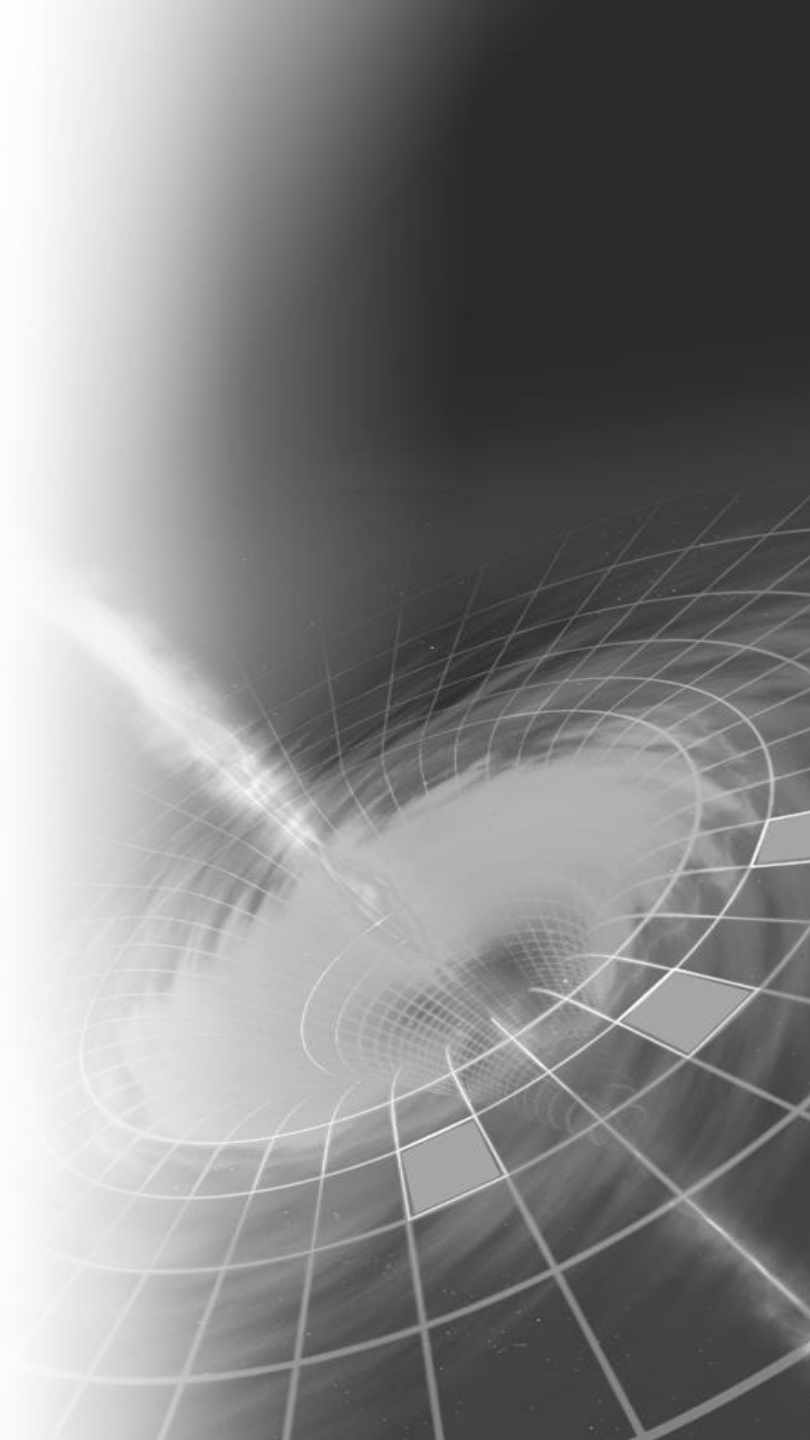
Bachelor-Programm

Compilerbau

im SoSe 2014

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

fischer@informatik.hu-berlin.de



4.6.4 LR(1) - Syntaxanalyse

- Motivation für LR(1)-Elemente
- Von FOLLOW-Mengen zu Look-Ahead-Betrachtungen
- Konstruktion von LR(1)-Elemente-Kollektionen mittels closure1 und goto1
- Konstruktion von LR(1)-Parsertabellen
- Beispiel: Behandlung der SLR(1)-Konfliktgrammatik mit LR(1)

FAZIT im Vergleich: SLR(1) – LR(1) (Wdh.)

Frage: Wann wird reduziert?

- SLR(1)-Analyse:
  FOLLOW-Menge entscheidet, ob das nächste Terminalsymbol der Variablen, zu der reduziert werden soll, überhaupt folgen darf:
  gehört es zur FOLLOW-Menge, wird reduziert.

- LR(1)-Analyse:
  LookAhead-Menge entscheidet, ob das nächste Terminalsymbol der Variablen, zu der reduziert werden soll, nach Anwendung der Reduktionsregel folgen darf:
  gehört es zur LookAhead-Menge, wird reduziert.

Unterschied:
Man betrachtet bei LR(1) nicht mehr Mengen von Terminalsymbolen, die einer Variablen generell folgen dürfen, sondern nur solche, die dieser Variablen nach Anwendung einer bestimmten **Reduktionsregel** folgen dürfen.

Konstruktion von LR(1)-Elementen

- ähnelt der Konstruktion von LR(0)-Elementen

- dabei Modifikation der Funktionen **Hülle** und **Sprung**
  (jetzt mit Berechnung der LookAhead-Mengen)

  closure0 → closure1

  goto0 → goto1

Prinzip der Hüllenbildung von LR(1)-Elementen

Gegeben seien

(1) Element: $[A \rightarrow \alpha \bullet B\beta, \mathbf{a}]$ mit α und β als beliebige Folgen (auch leer)

(2) Regel: $B \rightarrow \gamma$

enthält die FIRST-Menge mehrere Elemente, entstehen auch mehrere LR(1)-Elemente

Hinzunahme von

$[B \rightarrow \bullet\gamma, \text{FIRST}(\beta\mathbf{a})]$

zur Berechnung:
enthält FIRST(β) das leere Wort,
ergibt sich FIRST(β**a**) aus FIRST(β) und **a**
sonst FIRST(β)

closure1 (Erweiterung von closure0)

gegeben sei ein Element [A→ α•Bβ, **a**],
dann enthält die **Hülle**

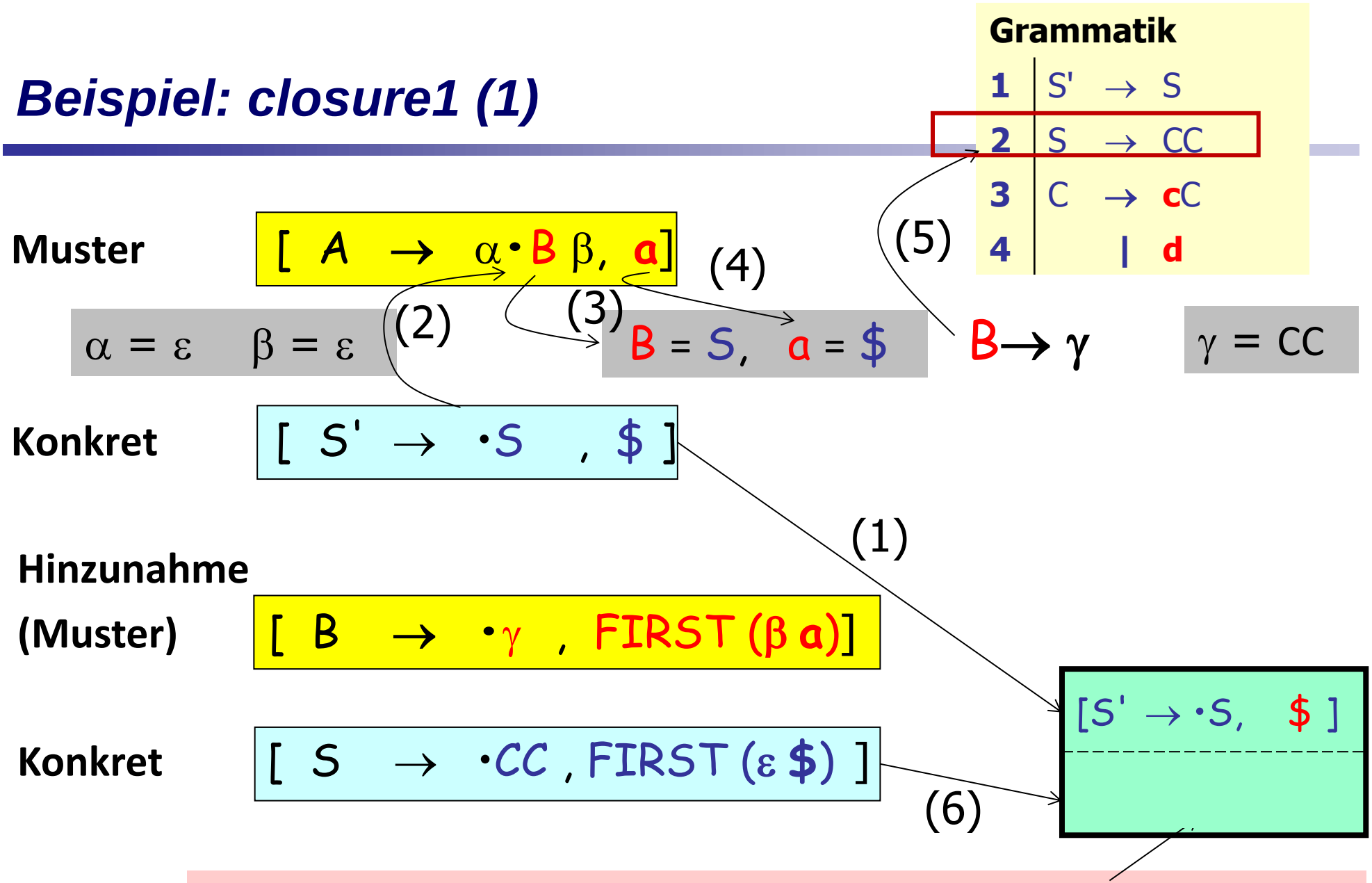
1. das Element selbst und
2. alle weiteren Elemente, die aus einer Teilzeichenkette gebildet werden können, die α folgt

