

**Bachelor-Programm**

# **Compilerbau**

im SoSe 2014

Prof. Dr. Joachim Fischer  
Dr. Klaus Ahrens  
Dipl.-Inf. Ingmar Eveslage

[fischer@informatik.hu-berlin.de](mailto:fischer@informatik.hu-berlin.de)



# Position

④ **Teil I**  
Die Programmier

④ **Teil II**  
Methodische Grund

④ **Teil III**  
Entwicklung ein

④ **Kapitel 1**  
Compilationsprozess

④ **Kapitel 2**  
Formalismen zur Sprachbeschreibung

④ **Kapitel 3**  
Lexikalische Analyse: der Scanner

④ **Kapitel 4**  
Syntaktische Analyse: der Parser

④ **Kapitel 5**  
Parsegeneratoren: Yacc, Bison

④ **Kapitel 6**  
Statische Semantikanalyse

④ **Kapitel 7**  
Laufzeitsysteme

④ **Kapitel 8**  
Ausblick: Codegenerierung

④ **4.1**  
Einführung in die Syntaxanalyse

④ **4.2**  
Restrukturierung von  
Grammatiken

④ **4.3**  
LL-Parser

④ **4.4**  
Beispiel: Ein-Pass-Compiler  
(Parser, Übersetzer)

④ **4.5**  
Tabellengesteuerter LL-Parser

④ **4.6**  
Tabellengesteuerter LR-Parser

## **4.6.1 Allgemeine Betrachtung**

- Einordnung von LR-Parsern
- Allgemeines Prinzip von Shift-Reduce-Verfahren
- Klassifikation von LR-Analysemethoden/Grammatiken

# LR-Parser (Wdh.)

---

... sind aus mehreren Gründen interessant:


- LR-Parser können für praktisch alle Programmiersprachen, für die kontextfreie Grammatiken existieren, konstruiert werden.
- LR-Parser sind allgemeiner als viele andere Techniken, besitzen aber dennoch die gleiche Effizienz.
- Selbst wenn die Grammatik Mehrdeutigkeiten enthält, können - mit notfalls **manuellen Eingriffen** - effiziente Parser erzeugt werden.
- LR-Parser erkennen Syntaxfehler zum frühesten möglichen Zeitpunkt.
- Für reale Programmiersprachen kann man diese Parser jedoch nicht mehr manuell konstruieren. Man benötigt Werkzeuge.  
**yacc** / **bison** als prominenteste Vertreter.
- Ein solcher Parser besteht aus zwei Teilen:
  1. einem **Treiber (= Automaten-Programm)**, der immer gleich ist, und
  2. einer **Tabelle**, die aus der Grammatik generiert wird.

# Unterschiedliche Methoden

---

... zur Erzeugung von LR-Parsertabellen:

- **Simple LR** oder kurz: **SLR**  
ist am einfachsten zu implementieren,  
funktioniert aber für einige Grammatiken nicht.
- **(kanonisches) LR**  
ist das mächtigste Verfahren,  
aber in der Implementierung sehr aufwendig.
- **LALR** oder **lookahead LR**  
liegt in der Komplexität zwischen den beiden Verfahren  
und  
wird von **yacc /bison** verwendet



gehören zur  
Klasse der  
**Shift-Reduce-  
Analysetechnik**

# Prinzip einer Shift-Reduce-Analyse

- sukzessive Anwendung von **Reduktionsschritten** im Bottom-Up-Stil, um letztendlich ein Eingabewort **w** (Programm) auf das Startsymbol **S** der Grammatik zu reduzieren

- einzelner Ersetzungsschritt (informal):

Suche nach einer geeigneten Teilzeichenkette in der resultierenden Satzform des vorhergehenden Schrittes

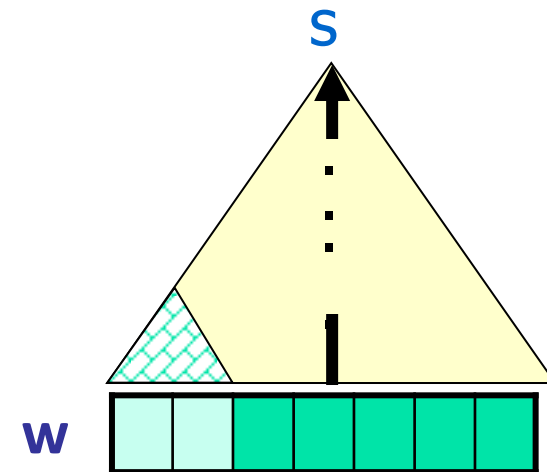
1 Teilzeichenkette

die mit rechter Seite einer Grammatik-Regel übereinstimmt

→ Variable

die mit linker Regel-Seite der bestimmten Grammatik-Regel übereinstimmt

- 2 wenn die Teilzeichenketten dabei so gewählt werden, dass eine Rechtsableitung in umgekehrter Reihenfolge vorliegt, handelt es sich um eine **Shift-Reduce-Analyse**



# Prinzip der Bottom-Up-Verfahren

## Strategie

- Suche den Rand des Baumes nach einer passenden RS ab.
- Wird eine passende RS gefunden, wird die die zur Variable (LS) reduziert.
- Verfahre so weiter, bis  $LS == \text{Startsymbol}$ .

## Schlüsselproblem

Situation: komplette RS einer Regel wird am Baumrand entdeckt

Frage: Soll jetzt reduziert werden oder

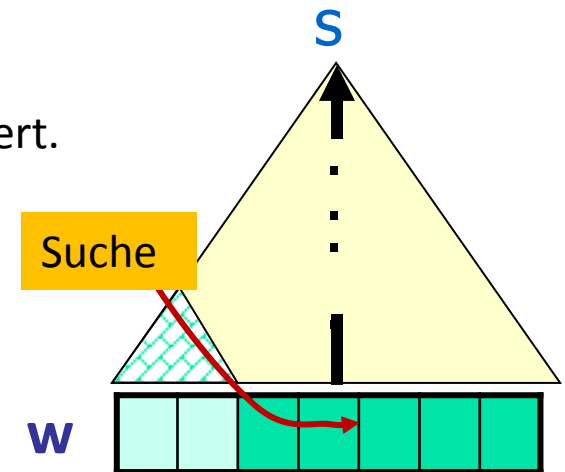
sollen noch weitere Symbole hinzugenommen und weiter gesucht werden?

(d.h. welches Teilstück im Baum ist zu reduzieren/beschneiden ?)

*shift oder reduce ?*

Welche Produktion soll zur Anwendung gebracht werden ?

→ wollen die Beantwortung zunächst noch vertagen u. beschäftigen uns zunächst weiter mit dem **Analyseprinzip**



# Beispiel: Aufbau des Syntaxbaumes ~ Bottom-Up

a b b c d e \$

Ausgangssatzform: Eingabewort

a b b c d e \$

Kandidaten für Teilzeichenkette: **b** (an 2 Positionen), **d**

a A b c d e \$

Reduktion: [3. Regel]

a A b c d e \$

Kandidaten: **Abc**, **b**, **d**

a A d e \$

Reduktion: [2. Regel]

a A c

Kandidaten: **c**

Grammatik

1.	S	→	aABe\$
2.	A	→	Abc
3.			b
4.	B	→	d

a A B e \$

Reduktion: [4. Regel]

a A B e \$

Kandidaten: **aABe\$**

a A c

Reduktion: [1. Regel]

FERTIG

## Beobachtung

tatsächlich ergeben diese 4 Reduktionen in umgekehrter Reihenfolge (bei richtiger Wahl der **Reduktionskandidaten**) eine Rechtsableitung

$S \xrightarrow{[1]} aABe \xrightarrow{[4]} aAde \xrightarrow{[2]} aAbcde \xrightarrow{[3]} abbcd e \$$



# Handle – als Beschreibung geeigneter Reduktionskandidaten

---

## Ziel

Suche in einer vorliegenden abgeleiteten **Satzform**  $\gamma$ , die

- aus Terminalsymbolen und Variablen bestehen kann und
- einer Rechtsableitung entstammt,

nach einem **Teilstring**  $\beta$ , der

1. mit der rechten Seite einer Regel  $A \rightarrow \beta$  übereinstimmt und
2. dessen **Reduktion** zum Nichtterminal  $A$  in einer **umgekehrten Rechtsableitung** führt.

Der Teilstring  $\beta$  wird auch **Griff** (engl. „Handle“) genannt.

# Handle

## Definition: Handle

Sei  $G$  eine kfG und sei

$S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$  eine Rechtsableitung in  $G$ ,

dann heißt  $\beta$  ein Handle der Satzform  $\alpha \beta w$

## Bemerkung

1. Teilwort  $w$  rechts vom Handle enthält nur (bislang noch nicht analysierte) Terminalsymbole:  $w \in \Sigma^*$ .
2. Ein **Handle**  $\beta$  einer **Satzform**  $\gamma$  ist bestimmt durch
  - a) eine Regel  $A \rightarrow \beta$  von  $G$  und
  - b) eine Position in  $\gamma$ , unter der  $\beta$  gefunden und durch  $A$  ersetzt werden kann.

