


Vorlesung Compilerbau (SoSe 2018)


Teil 5: Syntaktische Analyse – Parsing II



Vorlesung Compilerbau: Syntaktische Analyse – Parsing II Methoden der Syntaxanalyse

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

© Prof. J.C. Freytag, Ph.D. 5.1



Methoden der Syntaxanalyse

Hauptsächlich werden **zwei Klassen** in Abhängigkeit der Richtung, in der die Knoten des Parse-Baumes konstruiert werden, unterschieden:

- **Top-Down- Methoden**
Knotenkonstruktion beginnend mit Wurzel zu den Blättern:
 - effiziente Parser lassen sich per Hand erstellen
- **Bottom-Up-Methoden**
umgekehrt: Knotenkonstruktion von Blättern zur Wurzel
 - erlaubt die Behandlung einer größeren Klasse von Grammatiken bzw. Sprachen und Übersetzungsschemata
 - deshalb beliebter bei automatischer Parser-Generierung

© Prof. J.C. Freytag, Ph.D. 5.2

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Top-Down-Syntaxanalyse: Schwerpunkte

- grundlegende Ideen des Top-Down-Verfahrens
- Erstellung eines effizienten Top-Down-Parsers (sog. prädiktiver Parser, der ohne Rücksetzen auskommt)
- LL(1)-Grammatiken, für die sich prädiktive Parser automatisch ableiten lassen
- nichtrekursive Implementierung der Parser

© Prof. J.C. Freytag, Ph.D.

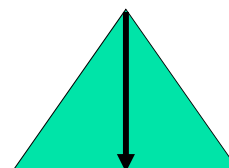
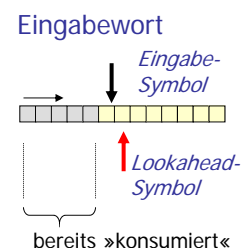
5.3



Allgemeine Form der Top-Down-Methode

Top-Down-Parsing

- vom Syntaxbaum ist zunächst nur die Wurzel (Startsymbol) bekannt
- Ziel ist es, den "Rest" des Baumes so zu konstruieren, dass das vom Syntaxbaum erzeugte Wort mit dem Eingabewort übereinstimmt
- dazu wählt man für das aktuelle Eingabe-Token eine passende Regel aus und erweitert den Baum, wobei die Knoten mit den Symbolen auf der rechten Regel-Seite markiert werden
- man versucht das aktuelle Eingabe-Token (evtl. mit Kenntnis zusätzlicher Lookahead-Symbole) zu verarbeiten
- falls gewählte Regel nicht zum Erfolg führt, muss die Analyse zurückgesetzt werden, um mit einer alternativen Regel den Baum zu erweitern (Backtracking)
- bei einigen Grammatiken kann ein Backtracking im Top-Down-Vorgehen vermieden werden



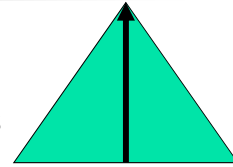
© Prof. J.C. Freytag, Ph.D.

5.4

Allgemeine Form der Bottom-Up-Methode

Bottom-Up-Parsing

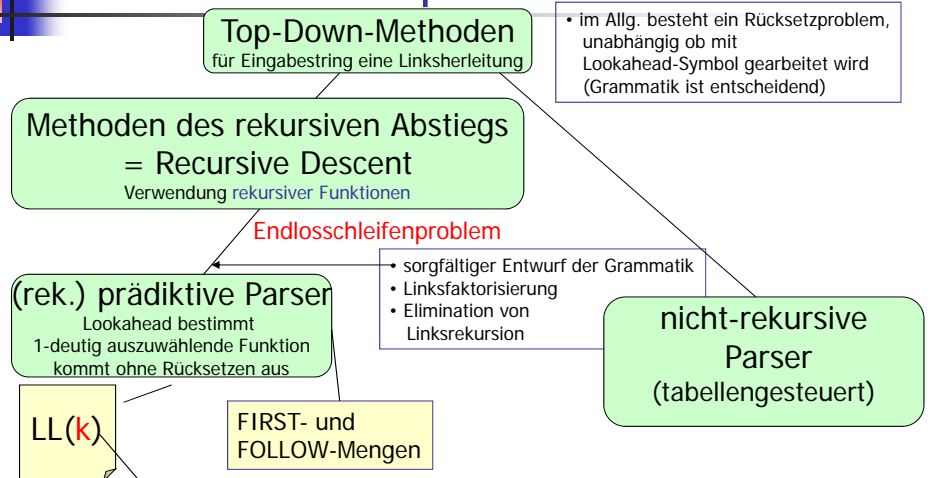
- baut den Baum mit den Blättern beginnend bis zur Wurzel auf
- startet in einem Zustand, der legal für das erste zu verarbeitende Token ist
- wird diese Eingabe »konsumiert«, geht man in einen neuen Zustand über, um die dann schon erkannten Präfixe zu »codieren« (Erkennung zulässiger Präfixe)
- man benutzt einen Keller um beides, Zustand und ein Teil des Wortes zu speichern



© Prof. J.C. Freytag, Ph.D.

5.5

Klassen von Top-Down-Methoden



© Prof. J.C. Freytag, Ph.D.

5.6

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Prinzip des TD-Verfahrens mit Rücksetzen

