



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 1

Kapitel 4

Syntaktische Analyse: LR Parsing



Autor:
Aho et al.

© Pearson Studium 2008



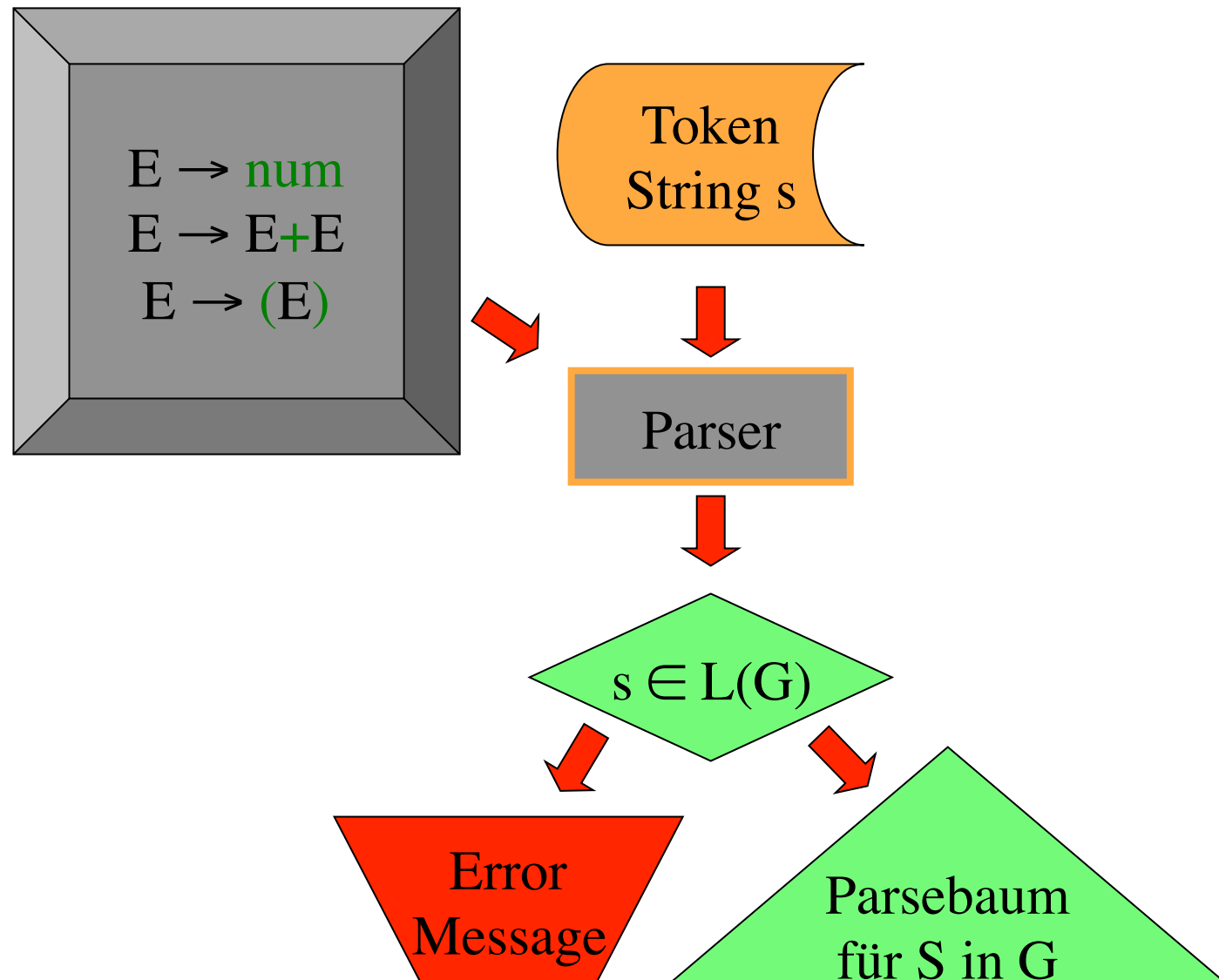
Compiler

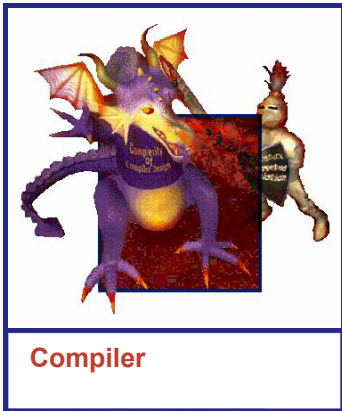
Kapitel 4

Syntaktische Analyse

2

Was ist Parsing





Top-Down/Bottom-up Parsing

$$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex \mid Ex * Ex$$

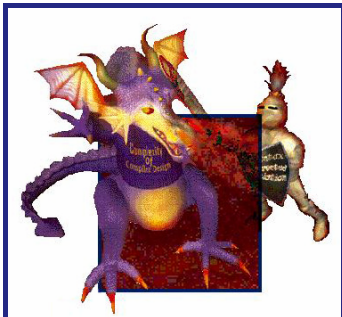
- **Top-down:**

- Man fängt mit dem “Startsymbol” an
- Man versucht den String abzuleiten

- **Bottom-up:**

- Man fängt mit dem String an
- Man versucht das Startsymbol zu erzeugen
- Produktionen werden von rechts nach links angewandt

Nat + Nat
Ex + Nat
Ex + Ex
Ex



Compiler

Kapitel 4

Syntaktische Analyse

LR(k) Bottom-up Parsing

- Links-nach-Rechts Parsing
- **R**echtsableitung
- **k** Token Lookahead

Warum Rechtsableitung ??