# Illustration: Handle-Bestimmung (1)

initiales  $\gamma$

ist eine Satzform, die nur aus Terminalsymbolen besteht

a b b c d e \$

Eingabewort

Beobachtung für letztes Beispiel

$S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abcde$

Grammatik

1.	S	$\rightarrow aABe\$$
2.	A	$\rightarrow Abc$
3.		b
4.	B	$\rightarrow d$

$\gamma = abcde$  entstammt einer Rechtsableitung, Handle  $\beta = b$  mit  $A \rightarrow b$  an Position 2

$\gamma = aAbcde$  entstammt einer Rechtsableitung, Handle  $\beta = Abc$  mit  $A \rightarrow Abc$  an Position 2

Kurzsprechweise: Teilstring  $\beta = b$  ist Handle von  $abcde$

Teilstring  $\beta = Abc$  ist Handle von  $aAbcde$

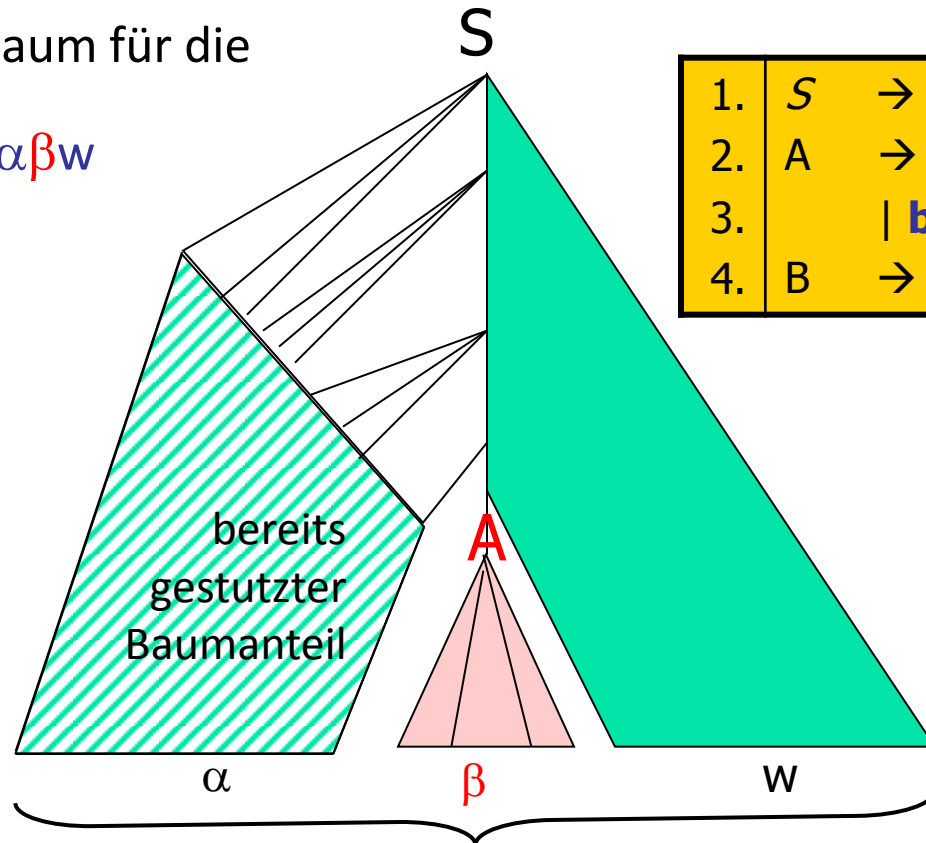
Teilstring  $\beta = b$  ist **kein** Handle von  $abcde$

# Illustration : Handle-Bestimmung (2)

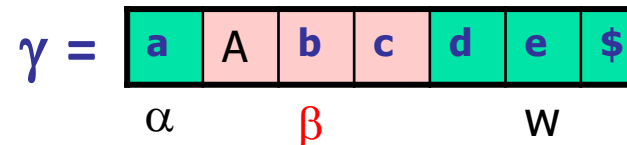
Handle  $\beta$  mit  $A \rightarrow \beta$  im Syntaxbaum für die rechtsabgeleitete Satzform  $\gamma = \alpha\beta w$

$A$  ist der **tiefste** und am **weitesten links** stehende innere Knoten, der noch **alle** seine Kindknoten im Baum besitzt

Reduktion von  $\beta$  zu  $A$  entspricht dem **Beschneiden eines Baumes** (von links unten nach rechts oben)



1.	$S$	$\rightarrow$	$aABe\$$
2.	$A$	$\rightarrow$	$Abc$
3.		$ $	$b$
4.	$B$	$\rightarrow$	$d$



# Eindeutigkeit eines Handles

## Satz: Existenz eindeutig bestimmter Handle

Falls die Grammatik  $G$  eindeutig ist,  
dann gibt es in jeder Satzform einer Rechtsableitung  
ein **eindeutig** bestimmtes Handle

Beweis-Idee:

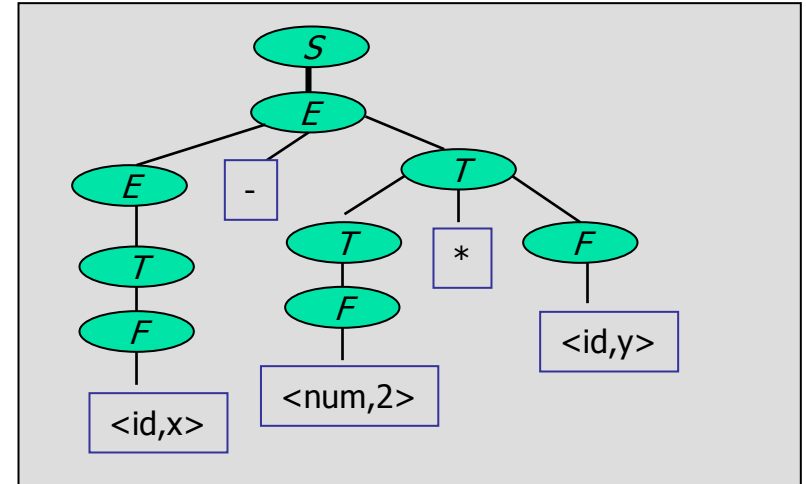
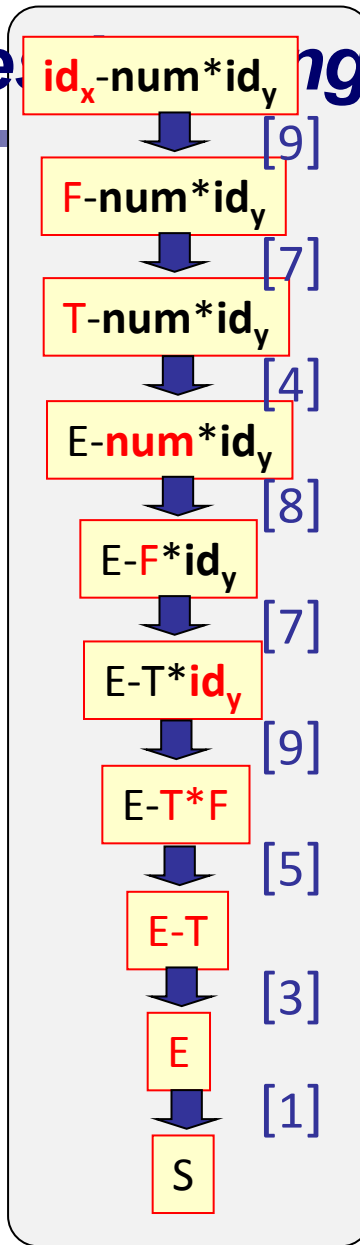
Anwendung der Definitionen

1.  $G$  ist eindeutig  $\rightarrow$  Rechtsableitung ist eindeutig
2.  $\rightarrow$  es existiert eine eindeutige Regel  $A \rightarrow \beta$ , die  $\gamma_{i-1}$  nach  $\gamma_i$  überführt
3.  $\rightarrow$  es existiert eine eindeutige Position  $k$ , an der die Regel  $A \rightarrow \beta$  angewandt wird
4.  $\rightarrow \beta$  ist das eindeutige Handle

# Beispiel: Handle-Belegung

1	S	→	E
2	E	→	E + T
3			E - T
4			T
5	T	→	T * F
6			T / F
7			F
8	F	→	num
9			id

Ergebnis  
umgekehrte Rechtsableitung



x - 2 \* y *Eingabewort*

Auswahl der Reduktionskandidaten  
id, F, T, num, F, id, T \* F, E - T, E  
als **Handle**  
in den jeweiligen Satzformen ist 1-deutig

d.h. andere Reduktionskandidaten  
führen **nicht** zu einer erfolgreichen  
Analyse

# Handle-Pruning-Prozess

## Definition: Handle-Pruning (Baum-Ausästung)

... bezeichnet einen Prozess,  
Syntaxbäume nach einer Bottom-Up-Strategie zu konstruieren

Sei  $w$  resultierender String einer Rechtsableitung

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

dann ist folgender einfacher Algorithmus ein *Handle-Pruning*-Algorithmus

**for**  $i = n$  **downto**  $0$  **do**

    finde die Handle  $A_i \rightarrow \beta_i$  in  $\gamma_i$ ;

    ersetze  $\beta_i$  durch  $A_i$ , um  $\gamma_{i-1}$  zu generieren

**endfor**

**Bemerkung:** Handle-Pruning wird durch Shift-Reduce-Parser implementiert

# Shift-Reduce-Parser (SR-Parser)

## verwendete Datenstrukturen

Kellerspeicher und Eingabepuffer

## Aktionen (grob)

- 1) **initialisiere** den Keller mit einem Zeichen (z.B. **\$** oder **#**) zur Kennzeichnung des Kellerbodens
  