Beispiel-Grammatik, die Rücksetzen bei Nicht-Vorausschau erfordert

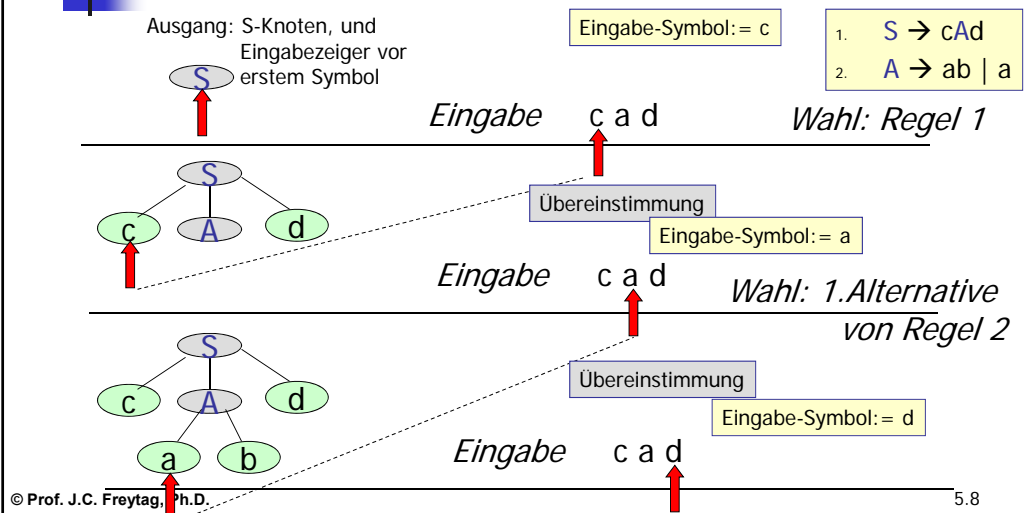
1. $S \rightarrow cAd$
 2. $A \rightarrow ab \mid a$
- (groß: Nichtterminal, klein: Terminal)

- Eingabestring: $w = cad$
- **Ziel:** Aufbau eines Syntaxbaums (von oben nach unten)

© Prof. J.C. Freytag, Ph.D.

5.7

Prinzip des TD-Verfahrens mit Rücksetzen (2)

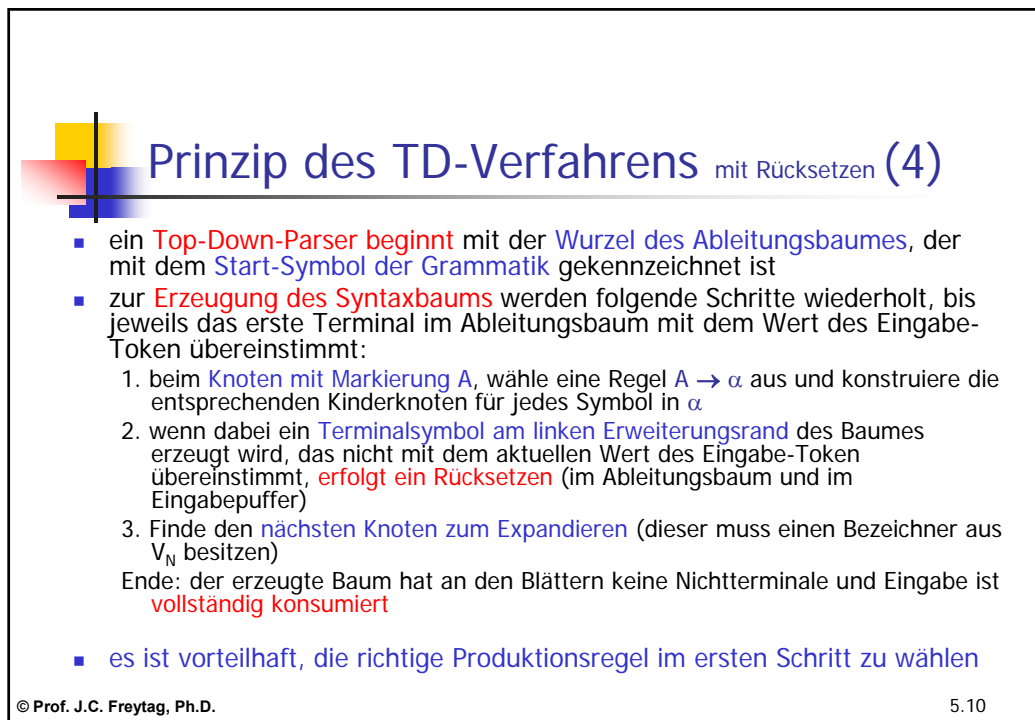
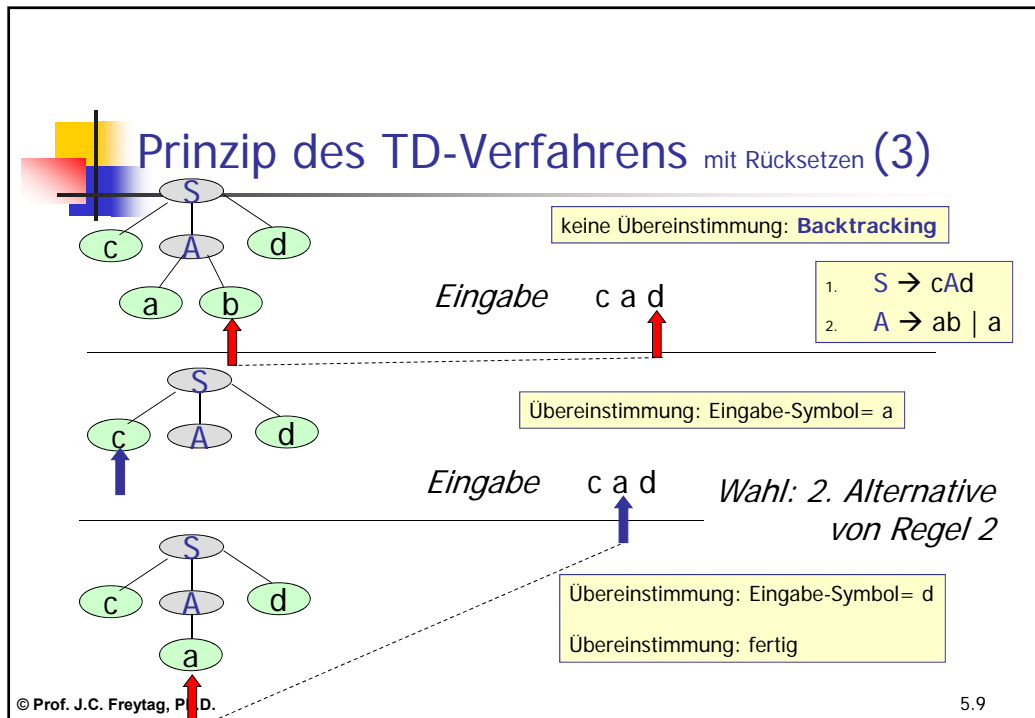


© Prof. J.C. Freytag, Ph.D.