D.h., falls der Parser einen brauchbaren Präfix α im Keller gespeichert hat, dann sollte die Eingabe **a** zu Bβ reduzieren
oder zu γ für ein anderes Element [B→ •γ, **b**] in der Hülle von [A→ α•Bβ, **a**]
mit **b** ∈FIRST(β**a**)

```
function closure1 (I)
repeat
    if [A→ α•Bβ, a] ∈ I then add [B→ •γ,b] to I where b∈FIRST(βa)
until (no more items can be added to I)
return I
```

α und β können auch ε sein

Beispiel: closure1 (1)



nächster zu untersuchender Kandidat ist Variable C (Regeln 3, 4)

Beispiel: closure1 (2)

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	**c**C
4		|	**d**

Muster [A → α • B β, **a**]

α = ε, β = C

B = C, **a** = $

B → γ

γ = **c**C

Konkret: [S → • CC, $]

(7)

Hinzunahme

(Muster) [B → • γ , FIRST (β **a**)]

Konkret [C → • **c**C , FIRST (C)]

FIRST(C)= {**c**, **d**}

(8)

[S' → •S, $]

[S → •CC, $]

Beispiel: closure1 (3)

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	**d**

Hinzunahme

Konkret [C → · **d** , FIRST(C)]

FIRST(C)= {**c**, **d**}

(9)

[S' → ·S,	$]
[S → ·CC,	$]
[C → ·**c**C,	**c**]
[C → ·**c**C,	**d**]
[C → ·**d**,	**c**]
[C → ·**d**,	**d**]

keines der neu hinzugenommenen Elemente hat ein Nichtterminalsymbol rechts vom Punkt:

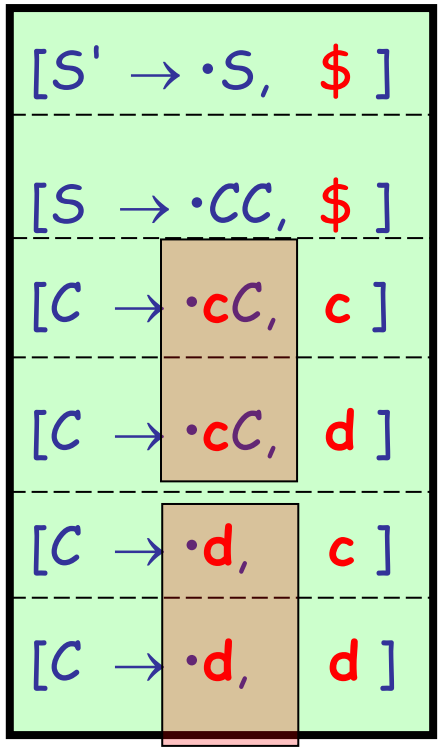
damit ist **closure1** komplett

Notation

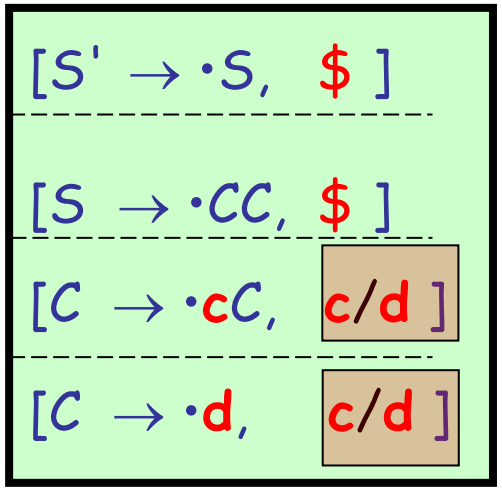
Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	**d**

Closure1 ([S → •S, $]) =



=



kompakte Schreibweise
für 6 Elemente

Goto1: (Erweiterung von goto0)

Sei I eine Menge an LR(1)-Elementen

$[A \rightarrow \alpha \bullet X \beta, \mathbf{a}] \in I$

und X ein Grammatiksymbol,

dann ist **goto1(I, X)** die Hülle der Menge aller Elemente

$[A \rightarrow \alpha X \bullet \beta, \mathbf{a}]$

goto1(I, X) repräsentiert den (Folge-)Zustand,
nachdem X im Zustand I erkannt worden ist

```
function goto1 (I, X)
        sei L die Menge aller Elemente [A→ αX•β, a] mit [A→ α•Xβ, a] ∈ I;
        return (closure1 (L))
```

Beispiel: goto1 (1)

I_0:	S'	→ ·S,	$
	S	→ ·CC,	$
	C	→ ·cC,	c/d
	C	→ ·d,	c/d

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

Bildung von **goto1 (I_0,** X**)** für alle möglichen X

X= S : Bildung von **goto1(I_0,** S**)**: [S'→ S•, $])

I_1:	S'	→S·,	$

jetzt **closure1** – Bildung → **aber** Punkt steht bereits am Ende→ **fertig**

Beispiel: goto1 (2)

I_0:	S'	→ ·S,	$
	S	→ ·CC,	$
	C	→ ·cC,	c/d
	C	→ ·d,	c/d

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

X= C Bildung von **goto1 (I_0, C)**:

FIRST(ε$) = { $ }

I_2:	S	→ C·C ,	$

I_2:	S	→ C·C ,	$
	C	→ ·cC,	
	C	→ ·d,	

I_2:	S	→ C·C ,	$
	C	→ ·cC,	$
	C	→ ·d,	$

Beispiel: goto1 (3)

I_0:	S'	→ ·S,	$
	S	→ ·CC,	$
	C	→ ·cC,	c/d
	C	→ ·d,	c/d

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

X= c Bildung von **goto1(I_0, c)**

FIRST(εc) = { c }

FIRST(εd) = { d }

Bildung von **closure1** ({ [C→ c•C, c/d] })

I_3:	C	→ c·C,	c/d
	C	→ ·cC,	c/d
	C	→ ·d,	c/d

FIRST(εc) = { c }

FIRST(εd) = { d }

Beispiel: goto1 (4)

I_0:	S'	→ ·S,	$
	S	→ ·CC,	$
	C	→ ·cC,	c/d
	C	→ ·d,	c/d

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

X= d Bildung von **goto1(I_0, d)**

Bildung von **closure1** ({ [C→ d•, c/d] })

I_4:	C	→ d·,	c/d

4.6.4 LR(1) - Syntaxanalyse

- Motivation für LR(1)-Elemente
- Von FOLLOW-Mengen zu Look-Ahead-Betrachtungen
- Konstruktion von LR(1)-Elemente-Kollektionen mittels closure1 und goto1
- Konstruktion von LR(1)-Parsertabellen
- Beispiel: Behandlung der SLR(1)-Konfliktgrammatik mit LR(1)

Erzeugung der LR(1)-Elemente-Kollektion

- **Beginn** der Erzeugung mit dem **Element** i= [S'→ •S, **$**] mit
  - S' als (neuem) Startsymbol der erweiterten Grammatik G'
  - S ist das Startsymbol der Grammatik G
  - **$** repräsentiert das Ende der Eingabe