Autor:
Aho et al.

© Pearson Studium 2008



Compiler

Kapitel 4

Syntaktische Analyse

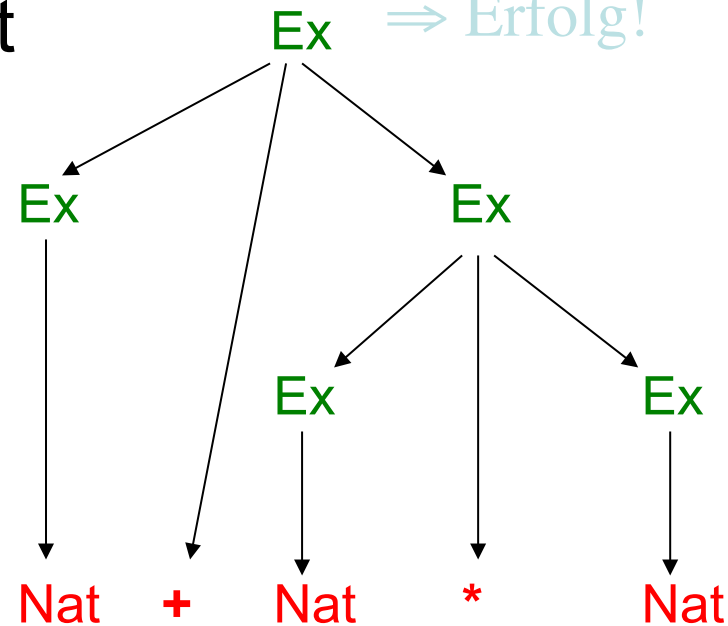
$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex \mid Ex * Ex$

Bottom-Up Parsing

Startsymbol erreicht

\Rightarrow Erfolg!

- $Nat + Nat * Nat$
- $Ex + Nat * Nat$
- $Ex + Ex * Nat$
- $Ex + Ex * Ex$
- $Ex + Ex$
- Ex



Entspricht einer
(top-down) Rechtsableitung!

Parse:

$Nat + Nat * Nat$

P
S

Autor:
Aho et al.

© Pearson Studium 2008



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 6



Autor:
Aho et al.

© Pearson Studium 2008

Warum LR-Parser?

- Praktisch alle Programmiersprachenkonstrukte können erkannt werden
- Mächtiger als LL
 - Weniger Umschreibung von Grammatiken
 - Kompaktere, elegantere Grammatiken
- Aber: mehr Aufwand um einen LR-Parser zu generieren.



Compiler

Kapitel 4

Syntaktische Analyse

Handle

$$E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid \text{id}$$

Teilstring, Rumpf einer Produktion,
Reduktion: Schritt in umgek. Rechtsableitung

id * id

F * id
|
id

T * id
|
 F
|
id

T * F
| |
 F **id**
|
id

T
/ | \
 T * F
| |
 F **id**
|
id

E
|
 T
/ | \
 T * F
| |
 F **id**
|
id

Rechte Satzform	Handle	Reduzierende Produktion
$\text{id}_1 * \text{id}_2$	id_1	$F \rightarrow \text{id}$
$F * \text{id}_2$	F	$T \rightarrow F$
$T * \text{id}_2$	id_2	$F \rightarrow \text{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$$



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 8

Handle II

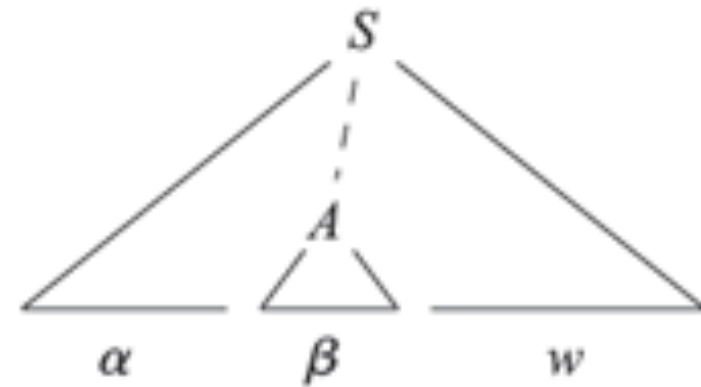


Abbildung 4.27: Ein Handle $A \rightarrow \beta$ im Parse-Baum $\alpha\beta w$

w: nur Terminalsymbole



Compiler

Kapitel 4

Syntaktische Analyse

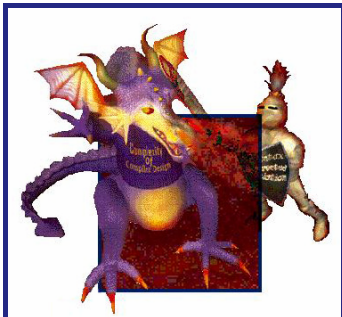
Folie: 9

Shift-Reduce Parsing

Stack	Eingabe
\$	w\$

Verschieben von Eingabe auf Stack (shift)
Reduzieren auf dem Kopf vom Stack (reduce)
Akzeptieren
Fehler erkennen