- 2) wiederhole die Aktionen
  - a) **Shift** (Keller füllen) und
  - b) **Reduce** (Kellerinhalt ersetzen)solange,  
bis das oberste Kellerzeichen das **Startsymbol** ist  
und das Eingabesymbol das **\$-Zeichen**
  
- 3) führe **Accept** aus



# Aktionen: Shift-Reduce-Parsing

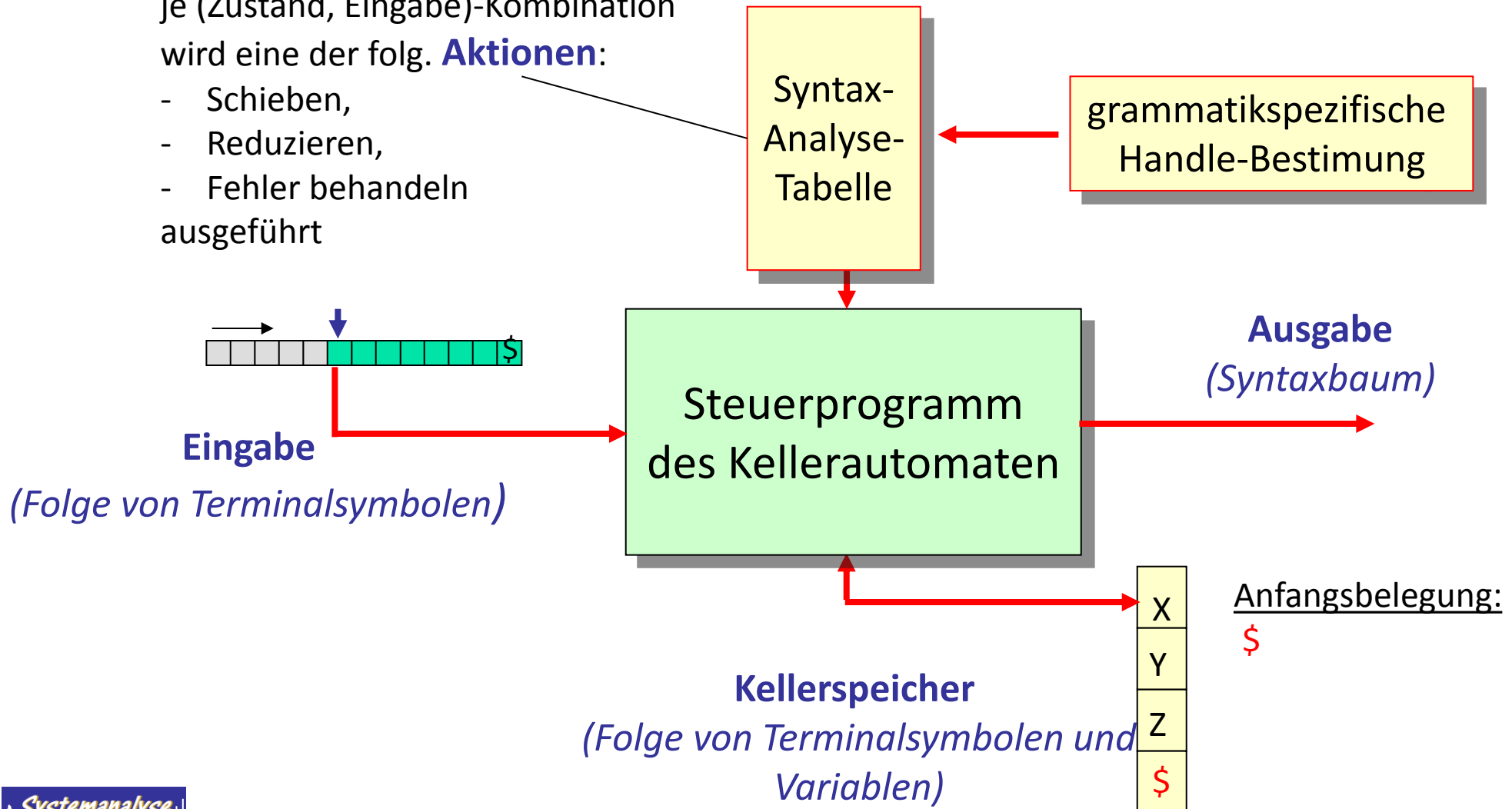
- **Shift**  
**schiebe** aktuelles Symbol der **Eingabe** auf den Keller:  
solange null oder mehrere Eingabesymbole auf den Keller,  
bis ein **Handle**  $\beta$  auf dem Keller zum Liegen kommt  
[rechtes Ende vom Handle liegt dann oben auf dem Keller]
- **Reduce**  
falls ein **Handle**  $\beta$  (mit  $A \rightarrow \beta$  auf dem Keller liegt, **reduziere** wie folgt:
  - nimm  $|\beta|$  Symbole aus dem Keller (**pop-Operation**)
  - ersetze diese durch  $A$  auf dem Keller (**push-Operation**)
- **Accept**  
**beende** das Erkennen und signalisiere Erfolg
- **Error**  
**rufe** eine Fehlerstabilisierungsroutine auf

Nach wie vor offenes **Problem**:  
Erkennung der einzelnen Handle

# Prinzip aller Shift-Reduce-Parser

je (Zustand, Eingabe)-Kombination  
wird eine der folg. **Aktionen**:

- Schieben,
- Reduzieren,
- Fehler behandeln  
ausgeführt

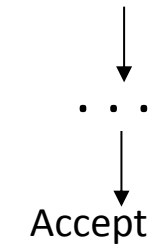


Nr.	Keller	Eingabe	Aktion
1.	\$	id - num * id \$	Shift: id
2.	\$ id	- num * id \$	Reduce: 9
3.	\$ F	- num * id \$	Reduce: 7
4.	\$ T	- num * id \$	Reduce: 4
5.	\$ E	- num * id \$	Shift: -
6.	\$ E -	num * id \$	Shift: num
7.	\$ E - num	* id \$	Reduce: 8
8.	\$ E - F	* id \$	Reduce: 7
9.	\$ E - T	* id \$	Shift: *
10.	\$ E - T *	id \$	Shift: id
11.	\$ E - T * id	\$	Reduce: 9
12.	\$ E - T * F	\$	Reduce: 5
13.	\$ E - T	\$	Reduce: 3
14.	\$ E	\$	Reduce: 1
15.	\$ S	\$	Accept

1	S → E
2	E → E + T
3	E - T
4	T
5	T → T * F
6	T / F
7	F
8	F → num
9	id

## Ablauf ...

Startkonfiguration [1.]



### Offene Probleme

auch bei eindeutiger Grammatik:

- (1) Wie erkennt man die eindeutig bestimmten Handle ?
- (2) Wie sind die Aktionen auszuwählen, um die korrekte Arbeit des Parsers zu garantieren?

Probleme werden in Abh. des Grammatiktyps unterschiedlich behandelt



# LR(k)-Grammatiken und Shift-Reduce-Parser

## Anschauliche Charakterisierung:

Eine **Grammatik G** ist vom **Typ LR(k)**, falls

- ein von links nach rechts arbeitender **Shift-Reduce-Parser** in der Lage ist,

für eine gegebene **Rechtsableitung**

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

und

für **jede rechte Satzform  $\gamma_i$**  in der obigen Ableitung

- den **Handle** in dieser rechten Satzform sowie
- die **Reduktionsregel**

zu bestimmen,

- wobei er maximal **k** Symbole über das rechte Ende der Handle für  $\gamma_i$  **hinausschauen** darf, um seine **Shift-Reduce-Entscheidungen** zu treffen

SR-Parser dürfen niemals eine „**gebotene**“ Reduktion verpassen, d.h. Kellerinhalt darf nicht verdeckt werden

## **4.6.1 Allgemeine Betrachtung**

- Allgemeines Prinzip von Shift-Reduce-Verfahren
- Klassifikation von LR-Analysemethoden/Grammatiken/Sprachen

# Konstruktionsarten von LR-Syntaxanalysetabellen

verschiedene Techniken zur **Konstruktion** einer LR-Syntaxanalysetabelle

## Varianten

- einfache LR- (kurz: **SLR-**) Analyse
- kanonische **LR**-Analyse (mächtigstes Verfahren)
- vorausschauende LR- (kurz: **LALR-**) Analyse
- ...