5.8

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II





Gefahr beim Top-Down-Verfahren mit Rücksetzen

- es kann vorkommen, dass immer wieder versucht wird, A zu expandieren, ohne dass irgendein Eingabesymbol konsumiert wird
- Ursache für das Nicht-Terminieren: Top-Down Parser können
 - keine Linksrekursion handhaben
 - Keine Regeln handhaben, die nach ϵ ableiten

Vorsicht

- ist die Grammatik der Sprache linksrekursiv, kann ein Recursive-Descent-Parser (auch wenn er mit Backtracking arbeitet) in Endlosschleifen geraten



Linksrekursion

- Def.:
eine Grammatik ist linksrekursiv, falls ein $A \in V_N$ existiert, so dass
$$A \Rightarrow^+ A\alpha$$
für eine beliebige Zeichenkette α gilt

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel für top-down Parsing

- Beispiel: Grammatik für einfache Ausdrücke (linksrekursiv)
- Eingabe: $x - 2 * y$

```

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

© Prof. J.C. Freytag, Ph.D.

5.13

Beispiel für TD-Parsing (2)

mit willkürlicher Regelwahl

Prod	Satzform	Eingabe
-	<goal>	^ x - 2 * y
1	<expr>	^ x - 2 * y
2	<expr> + <term>	^ x - 2 * y
4	<term> + <term>	^ x - 2 * y
7	<factor> + <term>	^ x - 2 * y
9	id + <term>	^ x - 2 * y
-	id + <term>	^ x - 2 * y
-	<expr>	^ x - 2 * y
3	<expr> - <term>	^ x - 2 * y
4	<term> - <term>	^ x - 2 * y
7	<factor> - <term>	^ x - 2 * y
9	id - <term>	^ x - 2 * y
-	id - <term>	x ^ - 2 * y

Backtracking

© Prof. J.C. Freytag, Ph.D.

5.14

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel für TD-Parsing (3)

Prod	Satzform	Eingabe
-	id - <term>	x - ^ 2 * y
7	id - <factor>	x - ^ 2 * y
8	id - num	x - ^ 2 * y
-	id - num	x - 2 ^ * y
-	id - <term>	x - ^ 2 * y
5	id - <term> * <factor>	x - ^ 2 * y
7	id - <term> * <factor>	x - ^ 2 * y
8	id - num * <factor>	x - ^ 2 * y
-	id - num * <factor>	x - 2 ^ * y
-	id - num * <factor>	x - 2 * ^ y
9	id - num * <id>	x - 2 * ^ y
-	id - num * <id>	x - 2 * y ^

Backtracking

© Prof. J.C. Freytag, Ph.D.

5.15

Beispiel für TD-Parsing (5)

- weitere Möglichkeit der Erkennung von $x - 2^* y$

Prod	Satzform	Eingabe
-	<goal>	^ x - 2 * y
	<expr>	^ x - 2 * y
2	<expr> + <term>	^ x - 2 * y
2	<expr> + <term> + <term>	^ x - 2 * y
2	<expr> + <term> + <term> + <term>	^ x - 2 * y
2	<expr> + <term> + ...	^ x - 2 * y
2	...	^ x - 2 * y

- werden vom Parser die falschen Entscheidungen getroffen, terminiert die Erkennung nicht
 - keine gute Eigenschaft eines Parsers (sollte immer terminieren)

© Prof. J.C. Freytag, Ph.D.

5.16

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Entfernen von Linksrekursion

Grammatik-Umformung zur Vermeidung der Linksrekursion:

- gegeben sei folgende links-rekursive Produktion
$$\langle \text{links} \rangle ::= \langle \text{links} \rangle \alpha \mid \beta$$
wobei α und β nicht mit $\langle \text{links} \rangle$ beginnen
- dann wird die Regel wie folgt umgeschrieben
$$\begin{aligned} \langle \text{links} \rangle &::= \beta \langle \text{neu} \rangle \\ \langle \text{neu} \rangle &::= \alpha \langle \text{neu} \rangle \quad /* \text{ dafür rechts-rekursiv } */ \\ &\mid \varepsilon \end{aligned}$$
mit $\langle \text{neu} \rangle$ als neuem Nicht-Terminalsymbol
- Bemerkung:** neue Regeln enthalten keine Linksrekursionen

© Prof. J.C. Freytag, Ph.D.

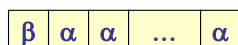
5.17



Vergleich: Links- und Rechtsrekursionen

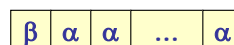
$$A \rightarrow A\alpha \mid \beta$$

wiederholte Anwendung der Regeln



$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \varepsilon \end{aligned}$$

wiederholte Anwendung der Regeln



© Prof. J.C. Freytag, Ph.D.

5.18

Beispiel: Entfernen von Linksrekursion

- Beispiel enthält zwei Regeln mit Linksrekursion:

```
<expr> ::= <expr> + <term>
          | <expr> - <term>
          | <term>
<term> ::= <term>* <factor>
          | <term> / <factor>
          | <factor>
```

- nach Umformung:

```
<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'>
          | - <term> <expr'>
          | ε
<term> ::= <factor> <term'>
<term'> ::= * <factor> <term'>
          | / <factor> <term'>
          | ε
```

Bemerkung:

- mit umgeformter Grammatik wird der Parser terminieren
- bei einigen Eingaben muss trotzdem zurückgesetzt werden

Beispiel Linksrekursion (3)

- Vorherige Grammatik umgeformt nach Regeln zur Entfernung der Linksrekursion

```
1. <goal> ::= <expr>
2. <expr> ::= <term> <expr'>
3. <expr'> ::= + <term> <expr'>
4.           | - <term> <expr'>
5.           | ε
6. <term> ::= <factor> <term'>
7. <term'> ::= * <factor> <term'>
8.           | / <factor> <term'>
9.           | ε
10. <factor> ::= num
11.           | id
```

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Alternative Rechtsrekursion

- Folgende **alternative** Grammatik:

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

- Diese ist
 - rechtsrekursiv ($A \Rightarrow^+ \alpha A$)
 - frei von ϵ -Produktionen
- aber**, sie generiert andere »Assoziativitäten«:
 - dieselbe Syntax, aber andere Semantik (Bedeutung) (Warum und Wo?)

© Prof. J.C. Freytag, Ph.D.

5.21

Alternative Rechtsrekursion

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

links-rekursiv