Stack	Eingabe
\$ S	\$



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 10

Beispiel 1

$$E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid \text{id}$$

Stack	Eingabe	Aktion
\$	id ₁ * id ₂ \$	Verschieben (shift)
\$ id ₁	* id ₂ \$	Reduzieren durch $F \rightarrow \text{id}$
\$ F	* id ₂ \$	Reduzieren durch $T \rightarrow F$
\$ T	* id ₂ \$	Verschieben
\$ T *	id ₂ \$	Verschieben
\$ T * id ₂	\$	Reduzieren durch $F \rightarrow \text{id}$
\$ T * F	\$	Reduzieren durch $T \rightarrow T * F$
\$ T	\$	Reduzieren durch $E \rightarrow T$
\$ E	\$	Akzeptieren

Abbildung 4.28: Konfigurationen eines Shift-Reduce-Parsers bei der Eingabe id₁ * id₂



Compiler

Kapitel 4

Syntaktische Analyse

Seite 11

- Warum ist der Handle immer oben auf dem Stack (und nicht in der Mitte) ?

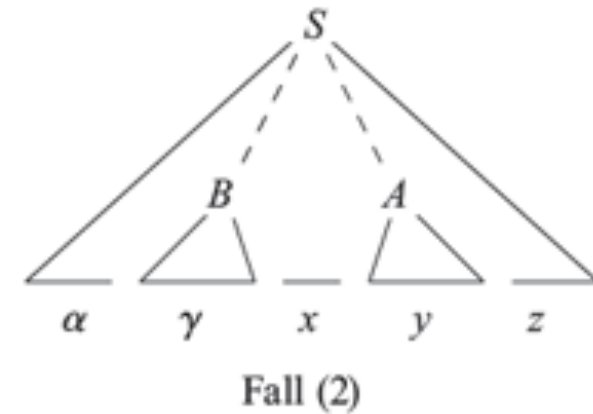
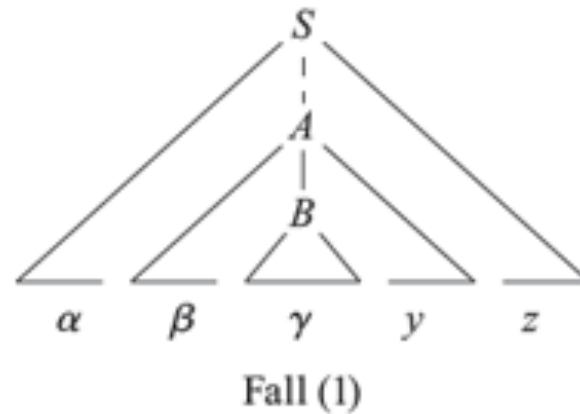


Abbildung 4.29: Mögliche Fälle bei zwei aufeinanderfolgenden Schritten einer Rechtsableitung

Stack	Eingabe		
$\$ \alpha \beta \gamma$	$yz \$$	$\$ \alpha \gamma$	$xyz \$$
$\$ \alpha \beta B$	$yz \$$		
$\$ \alpha \beta B y$	$z \$$	$\$ \alpha B x y$	$z \$$

In beiden Fällen: shift um Handle **oben** auf Stack zu bringen



$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex \mid Ex * Ex$

Shift-Reduce Parsing

Compi

Kapitel 4
Syntak

PEARSON
Stud

Autor:
Aho et al.

© Pearson Studium 2004 **SUCCESS**

STACK		INPUT FILE	ACTION
■		Nat + Nat * Nat	←shift
■	Nat	+ Nat * Nat	↓reduce
■	Ex	+ Nat * Nat	←shift
■	Ex +	Nat * Nat	←shift
■	Ex + Nat	* Nat	↓reduce
■	Ex + Ex	* Nat	←shift
Shift reduce Konflikt			
■	Ex + Ex*	Nat	←shift
■	Ex + Ex* Nat		↓reduce
■	Ex + Ex* Ex		↓reduce
■	Ex + Ex		↓reduce
■	Ex		

Idee vom LR(1) parsing:
Vorhersage der Aktion auf Basis:

- Status vom Stack
- nächstes Token der Eingabe



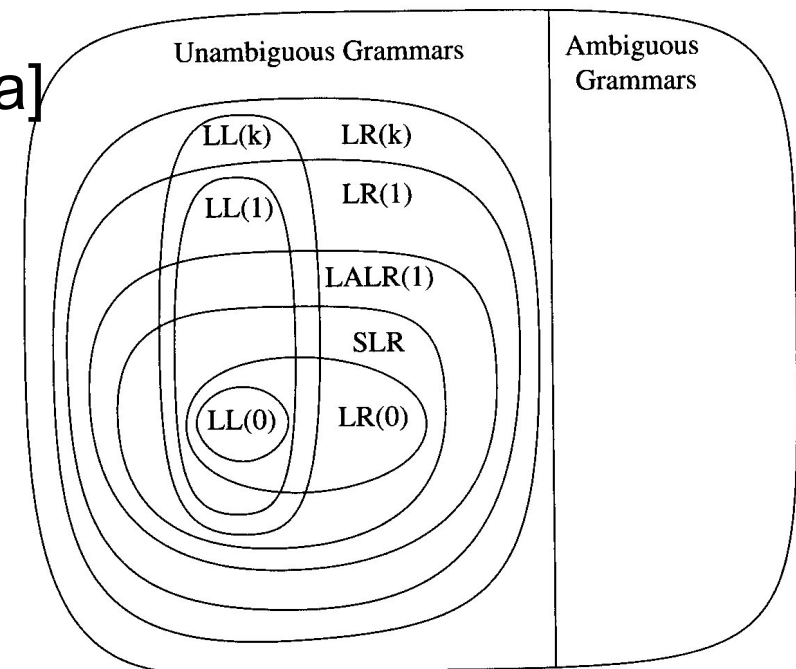
Compiler

Kapitel 4

Syntaktische Analyse

Algorithmen für das Shift-Reduce Parsing

- **LR(0), SLR, LR(k), LALR(k)**
 - LR(k) mächtiger als LL(k)
- Kompliziert
- aber es gibt Parsergeneratoren:
 - **Yacc** [C],
CUP, SableCC [Java]
→ LALR(1) Parser
- In der Vorlesung:
nur ein Beispiel
 - Intuition bekommen