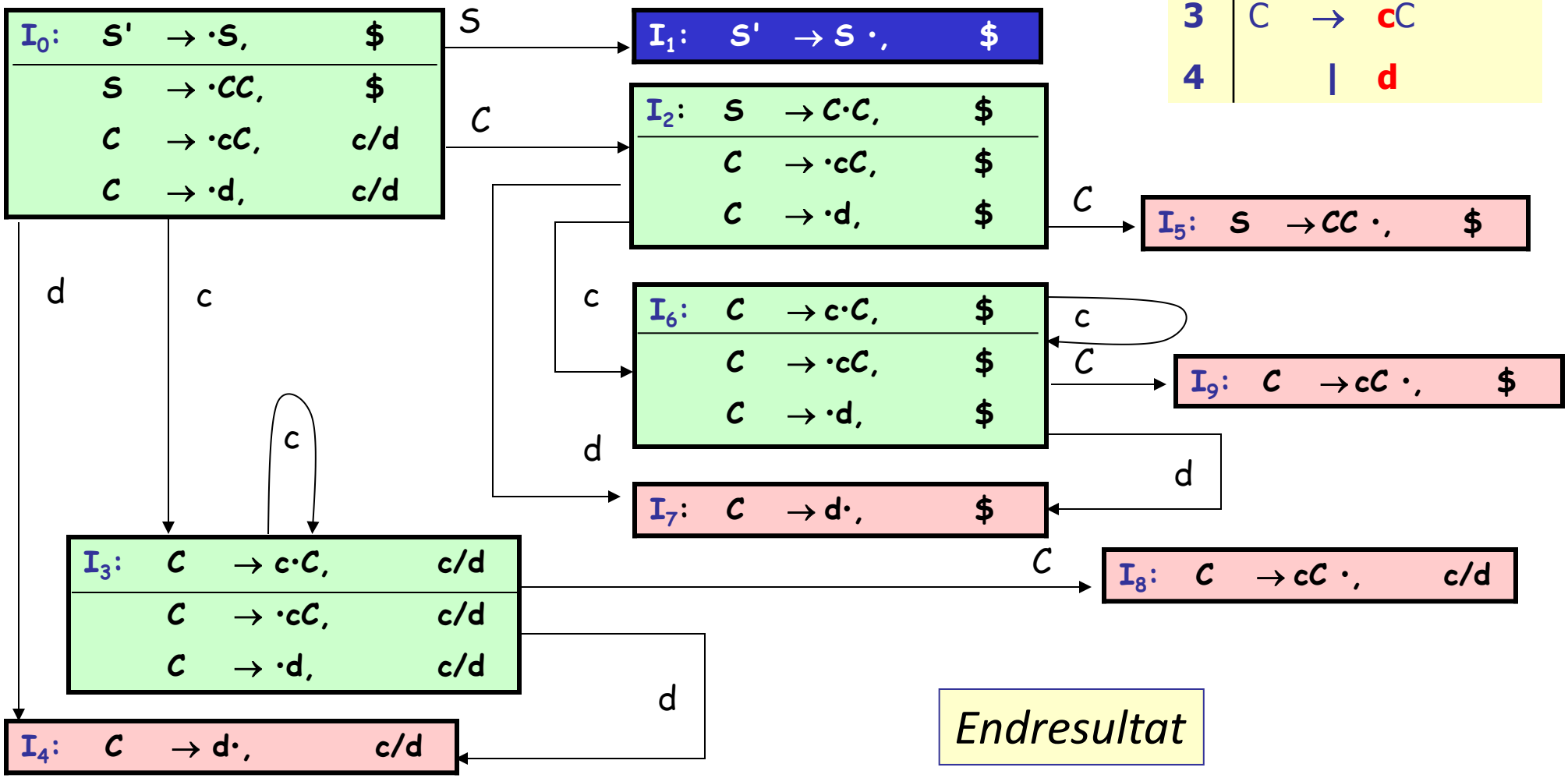
- Erzeugen der Sammlung aller **LR(1)-Element-Mengen**

```
function items (G')
    I0 := closure1({[S'→ •S, $]}); //Menge
    I  := { I0 };  //Multimenge
    repeat
            for each set of item i ∈ I  do
                for each grammar symbol X do
                    if goto1(i, X) ≠ ∅ and goto1(i, X) ∉ I  then add goto1(i, X) to I ;
    until (no more item sets can be added to I)
    return (I)
```

Beispiel: LR(1)-Elemente-Sammlung

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d



Endresultat

4.6.4 LR(1) - Syntaxanalyse

- Motivation für LR(1)-Elemente
- Von FOLLOW-Mengen zu Look-Ahead-Betrachtungen
- Konstruktion von LR(1)-Elemente-Kollektionen mittels closure1 und goto1
- Konstruktion von LR(1)-Parsertabellen
- Beispiel: Behandlung der SLR(1)-Konfliktgrammatik mit LR(1)

Konstruktion der LR(1)-Syntaxtabelle

(1) Konstruktion der Sammlung $\{I_0, I_1, \ldots, I_n\}$ von **LR(1)**-Elemente-Mengen
für G' (Erweiterung von G)

LookAhead-Info beim Schieben ohne Bedeutung

(2) Aufbau der ACTION-Tabelle:

(a) $[A \rightarrow \alpha \bullet a\beta, \mathbf{b}] \in I_i$ und **goto1**$(I_i, a) = I_j$, dann **ACTION**[i, **a**] := "shift j"

(b) $[A \rightarrow \alpha \bullet \quad , \mathbf{a}] \in I_i$ und $A \neq S'$, dann **ACTION**[i, **a**] := "reduce $A \rightarrow \alpha$"

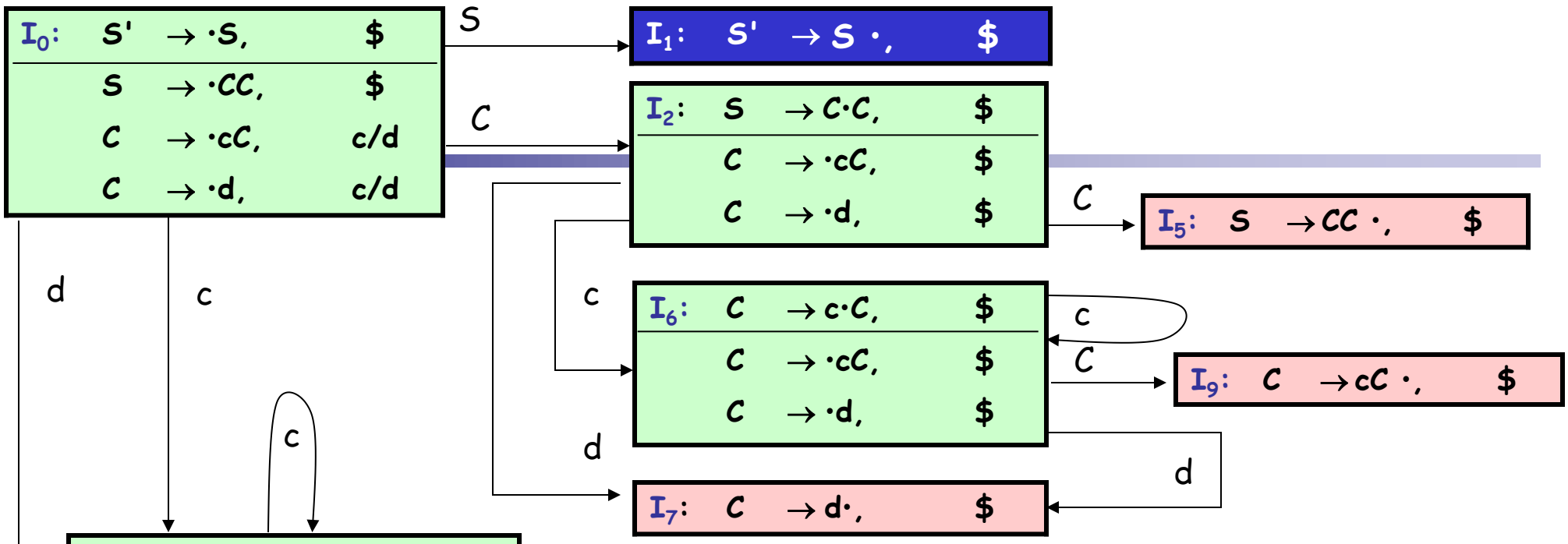
(c) $[S' \rightarrow S \bullet \quad , \mathbf{\$}] \in I_i$, dann **ACTION**[i, **$**] := "accept"

(3) **goto1**$(I_i, A) = I_k$ dann **GOTO**[i, A] := k

(4) setze undefinierte Einträge in **ACTION** und **GOTO** auf "error"

(5) Anfangszustand des Parsers s_0 ist **closure1**$([S' \rightarrow \bullet S, \$])$

Bem.: "LookAhead" wurde damit für Regelbestimmung bei Reduce in den CFSM eingebaut.



