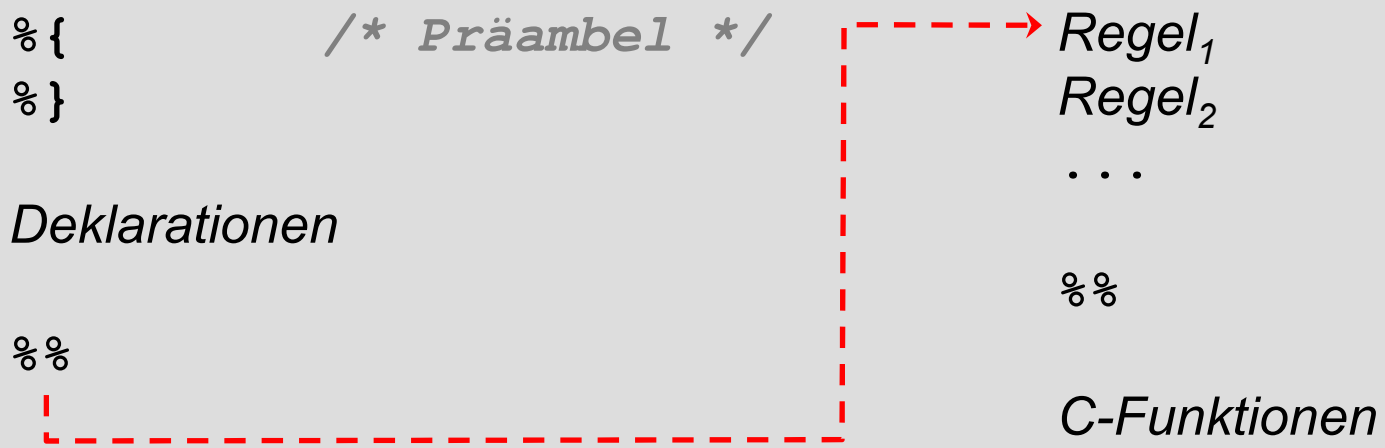
- Ein *Parser-Generator* erhält eine Beschreibung der zu erkennenden LR(1)-Grammatik als Eingabe und erzeugt daraus einen Parser für eben diese Grammatik.



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:



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 lineCounter; /* 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, lineCounter ); }

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

Parser-Generierung mit BISON (7)

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

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

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;
{identifier}  return tID;
[ \t]         /* Consumes white spaces & tabs. */
\n            incrementLineCounter();

%%

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

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.**