Compiler

Kapitel 4

Syntaktische Analyse

Folie: 14

PEARSON
Studium **it**
informatik

Autor:
Aho et al.

© Pearson Studium 2008

Item

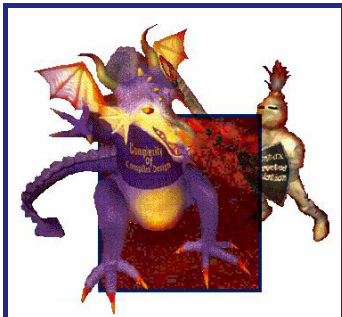
- LR(0)-Item (kurz Item):
 - eine Produktion mit einem Punkt im Rumpf
 - Intuitiv:
 - potenziell anwendbare Regel
 - was links vom Punkt steht haben wir schon gesehen; wir erwarten noch was rechts vom Punkt steht
- Für $A \rightarrow XYZ$:

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 15

Closure (Hülle) von Item-Mengen

Wenn I eine Item-Menge für eine Grammatik G ist, dann ist $\text{CLOSURE}(I)$ die nach den beiden folgenden Regeln aus I konstruierte Item-Menge:

1. Füge zuerst jedes Item von I in $\text{CLOSURE}(I)$ ein.
2. Wenn $A \rightarrow \alpha \cdot B \beta$ in $\text{CLOSURE}(I)$ und $B \rightarrow \gamma$ eine Produktion ist, füge das Item $B \rightarrow \cdot \gamma$ in $\text{CLOSURE}(I)$ ein, falls es noch nicht vorhanden ist. Wende diese Regel an, bis keine weiteren Items mehr in $\text{CLOSURE}(I)$ eingefügt werden können.



Compiler

Kapitel 4

Syntaktische Analyse

PEARSON
Studium **it**
informatik

Autor:
Aho et al.

© Pearson Studium 2008

Ein Beispiel für Closure

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid \text{id}$$

- $I = \{[E' \rightarrow \cdot E]\}$
- Closure(I):

$$\begin{array}{c} I_0 \\ E' \rightarrow \cdot E \\ \hline E \rightarrow \cdot E + T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot \text{id} \end{array}$$



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 17

Funktion GOTO

- GOTO(I,X):
 - Hülle aller Items $[A \rightarrow \alpha X \beta]$ so dass $[A \rightarrow \alpha \cdot X \beta] \in I$
- Wird für Übergänge in einem Automaten verwendet



Compiler

Kapitel 4

Syntaktische Analyse

Ein Beispiel für Goto

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid \text{id}$$

- $I = \{[E' \rightarrow E.], [E \rightarrow E.+T]\}$
- **GOTO(I,+):**

$$E \rightarrow E + \cdot T$$

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

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

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



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 19

Kanonische Sammlung von LR(0)-Item-Mengen

```
void items( $G'$ ) {  
   $C = \{\text{CLOSURE}(\{[SO \rightarrow \cdot S]\})\};$   
  repeat  
    for ( jede Item-Menge  $I$  in  $C$  )  
      for ( jedes Grammatiksymbol  $X$  )  
        if (  $\text{GOTO}(I, X)$  ist nicht leer und nicht in  $C$  )  
          füge  $\text{GOTO}(I, X)$  zu  $C$  hinzu;  
  until es werden keine neuen Item-Mengen mehr in einer Runde zu  $C$  hinzugefügt;  
}
```

Abbildung 4.33: Berechnung der kanonischen Sammlung von LR(0)-Item-Mengen



Compiler

Kapitel 4

Syntaktische Analyse

Ein Beispiel für Items

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

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

$$F \rightarrow (E) \mid \text{id}$$

$$\begin{array}{l} I_0 \\ E' \rightarrow \cdot E \\ \hline E \rightarrow \cdot E + T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot \text{id} \end{array}$$