→ Führen zur Akzeptanz unterschiedlicher Sprachklassen

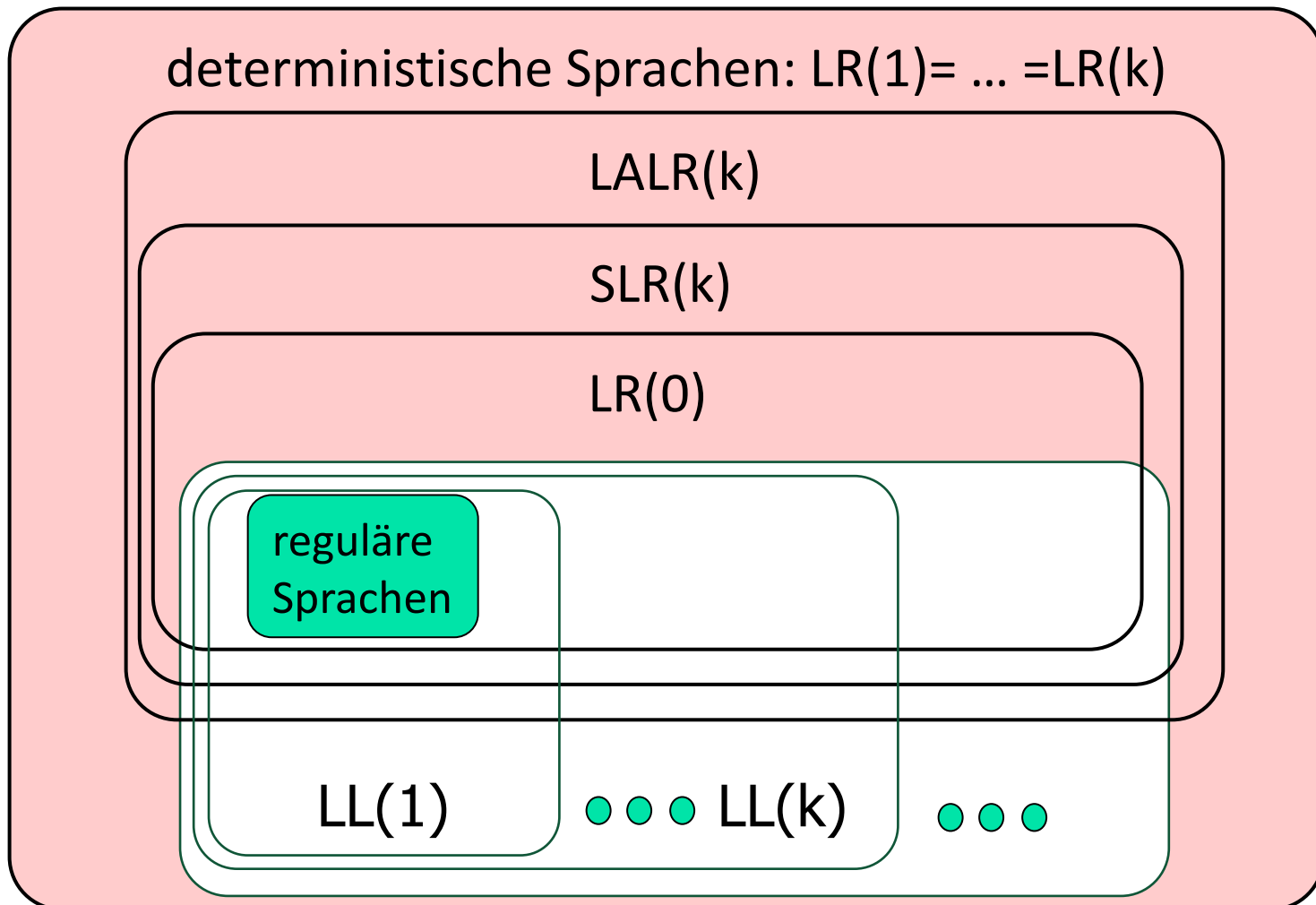
**OP**  $\subset$  LR(0)  $\subset$  SLR(1)  $\subset$  LALR(1)  $\subset$  LR(1) = LR(k)

Operator-Präzedenz-Analyse  
(einfach, leicht zu implementieren)

*Betrachtung  
in der Vorlesung:*

1. LR(0)
2. SLR(1)
3. LR(1)
4. LALR(1)
5. LR(k)

# Sprachmächtigkeit ~ Shift-Reduce-Verfahren



## 4.6.1 Allgemeine Betrachtung

- Allgemeines Prinzip von Shift-Reduce-Verfahren
- Klassifikation von LR-Analysemethoden/Grammatiken/Sprachen
- Präzisiertes Automatenmodell eines beliebigen LR-Parsers

**gültig für**

1. LR(0)
2. SLR(1)
3. LR(1)
4. LALR(1)
5. LR(k)



# Präzisiertes Automatenmodell (Merkmale)

1 zweiteilige austauschbare  
Syntaxanalysetabelle

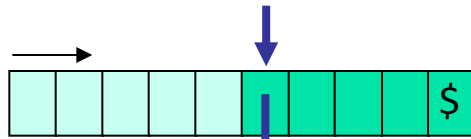
State	Action			Goto		
	id	+	...	S	E	...
0						
1						

neuer Folgezustand  
im Fall einer Reduktion  
zu einer Variablen

shift/reduce/  
accept/error

Eingabe

(Folge von Terminalen)



LR-  
Syntaxanalyse-  
Programm

Ausgabe

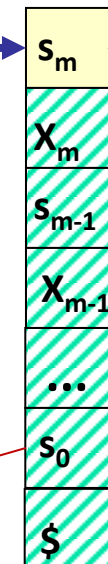
(Syntaxbaumaufbau)

2 zustandsorientierte  
Arbeitsweise

der Automat besitzt einen  
Startzustand

Aktion und Folgezustand  
sind für einen  
Ausgangszustand bei  
gegebenen Eingangssymbol  
eindeutig bestimmt

3 Belegung  
Folge von Paaren  
(Zustand,  
Grammatiksymbol)

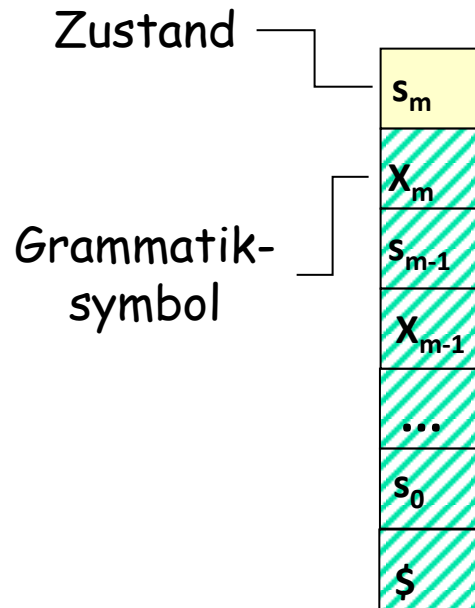


Kellerspeicher

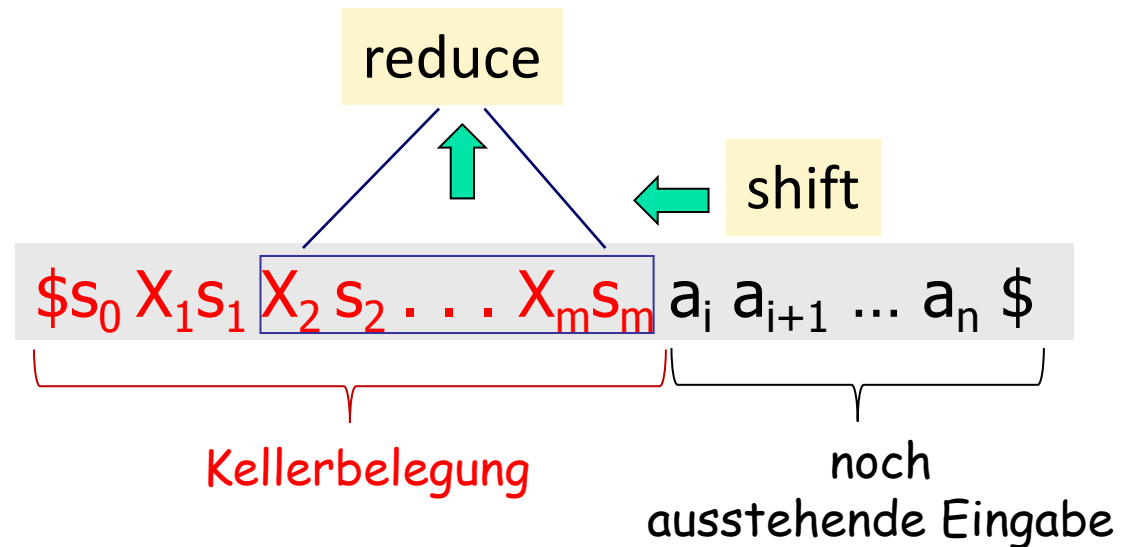
Zustandssymbol  
als Abstraktion  
des  
darunter  
liegenden  
Kellerinhaltes

Startzustand

# Beliebiger LR-Parser: Zustandsverwaltung



- Zustände charakterisieren den Inhalt des Stacks „unter“ ihnen
- Speicherung der Grammatiksymbole ist nicht notwendig - werden der Übersichtlichkeit halber weiterhin angegeben



# Beliebiger LR-Parser

nicht betroffen von der Konstruktionsart:

- **Steuerprogramm**  
des LR-Parsers ist für alle LR-Parser (Shift-Reduce-Verfahren) prinzipiell gleich  
nur die Syntaxanalysetabelle variiert
- **Struktur der Syntaxanalysetabelle** (besteht immer aus zwei Teilen):
  - (1) per Trigger ausgelöste Aktion (**Action**)
  - (2) Zustandsübergang (**Goto**)

sämtliche  
Terminalsymbole

sämtliche  
Nichtterminalsymbole

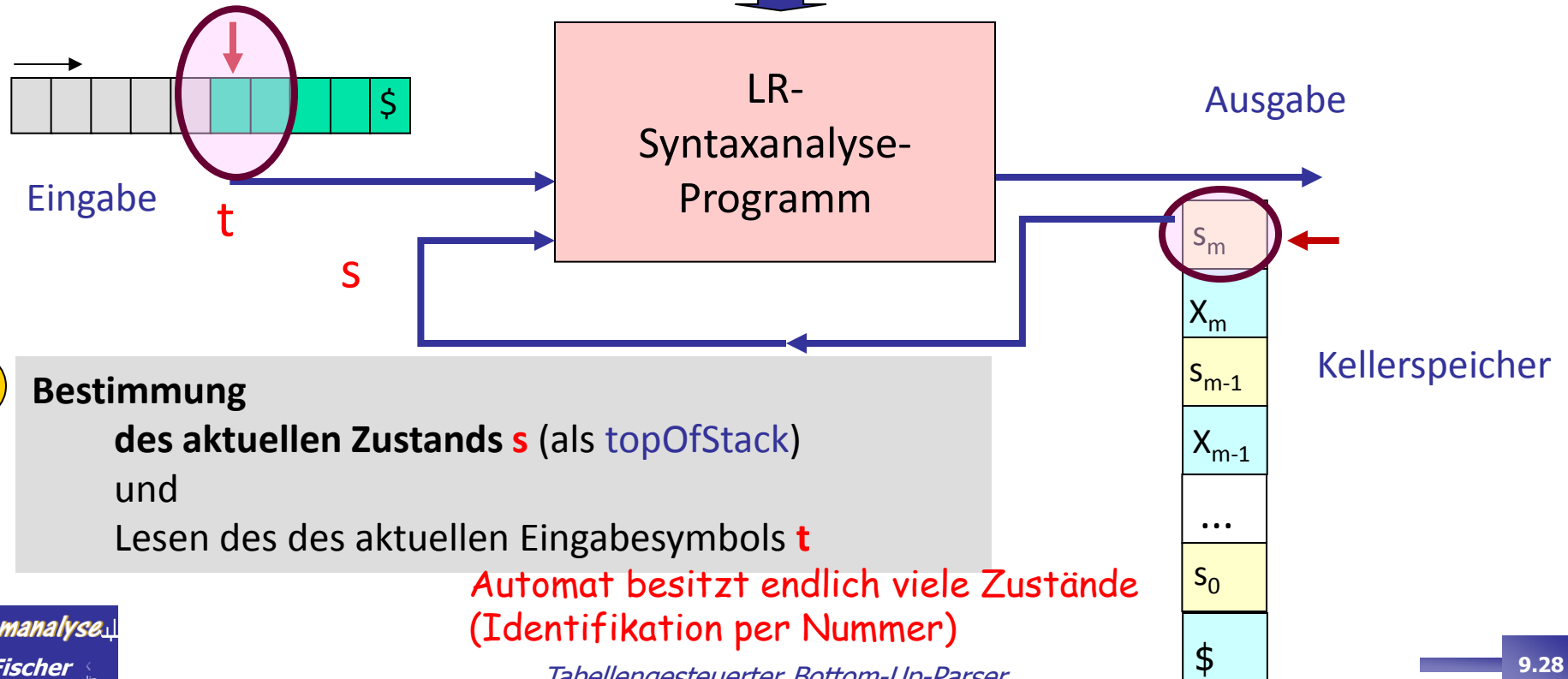
State	Action			Goto	
	id, +	id, (	...	S	...
0					
1					
...					

**hier:**  
für ein Symbol vorausschauend

# Prinzipielle Arbeitsweise eines beliebigen LR-Parsers (Schritt 1)

Syntaxanalysetabelle

State	Action		Goto	
	id	+	S	E
0				
1				



# Prinzipielle Arbeitsweise eines beliebigen LR-Parsers (Schritt 2)

2

Bestimmung  
der nächsten **Automatenaktion** für  
vorliegende Situation:  
(aktueller Zustand, aktuelles Eingabezeichen)  
aus  
**Action [s, t]**

State	Action			Goto	
	id	+	...	A	...
0					
1					
...					

- (1) **Shift s'**: schiebe aktuelles Eingabesymbol **t** und Zustand **s'** auf den Keller
- (2) **Reduce j**: **j** bestimmt das Handle reduziere Handle im Keller zur LS der Grammatikproduktion **j**
- (3) **Accept**: akzeptiere das Eingabewort
- (4) **Error**: behandle Fehler

Fall A:

... und Weiterrücken in der Eingabe

Fall B:

... keine weiteren Aktionen

# Prinzipielle Arbeitsweise eines beliebigen LR-Parsers (Schritt 3)

## 3 Bestimmung des Folgezustandes

State	Action			Goto	
	id	+	...	A	...
0					
1					
...					

(1) **Shift  $s'$** : schiebe aktuelles Eingabesymbol  $t$  und Zustand  $s'$  auf den Keller

Fall A:

$s'$  wird als oberstes Kellerelement zum neuen Folgezustand

(2) **Reduce  $j$** :  $j$  bestimmt das Handle reduziere Handle im Keller zur LS der Grammatikproduktion  $j$

Fall B:

aus freigelegtem Zustand  $s''$  und  $A$  als Reduktionsresultat vom Handle auf dem Stack liefert Eintrag von **Goto** [ $s''$ ,  $A$ ] zum neuen Folgezustand

(3) **Accept**: akzeptiere das Eingabewort

(4) **Error**: behandle Fehler

# Prinzipielle Arbeitsweise eines beliebigen LR-Parsers: Pseudocode

```
push(s0); /* Kellereinhalt: $s0 oder #s0 */
token:= nextToken();
repeat forever begin
  s:= topOfStack(); /* aktueller Zustand */
  switch Action[s, token]
    case "Shift s' " :      push(token); /* Grammatiksymbol */
                           push(s'); /* neuer Zustand */
                           token:= nextToken();
                           break;
    case "Reduce A →β " :
                           pop(); /* 2|β| Symbole !!! */
                           ... /* weitere pop()- Rufe */
                           s'':= topOfStack(); /* frei gelegter Zustand */
                           push(A); /* Grammatiksymbol */
                           push( Goto[s'', A]); /* neuer Zustand */
                           output("A→b");
                           break;
    case "Accept" :      return;
    default:             error();
  endswitch;
end;
```

als Zustandsnummer **i**

als Grammatikregelnummer **j**

# Prinzipielle Arbeitsweise eines beliebigen LR-Parsers: **Operationsanzahl**

Anzahl von Parser-Operationen

- **k Shift-** Operationen  
 $k$  = Länge der Eingabe (Anzahl von Terminalsymbolen)
- **l Reduce-** Operationen  
 $l$  = Tiefe des Syntaxbaums entspr. Rechtsableitung
- **1 Accept-** Operation



## 4.6.1 Allgemeine Betrachtung

- Allgemeines Prinzip von Shift-Reduce-Verfahren
- Klassifikation von LR-Analysemethoden/Grammatiken
- Präzisiertes Automatenmodell eines beliebigen LR-Parsers
- Arbeitsweise an einem Beispiel
- Konstruktionsvarianten von LR-Syntaxanalysetabellen im Überblick

# Beispiel: Arbeitsweise eines beliebigen LR-Parsers (1)

## Beispiel-Grammatik

1	S	→	E
2	E	→	T + E
3			T
4	T	→	F * T
5			F
6	F	→	id

Ann.: Syntaxanalysetabelle sei gegeben

State	Action				Goto		
	id	+	*	\$	E	T	F
0	s4	-	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	s5	-	r3	-	-	-
3	-	r5	s6	r5	-	-	-
4	-	r6	r6	r6	-	-	-
5	s4	-	-	-	7	2	3
6	s4	-	-	-	-	8	3
7	-	-	-	r2	-	-	-
8	-	r4	-	r4	-	-	-

## Codierung:

$s_i$	schieben des Zustand $i$
$r_j$	reduzieren nach Regel $j$
acc	akzeptieren
-	Fehler

# Beispiel: Arbeitsweise eines beliebigen LR-Parsers (33)

aktueller Trigger

aktuellerZustand	<b>1</b>
aktuelle Eingabe	<b>\$</b>

State	Action				Goto		
	id	+	*	\$	E	T	F
0	s4	-	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	s5	-	r3	-	-	-
3	-	r5	s6	r5	-	-	-
4	-	r6	r6	r6	-	-	-
5	s4	-	-	-	7	2	3
6	s4	-	-	-	-	8	3
7	-	-	-	r2	-	-	-
8	-	r4	-	r4	-	-	-

Keller	Eingabe	Aktion
\$ 0	id * id + id \$	s4
\$ 0 id 4	* id + id \$	r6
\$ 0 F 3	* id + id \$	s6
\$ 0 F 3 * 6	id + id \$	s4
\$ 0 F 3 * 6 id 4	+ id \$	r6
\$ 0 F 3 * 6 F 3	+ id \$	r5
\$ 0 F 3 * 6 T 8	+ id \$	r4
\$ 0 T 2	+ id \$	s5
\$ 0 T 2 + 5	id \$	s4
\$ 0 T 2 + 5 id 4	\$	r6
\$ 0 T 2 + 5 F 3	\$	r5
\$ 0 T 2 + 5 T 2	\$	r3
\$ 0 T 2 + 5 E 7	\$	r2
\$ 0 E 1	\$	acc