Zust.	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4	-	1	2
1	-	-	acc	-	-
2	s6	s7	-	-	5
3	s3	s4	-	-	8
4	r4	r4	-	-	-
5	-	-	r2	-	-
6	s6	s7	-	-	9
7	-	-	r4	-	-
8	r3	r3	-	-	-
9	-	-	r3	-	-

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

Beobachtung: Zustandskomplexität eines LR-Parsers

- Jede SLR(1)-Grammatik ist eine LR(1)-Grammatik

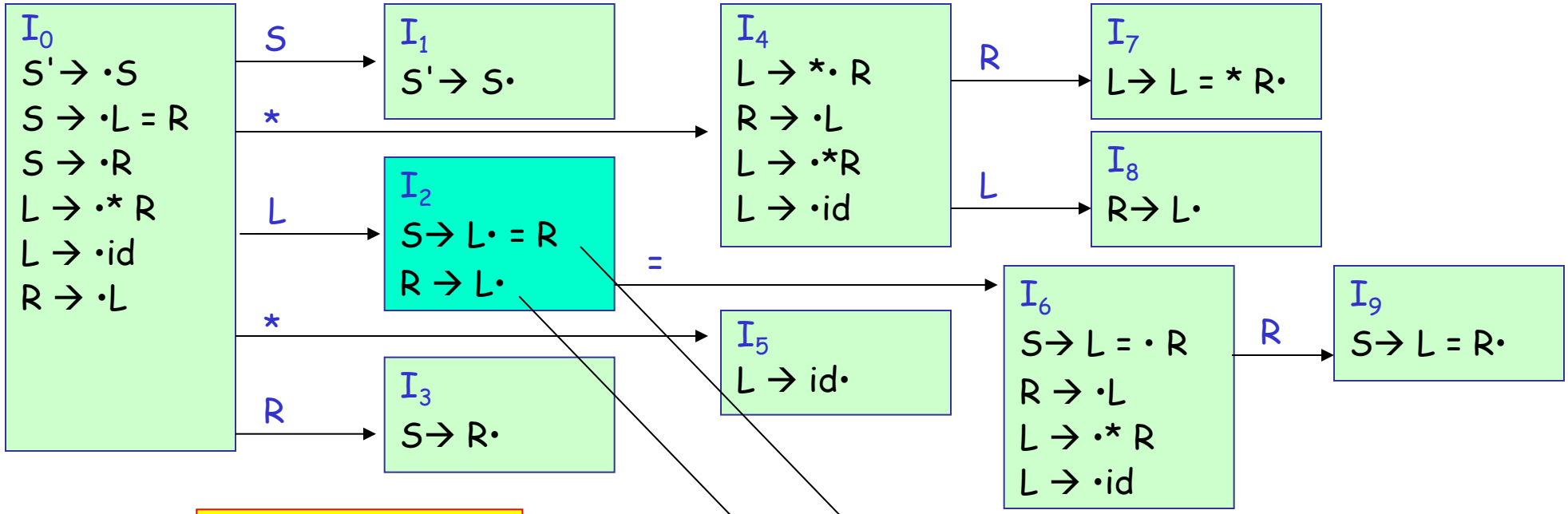
- **Aber**
  Für eine SLR(1)-Grammatik kann der entsprechende kanonische LR(1)-Parser mehr Zustände haben als der SLR(1)-Parser

  Betrachen nochmal die SLR(1)-Konflikt-Grammatik
  - SLR(1)-Parser: 10 Zustände
  - LR(1)-Parser: ???

4.6.4 LR(1) - Syntaxanalyse

- Motivation für LR(1)-Elemente
- Von FOLLOW-Mengen zu Look-Ahead-Betrachtungen
- Konstruktion von LR(1)-Elemente-Kollektionen mittels closure1 und goto1
- Konstruktion von LR(1)-Parsertabellen
- Beispiel: Behandlung der SLR(1)-Konfliktgrammatik mit LR(1)

I_2: Konfliktzustand bei SLR(1) (zur Erinnerung)



Grammatik

1	S	→	L = R
2		|	R
3	L	→	*R
4		|	id
5	R	→	L

ACTION[2, =] := s6

ACTION[2, =] := r5
weil = ∈ FOLLOW(R)

Kein Konflikt bei LR(1)-Analyse

SLR(1)-ACTION[2, =]:= { s6, r5 }

Grammatik

1	S	→	L = R
2		|	R
3	L	→	*R
4		|	id
5	R	→	L

für I_2 gibt es **keinen** Shift-Reduce-Konflikt mehr:

- reduce auf $ und
- shift auf =

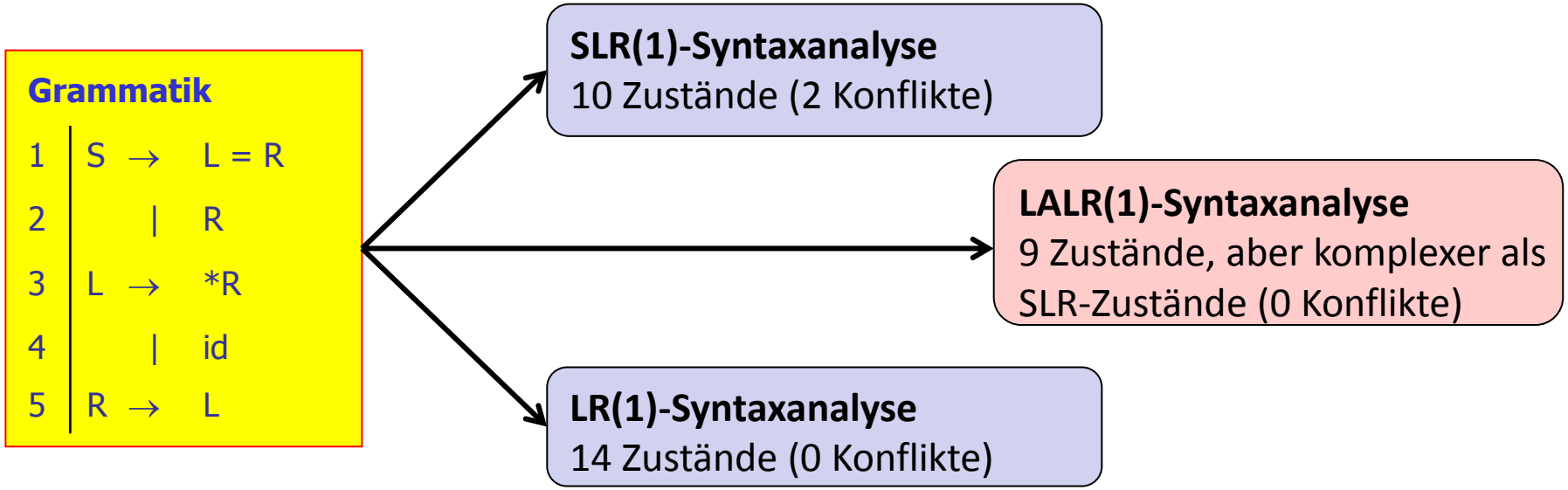
I_0:	S'	→	·S,	$
	S	→	·L=R,	$
	S	→	·R,	$
	R	→	·L,	$
	L	→	·*R,	=/ $
	L	→	·id,	=/ $
I_1:	S'	→	S·,	$
I_2:	S	→	L·=R,	$
	R	→	L·,	$
I_3:	S	→	R·,	$
I_4:	L	→	*·R,	=/ $
	R	→	·L,	=/ $
	L	→	·*R,	=/ $
	L	→	·id,	=/ $

I_5:	L	→	id·,	=/ $
I_6:	S	→	L=·R,	$
	R	→	·L,	$
	L	→	·*R,	$
	L	→	·id,	$
I_7:	L	→	*R·,	=/ $
I_8:	R	→	L·,	=/ $
I_9:	S	→	L=R·,	$
I_{10}:	R	→	L·,	$
I_{11}:	L	→	*·R,	$
	R	→	·L,	$
	L	→	·*R,	$
	L	→	·id,	$
I_{12}:	L	→	id·,	$
I_{13}:	L	→	*R·,	$

4.6.5 LALR(1)- Syntaxanalyse

- Motivation und Konzept zur Zustandsreduktion eines LR(1)-Parsers
- Beispiel: Anwendung der Zustandsreduktion
- Präzisierung der LALR(1)-Analysemethode
- Vergleich von LALR(1) und LR(1)

Motivation für LALR(1)



LALR(1)- und **SLR(1)**-Parser haben die **gleiche** Zustandsanzahl

LALR-Technik wird in der Praxis bevorzugt

Pascal (Zustände): LALR ~ 100 ← → LR(1) ~ 4.000

LR(1)-Analyse ohne Shift-Reduce-Konflikt

Grundlage für Zustandsreduktion: Zustände mit gleichen Kernen

Grammatik

1	S	→	L = R
2		|	R
3	L	→	*R
4		|	id
5	R	→	L

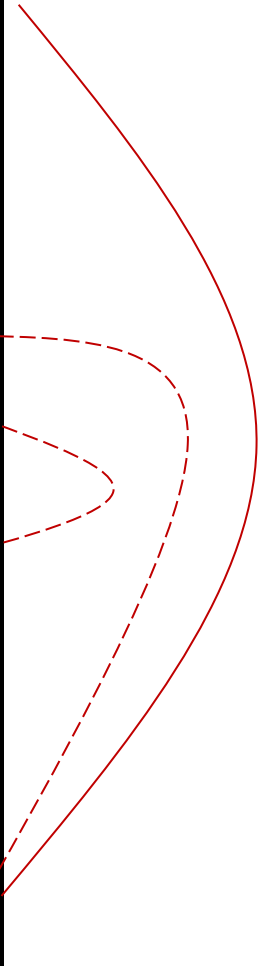
I_5 - I_{12}

I_7 - I_{13}

I_8 - I_{10}

I_0:	S'	→	·S,	$
	S	→	·L=R,	$
	S	→	·R,	$
	R	→	·L,	$
	L	→	·*R,	=/ $
	L	→	·id,	=/ $
I_1:	S'	→	S·,	$
I_2:	S	→	L·=R,	$
	R	→	L·,	$
I_3:	S	→	R·,	$
I_4:	L	→	*·R,	=/ $
	R	→	·L,	=/ $
	L	→	·*R,	=/ $
	L	→	·id,	=/ $

I_5:	L	→	id·,	=/ $
I_6:	S	→	L=·R,	$
	R	→	·L,	$
	L	→	·*R,	$
	L	→	·id,	$
I_7:	L	→	*R·,	=/ $
I_8:	R	→	L·,	=/ $
I_9:	S	→	L=R·,	$
I_{10}:	R	→	L·,	$
I_{11}:	L	→	*·R,	$
	R	→	·L,	$
	L	→	·*R,	$
	L	→	·id,	$
I_{12}:	L	→	id·,	$
I_{13}:	L	→	*R·,	$



Charakterisierieng von LALR(1)

- LALR(1) setzt eine heuristische Optimierungsstrategie ein,
  bei der Zustände, die

  sich **nur** durch die **LookAhead-Symbole** unterscheiden,

  zusammengelegt werden.