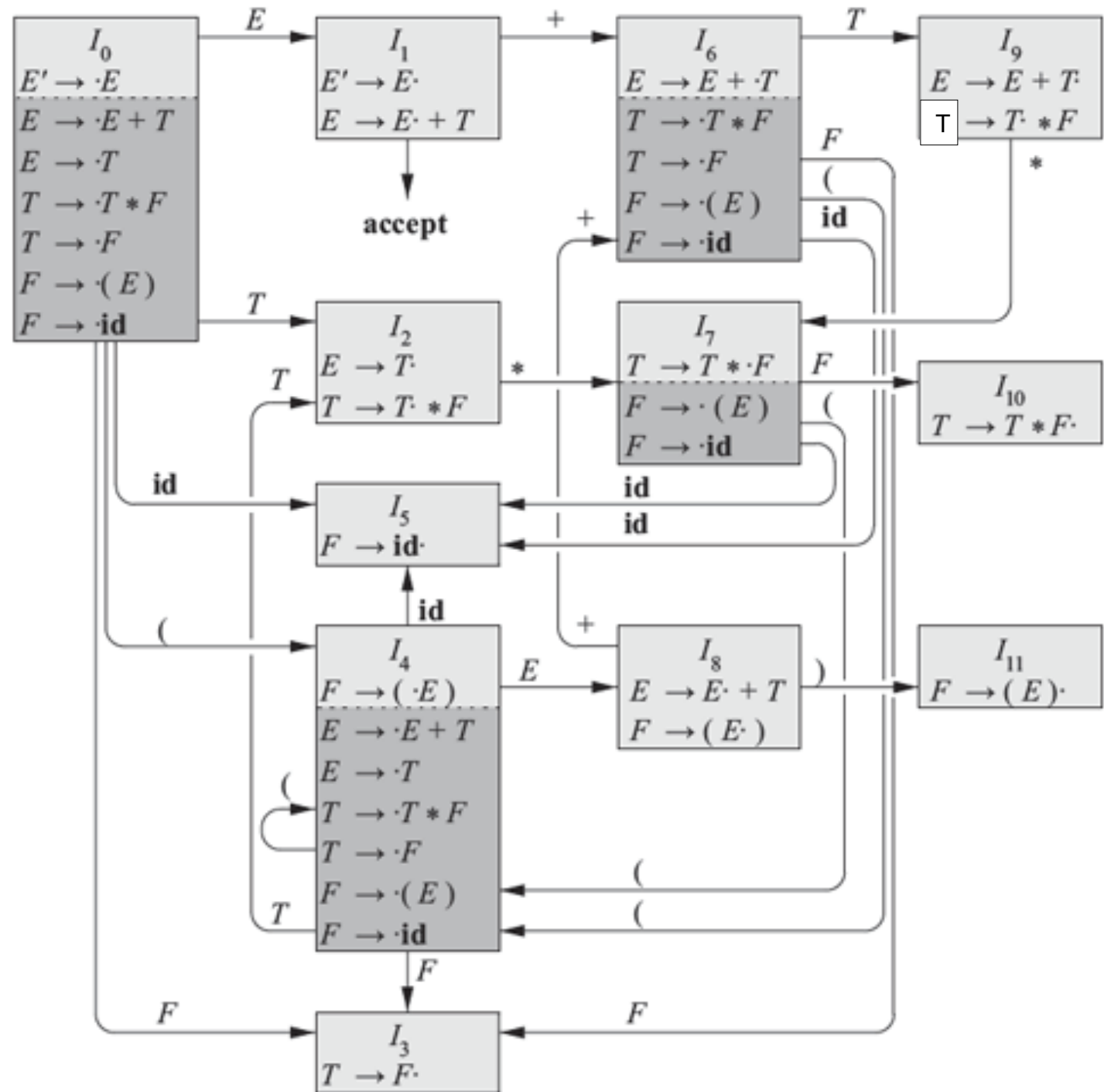


Compiler

Kapitel 4

Syntaktische Analyse

Folie: 21





Compiler

Kapitel 4

Syntaktische Analyse

Folie: 22



Autor:
Aho et al.

© Pearson Studium 2008

Verwendung des LR(0) Automaten: SLR-Parsing

- Startzustand: $\text{Closure}(\{[S' \rightarrow \cdot S]\})$
- Nächstes Eingabesymbol: a
 - Falls Übergang für a existiert: **shift**
 - Falls $[A \rightarrow \alpha \cdot]$ in Item-Menge und $a \in \text{Follow}(A)$:**
 - **reduce** mit Regel $A \rightarrow \alpha$
 - Zustand vom Stapel nehmen, Goto A anwenden
 - Falls $a = \$$ und $[S' \rightarrow S \cdot]$ in Item-Menge: akzeptieren
 - Sonst: Fehlermeldung

** : *Unterschied zwischen LR(0) und SLR*



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 23

s6 =
shift nach Zustand 6
r6 =
reduce mit Regel 6



Autor:
Aho et al.

© Pearson Studium 2008

$$E' \rightarrow E$$

$$E \rightarrow E + T^1 \mid T^2$$

$$T \rightarrow T * F_3 \mid F_4$$

$$F \rightarrow (E)_5 \mid id_6$$

Zustand	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abbildung 4.37: Parsertabelle für Ausdrucksgrammatik