## Beispiel-Grammatik

1	S	→	E
2	E	→	T + E
3			T
4	T	→	F * T
5			F
6	F	→	id

accept

# Beispiel: Arbeitsweise eines beliebigen LR-Parsers (Fazit)

Keller	Eingabe	Aktion
\$ 0	id * id + id \$	s4
\$ 0 id 4	* id + id \$	r6
\$ 0 F 3	* id + id \$	s6
\$ 0 F 3 * 6	id + id \$	s4
\$ 0 F 3 * 6 id 4	+ id \$	r6
\$ 0 F 3 * 6 F 3	+ id \$	r5
\$ 0 F 3 * 6 T 8	+ id \$	r4
\$ 0 T 2	+ id \$	s5
\$ 0 T 2 + 5	id \$	s4
\$ 0 T 2 + 5 id 4	\$	r6
\$ 0 T 2 + 5 F 3	\$	r5
\$ 0 T 2 + 5 T 2	\$	r3
\$ 0 T 2 + 5 E 7	\$	r2
\$ 0 E 1	\$	acc

Parser-Generatoren verzichten tatsächlich auf die *ausgegrauten Kellereinträge*, (bei Verringerung der Anzahl benötigter Kellerooperationen) sie dienen hier lediglich einer besseren Lesbarkeit

$S \Rightarrow E$   
 $\Rightarrow T + E$   
 $\Rightarrow T + T$   
 $\Rightarrow T + F$   
 $\Rightarrow T + id$   
 $\Rightarrow F * T + id$   
 $\Rightarrow F * F + id$   
 $\Rightarrow F * id + id$   
 $\Rightarrow id * id + id$

Eingabe-Wort: id \* id + id

**Fazit:** Resultat ist eine Rechtsableitung

## **4.6.1 Allgemeine Betrachtung**

- Allgemeines Prinzip von Shift-Reduce-Verfahren
- Klassifikation von LR-Analysemethoden/Grammatiken
- Präzisiertes Automatenmodell eines beliebigen LR-Parsers
- Automatenmodell eines beliebigen LR-Parsers
- Arbeitsweise an einem Beispiel
- Konstruktionsvarianten von LR-Syntaxanalysetabellen im Überblick

# Allgemeiner Tabellenkonstruktionsablauf

unabhängig von konkreter LR-Technik:

- (1) Hinzufügen einer neuen **Startproduktion** ( $S' \rightarrow S$ ) zu  $G$

**denn:**  $S$  als altes Startsymbol könnte auch in einer RS einer Regel auftauchen

wenn jetzt  $S$  nach  $S'$  reduziert wird,  
ist dies einmalig und damit Garantie für vollzogenen Endschritt  
der umgekehrten Rechtsableitung

- (2) Konstruktion des **Zustandsübergangsgraphen** eines DFAs

Zustände entstehen durch Mengengbildung von **LR(k)-Elementen**

**LR(0)-Element** einer Grammatik  $G$  ist eine Produktion von  $G$   
mit einer Bearbeitungs-Markierung  
(dargestellt als Punkt) in der rechten Seite: z.B.  $A \rightarrow X \bullet YZ$

**Einsatz:**  
verfahrensspezifisch

- (3) Ableitung der **Syntaxanalysetabelle** aus dem Zustandsübergangsgraphen des DFA

# LR(k)-Element

~ Hilfskonstrukt zur Darstellung des Analysezustandes eines DPDA

- Konstruktionsalgorithmen benutzen Mengen von LR(k)-Elementen, um die möglichen Zustände während des Parsens zu repräsentieren

## Definition: LR(k)-Element

... ist ein Paar  $[\alpha, \beta]$ , wobei

- $\alpha$  eine Produktion der Grammatik  $G$  (z.B. Regelnummer) und " $\bullet$ " eine Markierung in der RS der Regel ist, die anzeigt, wie viel von der RS einer Produktion schon erkannt worden ist, d.h. sich bereits auf dem Keller befindet;
- $\beta$  die LookAhead-Zeichenkette ist, die  $k$  Symbole (Terminalsymbole inkl. "\$") als Gedächtnis umfasst.

zwei Ausprägungen spielen eine besondere Rolle:  $k=0$  und  $k=1$

- **LR(0)-Elemente** zur Konstruktion der Tabelle für die Verfahren **LR(0)** und **SLR(1)** kommen ohne  $\beta$  aus
- **LR(1)-Elemente** zur Konstruktion der **LR(1)**- und **LALR(1)**-Tabellen

## 4.6.2 *LR(0)-Syntaxanalyse*

- LR(0)-Elemente und Idee zur Zustandsbildung
- Die Operatoren Closure0 und Goto0
- Kanonische LR(0)-Kollektion, charakteristischer Automat und Übergangstabellekonstruktion für einen LR(0)-Parser
- Beispiel: Konstruktion eines LR(0)-Parsers
- LR(0)-Konfliktbeispiel



# Zustandsabstraktion

## konzeptionelle Basis

**Zustände** des jeweils gesuchten DPDA als LR(k)-Parser entstehen durch geeignete Zusammenfassungen von **LR(k)-Elementen** als Paare  $[\alpha, \beta]$

LR(1)-Parser  
LALR(1)-Parser

**LR(1)-Element**  $[\alpha, \beta]$

$[A \rightarrow X \bullet YZ]$

LookAhead-Zeichenkette

Symbol " $\bullet$ " zeigt  
Verarbeitungs-/Erkennung-  
zustand der rechten Seite an:  
(X liegt bereits auf dem Stack)

$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) = LR(k)$

**LR(0)-Element**  $[\alpha]$

**LR(k)-Element**  $[\alpha, \beta]$

LR(0)-Parser  
SLR(1)-Parser

LR(k)-Parser

# LR(0)-Element als spezielles LR(k)-Element

- LR(0)-Element: **ohne** gespeichertes LookAhead
- allg. Regel  $A \rightarrow XYZ$  generiert vier LR(0)-Elemente:
  - $[A \rightarrow \bullet XYZ]$
  - $[A \rightarrow X \bullet YZ]$
  - $[A \rightarrow XY \bullet Z]$
  - $[A \rightarrow XYZ \bullet]$

Spezialfall: Regel  $A \rightarrow \varepsilon$  generiert nur ein LR(0)-Element:

$[A \rightarrow \bullet]$

- Symbol " $\bullet$ " zeigt an:  
wie viel von einem Element bereits in einem Zustand der Erkennung gesehen worden ist:

- $[A \rightarrow \bullet XYZ]$  zeigt an, dass der Parser eine Zeichenkette sucht, die aus der Zeichenkette  $XYZ$  abgeleitet werden kann
- $[A \rightarrow XY \bullet Z]$  zeigt an, dass der Parser bereits eine Zeichenkette gesehen hat, die aus  $XY$  abgeleitet werden konnte und dass er nach einer Zeichenkette sucht, die aus  $Z$  abgeleitet werden kann

# Illustrationsbeispiel: Bildung von Parser-Zuständen

## Grammatik

1	$S' \rightarrow S\$$
2	$S \rightarrow (L)$
3	$S \rightarrow x$
4	$L \rightarrow S$
5	$L \rightarrow L,$

**initialer Zustand** des Parsers als LR(0)-Element

- Stack leer
- Eingabe: kompletter  $S$ -Satz gefolgt von  $\$$

1.Schritt

$$S' \rightarrow \bullet S\$$$

$[S' \rightarrow \bullet S\$]$  zeigt an, dass der Parser eine Zeichenkette sucht, die aus der Zeichenkette  $S\$$  abgeleitet werden kann

2.Schritt

Hinzunahme der rechten Seiten von  $S$ -Produktionen

$$\begin{array}{l} S' \rightarrow \bullet S\$ \\ \hline S \rightarrow \bullet x \\ S \rightarrow \bullet (L) \end{array}$$

LR-Kernelement

LR-Erweiterung

3.Schritt

Zustand wird als Menge von LR-Elementen konstruiert, erhält o.B.d.A. die Nummer 1

$S' \rightarrow \bullet S\$$	1
$S \rightarrow \bullet x$	
$S \rightarrow \bullet (L)$	

# Illustrationsbeispiel: Parser-Zustände und Übergänge

## Shift-Operation im Zustand 1

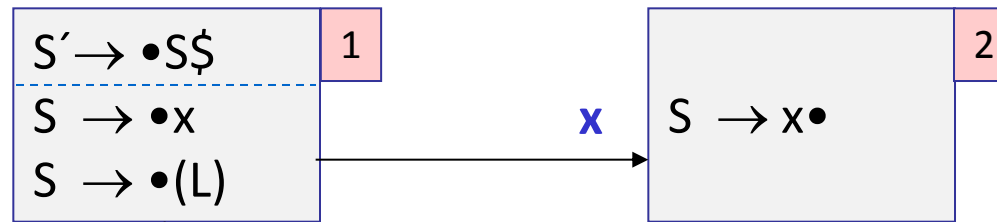
Grammatik		
1	$S'$	$\rightarrow S\$$
2	$S$	$\rightarrow (L)$
3	$S$	$\rightarrow x$
4	$L$	$\rightarrow S$
5	$L$	$\rightarrow L, S$