- Eine Grammatik heißt LALR(1),
  wenn nach diesem Prozess keine Konflikte entstehen.

- Bei SLR(1) wird auf LookAhead-Symbole in den Items verzichtet,
  stattdessen verwendet man FOLLOW-Mengen, um Konflikte aufzulösen

- LALR(1) und SLR(1) sind bereits mächtige Analyseverfahren, etwas schwächer als LR(1)-
  benötigen aber nur so viele Zustände wie das LR(0)-Verfahren.
  Defizit von LALR(1) gegenüber LR(1) fällt praktisch nicht ins Gewicht

4.6.5 LALR(1)- Syntaxanalyse

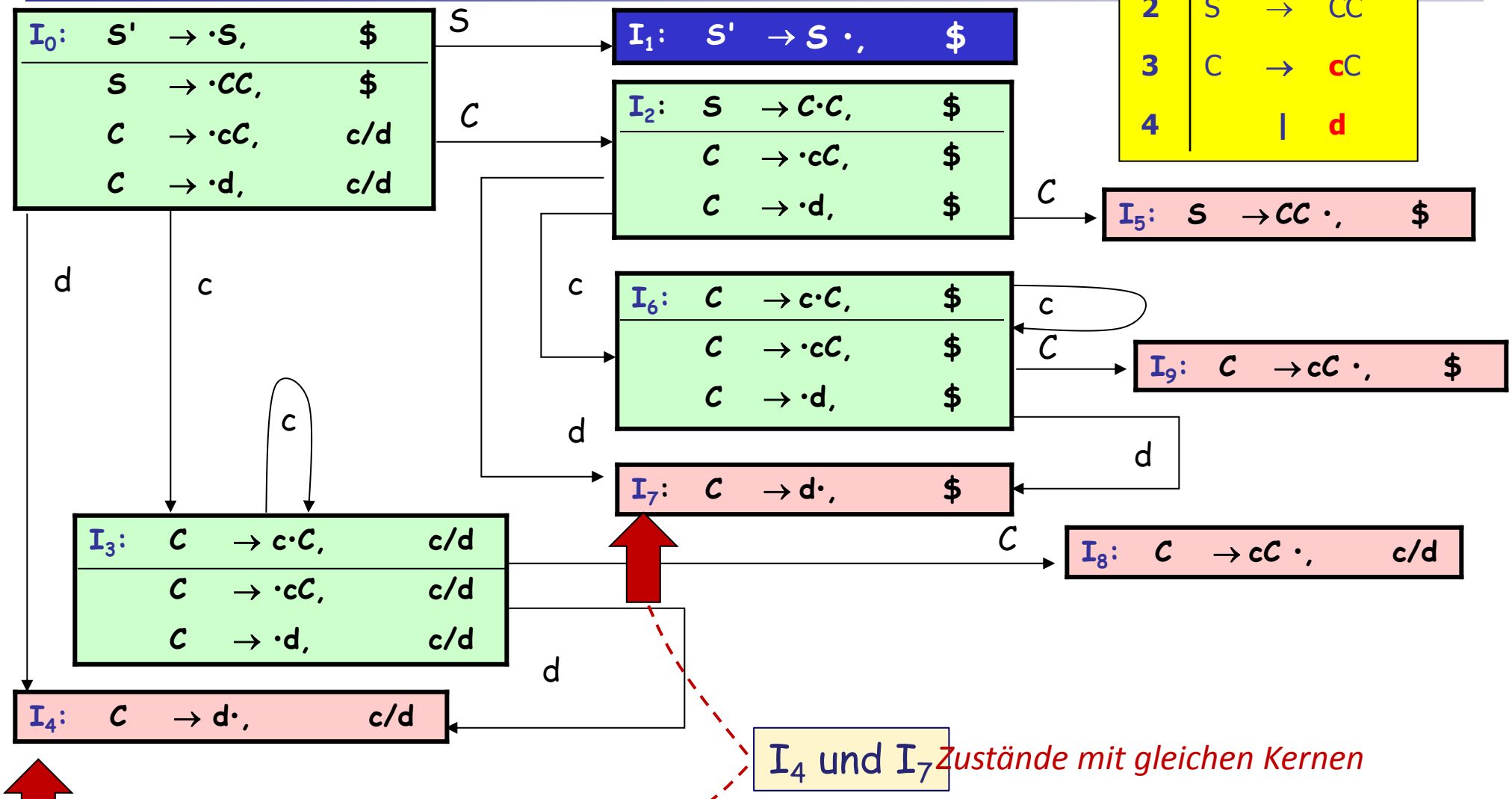
- Konzept zur Zustandsreduktion eines LR(1)-Parsers
- einfaches Beispiel: Anwendung der Zustandsreduktion
- Konstruktion der LALR(1)-Parser-Tabelle
- Vergleich von LALR(1) und LR(1)

für LR(1) bereits gezeigt

Konstruktion von CFSM

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d



I_4 und I_7 *Zustände mit gleichen Kernen*

Idee einer Zustandsreduktion

betrachten dazu die LR(1)-Elemente

I_4:	C	→ d·,	c/d
I_7:	C	→ d·,	$

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

versuchen dabei

die spezifischen Rollen ähnlicher Zustände beim Parser zu erkennen

reguläre Menge

c*dc*d

Zustand: I_4

1. Parser schiebt erste c-Gruppe und das folgende d auf den Stack und erreicht nach Lesen von d den Zustand I_4
2. Reduktion C→d
   (Vor. nächstes Eingabezeichen ist c oder d, sonst Fehler;
   folgt also $ dem ersten d, wird Fehler erkannt)

Idee einer Zustandsreduktion (Forts.)

I_4:	C	→ d·,	c/d

I_7:	C	→ d·,	$

Zustand: I_7

1. I_7 wird nach Lesen des zweiten d erreicht
2. Reduktion C→d
   (Vor. nächstes Eingabezeichen ist $, sonst Fehler)

Grammatik

1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

c*dc*d

Idee einer Zustandsreduktion (Forts.)

ersetzen zwei Zustände (I_4 und I_7) durch einen Zustand I_{47}

I_4: C →d·, c/d	I_7: C →d·, $

I_{47}: C → d·, c/d/$

Zustand: I_{47}

in jedem Fall: Reduktion C→d
(Vor. nächstes Eingabezeichen ist $, c oder d sonst Fehler)

Problem:
bei Fehlereingaben (wie ccd oder cdcdc) wird d zu C reduziert
Fehler wird nicht gleich, aber dennoch erkannt