```
1. <goal> ::= <expr>
2. <expr> ::= <term> <expr'>
3. <expr'> ::= + <term> <expr'>
4.           | - <term> <expr'>
5.           | ε
6. <term> ::= <factor> <term'>
7. <term'> ::= * <factor> <term'>
8.           | / <factor> <term'>
9.           | ε
10. <factor> ::= num
11.           | id
```

rechts-rekursiv

rechts-rekursiv
und ϵ -frei

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

© Prof. J.C. Freytag, Ph.D.

5.22



Prädiktive Syntaxanalyse

- als rekursiver Abstieg, d.h.
 - Eingabe wird dabei durch Menge **rekursiver Funktionen** abgearbeitet
 - jedem Nichtterminal der Grammatik entspricht eine Funktion
- Besonderheit der prädiktiven Analyse
 - **Lookahead-Symbol(e)** für jedes Nichtterminal bestimmt **eindeutig** die Auswahl der Funktion
 - Folge der Funktionen, die während der Abarbeitung der Eingabe aufgerufen werden, definiert **implizit** einen **Parse-Baum** für diese Eingabe

© Prof. J.C. Freytag, Ph.D.

5.23



Aufbau eines prädiktiven Parsers

es gibt verschiedene Funktionen

- für jedes **Nicht-Terminal** der Grammatik (Umsetzung der Ableitung)
- Funktion **advance** (aktualisiert **lookahead** mit dem nächsten Token)
- Funktion **eat** (steuert **advance**)
 - sollte Argument **t** von **eat** mit dem Lookahead-Symbol übereinstimmen, wird zum nächsten Eingabesymbol übergegangen


das Lookahead-Symbol bestimmt, welche Produktion anzuwenden ist

© Prof. J.C. Freytag, Ph.D.

5.24

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Bsp. eines prädikt. Par

Grammatik:
 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
 $S \rightarrow \text{begin } S \text{ L}$
 $S \rightarrow \text{print } E$
 $L \rightarrow \text{end}$
 $L \rightarrow ; S \text{ L}$
 $E \rightarrow \text{num} = \text{num}$

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};

extern enum token getToken(void);


enum token lookahead;

void advance() {lookahead = getToken(); }
void eat (enum token t) {if (lookahead == t) advance(); else error(); }

void S(void) {switch(lookahead) {
    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();
}}


```

© Prof. J.C. Freytag, Ph.D. 5.25



Bsp. eines prädikt. Parsers (2)

Grammatik:
 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
 $S \rightarrow \text{begin } S \text{ L}$
 $S \rightarrow \text{print } E$
 $L \rightarrow \text{end}$
 $L \rightarrow ; S \text{ L}$
 $E \rightarrow \text{num} = \text{num}$

```
...

void L(void) {switch(lookahead) {
    case END: eat(END); break;
    case SEMI:      eat(SEMI); S(); L(); break;
    default: error();
}}

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

Bei geeigneter Definition von **error** und **getToken** haben wir mit dieser Methode für die Grammatik einen geeigneten Parser.
Versuch mit einer weiteren Grammatik folgt

© Prof. J.C. Freytag, Ph.D. 5.26

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Weiteres Beispiel

```
void S(void) { E(); eat(Eof); }
void E(void) {switch(lookahead) {
    case '?': E(); eat(PLUS); T(); break;
    case '?': E(); eat(MINUS); T(); break;
    case '?': T(); break;
    default: error();
}}
void T(void) {switch(lookahead) {
    case '?': T(); eat(TIMES); F(); break;
    case '?': T(); eat(DIV); F(); break;
    case '?': F(); break;
    default: error();
}}
```

Grammatik:

```
S → E $
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → id
F → num
F → ( E )
```

Dateiende

Konflikt

© Prof. J.C. Freytag, Ph.D.

5.27

FIRST- Mengen

... sind hilfreich bei der Konstruktion prädiktiver Parser

Grundidee:

- für jeweils zwei Produktion $A \rightarrow \alpha | \beta$ brauchen wir „Unterscheidungsmerkmale“, um die richtige Regel beim Expandieren des Syntaxbaumes zu bestimmen

FIRST- Definition:

(α setzt sich aus Terminalen und Nichtterminalen zusammen)

- sei α die rechte Seite einer Produktion dann ist **FIRST(α)** *die Menge aller Terminalsymbole*, die bei mindestens einem aus α hergeleiteten Wörtern als Anfangssymbol auftreten
- ϵ gehört auch zur Menge, falls $\alpha \Rightarrow^* \epsilon$

FIRST(α) = {t: t ∈ V_T ∪ {ε} mit $\alpha \Rightarrow^* t\gamma$ } bei beliebigem γ

© Prof. J.C. Freytag, Ph.D.

5.28

Beispiel: FIRST-Mengen

$\text{FIRST}(\alpha) = \{t: t \in V_T \text{ und } \alpha \Rightarrow^* t\gamma\}$ bei beliebigem γ

Grammatik:

$S \rightarrow E \$$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

$\text{FIRST}(F) = \{\text{id}, \text{num}, (\}$

$\text{FIRST}(T) = \text{FIRST}(F)$

$\text{FIRST}(E) = \text{FIRST}(T)$

Konstruktion von FIRST- Mengen

Folgende Regeln

werden solange zur Berechnung von $\text{FIRST}(X)$ angewendet,
bis zu keiner FIRST -Menge mehr

- ein neues Terminal oder
- ϵ hinzukommt:

1. ist X Terminal, dann ist $\text{FIRST}(X) = \{X\}$
2. gibt es eine Produktion $X \rightarrow \epsilon$, **ist ϵ der Menge hinzuzufügen**
3. ist X ein Nichtterminal und $X \rightarrow Y_1 Y_2 \dots Y_k$ eine Produktion, dann ist **a** der Menge zuzuführen, falls **a** für **irgendein i** in $\text{FIRST}(Y_i)$ **und ϵ in allen $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$** enthalten ist
4. falls ϵ in allen $\text{FIRST}(Y_i)$ enthalten, dann gehört ϵ auch zu $\text{FIRST}(X)$

Beispiel: Konstruktion von FIRST-Mengen

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

nach Regel (3) zu FIRST(F), wobei jeweils $i=1$ ist, und da nach Regel (1)
 $\text{FIRST}(\text{id}) = \{\text{id}\}$ und $\text{FIRST}('(') = \{($ ist

nach Regel (3) mit $i=1$ und infolge $T \rightarrow FT'$ gehören id und '(' dann auch zu
 $\text{FIRST}(T)$

FIRST(E') = { +, ε }

ε ist nach Regel (2) in FIRST(E') enthalten

FIRST(T') = { *, ε }

Grammatik:

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \varepsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \varepsilon$
 $F \rightarrow (E) \mid \text{id}$

First-Mengen (vollständige Def.)

Menge aller Terminale, die am Anfang eines aus α ($\alpha \in V^*$)
ableitbaren Wortes stehen können

$\text{FIRST}_T(\alpha) = \{t: t \in V_T \text{ mit } \alpha \Rightarrow^* t\gamma\}$ bei beliebigem γ

$$\text{FIRST}(\alpha) = \begin{cases} \text{FIRST}_T(\alpha) \cup \{\varepsilon\}, & \text{falls } \alpha \Rightarrow^* \varepsilon \\ \text{FIRST}_T(\alpha) & , \text{sonst} \end{cases}$$

Eigenschaft von First-Mengen

Schlüsseleigenschaft:

- Falls zwei Produktionen $A \rightarrow \alpha | \beta$ in einer Grammatik vorkommen, soll die Eigenschaft gelten:

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

- Diese Eigenschaft würde es dem Parser erlauben, eine korrekte Wahl der richtigen Produktionsregel mit einer Vorausschau von einem Symbol zu treffen

FOLLOW-Mengen

FOLLOW- Definition:

- Sei A ein Nichtterminal,
- dann ist $\text{FOLLOW}(A)$ die Menge aller Terminalsymbole a , die in einem Wort direkt rechts neben A stehen können
- $\$$ gehört zu $\text{FOLLOW}(A)$, wenn A das am weitesten rechts stehende Symbol in einer Satzform ist

$$\text{FOLLOW}_T(A) = \{a: a \in V_T \text{ mit } S \Rightarrow^* \alpha A a \beta\} \text{ mit beliebigen } \alpha, \beta \in V^*$$

$$\text{FOLLOW}(A) = \begin{cases} \text{FOLLOW}_T(A) \cup \{\$, \} & \text{falls } s \Rightarrow^* \alpha A \\ \text{FOLLOW}_T(A) & , \text{sonst} \end{cases}$$

Achtung: es könnten durchaus zwischen A und a während der Bearbeitung Symbole gestanden haben, die in ϵ übergegangen sind

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Konstruktion von FOLLOW- Mengen

Folgende Regeln werden solange zur Berechnung von $FOLLOW(A)$ angewendet, bis keine $FOLLOW$ -Menge mehr vergrößert werden kann

1. $\$$ gehört zu $FOLLOW(s)$ falls s das Startsymbol ist
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird **jedes** Element von $FIRST(\beta)$ mit **Ausnahme von ϵ** auch in $FOLLOW(B)$ aufgenommen
3. gibt es Produktionen
 $A \rightarrow \alpha B$ oder
 $A \rightarrow \alpha B \beta$ und $FIRST(\beta)$ enthält dabei ϵ (d.h. $\beta \Rightarrow^* \epsilon$),
dann wird **jedes** Element von $FOLLOW(A)$ auch Element von $FOLLOW(B)$

© Prof. J.C. Freytag, Ph.D.

5.35

Beispiel: Mengen-Konstruktion (1)

Start :

	ϵ -ableitbar	FIRST	FOLLOW
X			
Y			
Z			

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.36

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	nein		
Y	ja		
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.37

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	nein	a	
Y	ja		
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.38

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	nein	a Y?	
Y	ja		
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.39

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	nein	a c	
Y	ja		
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.40

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja		
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.41

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein		

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.42