Bem.:

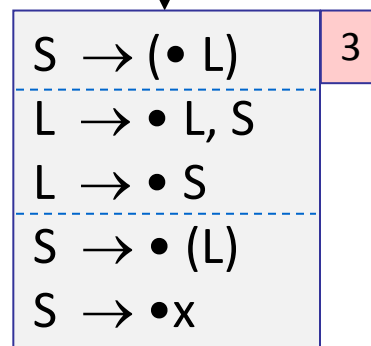
$[A \rightarrow XY \bullet u]$

per **Shift** wird  $u$  auf den Stack gebracht und der Parser nimmt neuen Zustand an

$x$  wird bei diesem Übergang auf den Stack geschoben



$($  wird bei diesem Übergang auf den Stack geschoben



LR-Kernelement

LR-Erweiterung

Anwendung  
Hüllenoperator

## 4.6.2 LR(0)- Syntaxanalyse

- LR(0)-Elemente und Idee zur Zustandsbildung
- Die Operatoren Closure<sub>0</sub> und Goto<sub>0</sub>
- Kanonische LR(0)-Kollektion, charakteristischer Automat und Übergangstabellekonstruktion für einen LR(0)-Parser
- Beispiel: Konstruktion eines LR(0)-Parsers
- LR(0)-Konfliktbeispiel

# Hüllenoperation: $\text{closure}_0$

## Definition: $\text{closure}_0$

Sei  $I$  eine Menge von LR(0)-Elementen für eine Grammatik  $G$ ,  
dann ist die Hülle  $\text{closure}_0(I)$  die Menge von LR(0)-Elementen,  
die aus  $I$  nach folgenden Regeln konstruiert wird:

- (1) jedes Element von  $I$  wird der Hülle  $\text{closure}_0(I)$  hinzugefügt
- (2) wenn  $[A \rightarrow \alpha \bullet B \beta]$  zur Hülle gehört und  $B \rightarrow \gamma$  eine Produktion,  
dann füge das Element  $[B \rightarrow \bullet \gamma]$  ebenfalls der Hülle zu

## Bemerkung:

D.h., falls der Parser einen brauchbaren Präfix  $\alpha$  im Keller gespeichert hat, dann sollte sich die Eingabe zu  $B\beta$  reduzieren (oder zu  $\gamma$  für ein anderes LR(0)-Element  $[B \rightarrow \bullet \gamma]$  in der Hülle von  $[A \rightarrow \alpha \bullet B \beta]$  )

# Berechnung der Hülle

I sei Kollektion von LR(0)-Elementen

```
function closure0 (I)
  J := I;
  repeat
    for jedes Element  $[A \rightarrow \alpha \bullet B \beta] \in J$  and
      jede Produktion  $B \rightarrow \gamma \in G$  mit  $[B \rightarrow \bullet \gamma] \notin J$ 
    do add  $[B \rightarrow \bullet \gamma]$  to J
  until (keine weiteren Elemente können J zugeführt werden)
  return J
endfunction
```

# closure<sub>0</sub> - Beispiele

## Grammatik

1	S	→	E\$
2	E	→	E + T
3			T
4	T	→	id
5			(E)

- $\text{closure}_0([S \rightarrow \bullet E\$]) = \{ [S \rightarrow \bullet E\$], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet \text{id}], [T \rightarrow \bullet (E)] \}$
- $\text{closure}_0([E \rightarrow E + \bullet T]) = \{ [E \rightarrow E + \bullet T], [T \rightarrow \bullet \text{id}], [T \rightarrow \bullet (E)] \}$

**Bemerkung:** "•" gibt an,

- was als nächstes (rechts von ihm) von der Eingabe erwartet wird und
- woher es evtl. über mehrere Stufen kommt (durch die Hüllenberechnung)

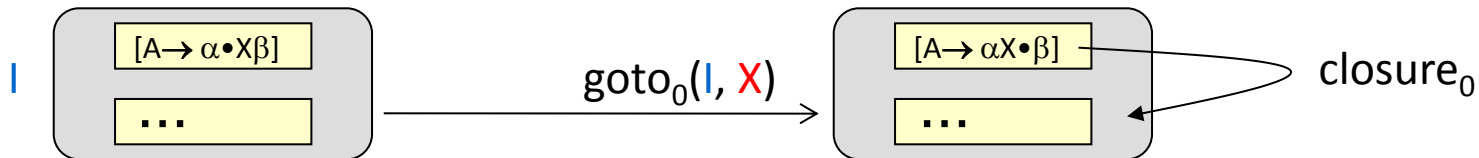


# Sprungoperation: $\text{goto}_0$

## Definition: $\text{goto}_0$

Sei  $I$  eine Menge von LR(0)-Elementen mit  $[A \rightarrow \alpha \bullet X \beta] \in I$   
und  $X$  dabei ein Grammatiksymbol,

dann ist  $\text{goto}_0(I, X)$  die **Hülle** der Menge aller Elemente  $[A \rightarrow \alpha X \bullet \beta]$



**anschaulich:**  $\text{goto}_0(I, X)$  repräsentiert den (Folge-)Zustand,  
nachdem  $X$  im Zustand  $I$  erkannt worden ist

# goto<sub>0</sub> - Beispiel

## Beispiel

sei  $I$  die Menge

$\{ [S \rightarrow E\bullet], [E \rightarrow E\bullet+T], [E \rightarrow \bullet E+T] \}$ ,

dann besteht  $\text{goto}_0(I, +)$  zunächst aus  
 $[E \rightarrow E+\bullet T]$

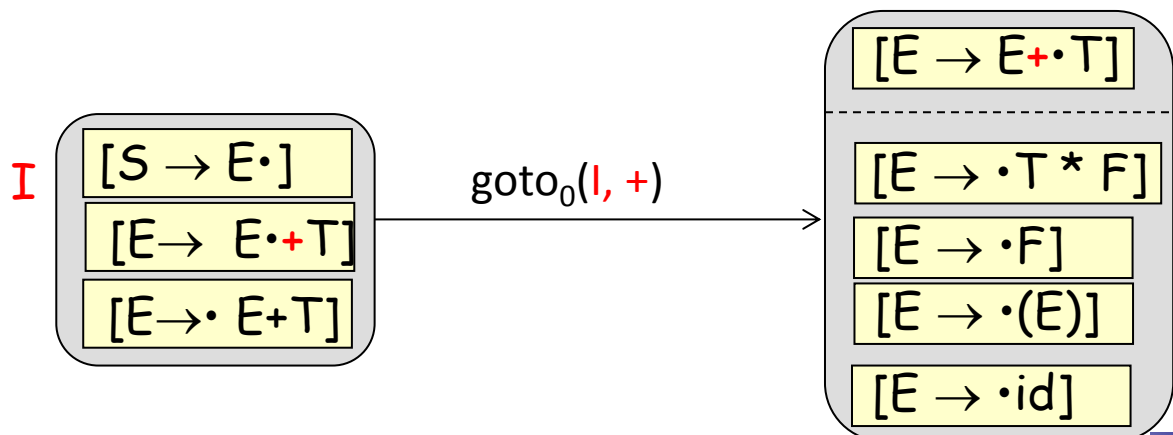
und aus der Hülle davon:  $\text{closure}_0(\{[E \rightarrow E+\bullet T]\})$

$[T \rightarrow \bullet T * F]$

$[T \rightarrow \bullet F]$

$[F \rightarrow \bullet (E)]$

$[F \rightarrow \bullet \text{id}]$



## Beispiel-Grammatik

1	$S \rightarrow E$
2	$E \rightarrow E + T$
3	$\quad \quad   \quad T$
4	$T \rightarrow T * F$
5	$\quad \quad   \quad F$
5	$F \rightarrow \text{id}$
6	$\quad \quad   \quad ( E )$

# Idee zur Konstruktion von LR(0)-Syntaxtabellen

Konstruktion eines **Deterministischen Endlichen Zustandsautomaten (DFA)** für die erweiterte Grammatik  $G'$ ,

der die **eindeutig bestimmten Handle** erkennt

$G'$  wurde aus Ausgangsgrammatik  $G$  unter Hinzunahme der Regel  $S' \rightarrow S$  gebildet

## Vorgehensweise

1. Zusammenfassung von **LR(0)-Elementen** zu Mengen, die Ausgangspunkte der Zustände des LR(0)-Parsers bilden
2. Identifikation von **Zustandsübergängen** (bei Aufbau des Zustandsgraphen)
3. Belegung der Tabellen **ACTION** und **GOTO** unter Nutzung des Zustandsübergangsgraphen und der Grammatik  $G'$

**Bem.:** Automat wird auch **charakteristischer Zustandsautomat (CFSM)** genannt

## 4.6.2 LR(0)- Syntaxanalyse

- LR(0)-Elemente und Idee zur Zustandsbildung
- Die Operatoren Closure<sub>0</sub> und Goto<sub>0</sub>
- Kanonische LR(0)-Kollektion, charakteristischer Automat und Übergangstabellekonstruktion für einen LR(0)-Parser
- Beispiel: Konstruktion eines LR(0)-Parsers
- LR(0)-Konfliktbeispiel