Grammatik

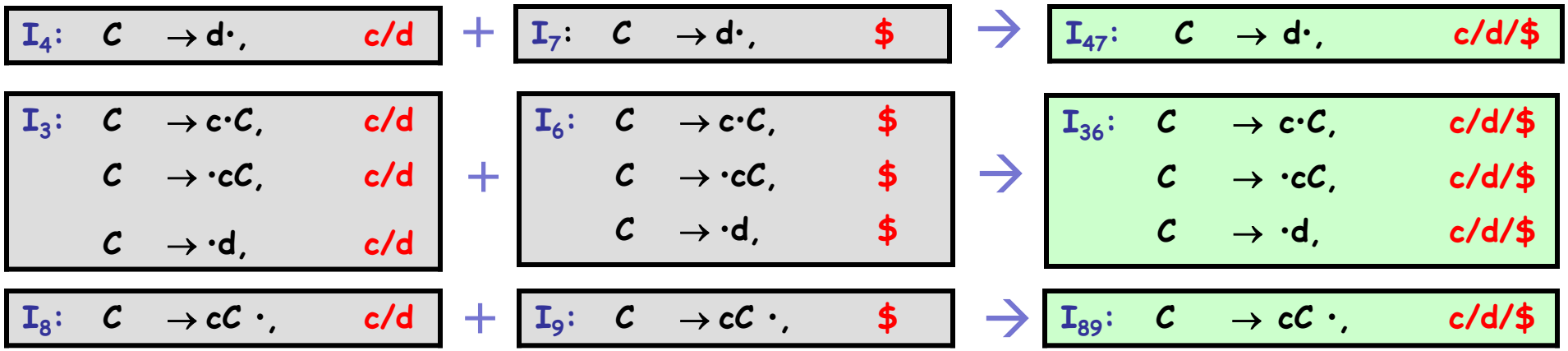
1	S'	→	S
2	S	→	CC
3	C	→	cC
4		|	d

reguläre Menge

c*dc*d

Verallgemeinerung der Idee *zur Zustandsreduktion*

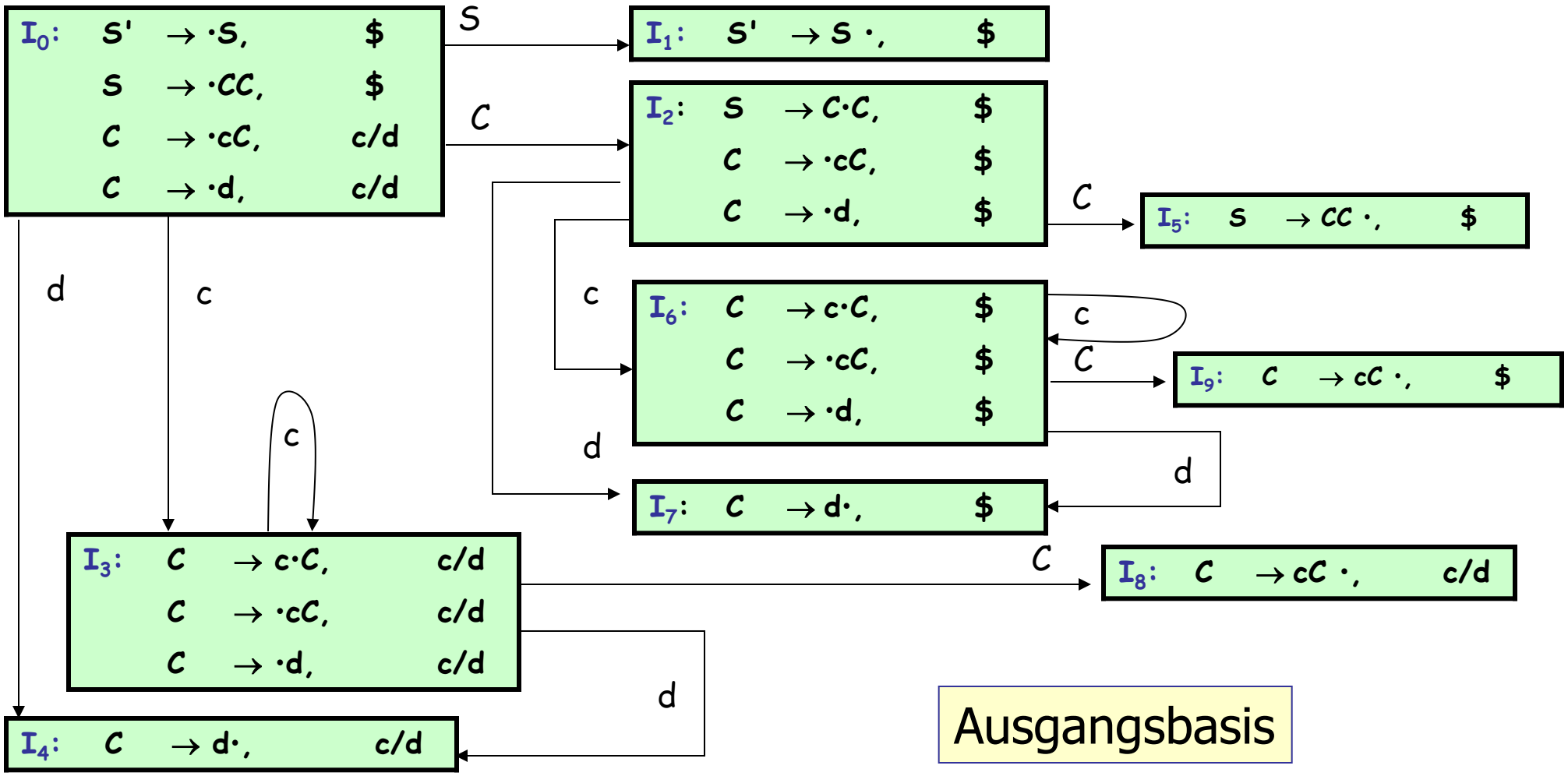
Suche nach LR(1)-Elementen mit gleichem Kern
(erste Komponenten sind gleich)



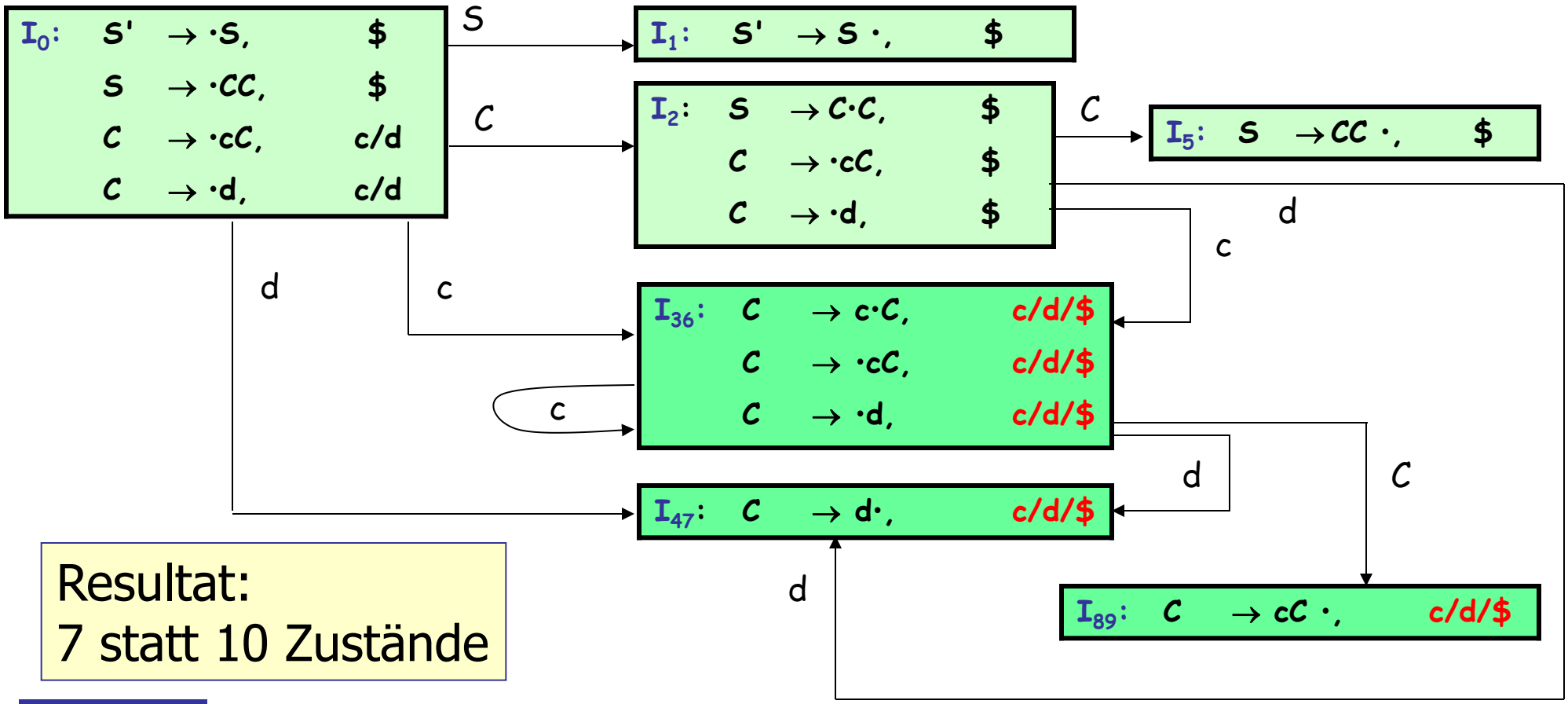
Bem.: im Beispiel haben (nur) Elementpaare einen gemeinsamen Kern
(i.allg. sind mehr Elemente möglich)

Feststellung: Der Kern von goto1(I, X) hängt nur von I ab, nicht von X

Realisierung der Idee *zur Zustandsreduktion*



Realisierung der Idee *zur Zustandsreduktion*



Resultat:
7 statt 10 Zustände

4.6.5 LALR(1)- Syntaxanalyse

- Konzept zur Zustandsreduktion eines LR(1)-Parsers
- Beispiel: Anwendung der Zustandsreduktion
- Konstruktion der LALR(1)-Parser-Tabelle
- Vergleich von LALR(1) und LR(1)

LALR(1)- Syntaxtabelle

- Um die LALR(1)-Syntaxtabelle zu konstruieren, braucht nur ein **zusätzlicher** Schritt in den LR(1)-Algorithmus eingefügt zu werden

  - für jeden Kern, der unter den Mengen an **LR(1)-Elementen** existiert, finde alle Mengen,

    die diesen Kern besitzen und ersetze diese Mengen durch ihre Vereinigung

  - die goto1-Funktion muss verändert werden, um die neue Zustandsmenge zu reflektieren

LALR(1)- Syntaxtabelle (Forts.)

geänderter Algorithmus

(1) Konstruiere die **LR(1)**-Elementmengen von **G'**

(2) Für **jeden Kern**, der unter den Mengen der LR(1)-Elemente existiert, finde alle Menge, die denselben Kern haben, und ersetze diese Mengen durch ihre **Vereinigung** (Verändere die **goto1**-Funktion inkrementell)

(3) **Zustand i** des LALR(1)-Automaten wird von I_i wie folgt konstruiert:

(a) $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ und $goto1(I_i, a) = I_k$ dann **ACTION**[i, a]:= "shift k"

(b) $[A \rightarrow \alpha \bullet, a] \in I_i$ und $A \neq S'$, dann **ACTION**[i, a]:= "reduce $A \rightarrow \alpha$"

(c) $[S' \rightarrow S \bullet, \$] \in I_i$ dann **ACTION**[i, $]:= "accept"

LALR(1)- Syntaxtabelle (Forts.)