Compiler

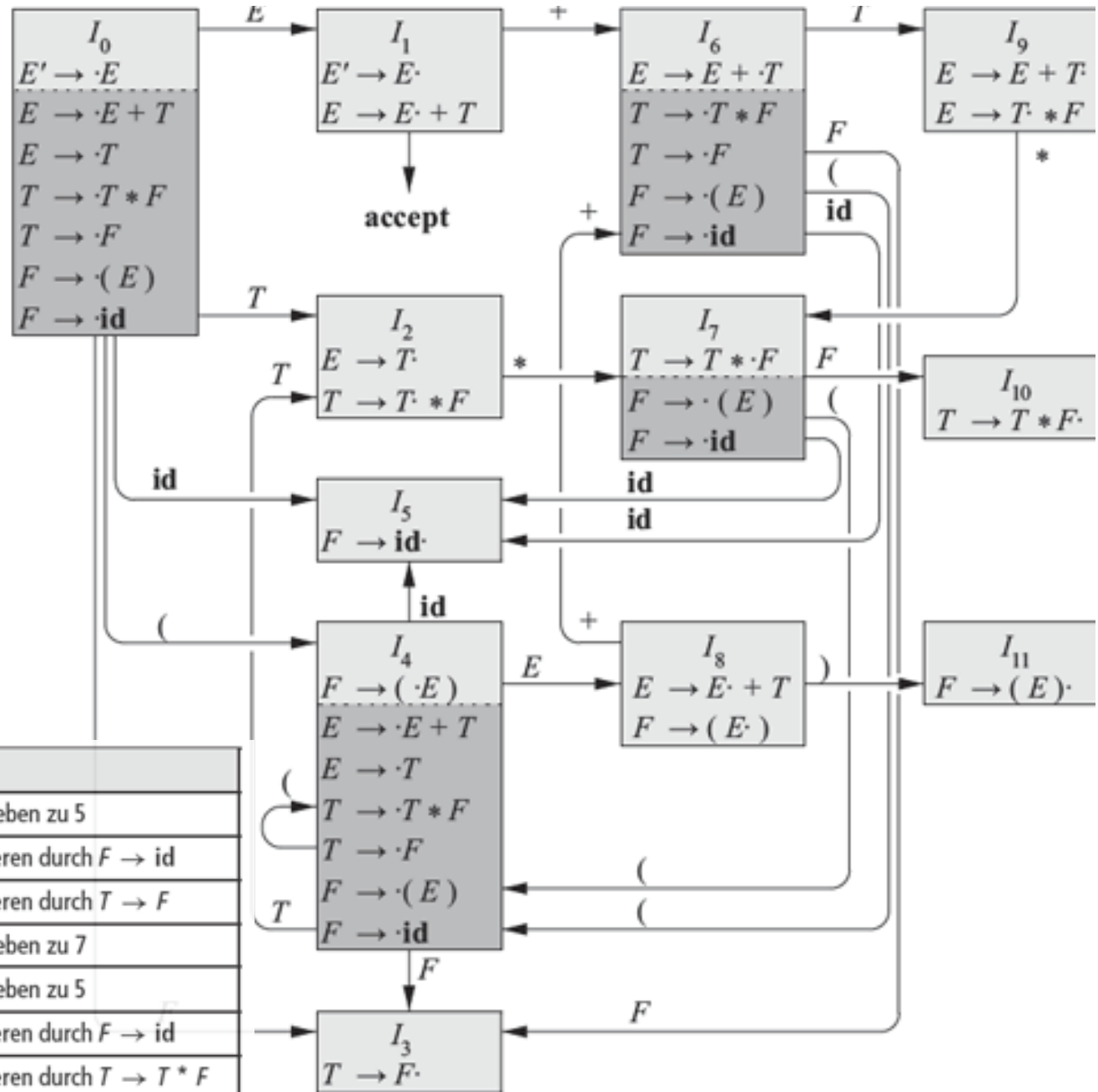
Kapitel 4

Syntaktische Analyse

Folie: 24

id*id

Stack	Symbole	Eingabe	Aktion
0	\$	id * id \$	Verschieben zu 5
05	\$ id	* id \$	Reduzieren durch $F \rightarrow id$
03	\$ F	* id \$	Reduzieren durch $T \rightarrow F$
02	\$ T	* id \$	Verschieben zu 7
027	\$ T *	id \$	Verschieben zu 5
0275	\$ T * id	\$	Reduzieren durch $F \rightarrow id$
02710	\$ T * F	\$	Reduzieren durch $T \rightarrow T * F$
02	\$ T	\$	Reduzieren durch $E \rightarrow T$
01	\$ E	\$	Akzeptieren