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	<i>ja</i>	a c ϵ	
Y	<i>ja</i>	c ϵ	
Z	<i>nein</i>	d	

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.43

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	<i>ja</i>	a c ϵ	
Y	<i>ja</i>	c ϵ	
Z	<i>nein</i>	d XYZ?	

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.44

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein	d a c YZ?	

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.45

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein	d a c Z?	

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.46

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein	d a c	

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

© Prof. J.C. Freytag, Ph.D.

5.47

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein	d a c	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von $FIRST(\beta)$ mit Ausnahme von ϵ auch in $FOLLOW(B)$ aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei $FIRST(\beta)$ ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von $FOLLOW(A)$ auch zu $FOLLOW(B)$

Z ist Start-Symbol

© Prof. J.C. Freytag, Ph.D.

5.48

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	
Z	nein	a c d	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$Z \rightarrow \alpha Y \beta$

FIRST(β) = FIRST(Z)

© Prof. J.C. Freytag, Ph.D.

5.49



Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	a c d
Z	nein	a c d	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$A \rightarrow \alpha Y \beta$

FIRST(β) = FIRST(Z)

© Prof. J.C. Freytag, Ph.D.

5.50

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	
Y	ja	c ϵ	a c d
Z	nein	d a c	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$Z \rightarrow \alpha X \beta$

$\text{FIRST}(\beta) = \text{FIRST}(YZ)$
 $\alpha = \epsilon$

© Prof. J.C. Freytag, Ph.D.

5.51

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	a c d
Y	ja	c ϵ	a c d
Z	nein	a c d	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$Z \rightarrow \alpha X \beta$

$\text{FIRST}(\beta) = \text{FIRST}(YZ)$

© Prof. J.C. Freytag, Ph.D.

5.52

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel: Mengen-Konstruktion (1)

1. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	a c d
Y	ja	c ϵ	a c d
Z	nein	a c d	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$A \rightarrow \alpha Z$

FOLLOW(A) = FOLLOW(Z)

© Prof. J.C. Freytag, Ph.D.

5.53

Beispiel: Mengen-Konstruktion (2)

2. Iteration:

	ϵ -ableitbar	FIRST	FOLLOW
X	ja	a c ϵ	a c d
Y	ja	c ϵ	a c d
Z	nein	a c d	\$

Grammatik:

$Z \rightarrow d\$$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

1. \$ gehört zu FOLLOW(S)
2. gibt es eine Produktion $A \rightarrow \alpha B \beta$, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen
3. gibt es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ (wobei FIRST(β) ϵ enthält, d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B)

$A \rightarrow \alpha Y \beta$

$\alpha = \epsilon$
 $\beta = \epsilon$

terminiert

© Prof. J.C. Freytag, Ph.D.

5.54

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Übungsbeispiel: Mengen-Konstruktion

Grammatik:

$S \rightarrow E\$$
 $E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \varepsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \varepsilon$
 $F \rightarrow (E) \mid id$

	ε -ableitbar	FIRST	FOLLOW
E'	ja	$+, \varepsilon$	$) \$$
E	nein	$(id$	$) \$$
F	nein	$(id$	$* +) \$$
T	nein	$(id$	$+) \$$
T'	ja	$*, \varepsilon$	$+) \$$

© Prof. J.C. Freytag, Ph.D.

5.55

Kriterien für LL(1)- Grammatiken

Eine beliebige kontextfreie Grammatik G heißt **LL(1)-Grammatik**, wenn folgende zwei Bedingungen durch G erfüllt sind:

1. für **alle** Produktionen $A \rightarrow \alpha|\beta$ eines **beliebigen** Nichtterminals A ($A \in V_N$) gilt:
$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset, \quad \alpha, \beta \in V^*$$
2. für **jedes** Nichtterminal A , für das eine ε -Ableitung möglich ist, sind die **FIRST**- und **FOLLOW**-Mengen disjunkt

Bem.: es gibt sowohl Algorithmen zur Berechnung von FIRST- und FOLLOW-Mengen als auch zur Bestimmung der ε -Ableitungen sämtlicher Nichtterminale einer kontextfreien Grammatik

© Prof. J.C. Freytag, Ph.D.

5.56

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



LL(1)-Analysealgorithmus

Ausgangssituation:

sei G eine LL(1)-Grammatik und

sei der Stand der Analyse mit der Auswertung der Regel

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

Algorithmus entscheidet, welche Ableitung für A anzuwenden ist

Entscheidungsalgorithmus:

zunächst wird überprüft, ob das Lookahead-Symbol in einer der $\text{First}(\alpha_i)$ -Mengen enthalten ist

- wenn **ja** : ist die Wahl der Regel eindeutig
- wenn **nicht**: gibt es die Möglichkeit der Ableitung: $A \Rightarrow \varepsilon$
 - wenn **ja** : sollte $\text{Follow}(A)$ die Entscheidung bringen
 - wenn **nicht**: (also keine Ableitung u. keine FOLLOW-Lösung) liegt ein syntaktischer Fehler vor



Implementation der LL(1)-Analyse

Prädiktive Top-Down-Verfahren:

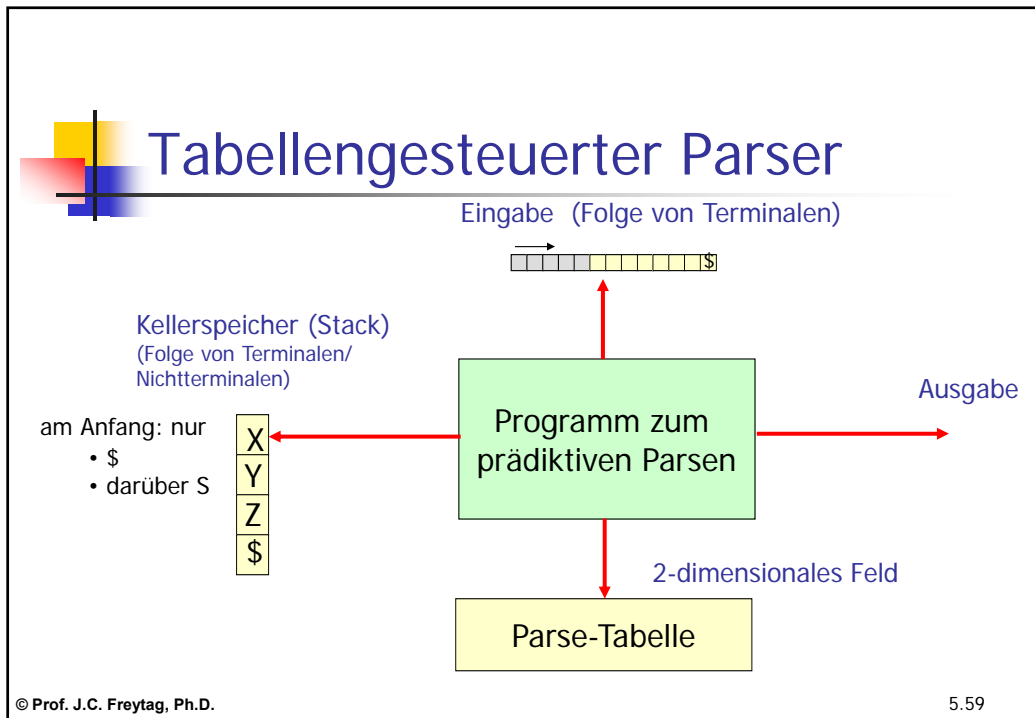
- Rekursiver Abstieg (s. vorherige Folien)
- nicht-rekursive, d.h. tabellengesteuerte Analyse

Bemerkung:

Umsetzung der FIRST- und FOLLOW-Mengen: am günstigsten in Form von Mengen als abstrakter Datentyp (muss in C aber selbst implementiert werden)

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Prädiktive Parse-Tabellen

- für einige Nichtterminale A war zu entscheiden welche Regel (von mehreren Alternativen) aufgrund des nächsten Eingabe-Tokens t zu wählen ist
- wenn die richtige Produktion für jedes (A, t) gefunden werden kann, lässt sich ein prädiktiver Parser mit Hilfe einer Tabelle $M = [A, t]$ bauen