# Konstruktion des CFSM-Zustandsübergangsgraphen (informal)

## Startzustand als Menge von LR(0)-Elementen

- Bestimmung des ersten LR(0)-Elementes  
(aus zusätzlicher Regel:  $S' \rightarrow S$ ):  $[S' \rightarrow \bullet S]$

$$I_0 := \text{closure}_0(\{ [S' \rightarrow \bullet S] \})$$

Kollektion bekommt Nummer  
zur Identifikation des CFSM-Zustandes

## Folgezustände mit Übergängen

- für **jedes** Symbol  $X$  **unmittelbar rechts vom Punkt** in den einzelnen LR(0)-Elementen des Ausgangszustandes wird der Folgezustand konstruiert:

$$I_1 := \text{goto}_0(I_0, X=S)$$

...

$$I_2 := \text{goto}_0(I_0, X=\dots)$$

Mengen bekommen als  
CFSM-Zustände Nummern  
zur Identifikation

- Mengen bekommen nur dann eine neue Nummer, wenn sie sich von bereits existierenden nummerierten Zuständen unterscheiden

# Kanonische Kollektion von LR(0)-Elementen

sei  $G'$  eine erweiterte Grammatik von  $G$  und  $M$  die zu konstruierende (Multi-)Menge  $\{I_0, I_1, \dots, I_n\}$  Kollektionen von LR(0)-Elementen

```
function items (G')
  M := closure0 ( {[S' → •S]} );
  repeat
    for each set of items I ∈ M do
      for each grammar symbol X do
        if goto0(I,X) ≠ ∅ and goto0(I,X) ∉ M
          then add goto0(I,X) to M ;
  until (no more item sets can be added to M)
  return (M) ;
endfunction
```

/\* I = I<sub>k</sub> = {[...], ..., [...]} \*/

**Bem.:** items generiert die Menge sämtlicher CFSM-Zustände für eine gegebene Grammatik  $G'$

## 4.6.2 LR(0)-Syntaxanalyse

- LR(0)-Elemente und Idee zur Zustandsbildung
- Die Operatoren Closure<sub>0</sub> und Goto<sub>0</sub>
- Kanonische LR(0)-Kollektion, charakteristischer Automat und Übergangstabellekonstruktion für einen LR(0)-Parser
- Beispiel: Konstruktion eines LR(0)-Parsers
- LR(0)-Konfliktbeispiel

# Beispiel: Kanonische LR(0)-Kollektion (1)

Grammatik G	
1	E → E + T
2	T
3	T → id
4	( E )

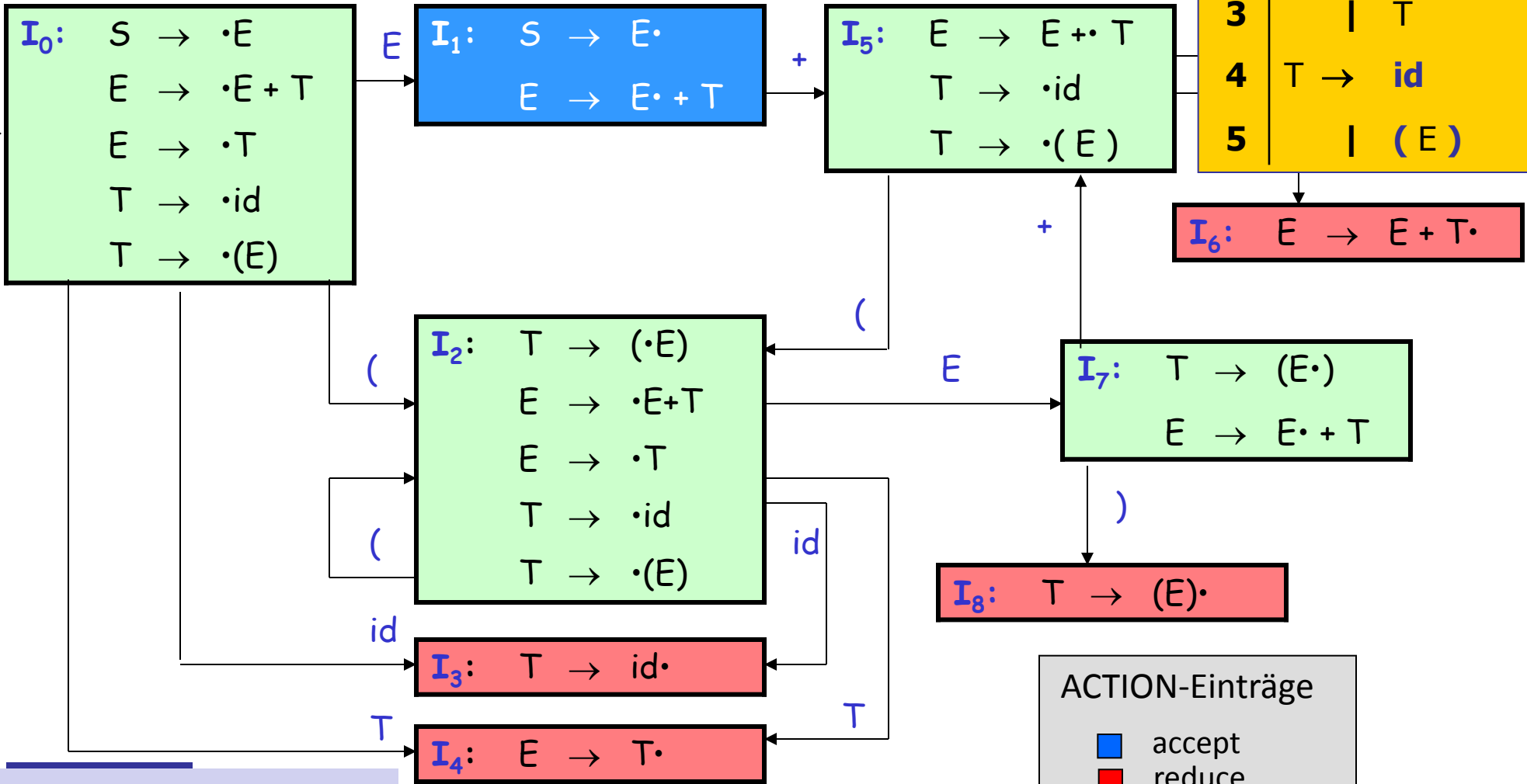


Grammatik G'	
1	S → E
2	E → E + T
3	T
4	T → id
5	( E )



# Beispiel: Kanonische LR(0)-Kollektion (16)

Grammatik	
1	$S \rightarrow E$
2	$E \rightarrow E + T$
3	$\quad   T$
4	$T \rightarrow id$
5	$\quad   (E)$



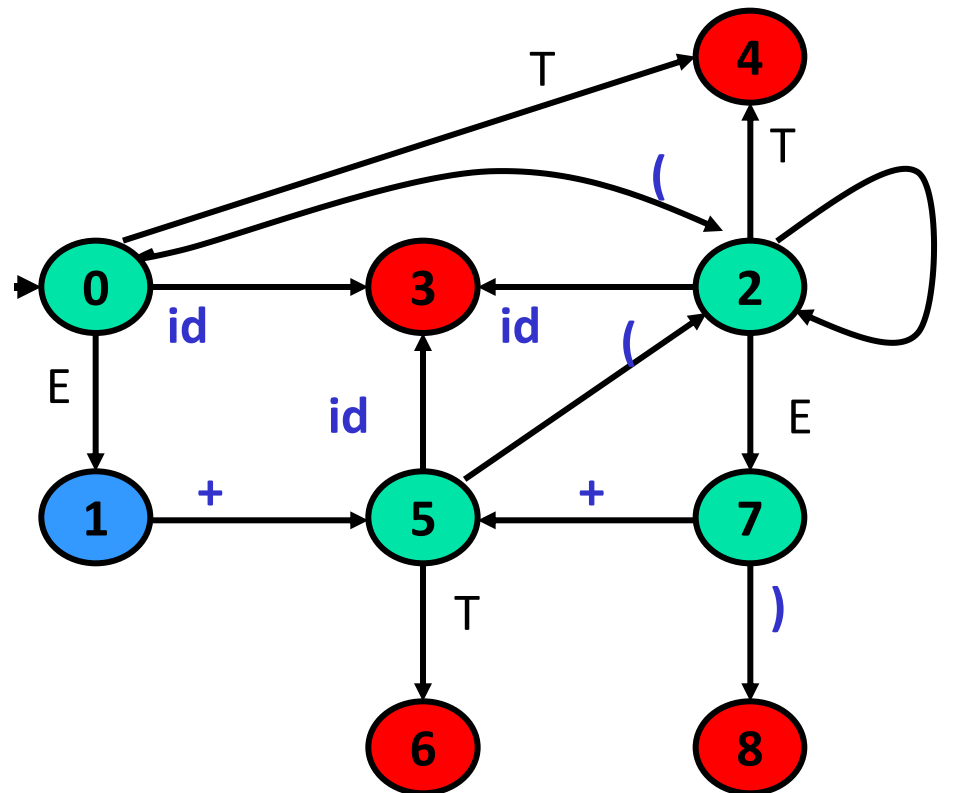
endgültiger Zustandsgraph

ACTION-Einträge

- accept
- reduce
- shift

# Beispiel: Kanonische LR(0)- Kollektion (17)

kompakter Zustandsgraph des charakteristischen Automaten (als DFA)



ACTION-Einträge

- accept
- reduce
- shift