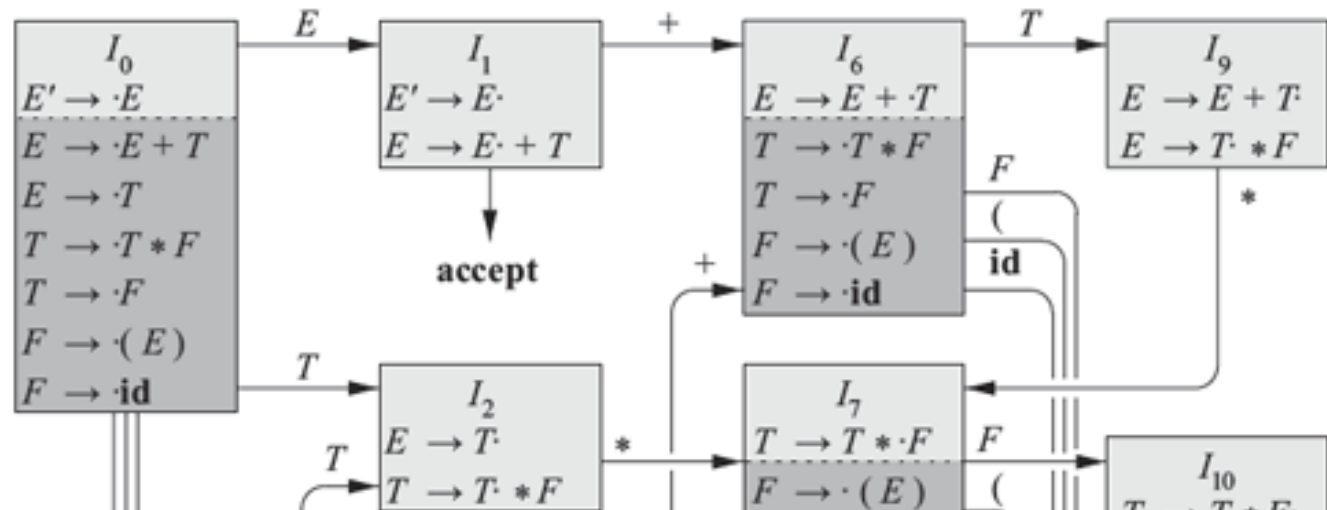


Compiler

Kapitel 4

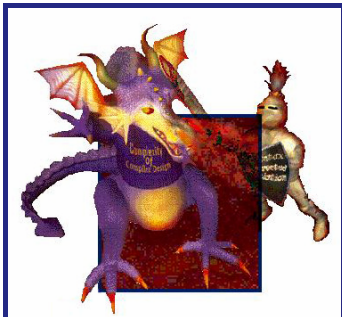
Syntaktische Analyse

Folie 26



	Stack	Symbole	Eingabe	Aktion
(1)	0		id * id + id \$	Verschieben
(2)	05	id	* id + id \$	Reduzieren durch $F \rightarrow id$
(3)	03	F	* id + id \$	Reduzieren durch $T \rightarrow F$
40	02	T	* id + id \$	Verschieben
(5)	027	T*	id + id \$	Verschieben
(6)	0275	T*id	+ id \$	Reduzieren durch $F \rightarrow id$
(7)	02710	T*F	+ id \$	Reduzieren durch $T \rightarrow T * F$
(8)	02	T	+ id \$	Reduzieren durch $E \rightarrow T$
(9)	01	E	+ id \$	Verschieben
(10)	016	E+	id \$	Verschieben
(11)	0165	E+id	\$	Reduzieren durch $F \rightarrow id$
(12)	0163	E+F	\$	Reduzieren durch $T \rightarrow F$
(13)	0169	E+T	\$	Reduzieren durch $E \rightarrow E + T$
(14)	01	E	\$	Akzeptieren

© Pearson Studium 2008



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 26

Einschränkungen von SLR

$$S \rightarrow L = R \mid R$$

$$L \rightarrow *R \mid id$$

$$R \rightarrow L$$

Grammatik eindeutig

I_2 bei =:
 - shift nach I_6
 - reduce $R \rightarrow L$

$$I_0: S' \rightarrow \cdot S$$

$$S \rightarrow \cdot L = R$$

$$S \rightarrow \cdot R$$

$$L \rightarrow \cdot * R$$

$$L \rightarrow \cdot id$$

$$R \rightarrow \cdot L$$

$$I_1: S' \rightarrow S \cdot$$

$$I_2: S \rightarrow L \cdot = R$$

$$R \rightarrow L \cdot$$

$$I_3: S \rightarrow R \cdot$$

$$I_4: L \rightarrow * \cdot R$$

$$R \rightarrow \cdot L$$

$$L \rightarrow \cdot * R$$

$$L \rightarrow \cdot id$$

$$I_5: L \rightarrow id \cdot$$

$$I_6: S \rightarrow L = \cdot R$$

$$R \rightarrow \cdot L$$

$$L \rightarrow \cdot * R$$

$$L \rightarrow \cdot id$$

$$I_7: L \rightarrow * R \cdot$$

$$I_8: R \rightarrow L \cdot$$

$$I_9: S \rightarrow L = R \cdot$$

Abbildung 4.39: Kanonische LR(0)-Sammlung für die Grammatik (4.16)



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 27



Autor:
Aho et al.

© Pearson Studium 2008

LR(1)-Items

- $[A \rightarrow \alpha.\beta, a]$ wobei a der Lookahead ist
- $[A \rightarrow \alpha., a]$ bedeutet dass eine Reduktion nur möglich ist wenn das nächste Eingabesymbol a ist
- LALR(k): “komprimierte” Fassung von LR(k)