Ersetzung der Übergänge der einzelnen Teilzustände durch Übergänge der Vereinigungszustände

geänderter Algorithmus

...

(4) Sei J die Vereinigung einer oder mehrerer Mengen von LR(1)-Elementen (aus Schritt 2):

$J = I_1 \cup I_2 \cup \ldots \cup I_k$,

dann sind die Kerne von **goto1**(I_1, X), **goto1**(I_2, X), ..., **goto1**(I_k, X) gleich!!!

sei nun K die Vereinigung aller Elementmengen, die den gleichen Kern wie **goto1**(I_1, X) haben, dann ist **GOTO**[J, X] := neueNummer(K)

(5) Setze undefinierte Einträge in **ACTION** und **GOTO** auf **error**

(6) **Anfangszustand** des Parsers s_0 ist closure1([S'→ •S, $])

Problem: Dieser Algorithmus erfordert leider jedoch genauso viel **Speicherplatz** wie LR(1) plus eines gewissen Mehraufwandes (aber erklärt das LALR-Prinzip zumindest anschaulich)

Beispiel wie zuvor …Syntaxtabelle

Grammatik

1	S'	→	S
3	S	→	CC
4	C	→	cC
5		|	d

vereinige folgende Zustände

I_0:	S'	→	·S,	$
	S	→	·CC,	$
	C	→	·cC,	c/d
	C	→	·d,	c/d
I_1:	S'	→	S·,	$
I_2:	S	→	C·C,	$
	C	→	·cC,	$
	C	→	·d,	$

I_3:	C	→	c·C,	c/d
	C	→	·cC,	c/d
	C	→	·d,	c/d
I_4:	C	→	d·,	c/d
I_5:	S	→	CC·,	$
I_6:	C	→	c·C,	$
	C	→	·cC,	$
	C	→	·d,	$
I_7:	C	→	d·,	$
I_8:	C	→	cC·,	c/d
I_9:	C	→	cC·,	$

I_{36}:	C	→	c·C,	c/d/$
	C	→	·cC,	c/d/$
	C	→	·d,	c/d/$
I_{47}:	C	→	d·,	c/d/$
I_{89}:	C	→	cC·,	c/d/$

Zust.	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47	-	1	2
1	-	-	acc	-	-
2	s36	s47	-	-	5
36	s36	s47	-	-	89
47	r3	r3	r3	-	-
5	-	-	r1	-	-
89	r2	r2	r2	-	-

4.6.5 LALR(1)- Syntaxanalyse

- Konzept zur Zustandsreduktion eines LR(1)-Parsers
- Beispiel: Anwendung der Zustandsreduktion
- Konstruktion der LALR(1)-Parser-Tabelle
- Vergleich von LALR(1) und LR(1)

Erste Zusammenfassung

Grammatik G

1	S'	→	S
3	S	→	CC
4	C	→	cC
5		|	d

Zust.	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47	-	1	2
1	-	-	acc	-	-
2	s36	s47	-	-	5
36	s36	s47	-	-	89
47	r3	r3	r3	-	-
5	-	-	r1	-	-
89	r2	r2	r2	-	-

- Tabelle heißt LALR-Syntaxtabelle für G
- gibt es **keine** Syntaxanalysekonflikte, dann ist die gegebene Grammatik G eine LALR(1)-Grammatik
- LR- und LALR-Parser verhalten sich bei fehlerfreien Eingaben gleich
- bei fehlerhaften Eingaben kann der LALR-Parser noch weitere Reduktionen ausführen, nachdem der LR-Parser bereits einen Fehler erkannt hat
- **ABER**: er kann aber niemals schieben, nachdem der LR-Parser einen Fehler erkannt hat.

→ vorgestellte Heuristiken der Fehlerbehandlung der Top-Down-Verfahren sind auch hier anwendbar

Syntaxanalysekonflikte

- **Annahme**: unsere Ausgangsgrammatik sei vom Typ LR(1)
  (d.h. die kanonische LR(1)-Analyse verursacht keine Syntaxanalysekonflikte)

- **Frage**: können Syntaxanalysekonflikte nach Zustandsreduktion bei einer LALR-Analyse entstehen?

- **Antwort**: Shift-Reduce-Konflikte sind nicht möglich,
  aber Reduce-Reduce-Konflikte können entstehen

Plausibilitätsbetrachtung

- **Ann.**:
  a) Grammatik sei vom Typ LR(1), also konfliktfrei!

  b) Shift-Reduce-Konflikt trete für LookAhead a in der Vereinigung auf
  d.h., es gibt zwei Elemente
  - [A→ α•, **a**] verlangt Reduktion für A→ α
  - [B→ β•**a**γ, **b**] verlangt Schieben

- dann muss es separate Zustände gegeben haben
  - [A→ α•, **a**] muss dann zu einer Menge I gehört haben
  - da außerdem alle Kerne der betreffenden ursprünglichen Terme gleich sein mussten, muss die Vereinigung auch ein Element [B→ β•**a**γ, **c**] für irgend ein **c** enthalten haben
  - dann hat aber dieser Zustand bereits ein Shift-Reduce-Konflikt bei **a**
    (**Widerspruch** zur Annahme Grammatik sei LR(1))

Fazit:
- Schiebe-Aktionen sind nur vom Kern abhängig (nicht vom LookAhead)
- Zustandsvereinigung bringt keine Shift-Reduce-Konflikte hervor

Reduce-Reduce-Konflikt

LR(1)- Grammatik

1	S'	→	S
2 | 3 | 4 | 5	S	→	**a** A **d** | **b** B **d** | **a** B **e** | **b** A **e**
6	A	→	**c**
7	B	→	**c**

generiert die Zeichenketten:

acd
ace
bcd
bce

gültige Vorsilbe ac: Zustand I_X= {[A→ c•, d], [B→ c•, e]}

gültige Vorsilbe bc: Zustand I_Y= {[A→ c•, e], [B→ c•, d]}

➔ keine der Mengen ruft einen Konflikt hervor, auch ihre Kerne sind gleich

➔ **Vereinigung**

I_{XY}:			
	A	→ c•,	**e/d**
	B	→ c•,	**e/d**

➔ **Konflikt**: Reduktionen sind sowohl mit A→c als auch mit B→c möglich

Effizientere Konstruktionen von LALR(1)-Syntaxtabellen

Frage: kann der Aufbau der vollständigen kanonischen Sammlung an Mengen von LR(1)-Elementen **kompakter** erfolgen?

Antwort: ja
(Compilergeneratoren arbeiten häufig nach diesem Prinzip)

- Aufbau eines LR(0)-Zustandsübergangsgraphen
- Erweiterung des LR(0)- zu einem LALR(1)- Zustandsübergangsgraphen durch Einführung der **LookAhead-Mengen**

Algorithmus aber komplizierter und weniger einsichtig
(Verzicht auf eine Darstellung)

Verdichtung von LR-Parsertabellen

- typische Programmiersprachen
  (50..100 Terminale, 100 Produktionen)

  ➔ LALR-Analyse benötigt
  - ≈ 100 Zustände
  - goto1 20.000 Einträge (**8 Bit pro Eintrag**)
    effiziente Kodierung 2-dimensionaler Felder

- Verwendung von Zeigern bei identischen Zeilen
  Zeiger
  für Zustände mit den gleichen Aktionen
  zeigen auf den gleichen Platz

Präzedenzen: „Aufbohren“ von LR(1)-Grammatiken

- Präzedenzen und Assoziativitäten können ebenfalls benutzt werden, um Shift-Reduce-Konflikte in mehrdeutigen Grammatiken aufzulösen
  - LookAhead mit höherer Präzedenz, dann **shift**
  - dieselbe Präzedenz, rechts-assoziativ, dann **shift**
  - dieselbe Präzedenz, links-assoziativ, dann **reduce**

- Vorteile:
  - bessere Darstellung/Auflösung mehrdeutiger Grammatiken
  - flachere Parse-Bäume und damit weniger Reduktionen