	a	c	d
X	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$ $Y \rightarrow c$	$Y \rightarrow \epsilon$
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$

Grammatik:

- $Z \rightarrow d$
- $Z \rightarrow XYZ$
- $Y \rightarrow \epsilon$
- $Y \rightarrow c$
- $X \rightarrow Y$
- $X \rightarrow a$

keine LL(1)-Grammatik

Bem.:
Eine Grammatik, deren Parse-Tabelle **keine** Mehrfach-Einträge besitzt, ist vom Typ LL(1)

$c \in \text{FIRST}(Y) \cap \text{FOLLOW}(Y)$

© Prof. J.C. Freytag, Ph.D. 5.60

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Tabellengesteuerter Parser: Programmablauf

Parser holt sich zyklisch Werte:

- oberstes Kellersymbol X und aktuelles Eingabesymbol t

mit drei Fortsetzungsmöglichkeiten

- $X = t$ und $t = \$$
der Parser stoppt und meldet erfolgreichen Abschluss der Syntaxanalyse
- $X = t$ und $t \neq \$$ ($t \in V_T$)
Entfernen des obersten Keller-Elementes
Umsetzen des Eingabezeigers auf das nächste Symbol
- $X = A$ ($A \in V_N$)
Parser wertet Eintrag $M[A, t]$ aus:
 - A-Produktion (z.B.: $A \rightarrow UVW$):
oberstes Kellersymbol A wird ersetzt durch UVW (U liegt oben) und die angewendete Produktion wird ausgegeben oder der Syntaxbaum konstruiert
 - **error** (als Eintrag in der Tabelle): Aufruf einer Fehlerbehandlungsroutine

© Prof. J.C. Freytag, Ph.D.

5.61



Vergleich: rekursiver Abstieg – tabellengesteuertes Verfahren

rekursiver Abstieg

- ist leichter zu implementieren und
- die Fehlerbehandlung (Information und Stabilisierung) ist präziser möglich

tabellengesteuertes Verfahren

- effektiver im Platz- und Zeitverbrauch
- Tabelle lässt sich von Hand nur schwer entwerfen (fehleranfällig)
- jedoch gibt es Werkzeuge zur Tabellengenerierung

© Prof. J.C. Freytag, Ph.D.

5.62

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Konstruktion prädiktiver Parse-Tabellen

Idee (Ausnutzung der LL(1)-Eigenschaft)

angenommen $A \rightarrow \alpha$ ist eine Produktion und a ist in $FIRST(\alpha)$, dann expandiert der Parser A zu α , wenn a aktuelles Eingabesymbol ist

im Fall $\alpha = \varepsilon$ oder $\alpha \Rightarrow^* \varepsilon$
muss A zu α expandiert werden, wenn das aktuelle Eingabesymbol a in $FOLLOW(A)$ ist **oder** wenn in der Eingabe die Endmarkierung $\$$ erreicht wurde und $\$$ in $FOLLOW(A)$ enthalten ist

© Prof. J.C. Freytag, Ph.D.

5.63



Konstruktionsalgorithmus

Eingabe: kontextfreie Grammatik G

Ausgabe: Parse-Tabelle M

Methode:

1. Führe für jede Produktion $A \rightarrow \alpha$ die Schritte 2 bis 4 aus
2. Trage für jedes Terminal a aus $FIRST(\alpha)$ die Produktion $A \rightarrow \alpha$ in $M[A, a]$ ein
3. Falls ε in $FIRST(\alpha)$, trage $A \rightarrow \alpha$ für jedes Terminal b aus $FOLLOW(A)$ an der Stelle $M[A, b]$ ein
4. Ist ε in $FIRST(\alpha)$ und $\$$ in $FOLLOW(A)$, so trage $A \rightarrow \alpha$ in $M[A, \$]$ ein
5. trage in jedem undefinierten Eintrag **error** ein

© Prof. J.C. Freytag, Ph.D.

5.64

Vorlesung Compilerbau (SoSe 2018)

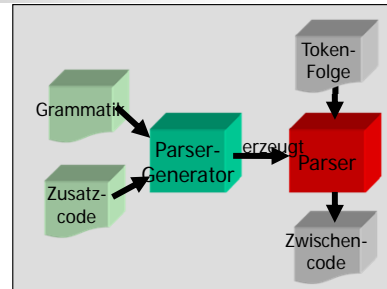
Teil 5: Syntaktische Analyse – Parsing II

Parser-Generatoren für LL(1)-Sprachen

Arbeitsweise

- **Eingabe:** beliebige kontextfreie Grammatik G
- **Ausgabe:**
 - Parser für LL(1)-Grammatik (tabellengesteuert)
 - negativer Fall: Infos über LL(1)-Verletzung
z.B.: Linksrekursivität

1. Berechnung von $M(\epsilon)$: $M(\epsilon) = \{A \mid A \Rightarrow^* \epsilon\}$
2. Bestimmung von $\text{FIRST}(\alpha_i)$ für alle Alternativen
3. Bestimmung von $\text{FOLLOW}(m)$ für alle $m \in M(\epsilon)$
4. Entscheidung, ob G LL(1)-Grammatik
5. aus Informationen oberer Schritte wird eine Parse-Tabelle zusammengefasst
6. der generierte Parser ist unabhängig von der Grammatik und arbeitet auf Grundlage der Parse-Tabelle



© Prof. J.C. Freytag, Ph.D.

5.65

Verfahren zur Sicherung der LL(1)-Eigenschaft (1)

Schlüsseleigenschaft (Wiederholung):

- Falls zwei Produktionen $A \rightarrow \alpha \mid \beta$
- in einer Grammatik vorkommen, soll die Eigenschaft gelten:
$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$
- Diese Eigenschaft würde es dem Parser erlauben, eine korrekte Wahl der richtigen Produktionsregel mit einer Vorausschau von einem Symbol zu treffen

© Prof. J.C. Freytag, Ph.D.

5.66

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Links-Faktorisierung

Frage: Was ist zu tun, wenn die Grammatik diese Eigenschaft nicht besitzt?

Umformung entsprechend folgender Vorgehensweise:

- bestimme für jedes Nicht-Terminalsymbol A den **längsten** Präfix β , den zwei oder mehr Alternativen gemeinsam haben
- Falls $\beta \neq \epsilon$, dann ersetze alle ausgewählten A -Regeln $A \rightarrow \beta\alpha_1 \mid \beta\alpha_2 \mid \dots \mid \beta\alpha_n$ mit
 - $A \rightarrow \beta A'$
 - $A' \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$mit A' als neuem Nicht-Terminalsymbol
- wiederhole die beiden Schritte solange, bis keine Anwendung mehr möglich ist

© Prof. J.C. Freytag, Ph.D.

5.67

Beispiel: Links-Faktorisierung

Grammatik

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

Um zwischen Regeln 2, 3 und 4 wählen zu können, muss der Parser die Token, die nach **num** oder **id** kommen, sehen können, um anhand der Operatoren entscheiden zu können, welche Regel zur Anwendung kommt, **aber**

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \emptyset$$

Grammatik ist damit **nicht** vorhersehbar (nicht-prädiktiv), nicht LL(1) !

© Prof. J.C. Freytag, Ph.D.

5.68

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel Links-Faktorisierung (2)

- Zwei Regeln zur Faktorisierung:

```
<expr> ::= <term> + <expr>
          | <term> - <expr>
          | <term>
```

```
<term> ::= <factor> * <term>
          | <factor> / <term>
          | <factor>
```

- Durch Umformung erhält man

```
<expr> ::= <term> <expr'>
<expr'> ::= + <expr>
          | - <expr>
          | ε
```

```
<term> ::= <factor> <term'>
<term'> ::= * <term>
          | / <term>
          | ε
```

© Prof. J.C. Freytag, Ph.D.

5.69

Beispiel Links-Faktorisierung (3)

- Einsetzen
der veränderten Regeln:

```
1. <goal> ::= <expr>
2. <expr> ::= <term> <expr'>
3. <expr'> ::= + <expr>
4.           | - <expr>
5.           | ε
6. <term> ::= <factor> <term'>
7. <term'> ::= * <term>
8. <term'> ::= / <term>
9.           | ε
10. <factor> ::= num
11.           | id
```

- jetzt ist nur noch eine Vorausschau von einem Zeichen notwendig

© Prof. J.C. Freytag, Ph.D.

5.70

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel Links-Faktorisierung (4)

Prod	Satzform	Eingabe
-	<goal>	^ x - 2 * y
1	<expr>	^ x - 2 * y
2	<term> <expr'>	^ x - 2 * y
6	<factor> <term'> <expr'>	^ x - 2 * y
11	id <term'> <expr'>	^ x - 2 * y
-	id <term'> <expr'>	^ x - 2 * y
9	id g <expr'>	x ^ - 2 * y
4	id - <expr>	x ^ - 2 * y
-	id - <expr>	x - ^ 2 * y
2	id - <term> <expr'>	x - ^ 2 * y
6	id - <term'> <expr'>	x - ^ 2 * y
10	id - num <term'> <expr'>	x - ^ 2 * y
-	id - num <term'> <expr'>	x - 2 ^ * y
7	id - num * <term> <expr'>	x - 2 ^ * y
-	id - num * <term> <expr'>	x - 2 * ^ y
6	id - num * <factor> <term'> <expr'>	x - 2 * ^ y
11	id - num * id <term'> <expr'>	x - 2 * ^ y
-	id - num * id <term'> <expr'>	x - 2 * y ^
9	id - num * id <expr'>	x - 2 * y ^
5	id - num * id	x - 2 * y ^

Das nächste Symbol bestimmt jede Regel korrekt

© Prof. J.C. Freytag, Ph.D.

5.71

Beispiel Links-Faktorisierung (4a)

Prod	Satzform	Eingabe
-	<goal>	^ x - 2 * y
1	<expr>	^ x - 2 * y
2	<term> <expr'>	^ x - 2 * y
6	<factor> <term'> <expr'>	^ x - 2 * y
11	id <term'> <expr'>	^ x - 2 * y
-	id <term'> <expr'>	^ x - 2 * y
9	id g <expr'>	x ^ - 2 * y
4	id - <expr>	x ^ - 2 * y
-	id - <expr>	x - ^ 2 * y

Das nächste Symbol bestimmt jede Regel korrekt

© Prof. J.C. Freytag, Ph.D.

5.72

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II

Beispiel Links-Faktorisierung (4b)

Prod	Satzform	Eingabe
2	id - <term> <expr'>	x - ^ 2 * y
6	id - <term'> <expr'>	x - ^ 2 * y
10	id - num <term'> <expr'>	x - ^ 2 * y
-	id - num <term'> <expr'>	x - 2 ^ * y
7	id - num * <term> <expr'>	x - 2 ^ * y
-	id - num * <term> <expr'>	x - 2 * ^ y
6	id - num * <factor> <term'> <expr'>	x - 2 * ^ y
11	id - num * id <term'> <expr'>	x - 2 * ^ y
-	id - num * id <term'> <expr'>	x - 2 * y ^
9	id - num * id ε <expr'>	x - 2 * y ^
5	id - num * id ε	x - 2 * y ^

Das nächste Symbol bestimmt jede Regel korrekt

© Prof. J.C. Freytag, Ph.D.

5.73

LR(k)- Analyseverfahren

- LR-Verfahren
 - Bottom-Up-Verfahren
 - ohne Rücksetzen und damit effizient (lineare Komplexität)
 - prädiktives Verfahren
- »maßgeschneiderte« Grammatiken für LR(k)-Analyseverfahren sind: ?

LR(k)-Grammatiken
bedeutet: LR(1) Grammatiken

© Prof. J.C. Freytag, Ph.D.

5.74

Vorlesung Compilerbau (SoSe 2018)

Teil 5: Syntaktische Analyse – Parsing II



Fragen??...