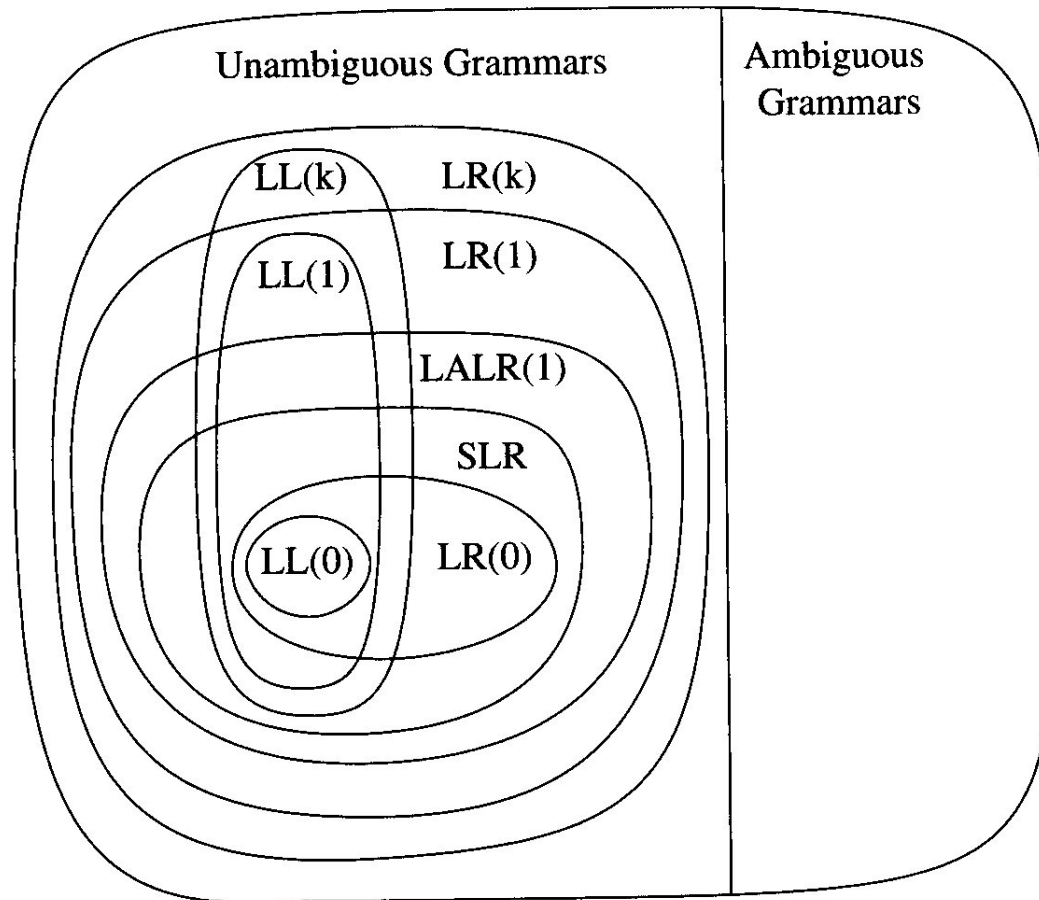


Compiler

Kapitel 4

Syntaktische Analyse

Overview





Compiler

Kapitel 4

Syntaktische Analyse

Mehrdeutige Grammatiken: Shift-Reduce Konflikte

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

- if a then if b then s1 else s2
 - if a then { if b then s1 else s2 } or
 - if a then {if b then s1} else s2
- if E then if E then S else s2
 - if E then if E then S else s2
 - if E then S else s2

← shift

↓ reduce

Eine Lösung: ← **shift** als “Default”

Autor:
Aho et al.

© Pearson Studium 2008



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 30

Yacc

```
yacc translate.y
```

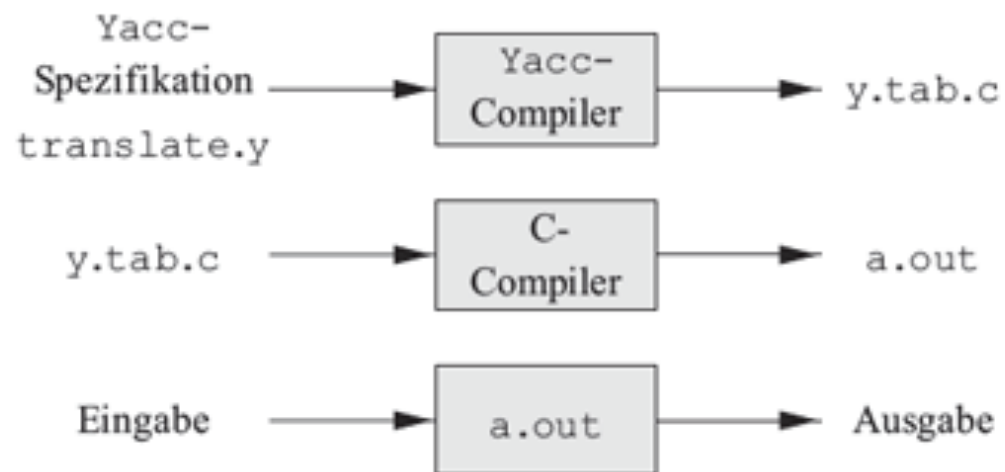


Abbildung 4.57: Erstellen eines Ein-/Ausgabe-Übersetzers mit Yacc

Deklorationen

```
%%
```

Übersetzungsregeln

```
%%
```

Unterstützende C-Routinen



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 31

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Typ double für den Yacc-Stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n' { printf("%g\n", $2); }
      | lines '\n'
      | /* leer */
      ;

expr  : expr '+' expr  { $$ = $1 + $3; }
      | expr '-' expr  { $$ = $1 - $3; }
      | expr '*' expr  { $$ = $1 * $3; }
      | expr '/' expr  { $$ = $1 / $3; }
      | '(' expr ')'   { $$ = $2; }
      | '-' expr %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;

%%

yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( ( c == '.' ) || ( isdigit(c) ) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Abbildung 4.59: Yacc-Spezifikation für einen verbesserten Taschenrechner



Compiler

Kapitel 4

Syntaktische Analyse

Für die Klausur

- Bottom-up Shift/Reduce Parsing
- Prinzip des SLR Parsing
- Benutzen eines (LA)LR Parsergenerators

- Nicht notwendig:
 - Unterschied zwischen LR,LALR,SLR



Autor:
Aho et al.

© Pearson Studium 2008



Compiler

Kapitel 4

Syntaktische Analyse

Folie: 33



Autor:
Aho et al.

© Pearson Studium 2008

Was haben wir gelernt

- **Lexer**
 - Reguläre Ausdrücke, NFA, DFA
 - lex
- **Parser**
 - kontextfreie Grammatiken
 - LL(1) Rekursiv-absteigendes Parsing
 - Shift-Reduce Bottom-up Parsing, SLR
- **Was kommt:**
 - SableCC Lexer+Parser Generierung
 - Semantische Analyse