- Beispiel:
  - Grammatiken für arithmetische Ausdrücke

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilations
- Kapitel 2
  Formalismen
- Kapitel 3
  Lexikalische
- Kapitel 4
  Syntaktische
- Kapitel 5
  Parser-Gener
- Kapitel 6
  Statische Ser
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

- 4.1
  Einführung i
- 4.2
  Restrukturie
  Grammatike
- 4.3
  LL-Parser
- 4.4
  Beispiel: Ein
  (Parser, Übe
- 4.5
  Tabellengest
- 4.6
  LR-Parser

- 4.6.1
  Allgemeine Betrachtung
- 4.6.2
  LR(0)- Syntaxanalyse
- 4.6.3
  SLR(1)-Syntaxanalyse
- 4.6.4
  LR(1)- Syntaxanalyse
- 4.6.5
  LALR(1)-Syntaxanalyse
- 4.6.6
  Zusammenfassung

Bisherige Themen

- Grammatiken und Automaten
- Restrukturierung von Grammatiken
- Mehrdeutigkeit von Grammatiken
- Syntaxbäume

- Rekursiv-absteigende Parser
- LL-Grammatiken
- FIRST- und FOLLOW
- Tabellengesteuerte Top-Down-Parser

- Tabellengesteuerte Bottom-Up-Verfahren
- Shift-Reduce-Verfahren (allgemein)
- Closure, Goto, CFSM
- LR(0)-Syntaxanalyse
- SLR(1)-Syntaxanalyse
- LR(1)-Syntaxanalyse
- LALR(1)-Syntaxanalyse
- LR-, LALR-, SLR-Grammatiken

- **Parser**
  übernimmt Eingabezeichen vom Lexer u. behandelt Tokennamen als Terminalsymbole einer kfG.

- **Ableitung**
  Verfahren, mit dem ersten NT einer Grammatik beginnend u. Schritt für Schritt durch die RS einer ihrer Produktionen zu ersetzen.
  Wird das am weitesten links (bzw. rechts) stehende NT ersetzt, handelt es sich um eine Links- (bzw. Rechts-)ableitung.

- **Parse-Baum**
  Bild einer Ableitung. Für jedes NT gibt es einen Knoten. Kinder eines Knoten sind die Symbole, durch die das NT in der Ableitung ersetzt wird.
  Zwischen Parse-Bäumen, Links- u. Rechtsableitungen besteht eine 1:1-Entsprechung.

- **Mehrdeutigkeit**
  Eine Grammatik, für die eine Eingabe zwei oder mehr unterschiedliche Parse-Bäume oder zwei Links- oder mehr bzw. Rechtsableitungen aufweist ist mehrdeutig.
  In den meisten Fällen lässt sich eine Grammatik zu einer eindeutigen Grammatik für die gleiche Sprache umformen.
  Mehrdeutige Grammatiken führen jedoch u.U. zu effizienteren Parsern,
  wenn sie mit bestimmten Tricks bearbeitet werden.

- **Top-Down- und Buttom-Up-Syntaxanalyse**
  Unterscheidung ob die Parser vom Startsymbol oder den Terminalsymbolen ausgehend den Parse-Baum aufbauen.
  Zu den Top-Down-Parsern gehören die rekursiv-absteigenden u. die LL-Parser.
  LR-Parser gehören zu den Bottom-Up-Verfahren.
- **Entwurf von Grammatiken**
  Grammatiken für die Top-Down-Analyse sind schwerer zu entwerfen als für die Bottom-Up-Analyse. Linksrekursion muss beseitigt werden. Außerdem muss eine Linksfaktorisierung realisiert werden.
- **Rekursiv absteigende Parser**
  Parser verwenden für jedes NT eine Prozedur. Diese untersucht die Eingabe u. entscheidet, welche Produktion für ihr NT anzuwenden ist. Zum geeigneten Zeitpunkt werden Terminale in der RS der Produktion mit der Eingabe verglichen (Match-Prozedur). Falls die falsche Prozedur gewählt worden ist, ist eine Zurückverfolgung möglich.
- **LL(1)-Parser**
  Eine Grammatik, bei der es möglich ist, die richtige Prozedur zum Expandieren eines bestimmten NT allein anhand des nächsten Eingabesymbols zu erkennen, heißt LL(1)-Grammatik. Solche Grammatiken machen es möglich, eine prädiktive Syntaxtabelle zu erstellen, die für die auszuwählende Produktion jedes NT- und Look-Ahead-Symbol korrekt angibt.
  Fehlerbehandlung: Fehleroutinen für Tabelleneinträge ohne zulässige Produktionen.

- **Shift/Reduce-Syntaxanalyse**
  Bottom-Up-Parser entscheiden auf der Grundlage des nächsten Eingabesymbols (LookAhead) und des aktuellen Stack-Inhaltes, ob sie die nächste Eingabe auf den Stack verschieben oder einige Symbole reduzieren, die oben auf dem Stack liegen.
  Ein Reduktionsschritt ersetzt die RS einer Regel durch die LS der Regel (NT).

- **Sinnvolle Präfixe**
  Bei der Shift/Reduce-Analyse ist der Stack-Inhalt immer ein sinnvolles Präfix (ein Präfix einer rechten Satzform) , das immer nicht weiter reicht als bis zum rechten Ende des Handle dieser Satzform.
  Das Handle ist der Teilstring, der im letzten Schritt der Rechtsableitung dieser Satzform eingeführt worden ist.

- **Gültiges Item**
  Ein Item ist eine Produktion mit einem Punkt an einer Stelle der RS der Produktion.
  Es ist für ein sinnvolles Präfix gültig, wenn
  a) die Produktion dieses Item dazu dient, das Handle zu generieren und
  b) das sinnvolle Präfix alle Symbole links vom Punkt umfasst, aber nicht die darauf folgenden.

  Items bilden durch Anwendung des Closure-Operators die LR-Elemente.

- LR-Parser
  LR-Parser beliebiger Art legen zuerst die Mengen gültiger Items (die LR-Zustände) für alle möglichen sinnvollen Präfixe fest und verfolgen den Zustand der einzelnen Präfixe auf dem Stack. Die aktuell gültige Item-Menge bestimmt die Entscheidung bei der Shift/Reduce-Analyse. Die Reduktion wird bevorzugt, wenn ein gültiges Item mit dem Punkt am rechten Ende der RS vorliegt. Steht jedoch das LookAhead in einem gültigen Item direkt nach dem Punkt, wird es auf den Stack verschoben.

- Einfache LR-Parser (SLR-Parser)
  Bei der Analyse wird eine Reduktion, ausgelöst durch ein gültiges Item mit einem Punkt am rechten Ende der RS, dann ausgeführt, wenn das Lookahead-Symbol in einer Satzform auf die LS der betreffenden Regel folgen kann (Element von FOLLOW(LS)). Dabei handelt es sich um eine SLR-Grammatik. Die SLR-Analyse kann angewendet werden, wenn es keine Konflikte zwischen Parser-Aktionen gibt (Reduce/Reduce- bzw. Shift/Reduce-Konflikte).

- Kanonische LR-Parser (LR(1)-Parser)
  Das Verfahren benutzt Items, die um die Menge der Look-Ahead-Symbole erweitert sind, die auf die Anwendung der zugrundeliegenden Produktion folgen können. Reduktionen werden nur vorgenommen, wenn ein gültiges Item mit einem Punkt am rechten Ende der RS vorliegt und das aktuelle LookAhead-Symbol für diese Produktion zulässig ist. Ein kanonischer Parser kann einige Aktionskonflikte vermeiden, die bei SLR-Parser auftreten, umfasst aber häufig mehr Zustände für dieselbe Grammatik.

- **Lookahead-LR-Parser (LALR(1)-Parser))**
  LALR-Parser bieten viele Vorteile gegenüber SLR- und LR-Parser, indem sie Zustände mit denselben Kernen zusammenfassen (Item-Mengen), wobei die zugehörigen LookAhead-Symbole ignoriert werden. Die Anzahl der Zustände ist also dieselbe wie bei den SLR-Parsern, aber einige der Parser-Konflikte konnten beseitigt werden.
  LALR-Parser haben sich in der Praxis zur Methode der Wahl entwickelt.

- **Bootom-Up-Syntaxanalyse mehrdeutiger Grammatiken**
  In vielen wichtigen Situationen, zum Beispiel bei der Analyse arithmetischer Ausdrücke, können mehrdeutige Grammatiken benutzt werden, weil Zusatzinformationen (wie die Präzedenz von Operatoren) genutzt werden können, um Konflikte zu lösen.
  LR-Parser-Techniken sind daher für zahlreiche mehrdeutige Grammatiken einsetzbar.

- **Yacc/Bison**
  Parser-Generatoren dieser Art bauen aus einer (möglicherweise) mehrdeutigen Grammatik sowie aus Informationen zur Konfliktlösung die LALR-Zustände auf.
  Anschließend erstellen sie eine Funktion, die mithilfe dieser Zustände eine Bottom-Up-Syntaxanalyse durchführt und bei jeder Reduktion eine zugehörige Aktion ausführt.

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parser-Generatoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

→ Praktikumsveranstaltung